A generic SPARC architecture simulator capable of emulating LEON-based computer systems

TSIM3

TSIM3 Simulator User's Manual

USER MANUAL

RELEASED MAY 2024

Frontgrade Gaisler AB | Kungsgatan 12 | SE-411 19 | Göteborg | Sweden | frontgrade.com/gaisler

Table of Contents

1. Introduction

1.1. General

TSIM is a generic SPARC[1] architecture simulator capable of emulating LEON-based computer systems.

TSIM provides:

- Emulation of LEON2/3/4 processors in general and tailored emulation of specific chips
- High precision multi core CPU models with bus contention and inter-processor effects modelled on a per instruction level (LEON 3/4)
- FPU and MMU emulation
- Accelerated processor standby mode, allowing faster-than-realtime simulation speeds
- Standalone operation with scriptable Tcl command line
- Operation via remote connection from GNU debugger (GDB)
- Provided as a library to be included in larger simulator frameworks
- 64-bit time for practically unlimited simulation periods
- Detailed instruction traces and AMBA bus traces
- Memory emulation, including SRAM, SDRAM, PROM, SPI memories, local data RAM and local instruction RAM.
- L2 cache emulation with support for configurable replacement, cache way locking, as well as and protected and uncached regions (currenly only supported for LEON4)
- Emulation of interrupt controllers, UARTs, timers, IOMMU, SpaceWire interfaces, CAN controllers, SPI controllers, GPIOs, Ethernet interfaces, DAC, PCI
- Loadable modules to include user-defined device models
- Non-intrusive execution time profiling
- Non-intrusive code coverage monitoring
- Stack backtrace with symbolic information
- Check-pointing capability to save and restore complete simulator state
- Unlimited number of breakpoints and watchpoints
- Predefined simulation models for GR740, GR712RC, GR716A, GR716B, UT699, UT700 and AT697

1.2. Supported host platforms and system requirements

TSIM supports the following host platforms: Linux and Windows 10.

1.3. Obtaining TSIM

The primary site for TSIM is https://www.gaisler.com where the latest version of TSIM can be ordered and evaluation versions downloaded.

1.4. License

TSIM3 LEON2, TSIM3 LEON3 and TSIM3 LEON4 are licensed separately as separate products. Emulation of the GR716A and GR716B LEON3FT microcontroller is available in TSIM3 LEON3, but can also be licensed separately as TSIM3 GR716 that simulates GR716A and GR716B only.

The license text can be found in `license.txt` in the top directory after installation. Please contact sales@gaisler.com to acquire a license.

1.5. Evaluation version

An evaluation version of TSIM3 LEON3 is available from https://www.gaisler.com. The evaluation version may only be used for evaluation and internal testing and only during a period of 21 days without purchasing a license. See the `license.txt` file that is included in the archive for details.

The evaluation version simulates a basic dual core LEON3 system, but can be made to simulate a single core system using the `-numcpus` option. The evaluation version is limited to 32-bit time. It does not support check-

[1] SPARC is a registered trademark of SPARC International

pointing, loadable modules, library interface, code coverage, configuration of caches, diagnostic cache accesses, configuration of memory or chip flags such as `-gr712rc` or `-ut700`. The `-help` option can show options that available in the evaluation version. There are also upper limits on number of simulations restarts, instruction and bus trace lengths and number of frontend command executions.

1.6. Upcoming TSIM developments

TSIM3 is in active development and additional general features as well as additional IP core models and IP core subfeatures will be added over time. Currently supported devices and interfaces of modelled components are listed in their respective chapters, with cross references to further details and possible limitations.

Upcoming features include AMBA split response from the GR740 L2 cache and the ability for user models to override builtin I/O core models.

1.7. Problem reports

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

Customers with a valid support agreement may send questions to support@gaisler.com. Include a TSIM log when sending questions, please. A log can be obtained by starting TSIM with the command line switch `-log` *`filename`*. Try to include as much details as possible from commands such as **reg**, **inst/ahb** (enable history with **inst len** *`len`* or **ahb len** *`len`*), **bt** and with relevant debug options turned on. See also Chapter 27.

2. Installation

2.1. General

TSIM is distributed as a tar-file (e.g. tsim-leon3-3.1.11.tar.gz) with the following contents:

Table 2.1. TSIM content

Directory	Description
coverage	Source level coverage helper scripts
doc	TSIM documentation
examples/input	Example loadable modules for interfaces
examples/test	Example programs
examples/modules	Example loadable modules
license.txt	TSIM license text
share	Tcl distribution
tsim/linux-x64	TSIM binary for Linux
tsim/win64	TSIM binary for Windows

On Linux, the tar-file can be installed at any location with the following command:

```
tar xf tsim-leon3-3.1.11.tar.gz
```

On windows, the archive can be unpacked e.g. with the freely available 7-zip application.

TSIM must find the `share` directory to work properly. TSIM will try to automatically detect the location of the folder. If TSIM fails to automatically detect the folder, then the environment variable `TSIM_SHARE` can be set to point a moved `share` folder. If TSIM fails to find the `share` folder altogether it will fail to start up.

2.2. License key installation

TSIM is licensed using a Sentinel LDK USB hardware key and has support for node-locked and floating license keys. The type of key can be identified by the colour of the USB dongle. The node-locked keys are purple and the floating license keys are red.

1. Node-locked keys (purple USB key)

For node-locked keys, the Sentinel LDK Run-time for the key must be installed before the key can be used (see below).

2. Floating keys (red USB key)

In the case of floating keys, the Sentinel LDK Run-time must be installed on the server and the client computer (see below).

Sentinel LDK communicates via TCP and UDP on socket 1947. This socket is IANA-registered exclusively for this purpose. By default the client will find the server by issuing a UDP broadcast to local subnets on port 1947.

If the broadcast method is not working or unwanted, then advanced network settings can be setup on the client side via the Sentinel Admin Control Center. The Sentinel Admin Control Center is accessed by opening the URL localhost:1947 in a web browser. The network settings are reached by selecting "Configuration" in the menu and then selecting the "Access to Remote License Managers" tab. Detailed information on how to setup the network settings can be found by selecting "Help" in the menu.

3. Sentinel LDK Runtime

The latest runtime can be found at the TSIM download page [https://www.gaisler.com/index.php/downloads/simulators]. Included in the downloaded Sentinel LDK runtime archive is a README file which contains detailed installation instructions.

Administrator privileges are required on Windows. On Linux it is required that the runtime is installed as root user.

3. Operation

3.1. Overview

TSIM can operate in tree modes: standalone, used as a library and attached to GDB. In standalone mode, LEON applications can be loaded and simulated using a scriptable Tcl based command line interface. A number of commands are available to drive, investigate and interact with the simulation. When TSIM is used is a library. TSIM can be driven via a C API (see Chapter 6). This API makes standalone commands available as well as additional functionality. When attached to GDB, TSIM acts as a remote GDB target (see Section 3.12). Applications are loaded and debugged through GDB (or a GDB front-end such as DDD or Eclipse). It this mode it is also possible to use the standalone commands through the “monitor” GDB command.

3.2. Starting TSIM

TSIM is started as follows on a command line:

tsim-leon2 [*options*] [*input_files*]
tsim-leon3 [*options*] [*input_files*]
tsim-leon4 [*options*] [*input_files*]

Please note that when starting TSIM with a chip option, e.g. `-gr712rc` or `-gr716a`, TSIM will configure chip specific features. This includes CPU configuration parameters like caches, MMU, FPU, as well as what interfaces are present and how they are configured. Thus, when using a chip option there is no need to manually configure parameters that affects configuration internal to the chosen chip. If a parameter is set by both a chip option and a option directly, TSIM will always use the direct option.

When using a TSIM3 GR716 license, the `-gr716a` option is implicitly declared and thus GR716A features are already enabled. For GR716B use the `-gr716b` option. Some of the following options that are not supported by GR716A/B or is already enabled by the `-gr716a`/`-gr716b` option have been disabled in the GR716 only release. This does not apply to `-gr716a` or `-gr716b` when using a TSIM LEON3 license.

Many options can be used without an argument to enable or disable a feature but can also take an optional `1` or `0` as an argument. Many of these are documented as having the optional argument `[0|1]` without any description of the optional argument. For these cases, regardless of if the option enables or disables something a `1` argument is the same as no argument and will work as per the description of the option and a `0` argument will invert the meaning of the option. This can be used to override something enabled or disabled by earlier options.

For standalone TSIM, command line options can also be specified in the file `.tsimcfg` in the home directory. This file, if present, will be read at startup the contents will be prepended to the options given on the command line. In other words, options from the command line will, when possible, override options specified in the config file. See the `-cfg` option for how to turn this off or how to use a different file.

The following command line options are supported by TSIM:

`-ahbstatus [0|1]`
: Adds AHB status register support.

`-asi1allocate [0|1]`
: Makes ASI 1 reads allocate cache lines (LEON3/4 only). This is enabled by default.

`-at697e`
: Set parameters according to the Atmel AT697E device (LEON2 only). See Chapter 14 for details on AT697 emulation.

`-banks <1|2|4>`
: Sets how many RAM banks the SRAM is divided on. Supported values are 1, 2 or 4. Default is 1.

`-batch`
: Exit after running startup scipts built from `-e`, `-c` instead of staring an interactive session. On Tcl errors a non-zero exit code will be returned by TSIM.

`-bootstrap` *`val`*
: Sets the GR716A/B bootstrap register to *`val`* (GR716A/B) only). Defaults to 0x0000c004.

`-bopt [0|1]`
: Enables idle-loop optimisation (see Section 4.1.6).

`-bp [0|1]`
: Enables emulation of LEON3/4 branch prediction

`-bus_ov [0|1]`
Allow modules to override models on the bus instead of resulting in an address space collision error message. When combined with `-v`, TSIM will inform about areas that are overridden.

`-bz [0|1]`
Halt execution on all traps except privileged_instruction, fpu_disabled, window_overflow, window_underflow, asynchronous_interrupt and trap_instruction (As GRMON does when not using GRMON's -nb option). This halts at the pc and in the register window of the trapping instruction. Note that this does not function as an ordinary break in execution; continuing from this halt will re-execute the trapping instruction. This does not affect debugging through GDB. Use instead the `-nb [0|1]` option to set up that behaviour.

`-c` *`file`*
Evaluate the contents in the file *`file`* at startup. This is run through TSIM's Tcl interpreter and can thus contain Tcl code in general, including TSIM commands. This is a convenient way for specifying additional Tcl procedure definitions, for specifying simple sequences of TSIM commands as well as setting up more elaborate Tcl scripting of TSIM. See also the `-e` option on how to specify commands on the command line. Multiple `-c` and/or `-e` options can be given and will be evaluated in order.

`-cfg` *`file`*`|none`
Reads extra configuration options from *`file`*. If file name is "none" it will prevent a default (for standalone TSIM) configuration file `.tsimcfg` from the home directory from being read. Options from the command line will override options specified in the config file.

`-cfgreg_and` *`and_mask`*, `-cfgreg_or` *`or_mask`*
LEON2 only: Patch the Leon Configuration Register (0x80000024). The new value will be: *`(reg & and_mask)`*`|` *`or_mask`*.

`-cas [0|1]`
Enable emulation of the CASA instruction, or disable with a 0 argument. (LEON3/4 only). Enabled by default. Chip options enables or disables CASA support according to the corresponding chip.

`-dcsize` *`size`*
Defines the set-size (KiB) of the LEON data cache. Allowed values are powers of two in the range 1 - 64 for LEON2 and 1-256 for LEON3/4. Default is 4 KiB.

`-dlock [0|1]`
Enable data cache line locking. Default is disabled.

`-dlram` *`addr size`*
Allocates *`size`* KiB of local data RAM (a.k.a. tightly coupled data memory and data scratch-pad RAM) at address *`addr`*. (LEON3/4)

`-dlsize <16|32>`
Sets the line size of the LEON data cache (in bytes). Allowed values are 16 or 32. Default is 16.

`-drepl <rnd|lru|lrr>`
Sets the replacement algorithm for the LEON data cache. Allowed values are `rnd` (default for LEON2) for random replacement, `lru` (default for LEON3/4) for the least-recently-used replacement algorithm and `lrr` for the least-recently-replaced replacement algorithm.

`-dsets` *`sets`*
Defines the number of sets in the LEON data cache. Allowed values are 1 - 4. Defaults to 1 for LEON2 and 4 in general. Is set to 1 in the evaluation version.

`-e` *`command(s)`*
Executes *`command(s)`* at simulator startup. This is run through TSIM's Tcl interpreter and thus does not not need to be a single command. For example, a string containing semicolon separated commands can be specified and will then run in sequence. See also the `-c` option on how to specify commands in a file. Multiple `-e` and/or `-c` options can be given and will be evaluated in order.

`-eclipse [0|1]`
Enable some special handling of the GDB protocol when connecting with Eclipse.

`-ext` *`nr`*
Enable extended IRQ in the interrupt controller with extended IRQ number *`nr`* (LEON3/4 only).

`-fast_uart [0|1]`
Run UARTs at infinite speed, rather than with correct baud rate.

`-freq` *`system_clock`*
Sets the simulated system clock in MHz. Default is 50.

`-gdb` *`[port]`*
Listen for GDB connection directly at start-up. If the port is not specified, the default port number 1234 is used. See also the `-port` option that changes the default GDB server port number without starting the server.

`-gdbuartfwd [0|1]`
Forward UART output to GDB when being connected over GDB. Which UART if any is forwarded is determined by the `-u` *`[index]`* option. The default behaviour is for GDB to not change UART forwarding behaviour.

`-gr712rc`
Set parameters to emulate the GR712RC device. See Chapter 7 for details on GR721RC emulation.

`-gr716a`
Set parameters to emulate the GR716A device. See Chapter 8 for details on GR716A emulation.

`-gr716b`
Set parameters to emulate the GR716B device. See Chapter 9 for details on GR716B emulation.

`-gr740_no_spwrtr`
For backwards compatibility reasons this option can disable and remove the router for GR740, and the AMBA ports will be turned to 4x GRSPW2 cores. This also changes the PnP entries as well as the IOMMU group ID.

`-grfpu`
Emulate the GRFPU floating point unit.

`-grfpulite`
Emulate the GRFPU-lite floating point unit (LEON3/4).

`-help` *`[option]`*
List short help on all available options or show specific help for a given option. Many options specific to certain cores will only be available when a chip option, that instantiates models that adds more options, is also given together with the `-help` option. Without an argument (i.e. it being the last option given), this displays short help for all available options. When the name of another option is given as an argument to `-help`, it will print, potentially more detailed, help about that option specifically.

`-swbp [0|1]`
Enable use of software breakpoints for GDB breakpoints. By default TSIM uses hardware breakpoints for GDB breakpoints. This does not affect standalone TSIM breakpoints.

`-stack` *`addr`*
Set initial stack pointer.

`-icsize` *`size`*
Defines the set-size (KiB) of the LEON instruction cache. Allowed values are powers of two in the range 1 - 64 for LEON2 and 1-256 for LEON3/4. Default is 4 KiB.

`-ilock [0|1]`
Enable instruction cache line locking. Default is disabled.

`-ilram` *`addr size`*
Allocates *`size`* KiB of local instruction RAM (a.k.a. tightly coupled instruction memory and instruction scratch-pad RAM) at address *`addr`*. (LEON3/4)

`-ilsize <16|32>`
Sets the line size of the LEON instruction cache (in bytes). Allowed values are 16 or 32. Default is 16 for LEON2/3 and 32 for LEON4.

`-irepl <rnd|lru|lrr>`
Sets the replacement algorithm for the LEON instruction cache. Allowed values are `rnd` (default for LEON2) for random replacement, `lru` (default for LEON3/4) for the least-recently-used replacement algorithm and `lrr` for the least-recently-replaced replacement algorithm.

`-isets` *`sets`*
Defines the number of sets in the LEON instruction cache. Allowed values are 1 - 4. Defaults to 1 for LEON2 and 4 in general. Is set to 1 in the evaluation version.

`-log` *`filename`*
Logs the console output to *`filename`*. If *`filename`* is preceded by '+' output is appended.

`-mcfgX` *`value`*
Set the reset value of memory configuration register *`X`*, where *`X`* can be 1, 2 or 3.

`-mflat [0|1]`
This switch should be used when the application software has been compiled with the gcc `-mflat` option, and debugging with GDB is done.

`-mmu [0|1]`
Enable MMU support, or disable with a 0 argument. By default LEON3 and LEON4 has MMU support, but LEON2 does not. Chip options enables or disables MMU support according to the corresponding chip.

`-mod` *`file`*
Loads an user specified `loadable_module` from *`file`*. The environment variable TSIM_MODULE_PATH can be used as a list of search paths. See Section 5.1.1 for details.

`-mul` *`value`*
Set instruction cost of smul/umul to *`value`*.

`-nb [0|1]`
Do not break on error exceptions when debugging through GDB. To affect standalone TSIM or TLIB behaviour, see instead the `-bz [0|1]` option.

`-nofpu [0|1]`
Disables the FPU to emulate a system without FPU. Any floating-point instruction will generate an FP disabled trap.

`-ni [0|1]`
Prevents the GDB server from bootloader-like initialisation when using the **gdb reset** command and when starting the GDB server before any simulation has been done. No other commands are affected.

`-mac [0|1]`
Enable LEON MAC instructions.

`-nosram [0|1]`
Disable SRAM on startup. When SRAM is disabled, SDRAM will appear at 0x40000000.

`-nothreads`
Disable threads support.

`-bmthreads`
Force bare metal thread support, even when an OS is detected. Bare metal thread support consists of reporting each CPU as a thread to GDB. Bare metal thread support is default if no OS is detected.

`-nov8 [0|1]`
Disable SPARC V8 MUL/DIV instructions.

`-nrtimers` *`val`*
Adds support for more than 2 timers (in one timer unit). Value *`val`* can be in the range of 2 - 7 (LEON3/4 only). Default: 2. See also the `-sametimerirq [0|1]` and `-timerirqbase` *`number`* switches.

`-numcpus` *`value`*
Set number of CPUs between 1 and 4. In the LEON3 evaluation version, the maximum number of CPUs is limited to 2.

`-nwin` *`win`*
Defines the number of register windows in the processor. Valid range is between 2 and 32. The default is 8. Only applicable to LEON3/4.

`-port` *`port`*
Set the port number *`port`* to be used for GDB communication. The default port number is 1234. The port number can also be specified with the `-gdb` option or the **gdb** command.

`-pr [0|1]`
Enable profiling automatically at startup.

`-ram` *`ram_size`*
Sets the amount of simulated RAM (KiB). Default is 8192 KiB.

`-ramwidth <8|16|32>`
By default, the RAM area at reset time is considered to be 32-bit. Specifying 8, 16 or 32 will initialise the memory width field in the memory configuration register to 8-, 16- or 32-bits. The only visible difference is in the instruction timing.

`-rfpart [0|1]`
Enable register window partitioning support.

`-rom` *`rom_size`*
Sets the amount of simulated ROM (KiB). Default is 8192 KiB.

`-romwidth <8|16|32>`
By default, the PROM area at reset time is considered to be 32-bit. Specifying 8, 16 or 32 will initialise the memory width field in the memory configuration register to 8-, 16- or 32-bits. The only visible difference is in the instruction timing.

`-rtems` *`ver`*
Override auto-detected RTEMS version for thread support. *`ver`* should be 46, 48, 48-edisoft or 410.

`-sametimerirq [0|1]`
Force the IRQ number to be the same for all timers (in one timer unit). Default: separate increasing IRQ numbers for each timer. (LEON3/4 only). See also the `-nrtimers` *`val`* and `-timerirqbase` *`number`* switches.

`-sdfreq frequency`
Set the frequency of the SDRAM in the SDCTRL in GR740. Default is 100 MHz.

`-sdramwidth <32|64>`
Set the SDRAM bus width of the SDCTRL in GR740 to 32 or 64 bit. Default is 64-bit. The only visible difference is in the instruction timing.

`-sdram` *`sdram_size`*
Sets the amount of simulated SDRAM (MiB). Default is 128.

`-sdbanks <1|2>`
Sets the number of SDRAM banks. Default is 1.

`-strict_reset [0|1]`
This enables strict reset behaviour for the memory controller. When this is not enabled, TSIM not only resets the memory controller configuration registers, but also sets up fields that are not reset in hardware. For example RAM banks sizes, RAM width are set up, and SDRAM is enabled if available and the RAM area is disabled of the `-nosram` option is used. This default behaviour is for convenience when working with RAM images. Enabling strict reset behaviour can be useful e.g. when testing boot loaders.

`-sym` *`file`*
Read symbols from *`file`*. This can be useful e.g. for self-extracting applications and applications that sets up non one-to-one MMU mapping.

`-tclwt`
Do all Tcl evaluation in a separate worker thread. This is needed for TLIB if a different thread than the one that called `tsim_init` needs to call a TSIM function that might result in Tcl evaluation. This is not only relevant to the `tsim_cmd` family of functions, but also `tsim_gdb`. See Section 6.5 for details. This option is only present for TLIB, not for standalone TSIM.

`-timer32 [0|1]`
Use 32 bit timers instead of 24 bit. (LEON2 only)

`-timerirqbase` *`number`*
Set the IRQ number of the first timer (in one timer unit) to interrupt number *`number`* (LEON3/4 only). Default: 8. See also the `-nrtimers` *`val`* and `-sametimerirq [0|1]` switches.

`-u` *`[index]`*
Connect the UART with the given index to stdin and stdout and set up up terminal or console in raw mode. If no index is given UART0 is chosen. By default this is enabled for UART0. A negative index makes sure that none of the UARTs are connected to stdin/stdout. See also `-uout` *`[index]`* for connecting a UART to stdout without configuring it. Maximum one UART can be connected to using the mutually exlusive `-u` *`[index]`* and `-uout` *`[index]`* options. See also `-uartX` *`device`* for connecting UARTs to serial devices.

`-uout` *`[index]`*
Connect the UART with the given index to stdout. If no index is given UART0 is chosen. This does not connect stdin and does no terminal or console configuration. This can be useful when stdin is closed and when stdin and/or stdout are redirected. A negative index makes sure that none of the UARTs are connected to stdout. See also `-u` *`[index]`* for connecting a UART to both stdin and stdout. Maximum one UART can be connected to using the mutually exlusive `-u` *`[index]`* and `-uout` *`[index]`* options. See also `-uartX` *`device`* for connecting UARTs to serial devices.

`-uartX` *`device`*
This option connects the chosen UART to a serial device. Here, X can be in the range 0 up to the number of UARTs (exclusive). See also `-u` *`[index]`* that is used to connect a UART to stdin/stdout.

On Linux, e.g. connecting the first uart to /dev/ttyUSB0 can be done with "-uart0 /dev/ttyUSB0". On Linux, using the device /dev/ptmx will create a pseudo-terminal pair with the chosen uart at one end. TSIM prints out the name of the other end of the pair to be opened by host software communicating with the chosen uart.

On Windows use //./com1, //./com2 etc. to access the serial ports. The serial port settings can be adjusted by opening the relevant entry under "Ports (COM and LPT)" entry in the Device Manager and choosing the "Port Settings" tab in the dialogue that pops up.

`-uart_fs <1|2|4|8|16|32>`
Set UART FIFO depth in characters (LEON3/4 only). This setting affects all APBUARTs in the system. Valid configurations are 1, 2, 4, 8 (default), 16 and 32 characters. If the FIFO depth is configured to 1 the UART FIFO is not present instead only the holding register is present and FIFO level interrupts are not present. The FIFO interface is available for both fast and accurate mode, however the transmitter side in fast mode never fills the FIFO since characters are always sent immediately.

`-upcounter [0|1]`
Enables upcounter registers (ASR22/23). For LEON3/4 only.

`-ut699`
Set parameters to emulate the UT699 device. Note that when `-ut699` is given, snooping will be set as non-functional. This also sets up TSIM to simulate only one APBUART core. See Chapter 11 for details on UT699 emulation.

`-ut699e`
Set parameters to emulate the UT699E device. This also sets up TSIM to simulate only one APBUART core. See Chapter 12 for details on UT699E emulation.

`-ut700`
Set parameters to emulate the UT700 device. This also sets up TSIM to simulate only one APBUART core. See Chapter 13 for details on UT700 emulation.

`-v`
Turn on verbose output.

`-vv`
Turn on very verbose output.

`input_files`
Executable files to be loaded into memory. The input files are loaded into the emulated memory according to the entry point for each segment. Recognised formats are elf32, aout and srecords.

3.3. Standalone mode commands

The TSIM command line interface is a Tcl driven command line interface with a number of different type of recognised commands. There are general TSIM commands that are always present, native Tcl commands (see Section 3.3.3) that allows for Tcl scripting, as well as core specific commands that are available if specific devices are present in the simulated hardware configuration. See Section 3.3.1, Section 3.3.3 and Section 3.3.5 respectively. The **help** command can also be used to show a listing with short help for all commands, and to show more detailed help about specific commands.

As long as there are no ambiguities, short forms of the commands are allowed. For example, **dis**, is interpreted as **disassemble**, but **re** is reported as ambiguous. TSIM offers tab completion on things like commands names, subcommand names, symbols, device names and debug flags. In addition tab completion on Tcl variables are possible when after typing a "$".

Commands that takes an address as an argument can in general also take a symbol as an argument in place of an address, as well as tab complete on symbols. Some commands can take optional *`cpuX`* arguments to select a specific CPU, or for some commands a set of CPUs. For such arguments *X* is in this case replaced with the CPU id. In other words to select CPU 0, "cpu0" is used. Such cpuX arguments must be placed last in the command call. For commands that does something in the context of a specific CPU, the current CPU is the one that is affected. The **cpu** can be used to change which CPU that is the current CPU.

Typing a 'Ctrl-C' will interrupt a running simulator. Note however that in order to abort user created Tcl loops, the script should manually break out of the loop if the Tcl `tsim::interrupt` variable is not zero.

If the file .tsimrc exists in the home directory, it will be used for standalone TSIM as a batch file and the commands in it will be evaluated as Tcl at startup. This can be used for commands to be executed as well as for defining Tcl procedures for later use.

3.3.1. General commands

Below is a description of general commands

ahb [-f *file*] [*length*]

Display the latest *length* (default 30) entries in the AMBA bus trace history. Using `-f` *filename* will write the AMBA bus trace to file rather than print it.

Note: CPU accesses to local instruction RAM and local data RAM do not in general go via the AMBA bus and thus do not show up in the AMBA bus trace history. The one exception is instruction fetch from dual-port local *data* RAM on GR716A/B.

ahb len *length*

Set the AMBA bus trace buffer length, clear the AMBA bus trace buffer and enable AMBA bus tracing. Setting it to zero clears and disables AMBA bus tracing.

batch *file* [*arguments...*]

Execute a Tcl script in the file *file*. During the script evaluation make argv0 contain the script filename, argv contain a list of all the arguments that appear after the filename and argc will be the length of argv. See also the `-c` option on how to specify commands in a file that is evaluated at startup.

bload *file* [*startaddr*]

Load the binary file *file* into memory starting at *startaddr*. The default *startaddr* is the start of RAM memory. If an L2 cache is present, it will be flushed and invalidated and the loaded content will be placed uncached in the memory behind the L2 cache.

boot [*address/symbol* | `-t`] [*instructions* | *amount timeunit*]

Performs a cold boot. In other words, resets the simulator and starts simulation from time 0 bootloader-like initialisation. The event queue is emptied but memory contents and any set breakpoints remain. If an L2 cache is present, it is flushed, invalidated and disabled. If an address or symbol is given, execution starts from there. Otherwise, the starting point is determined according to the following priority. If an entry point has been set with the **ep** command, execution starts from that entry point (which can be different for different CPUs). If no address is given and no entry point has been set, execution starts at the reset address. No entry points of loaded images are taken into account, in contrast to the **run** command.

The **boot** command never performs bootloader-like initialisation of the system before starting the simulation. Use the **run** command when such initialisation is desired.

If an address or symbol is specified, or `-t` is used instead of an address or symbol, an optional number of instructions or amount of time to stop after can also be specified. See Section 3.3.2 for the syntax for specifying time.

See Section 3.4 on Tcl return value.

bopt [**0**|**1**]

Enable (**bopt 1**), disable (**bopt 0**), or show the current status **bopt** of idle-loop optimisation (see Section 4.1.6).

bp [*cpuX...*]

Prints all breakpoints and watchpoints. With optional *cpuX* arguments, breakpoints and watchpoints can be shown for a subset of the available CPUs.

bp *address* [*cpuX...*]

Adds an breakpoint at *address*. With optional *cpuX* arguments, breakpoints can be set for a subset of the available CPUs.

bp delete [*num*]

Deletes breakpoint/watchpoint *num*. If *num* is omitted, all breakpoints and watchpoints are deleted.

bp watch *address* [*cpuX...*]

Adds a watchpoint at *address*. With optional *cpuX* arguments, watchpoints can be set for a subset of the available CPUs.

bt [*cpuX...*]

Print backtrace for the current or specified CPUs.

cont [*instructions* | *amount timeunit*]

Continue execution at present position, optionally for a number of instructions or an amount of time. See Section 3.3.2 for the syntax for specifying time.

See Section 3.4 on Tcl return value.

coverage enable *[merge|percpu]*
Enable coverage. Data will be merged for all CPUs if merge flag is specified, or recorded per CPU if percpu flag is specified. If no flag is specified then default is to merge. Note that changing coverage mode will reset the coverage data. See Section 3.9 for more details.

coverage disable
Disables coverage.

coverage save [*file_name*] [*cpuX...*]
Merge and write coverage data for specified CPUs to file (file name and CPU is optional). The coverage data will be merged for all CPUs if no CPU is specified. See Section 3.9 for more details.

coverage lcov [*file_name*] [*cpuX...*]
Merge and write coverage data for specified CPUs to file using the lcov output format (file name and CPU is optional). The coverage data will be merged for all CPUs if no CPU is specified. See Section 3.9 for more details.

coverage clear
Resets coverage data.

coverage print *address [len] [cpuX...]*
Print coverage data to console, starting at address. If no CPU is specified the data will be merged for all CPUs. Else merged data for specified CPUs will be printed. See Section 3.9 for more details.

cpu [**active** *X*]
List CPUs or switch CPU *X* to be the active CPU.

dbgon *flag*
Toggle *flag* debug for all applicable cores. See the *coreX*_**dbg** commands for which flags are available for different cores.

dcache print [*cpuX...*]
Print the data cache contents for the current or specified CPUs.

dcache flush [*addr*|*sym*] [*cpuX...*]
Flush the current or specified CPUs data cache, optionally for given address or symbol only.

dcache query <*addr*|*sym*> [*cpuX...*]
Print current or specified CPUs data cache status for given address or symbol.

dump *file address length*
Dumps memory content to file *file*, in whole aligned words. The *address* argument can be a symbol.

disassemble [*addr*] [*count*] [*cpuX...*]
Disassemble [*count*] instructions at address [*addr*] for the current CPU or for the specified CPUs. Default value for count is 16 and for *addr* the current program counter.

ep [**clear**] [*cpuX...*]
Clear entry point for execution on all or given CPUs.

ep [*address*] [*cpuX...*]
Show or set entry point for execution on all or given CPUs. When an entry point has not explicitly been set for a CPU, the entry point printed and returned is the entry point that would be used by the **run** command. Setting the entry point overrides the default start of execution address for the **run** and **boot** commands. The Tcl return value for this command is a list of all affected CPUs entry points. The list is sorted in ascending CPU index order.

event
Print events in the event queue. Only user-inserted events are printed.

exit [*val*]
Exit the simulator with exit value *val*, when given, or zero.

float [**-v**] [*cpuX...*]
Prints the FPU registers for the current or given CPUs. With the optional **-v** argument, the fields of the FSR registers are listed and denormalized numbers are marked.

gdb [*port*]
Start GDB server, listening for GDB connection, optionally on the given port. The default port is 1234, unless changed by the `-gdb` or `-port` option.

gdb reset
Prepares TSIM for a new run via GDB. This is in some cases needed before loading an image from GDB (or via GDB e.g. from Eclipse). See Section 3.12 for details. This should only be used in/via GDB as “monitor gdb reset”.

gdb postload

Performs final preparations after loading an image from GDB (or via GDB e.g. from Eclipse). This is in some cases needed when debugging multicore images. See Section 3.12 for details. This should only be used in/via GDB as “monitor gdb postload”.

go *address/symbol* [*instructions* | *amount timeunit*]

Continues simulation after having set the PC of the current CPU to the given address.

The **go** command never restarts simulation, resets the system or does any bootloader-like initialisation. Use the **run** or **boot** command when that is desired.

An optional number of instructions or amount of time to stop after can be specified. See Section 3.3.2 for the syntax for specifying time.

See Section 3.4 on Tcl return value.

help [*command*|*topic*]

Without an argument, print a help menu for TSIM commands. Using **help***command*, will show help for *command* when available. The **tcl** topic will list help for native Tcl commands.

hist [-v] [-f *file*] [*length*] [*cpuX*]

Displays the latest *length* (default 30) entries from both the current or given CPUs instruction trace buffers and AMBA bus trace buffers interleaved. Not that only one CPU can be specified at a time. Using `-f` *filename* will write the trace to file rather than print it. Using `-v` enables verbose output.

icache print [*cpuX...*]

Print the instruction cache contents for the current or specified CPUs.

icache flush [*addr*|*sym*] [*cpuX...*]

Flush the current or specified CPUs instruction cache, optionally for given address or symbol only.

icache query <*addr*|*sym*> [*cpuX...*]

Print current or specified CPUs instruction cache status for given address or symbol.

info reg [options] [*devicename*|*registername*|*addr*]...

Shows system registers. If one or more device names are passed to the command, then only the registers belonging them are printed. If one or more register names/addresses are passed, only those registers will be printed. See Section 3.3.4 on how to address registers by name. Use the **leon** command to list the names of the available devices in the system. If the option `-v` is specified then TSIM will print the field names and values of each register. If the option `-u` is used then TSIM will print register values (and field values, when also using `-v`) in unsigned decimal notation instead of hexadecimal. Note that some registers are not implemented in TSIM and thus will not show up. Note that for LEON2, in this release only the memory controller and PCI interface shows up in **info reg**. On LEON2, the **leon** command is used to show the non-PCI device registers.

inst [-v] [-f *file*] [*length*] [*cpuX*]

Display the latest *length* (default 30) instructions in the instruction trace buffer, for the current or given CPUs. Using `-f` *filename* will write the instruction trace to file rather than print it. Using `-v` enables verbose output.

inst len [*length*] [*cpuX*]

Set the instruction trace buffer length, clear the instruction trace buffer and enable instruction tracing, for all or the given CPUs. Setting it to zero clears and disables instruction tracing.

iommu apv decode <*base*>

Decodes APV starting at base *base*.

iommu cache flush

Flushes the IOMMU cache.

iommu cache show <*line*> <*count*>

Shows the contents of the IOMMU cache. Shows *count* lines starting at line *line*.

iommu cache write <*line*> <*data0...dataN*> <*tag*>

Write full cache line including tag to cache line *line* , i.e. the number of data words depends on the size of the cache line.

iommu pagetable lookup <*base*> <*addr*>

Lookup specified IO address *addr* in page table starting at *base*.

l2cache

Show L2 cache settings.

l2cache show data [*way*] [*count*] [*start*]
Prints the data of *count* cache lines of way *way* starting at cache line *start*.

l2cache show tag [*count*] [*start*]
Prints the tags of *count* cache lines, for all ways, starting at cache line *start*.

l2cache enable
Enable the cache.

l2cache disable
Disable the cache.

l2cache disable flushinvalidate
Disable the cache and all dirty cache lines are invalidated and written back to memory as an atomic operation.

l2cache invalidate
Invalidate all cache lines.

l2cache flush
Perform a cache flush to all cache lines.

l2cache lookup *addr*
Prints the data and status of a cache line if *addr* generates a cache hit.

l2cache flushinvalidate
Flush and invalidate all cache lines (copy-back).

leon
Display an overview of available peripherals and display the current CPUs configuration registers. Registers of individual peripherals can be displayed in detail with the **info reg** command. Note that for LEON2, in this release only the memory controller and PCI interface shows up in **info reg**. On LEON2, the **leon** command is used to show the non-PCI device registers.

load *files*
Load *files* into simulator memory. If an L2 cache is present, it will be flushed and invalidated and the loaded content will be placed uncached in the memory behind the L2 cache.

mcfgX [*value*]
Set or show the user defined value that is used to set the memory configuration register *X* when TSIM acts as a boot loader (e.g on **run**, but not **boot**). These commands do *not* set the corresponding registers when the commands themselves are evaluated. Here *X* can be 1, 2 or 3.

mem [*option*] *addr* [*count*]
memh [*option*] *addr* [*count*]
memb [*option*] *addr* [*count*]
Display memory at *addr* for *count* bytes. The **mem**, **memh** and **memb** commands shows and returns the result as words, half-words and bytes respectively. An unaligned addresses and lengths are rounded down. Unimplemented address areas are displayed as zero. Possible options affecting the format of the Tcl return value are:

-ascii If the -ascii flag has been given, then a single ASCII string is returned instead of a list of values.

-cstr If the -cstr flag has been given, then a single ASCII string, up to the first null character, is returned instead of a list of values.

-hex Give the -hex flag to make the Tcl return value be list hex strings, but without any 0x prefix. The numbers are always 2, 4 or 8 characters wide strings regardless of the actual integer value.

-x Give the -x flag to make the Tcl return value be a list of hex strings prefixed with 0x. The numbers after 0x are always 2, 4 or 8 characters wide strings regardless of the actual integer value.

mmu [*cpuX*]
Display the MMU registers for the current or given CPUs. Individual registers can also be read and written using the **reg** command with the register names printed by the mmu command prepended with mmu, e.g. “reg mmuctrl”.

mmu debug [*value*] [*cpuX*]
Set debug level for the MMU on current or given CPU.

mmu ctrl [*value*] [*cpuX*]
Display or set the value of the MMU control to *value* for the current or given CPUs.

mmu ctx [*value*] [*cpuX*]
Display or set the value of the MMU context register to *value* for the current or given CPUs.

mmu ctxptr [*value*] [*cpuX*]
Display or set the value of the MMU context pointer register to *value* for the current or given CPUs.

mmu table [**-v**] [*ctx*] [*cpuX*]
Display MMU table, optionally specifying a context. If no context is given, the current context will be used. The command shows the MMU table for the currently active CPU, unless the cpuX is used to specify a different CPU. The optional **-v** option can be used to show one line per PTE, otherwise neighboring regions with a linear mapping and with the same access permissions are presented as one line in the table. The permissions are presented on the form `crwxrwx` where a letter means that something is enabled, and a `-` means that it is disabled. The first position shows if the mapping is set cacheable. The next three positions shows if the mapping is readable, writeable and executable in supervisor mode. The last three shows the same for user mode.

mmu tlb [*cpuX*]
Display the TLB for the current or given CPUs.

nolog *cmd*
Suppress the log output of a command.

perf [*cpuX...*]
The **perf** command will display various execution statistics. By default, the statistics information for all CPUs that has been started are merged. With optional *cpuX* arguments, profiling can be shown for a subset of the available CPUs. Restarting simulation (e.g. via **run**, **boot**, or **reset**) also resets the statistic information.

perf reset
Reset the performance statistics. This can be used if statistics shall be calculated only over a part of the program. Restarting simulation (e.g. via **run**, **boot**, or **reset**) also resets the statistic information.

profile enable [*stime*]
Enable profiling on all CPUs, clearing any previous profiling information. Default sampling period is 1000 clock cycles, but can be changed by specifying *stime* as the number of clock cycles between samples.

profile disable
Disable profiling, but do not clear profiling information.

profile [*num*] [*cpuX...*]
Show profiling information. By default all symbols with enough samples to reach 0.01% is printed. With a *num* argument the number of printed lines are limited to *num*. By default, the profiling information for all CPUs that has been started during the sampling (including being started but in power down) are merged. With optional *cpuX* arguments, profiling can be shown for a subset of the available CPUs.

quit
Exits the simulator. Use the **exit** command to exit with a given exit value.

reg [*reg_name* [*value*]|*window*]... [*cpuX*]

Prints and sets the IU registers in the current or given register window on current or given CPU, prints and sets individual registers and returns register values as a Tcl list. **reg** without parameters prints the IU registers of the current register window in addition to a number of special registers. **reg** *reg_name* shows the value of the corresponding register. **reg** *reg_name value* sets the corresponding register to *value*. Multiple register can be set and read in the same reg command call. E.g. `reg g1 123 g2` sets the g1 register to 123 and shows the value of g1 and g2 as well as returns a Tcl list with the values of g1 and g2

Valid register names include g0-g7, o0-o7, l0-l7, i0-i7, psr, tbr, wim, y, pc and npc. The aliases fp and sp for i6 and o6, as well as the r0-r31 aliases are also supported. Depending on what registers the system at hand has, the asr16, asr17, asr18, asr20, asr22, and asr23 registers can also be accessed.

For systems with FPU, the registers f0-f31 and fsr are valid. In addition the names d0-d15 can be used to set/get double precision floats where e.g. d15 represents a double precision floats stored in f30 and f31. They can be set either with an integer (to set the underlying binary representation) or a floating point value.

For systems with caches, the cache registers cctrl (with alias ccr), icfg and dcfg can be accessed.

For systems with MMU, the MMU registers can be accessed with the names mmuctrl, mmuctx, mmuctxptr (with alias mmuctp) mmufsr, and mmufar.

To view a certain register window, use **reg** w*n*, where *n* is the index of the register window. To show or set a single register from a specific window, prepend w*n* to the register name, e.g. w1i2.

The Tcl return value is a list (when multiple registers are given otherwise a single value) of the value of all the accessed registers, including the ones that were set where the value is the one after setting it. Each floating point registers instead of a single value gets a tuple returned in the list, a floating point value as well as the raw underlying integer representation.

reset

Restarts the simulation (simtime is set to zero) and resets the system. If an L2 cache is present, it will be flushed, invalidated and disabled.

restore *file*

Restore simulator state from *file*. See Section 3.10 for details.

run [*address/symbol* | `-t`] [*instructions* | *amount timeunit*]

Resets the simulator and starts simulation from time 0. The event queue is emptied but memory contents and any set breakpoints remain. If an address or symbol is given, execution starts from there. Otherwise, the starting point is determined according to the following priority. If an entry point has been set with the **ep** command, execution starts from that entry point (which can be different for different CPUs). Otherwise, if an image has been loaded, execution starts from the entry point of that image. If no image has been loaded either, execution starts at the reset address.

The run command always performs bootloader-like initialisation of the system before starting the simulation. Use the **boot** command when no such initialisation is desired. If an L2 cache is present, it will be flushed, invalidated and then enabled as part of the this initialisation.

If an address or symbol is specified, or `-t` is used instead of an address or symbol, an optional number of instructions or amount of time to stop after can also be specified. See Section 3.3.2 for the syntax for specifying time.

See Section 3.4 on Tcl return value.

save *file*

Save simulator state to *file*. See Section 3.10 for details.

shell *cmd*

Execute the shell command *cmd* in the host system shell.

silent *cmd*

Suppress stdout of a command.

stack [**clear**|*address*] [*cpuX...*]

Show, clear or set initial stack pointer for the current or given CPU. Setting the stack pointer will override the default stack pointer. Clearing a set stack pointer will make TSIM go back to setting a default stack pointer.

step [`-v`] [*instructions* | *amount timeunit*]

Execute and disassemble one or more instructions or for a certain amount of time on the current CPU. Using `-v` enables verbose output. Any other CPUs will execute as usual, silently, in the resulting timespan.

See Section 3.4 on Tcl return value.

symbols *file*

Load symbol table from *file*.

symbols clear

Clear all knowledge of symbols.

symbols list

Prints a list of the loaded symbols.

symbols lookup *symbol*

Look up the address of the given symbol. Prints and returns the result.

thread [*info* | *bt*]

Prints thread info or thread backtrace. See also Section 3.13.1.

version

Prints the TSIM version and build date.

vmem [*option*] *addr* [*count*]
vmemh [*option*] *addr* [*count*]
vmemb [*option*] *addr* [*count*]

Same as **mem**, **memh** and **memh** respectively, but does a MMU translation on *vaddr* first whenever MMU is present and enabled.

vwmem *vaddr val...*
Write word with value *val* to virtual address *vaddr*. If MMU is not present or not enabled, no address translation is done. If several values are given, they are written to consecutive virtual word addresses.

vwmems *vaddr val* [*cpuX*]
Write a string, including null-termination, directly to simulated memory space starting at given virtual address for current or given CPU. If MMU is not present or not enabled, no address translation is done.

walk <*address*|*symbol*>[*cpuX*]
If the MMU is enabled printout a table walk for the given address or symbol on the current or given CPUs.

wmem, wmemh, wmemb *address value...*
Write a word, half-word or byte directly to simulated memory space. If several values are given, they are written to the consecutive word, half-word or byte addresses respectively.

wmems *address string*
Write a string, including NULL termination, directly to simulated memory space.

xwmem *asi address value*
Write a word to simulated memory using ASI=*asi*.

3.3.2. Time specification for commands

Commands such as **run**, **boot**, **cont**, **go** and **step** supports simulating for a specified amount of time.

If an amount without a unit is specified, execution will stop after the specified number of instructions. If an amount and a time unit (with whitespace between) is specified, the execution will continue until the given time has passed (relative to the current time). The following time units are supported:

Table 3.1. Time units for commands that run simulation

Argument	Unit
c	cycles
us	microseconds
ms	milliseconds
s	seconds
min	minutes
h	hours
d	days

3.3.3. Tcl commands

TSIM has built-in support for Tcl 8.6. All command lines entered through the command line interface as well as via the GDB monitor command or executed from TLIB will pass through a Tcl-interpreter. This enables e.g. loops, variables, procedures, scripts, and arithmetic calculations for the user. Commands like **mem**, **reg**, **run**, **go**, **cont** and **step** gives useful Tcl return values that can be used for scripting.

Although this manual does not list all supported native Tcl commands, the TSIM **help tcl** can be used list short help for all supported native Tcl commands and **help** *cmdname* can be used to list full help for a given Tcl command. The help for the native Tcl **info** command can be listed with **help tclinfo**.

3.3.4. Tcl variables

TSIM provides Tcl variables for commonly used values. Such as core registers and fields. The notation for registers are *coreX::register* and for fields *coreX::register::field*. This notation can be used to both read from a specific register and to set the value of it. Tab completion on these variables are supported.

3.3.5. Core specific commands

Many cores in the system has their own commands on the format *coreX_commandname*, where *X* is the index (starting from 0) off the core within the set of cores of the same type. For example **gpio0_status** shows the status of the first GPIO in the system. The availability of these commands depends upon what cores are present in the simulated system. The available cores in the simulated hardware can be shown with the **leon** command.

For some cores in the system there is a *coreX*_**status** command shows some additional status information. For some cores it is possible to enable extra debug information with their *coreX*_**dbg** command. This command takes a debug flag or a subcommand as argument. The flags are specific for each core type and explained in the respective chapter. Common for all *coreX*_**dbg** commands are the subcommands **all**, **clean** and **list** which will enable, disable or list all applicable debug flags respectively for the core in question.

gpioX_status
Print status for the GPIO core.

gpioX_dbg [*flag*|**all**|**clean**|**list**]
Toggle specific flag, set all, clear all, or list debug flags for the given GPIO core. See Section 18.4 for a list of debug flags.

canbusX_status
Prints the status information on the given CAN bus. Note that this is only used for systems with one or more GRCAN devices, not for CAN_OC.

grcanX_dbg [*flag*|**all**|**clean**|**list**]
Toggle specific flag, set all, clear all, or list debug flags for the given GRCAN core. See Section 15.3 for a list of debug flags.

grspwX_dbg [*flag*|**all**|**clean**|**list**]
Toggle specific flag, set all, clear all, or list debug flags for the given GRSPW/GRSPW2 core. See Section 20.3 for a list of debug flags for GRSPW cores and Section 21.3 for GRSPW2 cores.

grspwX_status
Print status for GRSPW2 core *X*.

grethX_dbg [*flag*|**all**|**clean**|**list**]
Toggle specific flag, set all, clear all, or list debug flags for the given GRETH core. See Section 17.3 for a list of debug flags.

grethX_status
Prints the status of greth core X.

grethX_connect *[ip[:port]]*
Connect to packet server at given IP address and optional port. Default port is 2224. If no IP address is specified, the default is localhost.

grethX_ping [*ip*]
Send an ARP request followed by a ping packet (regarding/addressed to the given IP address) to the GRETH and present any ping reply send by the GRETH. If no IP address is specified the address 192.168.0.80 is used by default.

grethX_dump *file*
Dump packets to Ethereal readable *file*.

grethX_reconnect *<0|1>*
Turn GRETH autoreconnect on or off.

can_ocX_connect *host:[port]*
Connect CAN_OC core X to packet server to specified server and TCP port.

can_ocX_server *port*
Open a packet server for CAN_OC core X on specified TCP port.

can_ocX_ack *<0|1>*
Specifies whether the CAN_OC core will wait for a acknowledgement packet on transmission. This command should only be issued after a connection has been established.

can_ocX_status
Prints out status information for the CAN_OC core.

can_ocX_dbg [*flag*|**all**|**clean**|**list**]
Toggle, set, clear, list debug flags for the CAN_OC core.

grpciX_status
Print status for PCI core X

grpciX_dbg
Toggle specific flag, set all, clear all, or list debug flags for the given grpci core. See Section 19.2 for a list of debug flags.

esapci0_dbg
Toggle specific flag, set all, clear all, or list debug flags for the given PCI core. See Section 25.2 for a list of debug flags.

grspwrtr_portX_connect *host:[port]*
Connect SpaceWire routers SpaceWire port *X* to packet server at specified server and TCP port.

grspwrtr_portX_server *port*
Open a packet server for SpaceWire routers SpaceWire port *X* on specified TCP port.

spwrtr_dbg [*flag*|**all**|**clean**|**list**]
Toggle specific flag, set all, clear all, or list debug flags for the given GRSPWROUTER core. See Section 22.3 for a list of debug flags.

spiX_dbg [*flag*|**all**|**clean**|**list**]
Toggle specific flag, set all, clear all, or list debug flags for the given SPI core. See Section 23.4 for a list of debug flags.

bootstrap_status
Prints the bootstrap register.

print_dummies
List all dummy register areas, if any. For some configurations TSIM implements registers of some cores as dummy registers. They can be read and written, but writes do not stick and reads will always yield 0.

3.4. Return values for simulation stop reasons

Tcl commands such as **run**, **boot**, **cont**, **go** and **step** that starts simulation returns a Tcl result that indicates why simulation was stopped. They return it as a list of stopping reasons, in form of a signal name, for each CPU according to Table 3.2.

When `SIGINT` is the returned reason, each CPU will have that in its list entry in the returned list. Otherwise, the CPU that caused a stop will get the stop reason in its list entry and all other CPUs will have `SIGSTOP` in theirs. If simulation stops for another reason than a CPU triggering some condition, all CPUs will be marked `SIGSTOP`. This includes stopping due to a after a given number of instructions (even though it is tied to a certain CPU).

Table 3.2. Returned reasons for simulation stopping

SIGINT	Simulation stopped due to interruption, e.g. Ctrl-C. All CPUs marked thusly.
SIGSTOP	Simulation stopped, not because of a condition of the CPU itself.
SIGTRAP	Simulation stopped due to breakpoint hit
SIGSEGV	Simulation stopped due to processor in error mode
SIGTERM	Simulation stopped due to program termination

For example when CPU 1 in a dual core system hits a breakpoint:

```
tsim> set result [cont]

  CPU 1 stopped at breakpoint 2: t_wovf
tsim> puts $result
  SIGSTOP SIGTRAP
```

For TLIB, the returned stop reasons codes from `tsim_cont` and `tsim_get_stopreason` are logically the same, but in a different format. Instead of a list of names, the returned information is in form of one stop reason and one CPU ID for the CPU responsible for the stop. The stop reason is an integer matching macros with the names as in Table 3.2. When the reason is `SIGINT` and `SIGSTOP`, `-1` will be returned as CPU ID, and for the other reasons, the ID of the CPU causing the stop.

3.5. Symbolic debug information

TSIM will automatically extract (.text) symbol information from elf-files. The symbols can be used where an address is expected:

```
tsim> bp main
  breakpoint 1 at 0x310013b0:  main + 0x4

tsim> dis strcmp 5

   31004198  82120009   or        %o0, %o1, %g1            strcmp
   3100419c  80886003   andcc     %g1, 0x3, %g0            strcmp + 0x4
```

```
   310041a0  3280001e   bne,a     0x31004218              strcmp + 0x8
   310041a4  c24a0000   ldsb      [%o0], %g1              strcmp + 0xc
   310041a8  c2024000   ld        [%o1], %g1              strcmp + 0x10
```

The **symbols list** command can be used to lookup and display all symbols. Symbols are automatically read from files loaded with the **load** command. To to read in symbols from an alternate (elf) file use **symbols** *`file`*.

```
tsim> symbols dhrystone.elf
  read 476 symbols
tsim> symbols lookup strcmp
  Found address 0x31004198
tsim> symbols list
  ...
  0x31000000 L __text_start
  0x31000000 L __bcc_trap_table
  0x31000000 L __bcc_entry_point
  0x31001000 L __bcc_crt0
  0x310010c8 L deregister_tm_clones
  0x31001108 L register_tm_clones
  0x3100115c L __do_global_dtors_aux
  0x31001200 L call___do_global_dtors_aux
  0x3100120c L frame_dummy
  0x3100123c L call_frame_dummy
  0x31001248 L Proc_1
  0x310012f0 L Proc_2
  0x31001324 L Proc_3
  0x31001360 L Proc_4
  0x31001394 L Proc_5
  0x310013ac L main
  0x31001868 L Proc_6
  0x310018c4 L Proc_7
  0x310018d8 L Proc_8
  0x31001934 L Func_1
  0x31001964 L Func_2
  0x310019ac L Func_3
  ...
```

Reading symbols from alternate files is necessary when debugging applications were the image does not contain debugging symbols. This includes self-extracting applications and applications extracted by a bootrom, e.g. bootrom created with mkprom, application software images unpacked by the GR716A/B boot ROM and Linux images.

3.6. Breakpoints and watchpoints

TSIM supports execution breakpoints and write data watchpoints. In standalone mode, hardware breakpoints are always used and no instrumentation of memory or changes to memory are made. TSIM's hardware breakpoints are entirely handled outside the simulation model. No DSU hardware breakpoints are emulated. Breakpoints and watchpoints are set, displayed and deleted with the **bp** command.

When using the GDB interface, the GDB ‘break’ command requests TSIM to set breakpoints, which by default is handled using TSIM's internal hard breakpoints. If `-swbp` is enabled, TSIM lets GDB handle software breakpoints by itself overwriting the breakpoint address with a ‘ta 1’ instruction. In addition, hardware breakpoints can always be inserted by using the GDB ‘hbreak’ command. Data write watchpoints are inserted using the ‘watch’ command. A watchpoint can only cover one word address, block watchpoints are currently not available.

3.7. Profiling

The profiling function calculates the amount of execution time spent in and under each subroutine of the simulated program. The profiling is non-intrusive. The Profiling does not have any affect on the execution in terms of simulated time and no changes needs to be done to the instrumented code. The profiling is made by periodically sample the execution point and the associated call tree. In other words, the profiling is inclusive. At each sample point all functions in the call stack are considered to be be executing, e.g. time spent in a function `g` called by a function `f` will tally up samples for both `f` and `g`. Cycles in the call graph are properly handled, as well as sections of the code where no stack is available (e.g. trap handlers).

The profiling information is printed as a list sorted on highest execution time ratio using **profile**. For a particular symbol, the presented percentage number is the percentage of all samples that the symbol was found in the call stack. By default all symbols with enough samples to reach 0.01% is printed. With a numeric argument the number of printed lines are limited to the given number of lines. By default, the profiling information for all CPUs that has

been started during the sampling (including being started but in power down) are merged. With optional *cpuX* arguments, profiling can be shown for a subset of the available CPUs.

Profiling is enabled through the **profile enable** command. The sampling period is by default 1000 clocks which typically provides a good compromise between accuracy and performance. Other sampling periods can also be set through **profile enable** *n* where *n* is a the profile period in clock cycles. Profiling can be disabled through the **profile disable** command.

Below is an example profiling the Dhrystone benchmark:

```
tsim> load dhrystone.elf
...
tsim> profile enable
  Profiling enabled with sample period 1000
tsim> run
...
tsim> profile
  Merged profile for all started CPUs:

  function                      ratio(%)
  --------                      --------
  __bcc_crt0                       99.99
  main                             99.79
  Func_2                           29.22
  strcmp                           25.64
  memcpy                           17.09
  Proc_8                            8.34
  Func_1                            4.77
  Proc_7                            4.37
  Proc_6                            1.78
tsim>
```

3.8. Performance

The **perf** command will display various execution statistics. A **perf reset** command will reset the statistics. This can be used if statistics shall be calculated only over a part of the program. Restarting simulation (e.g. via run, boot, or reset) also resets the statistic information.

By default, the performance information for all CPUs that has been started are merged. With optional *cpuX* arguments, performance can be shown for a subset of the available CPUs.

Below is an example of performance statistics

```
tsim> perf
  Performance statistics for CPU 0
   Cycles : 467054246
   Instructions : 334033114
   Overall CPI : 1.40

   CPU performance (50.0 MHz)  :  35.76 MOPS (35.76 MIPS, 0.00 MFLOPS)
   Simulated time              :   9.34 s
   Processor utilisation       : 100.00 %

  Performance of the simulator:
   Real-time performance       : 123.93 %
   Simulator performance       :  44.32 MIPS
   Used time (sys + user)      :   7.54 s
tsim>
```

3.9. Code coverage

To aid software verification, TSIM includes support for code coverage. When enabled, code coverage keeps a record for each 32-bit word in the emulated memory and monitors whether the location has been read, written or executed. Coverage information can be recorded individually per CPU or merged for all CPUs. Coverage information will be recorded also for cache hits. The coverage function is controlled by the coverage command:

coverage enable [merge|percpu]	Enable coverage. Data will be merged for all CPUs if merge flag is specified, or recorded per CPU if percpu flag is specified. If no flag is specified then default is to merge. Note that changing coverage mode will reset the coverage data.

coverage disable	Disable coverage
coverage save [filename] [cpuX...]	Merge and write coverage data for specified CPUs to file (file name and CPU is optional). The coverage data will be merged for all CPUs if no CPU is specified.
coverage lcov [filename] [cpuX...]	Merge and write coverage data for specified CPUs to file using the lcov output format (file name and CPU is optional). The coverage data will be merged for all CPUs if no CPU is specified.
coverage print address [len] [cpuX...]	Print coverage data to console, starting at address. If no CPU is specified the data will be merged for all CPUs. Else data for specified CPUs will be merged and printed.
coverage clear	Reset coverage data

The coverage data for each 32-bit word of memory consists of a 5-bit field, with bit0 (lsb) indicating that the word has been executed, bit1 indicating that the word has been written, and bit2 that the word has been read. Bit3 and bit4 indicates the presence of a branch instruction; if bit3 is set then the branch was taken while bit4 is set if the branch was not taken.

As an example, a coverage data of 0x6 would indicate that the word has been read and written, while 0x1 would indicate that the word has been executed. When the coverage data is printed to the console or save to a file, it is presented for one block of 32 words (128 bytes) per line:

```
tsim> cov print strcmp
  31004198 :   1  1 11  0  1  1  1 11  0  1  1  1  1  1  1  1
  310041d8 :   9  1  0  0  1  1  1 11  0  1  1  1  1 19  1  1
  31004218 :   1 11  1  1  1  9  1  0  0  1  1 19  1  1  1  1
  31004258 :   1  9  1  0  0  0  0  1  1  1  0  0  1  9  1  0
  31004298 :   0  0  0  0  0  0  0  0  0  0  0  0  1  1  1  1
  310042d8 :   1  1  1  1  9  1  0  0  0  0  0  0  0  0  0  0
  31004318 :   0  0  0  0  1  1 19  1  1  1  0  0  0  0  0  0
  31004358 :   0  0  0  0  0  0  0  0  0  0  4  0  0  0  0  0
  31004398 :   0  0  4  0  0  0  0  0  0  0  0  0  0  0  0  0
  310043d8 :   0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0
  31004418 :   0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0
  31004458 :   4  0  4  0  0  0  0  0  0  0  0  0  0  0  0  0
  31004498 :   0  0  0  0  0  0  0  1  1  1  1  1  1  1  1  1
  310044d8 :   1 11  1  1  1 11  1  1  1  1 11  0  1  1  1 11
  31004518 :   1  1  1 11  0  1  1  1 19  1  1  1  1  1  1  1
  31004558 :  11  1  1  1 19  1  1 11  0  1  1  1  1  1  1  1
```

When the code coverage is saved to file, only blocks with at least one coverage field set are written to the file. Block that have all the coverage fields set to zero are not saved in order to decrease the file size.

Only internally emulated memory are currently subject for code coverage. Any memory emulated in the user modules must be handled by a user-defined coverage function.

The memory controller address ranges that are monitored are based on the memory configuration at the moment when coverage is enabled. When using TSIM's startup parameters to configure memory, coverage can be enabled before starting simulation. For instance, the range corresponding to SDRAM, for an FTMCTRL memory controller with the RAM area starting at 0x40000000, will begin at address 0x40000000 if TSIM was started with `-nosram` or `-ram 0`, or will begin at 0x60000000 otherwise. In case a bootloader or the application itself sets up the memory controller configuration, coverage should be enabled after this setup has been completed.

NOTE on MMU and coverage: The TSIM coverage system does not do any address translations. The monitored address ranges are based on the physical address ranges where TSIM emulates some kind of memory. There is currently no support for getting virtual address coverage for virtual addresses that untranslated would go outside these memory ranges.

When coverage is enabled, disassembly will include an extra column after the address, indicating the coverage data. This makes it easier to analyse which instructions has not been executed:

```
tsim> dis strcmp

   31004198  1 82120009   or        %o0, %o1, %g1           strcmp
   3100419c  1 80886003   andcc     %g1, 0x3, %g0           strcmp + 0x4
   310041a0 11 3280001e   bne,a     0x31004218              strcmp + 0x8
```

```
   310041a4  0 c24a0000   ldsb      [%o0], %g1              strcmp + 0xc
   310041a8  1 c2024000   ld        [%o1], %g1              strcmp + 0x10
   310041ac  1 c4020000   ld        [%o0], %g2              strcmp + 0x14
   310041b0  1 80a04002   cmp       %g1, %g2                strcmp + 0x18
   310041b4 11 32800019   bne,a     0x31004218              strcmp + 0x1c
   310041b8  0 c24a0000   ldsb      [%o0], %g1              strcmp + 0x20
   310041bc  1 093fbfbf   sethi     %hi(0xfefefc00), %g4    strcmp + 0x24
   310041c0  1 07202020   sethi     %hi(0x80808000), %g3    strcmp + 0x28
   310041c4  1 881122ff   or        %g4, 0x2ff, %g4         strcmp + 0x2c
   310041c8  1 8610e080   or        %g3, 0x80, %g3          strcmp + 0x30
   310041cc  1 84004004   add       %g1, %g4, %g2           strcmp + 0x34
   310041d0  1 82288001   andn      %g2, %g1, %g1           strcmp + 0x38
   310041d4  1 80884003   andcc     %g1, %g3, %g0           strcmp + 0x3c
```

The coverage data is not saved or restored during check-pointing operations.

In TLIB individual coverage fields can also be read and written using the TSIM function interface using the `tsim_cov_get` and `tsim_cov_set` functions (see Section 6.2). Enabling and disabling the coverage should be done using a call to a function in the `tsim_cmd` family.

Example scripts for annotating C code using saved coverage information from TSIM can be found in the coverage sub-directory.

Using the **coverage lcov** command the coverage information is stored in a format that can be easily processed using the lcov utility. This allows coverage data from multiple runs to be combined, compared, or filtered. It can also be used by the genhtml utility to create HTML pages with the coverage information in the form of annotated source code.

3.10. Check-pointing

TSIM can save and restore its simulation state, allowing to resume simulation from a saved check-point. Saving the state to file is done with the **save** *`file`* command. To restore the state, use the **restore** *`file`* command.

Restoring state can be done both in the same session in which it was saved, and in a different session. However, this relies upon the session in which the restore is done to have been set up to simulate the same simulated hardware as when the save was made. When applicable. TSIM shows discrepancies between startup options between the session in which the save was made and the one in which the restore is done. It is up to the user to ensure that they represent the same hardware. Restoring state from a different TSIM version is not supported

Some I/O cores do not have check-pointing support at the moment. See the respective tables for each chip, listing which cores that does not support check-pointing right now. Such cores keep their pre-restore state when restoring. Using **save -l** from within TSIM can also list check-pointing support status for cores in the simulated system.

User modules can use save and restore callback functions to save and restore their state. When using events, restoreable events should be used for the events to be saved and restored properly. See Section 5.7 for details.

```
$ tsim-leon3
...
tsim> load hello
  section: .text, addr: 0x40000000, size: 46768 bytes
  section: .data, addr: 0x4000b6b0, size: 2936 bytes
  read 392 symbols

tsim> bp main
  breakpoint 1 at 0x40001934:  main + 0x4
tsim> run
  Initializing and starting from 0x40000000

  CPU 0 stopped at breakpoint 1: main + 0x4
tsim> save at-main.tss
  State saved to at-main.tss
tsim> quit

$ tsim-leon3
...
tsim> restore at-main.tss
  Restored state from at-main.tss
tsim> cont
Hello world!

  Program exited normally on CPU 0.
```

3.11. Backtrace

The bt command will display the current call backtrace and associated stack pointer:

```
tsim> bt
      %pc          %sp
  #0   0x31004198   0x3000fcf8   strcmp + 0x0
  #1   0x31001980   0x3000fcf8   Func_2 + 0x1c
  #2   0x31001540   0x3000fd58   main + 0x194
  #3   0x310010b0   0x3000fe10   __bcc_crt0 + 0xb0
```

3.12. Connecting to GDB

TSIM can act as a remote target for GDB, allowing symbolic debugging of target applications. GDB versions 6.8 and 8.2 are actively supported.

To initiate GDB communication, start the simulator with the `-gdb` switch or use the TSIM **gdb** command:

```
tsim> gdb
  gdb interface: using port 1234
  Starting GDB server. Use Ctrl-C to stop waiting for connection.
```

Then, start GDB in a different window and connect to TSIM using the extended-remote protocol:

```
$ sparc-rtems-gdb example.exe
(gdb) target extended-remote localhost:1234
Remote debugging using localhost:1234
0x0 in ?? ()
(gdb)
```

To interrupt simulation, Ctrl-C can be typed in both GDB and TSIM windows. The program can be restarted using the GDB **run** command but a **monitor gdb reset** and **load** has first to be executed in/via GDB to set up TSIM for a new run and reload the program image into the simulator. The **monitor gdb reset** command can be omitted if the MMU is not in use when using extended-remote target type and using the GDB **run** to start new simulation.

```
(gdb) monitor gdb reset
(gdb) load
Loading section .text, size 0x14e50 lma 0x40000000
Loading section .data, size 0x640 lma 0x40014e50
Start address 0x40000000 , load size 87184
Transfer rate: 697472 bits/sec, 278 bytes/write.
(gdb) run
```

The **monitor gdb reset** can always be omitted when using the extended-remote target type, with remote exec-file and starting each new execution via the GDB **run** command.

```
(gdb) target extended-remote :1234
Remote debugging using :1234
0x00000000 in ?? ()
(gdb) set remote exec-file /full/path/to/example.exe
(gdb) run
The program being debugged has been started already.
Start it from the beginning? (y or n) y
...

(gdb) run
The program being debugged has been started already.
Start it from the beginning? (y or n) y
...
```

When using remote target type (as opposed to extended-remote, e.g. when running via GDB in Eclipse) or not using run to start simulation, the **monitor gdb reset** should never be omitted before loading an image. In addition, when debugging multicore images in this situation, **monitor gdb postload** needs to be issued after loading to prepare all CPUs for a new run.

```
(gdb) target remote :1234
(gdb) monitor gdb reset
(gdb) load
Loading section .text, size 0x14e50 lma 0x40000000
Loading section .data, size 0x640 lma 0x40014e50
Start address 0x40000000 , load size 87184
Transfer rate: 697472 bits/sec, 278 bytes/write.
(gdb) monitor gdb postload
(gdb) cont
...
```

If GDB is detached using the **detach** command, the simulator returns to the command prompt, and the program can be debugged using the standard TSIM commands. The simulator can also be re-attached to GDB by issuing the **gdb** command to the simulator (and the **target** command to GDB). While attached, normal TSIM commands can be executed using the GDB **monitor** command. Output from the TSIM commands is then displayed in the GDB console. UART output forwarded to stdout is forwarded to GDB when running the simulation from GDB if TSIM is started with the `-gdbuartfwd` option.

TSIM translates SPARC traps into (Unix) signals which are communicated to GDB. If the application encounters a fatal trap, simulation will be stopped exactly on the failing instruction. The target memory and register values can then be examined in GDB to determine the error cause. To disable this and let execution continue through the corresponding trap handler instead, use the `-nb [0|1]` startup option.

Profiling an application executed from GDB is possible if the symbol table is loaded in TSIM before execution is started. GDB does not download the symbol information to TSIM, so the symbol table should be loaded using the monitor command:

```
(gdb) monitor symbols example.exe
read 158 symbols
```

When an application that has been compiled using the gcc -mflat option is debugged through GDB, TSIM should be started with -mflat in order to generate the correct stack frames to GDB.

3.13. Thread support

TSIM has thread support for the RTEMS 4.8 and RTEMS 4.10 operating system. Additional OS support will be added to future versions. The GDB interface of TSIM is also thread aware and the related GDB commands are described later.

3.13.1. TSIM thread commands

thread info - lists all known threads. The currently running thread is marked with an asterisk. (The wide example output below has been split into two parts.)

```
tsim> thread info
  Name | Type     | Id         | Prio | Time (h:m:s)  | Entry point              ...
-------------------------------------------------------------------------------- ...
  Int. | internal | 0x09010001 |  255 |   5:30.682722 | bsp_idle_thread          ...
-------------------------------------------------------------------------------- ...
  UI1  | classic  | 0x0a010001 |  100 |      0.041217 | system_init              ...
-------------------------------------------------------------------------------- ...
  ntwk | classic  | 0x0a010002 |  100 |      0.251199 | soconnsleep              ...
-------------------------------------------------------------------------------- ...
  ETH0 | classic  | 0x0a010003 |  100 |      0.000161 | soconnsleep              ...
-------------------------------------------------------------------------------- ...
* TA1  | classic  | 0x0a010004 |    1 |      0.034739 | prep_timer               ...
-------------------------------------------------------------------------------- ...
  TA2  | classic  | 0x0a010005 |    1 |      0.025740 | prep_timer               ...
-------------------------------------------------------------------------------- ...
  TA3  | classic  | 0x0a010006 |    1 |      0.021357 | prep_timer               ...
-------------------------------------------------------------------------------- ...
  TTCP | classic  | 0x0a010007 |  100 |      0.002914 | rtems_ttcp_main          ...
-------------------------------------------------------------------------------- ...

... | PC                                       | State
```

```
... -----------------------------------------------------
... | 0x40044bec _Thread_Dispatch + 0xd8           | READY
... -----------------------------------------------------
... | 0x40044bec _Thread_Dispatch + 0xd8           | SUSP
... -----------------------------------------------------
... | 0x40044bec _Thread_Dispatch + 0xd8           | READY
... -----------------------------------------------------
... | 0x40044bec _Thread_Dispatch + 0xd8           | Wevnt
... -----------------------------------------------------
... | 0x40006a28 printf + 0x4                      | READY
... -----------------------------------------------------
... | 0x40044bec _Thread_Dispatch + 0xd8           | DELAY
... -----------------------------------------------------
... | 0x40044bec _Thread_Dispatch + 0xd8           | DELAY
... -----------------------------------------------------
... | 0x40044bec _Thread_Dispatch + 0xd8           | Wevnt
... -----------------------------------------------------
```

thread bt *id* prints a backtrace of a thread.

```
tsim> thread bt 0x0a010007

    %%pc
#0   0x40044bec   _Thread_Dispatch + 0xd8
#1   0x400418f8   rtems_event_receive + 0x74
#2   0x40031eb4   rtems_bsdnet_event_receive + 0x18
#3   0x40032050   soconnsleep + 0x50
#4   0x40033d48   accept + 0x60
#5   0x4000366c   rtems_ttcp_main + 0xda0
```

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

```
tsim> thread bt
    %pc          %sp
#0   0x40006a28   0x4008d7d0   printf + 0x0
#1   0x40001c04   0x4008d838   Test_task + 0x178
#2   0x4005c88c   0x4008d8d0   _Thread_Handler + 0xfc
#3   0x4005c78c   0x4008d930   _Thread_Evaluate_mode + 0x58
```

3.13.2. GDB thread commands

TSIM needs the symbolic information of the image that is being debugged to be able to check for thread information. Therefore the symbols needs to be read from the image using the **symbols** command before issuing the **gdb** command. When a program running in GDB stops TSIM 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/thread-
dispatch.c:109
#1   0x40013ee0 in rtems_event_receive (event_in=33554432, option_set=0, ticks=0,
event_out=0x43fecc14)
      at ../../../../leon3/lib/include/rtems/score/thread.inl:205
#2   0x4002782c in rtems_bsdnet_event_receive (event_in=33554432, option_set=2, ticks=0,
event_out=0x43fecc14)
      at ../../../../../../rtems-4.6.5/cpukit/libnetworking/rtems/rtems_glue.c:641
#3   0x40027548 in soconnsleep (so=0x43f0cd70) at ../../../../../../rtems-4.6.5/cpukit/libnetwork-
ing/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/threadhan-
dler.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
```

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             0x43ffafc8       1140830152
i4             0x0      0
i5             0x4007cd40       1074253120
fp             0x43fecd08       0x43fecd08
i7             0x40053380       1074082688
y              0x0      0
psr            0xf34000e0       -213909280
wim            0x0      0
tbr            0x0      0
pc             0x40004028       0x40004028 <daemon+148>
npc            0x4000402c       0x4000402c <daemon+152>
fsr            0x0      0
csr            0x0      0
```

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

3.14. Synchronising TSIM time to external time

To maximise simulation performance, TSIM executes as fast as possible doing no synchronisation of the simulation time with any external notion of time. This is especially apparent when the processor is in power-down mode and simulation time is increased by the events in the event queue alone.

To synchronise the simulation time with an external notion of time, events that handles synchronisation needs to be added to the event queue. The `walltimesync` example module in the `examples/modules` directory provides an example that makes sure that TSIM does not execute faster than real time. This example can be used as a template for synchronising to other notions of time. See Chapter 5 on how to use modules.

3.15. Debugging particular device types and devices

To enable printout of debug information one can issue the **dbgon** *`flag`* command on the TSIM3 command line to toggle the on/off state of a flag for all cores of a certain type. The debug flags that are available are described for each core in their chapters.

Many cores also have their own debug commands on the format *`coreX`***_dbg** that targets single cores instead of all of one kind and that have support to set all or none of the debug flags options and list the current setting for the debug flags. See the sections on the respective cores for details.

4. Emulation characteristics

4.1. Common behaviour

4.1.1. Timing

The simulator time is maintained and incremented in terms of clock cycles. The parallel execution between the IU and FPU is modelled, as well as stalls due to operand dependencies. Instruction timing has been modelled after the real devices. Integer instructions have a higher accuracy than floating-point instructions due to the somewhat unpredictable operand-dependent timing of the FPU. Typical usage patterns have higher accuracy than atypical ones, e.g. having vs. not having caches enabled on LEON systems. Tracing using the **inst**, **ahb** or **hist** command will display the corresponding simulator time in the left column. This time indicates when the instruction or bus access finished. Cache misses, waitstates or data dependencies will delay the following fetch according to the incurred delay.

4.1.2. UARTs

The UART model can be operating in two modes, accurate mode and fast mode. In the accurate mode the baud rate and frame length is taken into account but in fast mode the UARTs operate at infinite speed. In fast mode the transmitter FIFO/holding register is always empty and a transmitter empty interrupt is generated directly after each write to the transmitter data register. The receivers can never overflow or generate errors. Fast mode is enabled with the `-fast_uart` switch.

Note that in accurate mode, it is possible that the last character of a program is not displayed on the console. This can happen if the program forces a processor in error mode, thereby terminating the simulation, before the last character has been shifted out from the transmitter shift register. To avoid this, an application can poll the UART status register and not force the processor in error mode before the transmitter shift registers are empty. The real hardware does not exhibit this problem since the UARTs continue to operate even when the processor is halted.

When an application is running with UART forwarded to the console (as the first UART is by default, or some other UART using the `-u` option) the following key sequences will be available. The sequences can be used to send key sequences to the UART that would otherwise be intercepted by the host operating system or to adjust the input to what the target system expects. For a key sequence to take effect, both key presses must be pressed within 1.5 seconds of each other. Otherwise, they will be forwarded as is.

Table 4.1. Uart control sequences

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

4.1.2.1. APBUART model (LEON3/4 only)

The APBUART model used on LEON3 and LEON4 systems is by default set up for receiver and transmitter FIFO mode. In this mode the additional FIFO flags and level interrupts are also modelled like the APBUART IP. The FIFO depth can be configured with the `-uart_fs` switch. FIFO mode can be disabled altogether with `-uart_fs 1`. FIFO mode is supported with both accurate and fast mode. However in fast mode the transmitter operates in infinite speed always causing the FIFO to be empty.

Loopback mode is supported both in fast and accurate mode. In fast mode transmitted characters directly ends up in the receiver. Similar to the hardware the CTSN/RTSN signals are connected together in loop back mode making flow control possible regardless of operating mode.

Flow control bit is supported but has a different effect compared to hardware when loopback mode is disabled. TSIM UARTs interfaces to user controlled devices (see `-uartX`) which may/may not implement flow control in different ways. When flow control is enabled APBUART receiver never overflows, however the transmitter operates independently of the flow control setting as if CTSN is always 0 by pausing the simulator until the character is transferred to the UART device.

The following debug flags can be used with TSIM command **uartX_dbg** (or **dbgon**) to enable debug printouts for various events. Here X is the index of the APBUART core.

Table 4.2. APBUART debug flags

Flag/subcommand	Trace
APBUART_ACC	Trace register accesses
APBUART_IRQ	Trace interrupt generation
APBUART_RX	Trace data reception
APBUART_TX	Trace data transmission
all	Set all debug flags for the core
clean	Set none of the debug flags for the core
list	List the current setting of the debug flags for the core

4.1.2.2. UART model (LEON2 only)

The UART model of LEON2 automatically switch to fast mode when the scaler baud rate register is set to zero. This is different from the APBUART model where only the `-fast_uart` switch is used to determine the mode.

4.1.3. Floating point unit (FPU)

The models for the GRFPU-lite and GRFPU models supports parallel IU and FPU execution, deferred floating point traps and the floating point deferred trap queue. The model for the Meiko FPU on LEON2 models the FPU setup for AT697E and AT7913E with no parallel IU and FPU execution, no floating point queue and no deferred floating point traps.

The GRFPU model simulates all types calculation results and exceptions, including denormal numbers and NaN results. It does not however simulate the possibility of multiple outstanding floating point operations. The complex internal timing of the GRFPU is not modelled in detail.

The simulator implements (to some extent) data-dependent execution timing for the Meiko FPU and GRFPU-lite. The only discrepancy between TSIM and actual hardware in terms of results is that when NaN results are generated on Meiko FPU on LEON2 and GRFPU-lite, they can differ compared to real hardware in the significand bits (but not in the signalling/quiet bit).

4.1.4. Delayed write to special registers

The SPARC architecture defines that a write to the special registers (%psr, %wim, %tbr, %fsr, %y) may have up to 3 delay cycles, meaning that up to 3 of the instructions following a special register write might not ‘see’ the newly written value due to pipeline effects. While LEON have between 2 and 3 delay cycles, TSIM has 0. This does not affect simulation accuracy or timing as long as the SPARC ABI recommendations are followed that each special register write must always be followed by three NOP. If the three NOP are left out, the software might fail on real hardware while still executing ‘correctly’ on the simulator.

4.1.5. Peripherals registers

An overview of peripherals can be displayed with the **leon** command. Individual registers can be listed with the **info reg** *`coreX`* or **info reg** *`addr`* command.

4.1.6. Idle-loop optimisation

To minimise power consumption, LEON applications will typically place the processor in power-down mode when the idle task is scheduled in the operation system. In power-down mode, TSIM increments the event queue with-

out executing any instructions, thereby significantly improving simulation performance. However, some (poorly written) code might use a busy loop (BA 0) instead of triggering power-down mode. The `-bopt` switch will enable a detection mechanism which will identify such behaviour and optimise the simulation as if the power-down mode was entered.

4.1.7. Custom instruction emulation

TSIM/LEON allows the emulation of custom (non-SPARC) instructions. A handler for non-standard instruction can be installed using the `tsim_ext_ins()` callback function (see Section 6.2). The function handler is called each time an instruction is encountered that would cause an unimplemented instruction trap. The handler is passed the CPU ID of the executing CPU, *`cpuid`*, and a pointer, *`r`*, to a structure containing the opcode and all processor registers, allowing it to decode and emulate a custom instruction, and update the processor state.

The definition for the custom instruction handler is:

```
int (*func)(void *priv,
            int cpuid,
            uint32 inst,
            uint32 **window,
            uint32 *icnt),
```

The `priv` pointer is a private pointer registered by user and passed to func when it is called. The `cpuid` argument is the CPU index of CPU for which instruction is to be executed. The `inst` argument contsins the instruction to be executed. The `window` argument is an array of pointers to instructions. It can be used both for to read and write registers. It can be indexed using the `G*`, `I*`, `O*`, and `L*` constants from `enum regname`. It can be used both to get current register values as well as change variables in the current register window. In case other registers needs to be accessed, `tsim_get_reg` and `tsim_set_reg` can be used. Using the `icnt` argument, the number of cycles in the pipeline for the instruction can be set in `*icnt`. It defaults to 1, a fully pipelined instruction.

The return value of the custom handler indicates which trap the emulated instruction has generated, or 0 if no trap was caused. If the handler could not decode the instruction, 2 should be returned to cause an unimplemented instruction trap.

The number of clocks consumed by the instruction should be returned in `*icnt`. This value is by default 1, which corresponds to a fully pipelined instruction without data interlock. The handler should not increment the `%pc` or `%npc` registers, as this is done by TSIM.

4.1.8. Chip-specific errata

Incorrect behaviour described in errata documents for specific devices are not emulated by TSIM in general.

4.2. LEON2 specific emulation

4.2.1. Processor

The LEON2 version of TSIM emulates the behaviour of the LEON2 VHDL model. The (optional) MMU can be emulated by starting TSIM with the `-mmu` switch.

4.2.2. Cache memories

TSIM/LEON2 can emulate any permissible cache configuration using the `-icsize`, `-ilsize`, `-dcsize` and `-dlsize` options. Allowed sizes are 1 - 64 KiB with 16 - 32 bytes/line. The characteristics of the LEON multi-set caches can be emulated using the `-isets`, `-dsets`, `-irepl`, `-drelp`, `-ilock` and `-dlock` options. Diagnostic cache reads/writes are implemented. The simulator commands **icache** and **dcache** can be used to display cache contents. Starting TSIM with `-at697e` will configure that caches according to the Atmel AT697E device.

4.2.3. Interrupt controller

External interrupts are not implemented, so the I/O port interrupt register has no function. Internal interrupts are generated as defined in the LEON specification. All 15 interrupts can also be generated from the user defined I/O module using the `set_irq()` callback.

4.2.4. Power-down mode

The power-down register (0x80000018) is implemented as in the specification. In power-down mode, the simulator skips time until the next event in the event queue, thereby significantly increasing the simulation speed. A Ctrl-C in the simulator window will break execution, but will not make the CPU exit power-down mode.

4.2.5. Memory emulation

The memory configuration registers 1/2 are used to decode the simulated memory. The memory configuration registers has to be programmed by software to reflect the available memory, and the number and size of the memory banks. The waitstates fields must also be programmed with the correct configuration after reset. Both SRAM and functionally modelled SDRAM (with SRAM timing) can be emulated.

Using the `-banks` option, it is possible to set over how many RAM banks the external SRAM is divided in. For mkprom encapsulated programs, it is essential that the *same* RAM size and bank number setting is used for both mkprom and TSIM.

The memory EDAC of LEON2-FT is not implemented.

4.2.6. SPARC V8 MUL/DIV and V8E MAC instructions

TSIM/LEON2 by default supports the SPARC V8 multiply and divide instructions. To emulate LEON2 systems which do not implement these, use the `-nov8` option to disable multiply and divide instructions. TSIM/LEON2 optionally implements the SPARC V8E MAC instructions. To emulate LEON2 systems which implement these, use the `-mac` option to enable the MAC instructions, and make sure to not use `-nov8`.

4.2.7. FPU emulation

By default, TSIM/LEON emulates the Meiko FPU. The `-grfpu` command line option enables the GRFPU model. See Section 4.1.3 for details on the FPU models.

4.2.8. DSU and hardware breakpoints

The LEON debug support unit (DSU) and the hardware watchpoints (%asr24 - %asr31) are not emulated.

4.3. LEON3 specific emulation

4.3.1. General

The LEON3 version of TSIM by default emulates the behaviour of a generic LEON3. The system includes the following modules: LEON3 processor, APB bridge, IRQMP interrupt controller, FTMCTRL memory controller (but EDAC is not modelled), GPTIMER timer units with 32-bit timers and APBUART UARTs. Chip options instead sets up TSIM to emulate a particular chip. Other hardware configuration options can change parameters from either the default values or from the values set up by a chip option.

4.3.2. Processor

The instruction timing of the emulated LEON3 processor is modelled after the LEON3 in GRLIB IP library and after the specific chips that have their own chip options. The processor can be configured with 2 - 32 register windows using the `-nwin` switch. The MMU is emulated by default, but can be disabled using `-mmu 0`. Local instruction RAM and local data RAM can be added with the `-ilram` and `-dlram` switches.

4.3.3. Cache memories

TSIM can emulate any permissible cache configuration using the `-icsize`, `-ilsize`, `-dcsize` and `-dlsize` options. Allowed sizes are 1 - 256 KiB with 16 - 32 bytes/line. The characteristics of the LEON multi-way caches can be emulated using the `-isets`, `-dsets`, `-irepl`, `-drelp`, `-ilock` and `-dlock` options. Diagnostic cache reads/writes are implemented. The simulator commands **icache** and **dcache** can be used to display cache contents, flush caches and query cache status for given addresses.

The evaluation version of TSIM/LEON3 implements 2*4 KiB caches, with 16 bytes per line.

4.3.4. Power-down mode

The LEON3 power-down function is implemented as in the specification. When in power down mode, the simulator skips time until the next event in the event queue, thereby significantly increasing the simulation speed. A Ctrl-C in the simulator window will break execution,but will not make any CPU exit power-down mode.

4.3.5. Interrupt controller

The IRQMP interrupt controller model supports extended interrupts, multicore registers, interrupt maps, and interrupt timestamping. When having extended interrupts enabled, interrupts 1-31 can be generated. For GR716A/B interrupts 1-63 can be generated. Extended interrupts can be enabled by the `-ext` option, or with a chip option for a chip that has extended interrupts (e.g. `-gr712rc`, `-ut700` and `-ut699e`). Interrupts can be generated by user models using the `set_irq()` callback of TSIM's ioif struct. See Section 5.2.3 for details.

The debugflag IRQMP_ACC can be used with the TSIM command **irqmpX_dbg** to enable debug printouts each time a register access is made to the interrupt controller. Here X is the index of the interrupt controller.

4.3.6. Memory emulation

The FTMCTRL (without EDAC emulation) or the LEON2 memory controller is emulated in the LEON3 version of TSIM. The memory configuration registers 1 and 2 are used to decode the simulated memory. The memory configuration registers has to be programmed by software to reflect the available memory, and the number and size of the memory banks. Both SRAM and SDRAM can be emulated, however, the SDRAM model does not support sending commands using the SDRAM command field in mcfg2. The PROM area is basically modelled as MRAM.

The SRAM is configured using options like `-ram`, `-ramwidth`, `-banks` and `-nosram`. The SDRAM is configured using options like `-sdram` and `-sdbanks`. The PROM is configured using options like `-rom` and `-romwidth`.

When booting from PROM, it is important that the configuration done by the bootloader matches the system setup, just as for booting on actual hardware. TSIM however does not model any failure due to too few waitstates.

Options regarding memory characteristics are not available in the evaluation version of TSIM/LEON3.

4.3.7. CASA instruction

The `-cas` option or any chip option for a chip with CASA support enables emulation of the CASA instruction (LEON3/4 only). Using `-cas 0` can disable CASA support when otherwise already enabled.

4.3.8. SPARC V8 MUL/DIV and V8E MAC instructions

TSIM/LEON3 by default supports the SPARC V8 multiply and divide instructions. To emulate LEON3 systems which do not implement these, use the `-nov8` option to disable multiply and divide instructions. TSIM/LEON3 optionally implements the SPARC V8E MAC instructions. To emulate LEON3 systems which implement these, use the `-mac` option to enable the MAC instructions, and make sure to not use `-nov8`.

4.3.9. FPU emulation

By default, TSIM/LEON3 emulates the GRFPU-lite FPU unless a chip option for a chip with a GRFPU is used. The `-grfpu` command line option enables the GRFPU model. See Section 4.1.3 for details on the FPU models.

4.3.10. DSU and hardware breakpoints

The LEON debug support unit (DSU) and the hardware watchpoints (%asr24 - %asr31) are not emulated.

4.3.11. AHB status registers

When using `-ahbstatus` or a chip option for a chip that has AHB status registers, AHB status registers are enabled. As TSIM/LEON3 does not emulate FT, the CE bit will never be set by TSIM's internal memory models, but the `correctable_error()` function (Section 5.3.1) can be used in a user model to set it. Furthermore, the HMASTER field is set to the CPU index (starting at zero) when the CPU caused the error and one over the last CPU index (i.e. 1 in a one CPU system) when any other master caused the error.

4.3.12. GPTIMER emulation

The debugflag GPTIMER_ACC can be used with the TSIM command **gptimerX_dbg** to enable debug printouts each time a register access is made to the core. Here X is the index of the GPTIMER core.

4.3.13. GRTIMER emulation

When using `-gr712rc`, the GRTIMER core is modelled (in addition to the regular GPTIMER core).

The debugflag GRTIMER_ACC can be used with the TSIM command **grtimerX_dbg** to enable debug printouts each time a register access is made to the core. Here X is the index of the GRTIMER core.

4.4. LEON4 specific emulation

Currently, the only supported LEON4 configuration is GR740.

4.4.1. Processor

The four emulated LEON4 processors are modelled after the LEON4 VHDL model in GRLIB IP library and is configured to emulate GR740. Simulation of LEON4 in other configurations is not yet available.

4.4.2. L1 Cache memories

TSIM/LEON4 can emulate any permissible cache configuration using the `-icsize`, `-ilsize`, `-dcsize` and `-dlsize` options. Allowed sizes are 1 - 256 KiB with 16 - 32 bytes/line. The characteristics of the LEON multi-set caches can be emulated using the `-isets`, `-dsets`, `-irepl`, `-drelp`, `-ilock` and `-dlock` options. Diagnostic cache reads/writes are implemented. The simulator commands **icache** and **dcache** can be used to display cache contents, flush caches and query cache status for given addresses.

4.4.3. L2 Cache memory

GR740 has a 2 MiB L2 cache with 4 cache ways and 32 byte cache lines. The L2 cache has support for dynamically configurable replacement policies as well as locked ways. Individual memory regions can be write protected or marked uncacheable by the MTRR registers.

When starting TSIM the L2 cache is set up in reset state and thus disabled. The **run** command will as part of the boot-loading flush, invalidate and enable the L2 cache. The **boot** command will flush and invalidate the cache as part of restarting simulation but will otherwise leave it in its reset state. The L2 cache can otherwise be enabled or disabled via the register interface or the **l2cache enable** and **l2cache disable** commands.

The **l2cache** command shows the current overview of the state of the L2 cache. For more commands for flushing, invalidating the cache as well as investigating the L2 cache state, see the various **l2cache** subcommands in Section 3.3.1 or use the **help l2cache** command to list the available L2 cache commands.

4.4.3.1. Limitations of the L2 cache model

In this release, the L2 cache model has a number of features that are not yet supported. AMBA split responses and writethrough are not supported. In other words, only waitstate responses will be given and only copy-back will be performed. The different tuning settings available in the access control register are not modelled. Moreover, dirty cachelines are always modelled as fully dirty and not half dirty. These limitations have no functional effects on simulated software as long as the cache is flushed before disabling if the cache needs to be disabled.

No FT features are modelled. There is no EDAC emulation, error injection, scrubbing. This includes related registers and register fields, including the entire error status/control register. There is also no support for HPROT signals. These limitations are reflected in the registers shown by the **info reg** command for the L2 cache. Timing of cache clushing is not modelled.

Interacting with the L2 cache with commands such as the **mem** and **wmem** will affect the state the cache just as regular bus accesses would, including timing of future accesses when continuing (as opposed to restarting) execution. In the same way, **l2cache** commands that changes L2 cache state, will affect timing of future accesses when continuing execution.

4.4.4. Power-down mode

The LEON4 power-down function is implemented as in the specification. When in power down mode, the simulator skips time until the next event in the event queue, thereby significantly increasing the simulation speed. A Ctrl-C in the simulator window will break execution, but will not make any CPU exit power-down mode.

4.4.5. Interrupt controller

The IRQ(A)MP interrupt controller model supports multiple internal interrupt controllers, extended interrupts, multicore registers, interrupt maps, interrupt timestamping and extended interrupts. All 31 interrupts can be generated by user models using the `set_irq()` callback of TSIM's ioif struct. See Section 5.2.3 for details. The watchdog control and error mode status registers are not yet implemented.

The debugflag IRQMP_ACC can be used with the TSIM command **irqmpX_dbg** to enable debug printouts each time a register access is made to the interrupt controller. Here X is the index of the interrupt controller.

4.4.6. Memory emulation

The SDRAM controller behind the L2 cache is modelled for GR740. It can be configured with the `-sdram`, `-sdfreq`, `-sdbanks` and `-sdramwidth` options as well as through the SDRAM controller registers, sdcfg1/sdcfg2.

In this release the simulated timing is based on a CPU frequency of 250 or 50 MHz and a memory frequency of either 50 or 100 MHz, default memory frequency is 100 MHz. For CPU frequencies other than 250 or 50 MHz, timing is a rough estimate. Issuing commands to the SDRAM through the sdcfg1 register is not supported. No EDAC functionality is currently emulated.

The FTMCTRL for the PROM and I/O areas is also emulated. No EDAC functionality is currently emulated.

4.4.7. IOMMU

Two modes of protection are supported, access protection vector (APV) and MMU mode. Diagnostic accesses and error injection are not supported. But most diagnostic functionality is supported by commands, such as displaying the contents of the cache, writing cache lines/tags and looking up address translations.

4.4.8. CASA instruction

The `-cas` option or any chip option for a chip with CASA support enables emulation of the CASA instruction (LEON3/4 only). Using `-cas 0` can disable CASA support when otherwise already enabled.

4.4.9. SPARC V8 MUL/DIV and V8E MAC instructions

TSIM/LEON4 by default supports the SPARC V8 multiply and divide instructions. To emulate LEON4 systems which do not implement these, use the `-nov8` option to disable multiply and divide instructions. TSIM/LEON4 optionally implements the SPARC V8E MAC instructions. To emulate LEON4 systems which implement these, use the `-mac` option to enable the MAC instructions, and make sure to not use `-nov8`.

4.4.10. FPU emulation

By default, TSIM/LEON4 emulates the GRFPU FPU. See Section 4.1.3 for details on the FPU models.

4.4.11. DSU and hardware breakpoints

The LEON debug support unit (DSU) and the hardware watchpoints (%asr24 - %asr31) are not emulated.

4.4.12. AHB status registers

The AHB status register on the processor bus is modelled. The CE bit will never be set by TSIM's internal memory models, but the `correctable_error()` function (Section 5.3.1) can be used in a user model to set it. Furthermore, the HMASTER field is set to the CPU index (starting at zero) when the CPU caused the error, and 4 when any other master caused the error.

When using `-ahbstatus` or a chip option for a chip that has AHB status registers, AHB status registers are enabled. As TSIM currently does not emulate FT, the CE bit will never be set by TSIM's internal memory models, but the `correctable_error()` function (Section 5.3.1) can be used in a user model to set it. Furthermore, the HMASTER field is set to the CPU index (starting at zero) when the CPU caused the error and one over the last CPU index (i.e. 1 in a one CPU system) when any other master caused the error.

4.4.13. GPTIMER emulation

The debugflag GPTIMER_ACC can be used with the TSIM command **gptimerX_dbg** to enable debug printouts each time a register access is made to the core. Here X is the index of the GPTIMER core.

5. Loadable modules

NOTE: At this stage, the available interfaces are not entirely in their final form and are subject to change.

User-defined models using C APIs are all loaded into TSIM using the general module interface, from which the specific user modules can be registered with TSIM using different registration functions.

5.1. General module interface

This section describes the general module interface. General modules can in themselves be used in to run code from various callbacks called from TSIM, but are also used as spring boards for all kinds of user models that are registered from one or several general modules.

5.1.1. Loading modules

To load a general module in standalone TSIM, use the `-mod` option to specify a general module in form of a dynamic shared object for TSIM to load. In other words a .so file in Linux and DLL in Windows. Note that in Linux you generally cannot just specify the name of the file even if it is in the current directory.

```
$ tsim-leon3 -mod ./module.so
```

The environment variable TSIM_MODULE_PATH can be set to list of seach paths in order to find modules without specifying a full path. This should be a ':' separated list in Linux, and a ';' separated list in Windows. When using TLIB, the `tsim_register_module` function is an alternative to the `-mod` option. See Chapter 6.

See Section 5.1.3 on how to register other kind of modules from a general module and in which sections they are documented.

5.1.2. General module API

A module should be a dynamic library that should expose a public symbol `loadable_module` of type `struct loadable_module *`. Note that the module must be compiled to be position-independent, i.e. with the `-fPIC` switch (gcc).

The `struct loadable_module` is defined in `tsim.h` as:

```
struct loadable_module {
    void *priv; /* Free for the module to use */
    int  (*preinit)(struct loadable_module *module);
    int  (*init)(struct loadable_module *module);
    void (*exit)(struct loadable_module *module);
    void (*restart)(struct loadable_module *module);
    void (*reset)(struct loadable_module *module);
    void (*preset)(struct loadable_module *module);
    void (*start)(struct loadable_module *module);
    void (*stop)(struct loadable_module *module);
    int (*save)(struct loadable_module *module, struct user_checkpoint *ucp);
    int (*restore)(struct loadable_module *module, struct user_checkpoint *ucp);
};
```

The elements in the structure has the following meaning:

`void *priv;`
: Free for the module to use.

`int (*preinit)(struct loadable_module *module);`
: Called once before simulator startup. Startup options should be registered here. See Section 5.5.

`int (*init)(struct loadable_module *module);`
: Called once on simulator startup. Modules should be registered here.

`void (*exit)(struct loadable_module *module);`
: Called once on simulator exit.

`void (*restart)(struct loadable_module *module);`
: Called every time the simulator is restarted (simtime set to zero) including at startup. After a restart TSIM will also issue a call to `reset`.

`void (*reset)(struct loadable_module *module);`
Called every time the system is reset, including at startup and after a restart.

`void (*preset)(struct loadable_module *module);`
Called when the **run** command performs bootloader-like operations.

`void (*start)(struct loadable_module *module);`
Called each time simulation starts, both when starting for the first time using **boot** or **run** command and when continuing using **go**, **cont**, **step** and the like.

`void (*stop)(struct loadable_module *module);`
Called every time simulation stops, e.g. due to breakpoints, user pressing Ctrl-C, etc.

`int (*save)(struct loadable_module *mod, struct user_checkpoint *ucp);`
Called from the **save** command to let the user module save its state. See Section 5.7 for details.

`int (*restore)(struct loadable_module *mod, struct user_checkpoint *ucp);`
Called from the **restore** command to let the user module save its state. See Section 5.7 for details.

5.1.3. Connecting specific modules

Specific modules should be registered from the init function of a general module. The following functions are used for that:

`tsim_register_ahb_module(struct ahb_subsystem *ahbsystem)`
Register an AHB system module. See Section 5.3.

`tsim_register_io_module(struct io_subsystem *iosystem)`
Register a I/O system module. See Section 5.4.

`tsim_register_spim_module(struct spim_subsystem *subsystem, int index)`
Register `subsystem` to SPIM controller with index `index`. See Chapter 24.

`tsim_register_gpio_module(struct gpio_input *inp, int index)`
Register `inp` to GPIO controller with index `index`. See Chapter 18.

`tsim_register_spi_module(struct spi_input *inp, int index)`
Register `inp` to SPI controller with index `index`. See Chapter 23.

`tsim_register_dac_module(struct dac_input *inp, int index)`
Register `inp` to DAC controller with index `index`. See Section 8.3.

`tsim_register_can_node(struct can_node *node, int canbus_index)`
Register `node` to CAN bus with index `canbus_index`. See Section 15.4 for more information.

`tsim_register_grpci_module(struct grpci_input *inp, int index)`
Register `inp` to GRPCI controller with index `index`. See Section 19.3 for more information.

`tsim_register_spw_node(struct spw_node *node, int spw_index)`
Register `node` to SpaceWire port with index `spw_index`. See Section 22.7 for more information. (Only available for SpaceWire Router ports)

5.1.4. General module examples

The `walltimesync.c` example is a pure general module example that does not register another type of module. See Section 5.8 for a more complete list of module examples, that all are general modules as entry points.

5.2. TSIM exported emulation interfaces

TSIM exports three structures: simif, ioif and procif. The simif structure defines functions and data structures belonging to the simulator core, while ioif defines functions for bus accesses. The procif structure defines a few functions giving access to the processor emulation, cache behaviour and interrupt controller.

Pointers to simif, ioif and procif can be obtained by the functions `tsim_get_simif()`, `tsim_get_ioif()` and `tsim_get_procif()` defined in tsim.h.

5.2.1. API constraints

Note that in general the exported functions in these structures may only be called from user module functions that are called by TSIM, e.g. the init function, from event callbacks, from read and write functions. Unless explicitly documented to be able to be called asynchronously, do not call them from a separate thread or a signal handler.

For TLIB applications, calls to most of these functions can also be done when simulation is not ongoing and the control is in the hands of the TLIB application. See also Section 6.5.

5.2.2. simif structure

The simif structure is defined in `tsim.h` as:

```
struct sim_options {
   uint32 phys_ram;
   uint32 phys_rom;
   float64 freq;
   float64 wdfreq;
   uint32 phys_sdram;
};
struct sim_interface {
   struct sim_options *options;      /* tsim command-line options */
   uint64 (*simtime)(void);          /* current simulator time */
   void (*sys_reset)(void);          /* reset processor */
   void (*sim_stop)(void);           /* stop simulation */
   void (*sim_interrupt)(void);      /* interrupt simulation just as Ctrl-C */
   void (*event)(void (*cfunc)(void *), void *arg, uint64 offset);
   int  (*stop_event)(void (*cfunc)(void *));
   int  (*stop_event_arg)(void (*cfunc)(void *), void *arg);
   int  (*stop_one_event)(void (*cfunc)(void *));
   int  (*stop_one_event_arg)(void (*cfunc)(void *), void *arg);

   /* Restorable events */
   unsigned short (*reg_revent)(void (*cfunc) (void *arg));
   unsigned short (*reg_revent_prearg)(void (*cfunc) (void *arg),
                                       void *arg);
   int (*revent)(unsigned short index, void *arg, uint64 offset);
   int (*revent_prearg)(unsigned short index, uint64 offset);
   int (*stop_revent)(unsigned short index);
   int (*lprintf)(const char *format, ...); /* logged formatted output */
   int (*vlprintf)(const char *format, va_list ap); /* logged formatted output */

   /* Collected arguments from all sources, excluding executable name */
    int argc;
    char **argv;
};
```

The elements in the structure has the following meaning:

`struct sim_options *options;`
: Contains some tsim startup options. options.freq defines the clock frequency of the emulated processor and can be used to correlate the simulator time to the real time.

`uint64 (*simtime)(void);`
: Returns the current simulator time. Time is counted in clock cycles since start of simulation. To calculate the elapsed real time, divide simtime with options.freq.

`void (*sys_reset)(void);`
: This function for performing a system reset without restarting simulation is currently not supported. This can be worked around using sim_stop together with a Tcl loop that restarts simulation, in standalone TSIM, or a corresponding loop when using TLIB.

`void (*sim_stop)(void);`
: Stops current simulation, just as if getting to the end e.g. a step or cont with duration. This can be called asynchronously from any thread. Does not set the tsim::interrupt Tcl variable.

`void (*sim_interrupt)(void);`
: Interrupts current simulation, just as if Ctrl-C was pressed. This can be called asynchronously from any thread. Sets the tsim::interrupt Tcl variable to 1.

`void (*event)(void (*cfunc)(void *), void *arg, uint64 offset);`

: TSIM maintains an event queue to emulate time-dependent functions. The `event()` function inserts an event in the event queue. An event consists of a function to be called when the event expires, an argument with which the function is called, and an offset (relative the current time) defining when the event should expire.

: NOTE: The `event()` function may only be called from event callbacks or at start of simulation (e.g. not from from from a separate thread or a signal handler). The event queue can hold a maximum of 2048 events.

: NOTE: For save and restore support, restorable events should be used instead.

`int (*stop_event)(void (*cfunc)(void *));`
: Removes all events from the event queue which has the calling function equal to `cfunc()`. Returns the number of events stopped.

`int (*stop_event_arg)(void (*cfunc)(void *), void *arg);`
: Removes all events from the event queue which has the callback function equal to `cfunc()` and argument equal to *`arg`*. Can be useful when simulating multiple instances of an entity.

`int (*stop_one_event)(void (*cfunc)(void *));`
: Removes at most one event, closest in time, from the event queue which has the calling function equal to `cfunc()`. Returns the number of events stopped. Can be used when either only one of many events is to be removed or for better performance when knowing that there is only one possible match.

`int (*stop_one_event_arg)(void (*cfunc)(void *), void *arg);`
: Removes at most one event, closest in time, from the event queue which has the callback function equal to `cfunc()` and argument equal to *`arg`*. Can be useful when simulating multiple instances of an entity. Can be used when either one of many events is to be removed or for better performance when knowing that there is only one possible match.

`unsigned short (*reg_revent)(void (*cfunc)(void *arg));`
: Registers a restorable event that will use `cfunc` as callback. The returned index should be used when calling `revent()`. The event argument is supplied when calling `revent()`. The call to `reg_revent()` should be done once at module initialisation in the module's `init()` callback.

`unsigned short (*reg_revent_prearg)(void (*cfunc) (void *arg), void *arg);`
: Registers a restorable event that will use `cfunc` as callback and *`arg`* as argument. This can be used to register an argument that is a pointer to a data structure. The returned index should be used when calling `revent_prearg()`. The call to `reg_revent_prearg()` should be done once at module initialisation.

`int (*revent)(unsigned short index, void *arg, uint64 offset);`
: This inserts an event registered by `reg_revent()` into the event queue with the registered `cfunc` for the given *`index`*. Multiple events with the same *`index`* can be in the event queue at the same time. The *`arg`* and *`offset`* arguments are the same as for the `event()` function. On success 0 is returned, non-zero otherwise.

  NOTE: See the description of `event()` for limitations on number of events and from which contexts events can be added.

`int (*revent_prearg)(unsigned short index, uint64 offset);`
: This inserts an event registered by `reg_revent_prearg()` into the event queue with the registered `cfunc` and *`arg`* for the given *`index`*. Multiple events with the same *`index`* can be in the event queue at the same time. The *`offset`* argument is the same as for the `event()` function. On success 0 is returned, non-zero otherwise.

  NOTE: See the description of `event()` for limitations on number of events and from which contexts events can be added.

`int (*stop_revent)(unsigned short index);`
: This removes at most one event, closest in time, from the event queue that has been entered by `revent()` or `revent_prearg()` using the given *`index`*. Returns the number of events stopped.

`int (*lprintf)(const char *format, ...)`
: Function for formatted output that goes both to stdout and, when logging is enabled, to the log. The function interface works like for printf.

`int (*vlprintf)(const char *format, va_list ap)`
: Function for formatted output that goes both to stdout and, when logging is enabled, to the log. The function interface works like for vprintf.

`int argc, char** argv`
: `argv` is the collected arguments from all sources, excluding executable name. `argc` is the number of arguments.

5.2.3. ioif structure

The ioif structure is defined in tsim.h as:

```
struct io_interface {
   void (*set_irq)(uint32 irq);
   int (*dma_read)(uint32 master_id, uint32 addr, uint32 *data, int num);
   int (*dma_write)(uint32 master_id, uint32 addr, uint32 *data, int num);
   int (*dma_write_sub)(uint32 master_id, uint32 addr, uint32 *data, int sz);
};
```

The elements of the structure have the following meaning:

`void (*set_irq)(uint32 irq);`

Generate interrupt *`irq`* on the bus. Valid values of *`irq`* is 1 - 15 for systems without extended interrupts and 1-31 for systems with extended interrupts, and 1-63 for GR716A/B. Note that the interrupt controller controls how and when processor interrupts are actually generated.

`int (*dma_read)(uint32 master_id, uint32 addr, uint32 *data, int num);`
`int (*dma_write)(uint32 master_id, uint32 addr, uint32 *data, int num);`

Performs DMA transactions to/from the emulated processor memory. Only 32-bit word transfers are allowed, and the address must be word aligned. On bus error, 1 is returned, otherwise 0. DMA takes place on the AMBA AHB bus.

`int (*dma_write_sub)(uint32 master_id, uint32 addr, uint32 *data, int sz);`

Performs DMA transactions to/from the emulated processor memory on the AMBA AHB bus. On bus error, 1 is returned, otherwise 0. Write size is indicated by *`sz`* as follows: 0 = byte, 1 = half-word, 2 = word, 3 = double-word.

5.2.4. procif structure

The procif structure is defined in tsim.h as:

```
struct proc_interface {
   void (*set_irl)(int cpuid, int level); /* Generate external interrupt signal directly to CPU */
   void (*cache_snoop)(uint32 addr);
   void (*cctrl)(int cpuid, uint32 *data, uint32 read);
   void (*power_down)(int cpuid);
   void (*set_irq_level)(uin32 irq, int set);
   void (*set_irq)(uint32 irq); /* generate external interrupt */
};
```

The elements in the structure have the following meaning:

`void (*set_irl)(int cpuid, int level);`

Set the the current interrupt level (iui.irl in VHDL model) signal directly to the specified CPU. Allowed values are 0 - 15, with 0 meaning no pending interrupt. Once the interrupt level is set, it will remain until it is changed by a new call to `set_irl()`. The modules interrupt callback routine should typically reset the interrupt level to avoid new interrupts.

NOTE: For normal interrupt generation, use `set_irq` instead. This bypasses the built in interrupt controller model.

`void (*cache_snoop)(uint32 addr);`

The `cache_snoop()` function can be used to invalidate data cache lines (regardless of whether data cache snooping is enabled or not). The tags to the given address will be checked, and if a match is detected the corresponding cache lines will be flushed (i.e. the tag will be cleared). If an MMU is present and is enabled the argument should be a virtual address. See also the snoop function in `struct ahb_interface`.

`void (*cctrl)(int cpuid, uint32 *data, uint32 read);`

Read and write the specified CPUs cache control register (CCR). If read = 1, the CCR value is returned in *`*data`*, else the value of *`*data`* is written to the CCR.

`void (*power_down)(int cpuid);`

The specified processor enters power down-mode when called.

`void (*set_irq_level)(uin32 irq, int set);`

This is used to generate level interrupts. When calling `set_irq_level` with *`set`* set to 1 this enables a constant generation of interrupt *`irq`* that remains active until a subsequent call to `set_irq_level` with the same *`irq`* value and with *`set`* set to 0.

`void (*set_irq)(uint32 irq);`

Generate interrupt *`irq`* on the bus. Valid values of *`irq`* is 1 - 15 for systems without extended interrupts and 1-31 for systems with extended interrupts, and 1-63 for GR716A/B. Note that the interrupt controller controls how and when processor interrupts are actually generated.

5.3. LEON AHB emulation interface

TSIM allows user defined AHB modules simulating devices on the AMBA buses (both AHB and APB). The emulated processor core communicates with an AHB module using an interface similar to the AHB master interface

in the real LEON VHDL model. As the real processor, the simulator primarily interacts with the emulated device through read and write requests, while the emulated device can optionally generate interrupts and DMA requests.

To load and register an AHB system, the general module interface should be used to load it in, and from the general module init function call `tsim_register_ahb_module()` to register the AHB system The ahb_subsystem struct is described in Section 5.3.1.

The AHB module interface is made up of two parts; one that is exported by the AHB module and allows TSIM to access the emulated AHB devices; and one that is filled in by TSIM and defines TSIM functions and data structures that can be used by the AHB module. The data structures documented in Section 5.2 can also be used by the AHB module. The information there about from where those functions are allowed to be called applies to the TSIM provided function in the AHB module interface as well.

To register memory areas, use the `add_ahb_slave` and/or `add_apb_slave` functions. Whenever an access to that memory area is performed either the registered read or write callback will be called. To be able to use the **load** or **bload** command, a function `get_mem_ptr` needs to be registered when adding an AHB slave. This function should return a pointer to the module's internal underlying memory. The AHB module can use the `add_ahb_pp` or `add_ahb_pp_full`, and `add_apb_pp` functions to register plug&play entries that will show up in plug&play areas and thus can be seen during plug&play-scanning. Memory areas and plug&play entries should be registered from the ahb_subsystem `init` function.

5.3.1. Structure to be provided by AHB module

tsim.h defines the `ahb_subsystem` structure to be provided by the emulated AHB module:

```
struct ahb_subsystem {
    /* --- Initialied by module --- */
    void   (*init)(void);
    void   (*exit)(void);
    void   (*reset)(void);
    void   (*restart)(void);
    int    (*intack)(int level);
    void   (*irq_event)(uint32 irq, int kind);
    void   (*start)(void);
    void   (*stop)(void);

    /* --- Initialised by TSIM --- */
    void (*correctable_error)(uint32 addr, uint32 master, uint32 size, int write);
    int (*add_apb_slave)(uint32 base,
                         uint32 size,
                         void *priv,
                         int (*read)(void *priv, uint32 addr, uint32 *data),
                         int (*write)(void *priv, uint32 addr, uint32 data));
    int (*add_ahb_slave)(uint32 base,
                         uint32 size,
                         int cacheable,
                         void *priv,
                         uint8 *(*get_mem_ptr)(void *priv, uint32 base, uint32 size),
                         int (*read)(void *priv, struct ahb_access *access),
                         int (*write)(void *priv, struct ahb_access *access));
    int (*add_apb_pp)(uint32 vendor, uint32 device,
                      uint32 version, uint32 irq,
                      uint32 absolute_address,
                      uint32 absolute_mask);
    uint32 (*build_ahb_id)(uint32 vendor, uint32 device, uint32 version,
                           uint32 irq);
    uint32 (*build_ahb_membar)(uint32 start, uint32 size,
                               int cacheable, int prefetchable);
    uint32 (*build_ahb_iobar)(uint32 start, uint32 size,
                              int cacheable, int prefetchable);
    int (*add_ahb_pp)(int master, uint32 id,
                      uint32 bar0, uint32 bar1, uint32 bar2, uint32 bar3);
    int (*add_ahb_pp_full)(int busid, int master, uint32 id,
                           uint32 user0, uint32 user1, uint32 user2,
                           uint32 bar0, uint32 bar1, uint32 bar2, uint32 bar3);
};
```

5.3.1.1. Elements initialised by module

The elements of the structure initialised by modules have the following meanings:

`void (*init)(void);`
: Called once on simulator startup. Set to NULL if unused.

`void (*exit)();`
Called once on simulator exit. Set to NULL if unused.

`void (*reset)();`
Called every time the system is reset, including at startup and restart. Set to NULL if unused.

`void (*restart)();`
Called every time the simulator is restarted (simtime set to zero). Set to NULL if unused.

`int (*intack)(int level);`
`intack()` is called when the processor takes an interrupt trap (tt = 0x11 - 0x1f). The level of the taken interrupt is passed in level. This callback can be used to implement interrupt controllers. `intack()` should return 1 if the interrupt acknowledgement was handled by the AHB module, otherwise 0. If 0 is returned, the default LEON interrupt controller will receive the intack instead.

`void (*irq_event)(uint32 irq, int kind)`
Leon3/4 only: The `irq_event()` function is called when there is a change on the interrupt bus. When an edge interrupt happens, kind is IRQKIND_EDGE. When a level interrupt starts to be driven kind is IRQKIND_LEVELON, and when it stops kind is IRQKIND_LEVELOFF. The irq parameter is the interrupt number of the interrupt that is happening/changing, in the range 1-64 (maximum depends on system).

`void (*start)(void)`
Called each time simulation starts, both when starting for the first time using **boot** or **run** command and when continuing using **go**, **cont**, **step** and the like.

`void (*stop)(void)`
Called every time simulation stops, e.g. due to breakpoints, user pressing Ctrl-C, etc.

5.3.1.2. Elements initialised by TSIM

The elements of the structure initialised by TSIM have the following meanings:

`struct sim_interface *simif;`
Entry `simif` is initialised by tsim with the global `struct sim_interface` structure.

`void (*snoop) (unsigned int addr)`
The callback `snoop` is initialised by tsim. If data cache snooping is enabled (and functioning, i.e. not UT699) it flushes (i.e. invalidates) data cache lines corresponding to physical address *`addr`* (on LEON3/4 even when MMU is enabled). If the AHB module is doing DMA writes directly to memory pointers, it is the responsibility of the AHB module to call this for all changed words for snooping to work correctly.

`struct io_interface *io;`
Initialised with the I/O interface structure pointer.

`void (*dprint)(char *);`
Initialised by tsim with a callback pointer to the debug output function. Output ends up in log, when logging is enabled and gets forwarded to gdb when running TSIM via gdb. See `lprintf` and `vlprintf` for the formatted counterparts.

`struct proc_interface *proc;`
Initialised with the procif structure pointer.

`int (*lprintf)(const char *format, ...)`
Initialised by TSIM with a function for formatted output that goes both to stdout and, when logging, to the log. The function interface works like for printf.

`int (*vlprintf)(const char *format, va_list ap)`
Initialised by TSIM with a function for formatted output that goes both to stdout and, when logging is enabled, to the log. The function interface works like for vprintf.

`void correctable_error(uint32 addr, uint32 master, uint32 size, int write)`
Can be called by an AHB module to signal a correctable error to an AHBSTAT core (if present) or a LEON2 memstat. It is intended to be called during handling of a successful read or write. The parameters to supply corresponds to the register fields to the AHBSTAT registers or LEON2 FAILAR/FAILSR registers (the rw field in LEON2 FAILSR corresponding to `!write`).

```
int (*add_apb_slave)(uint32 base, uint32 size, void *priv, int (*read)(void
*priv, uint32 addr, uint32 *data), int (*write)(void *priv, uint32 addr,
uint32 data));
```

Registers an APB slave. The, *`base`* parameter is the start address of the area, *`size`* is the size of the area (in bytes). Both needs to be divisible by 0x100. The *`priv`* parameter is a pointer that can be set freely by the user and is provided to calls to the *`read`* and *`write`* functions. The registered read and write functions are called on bus reads and writes from and to the registered memory area respectively. APB slave models

(in contrast to AHB slave models) do not need to be concerned about access timing, different write sizes or number of multiple word reads. Those things are handled by the APB controller model. The APB slave only handles single word reads and single word writes.

The *read* function is called for reads from the registered area. The *priv* argument is the pointer registered in the add_apb_slave call. The *addr* parameter contains the address of the single word read. The *data* parameter points to a buffer where the read data should be placed into on a successful read. The function should return 0 for a successful access or 1 for a failed access.

The *write* function is called for writes to the registered area. The *priv* argument is the pointer registered in the add_apb_slave call. The *addr* parameter contains the address of the single word write. The *data* parameter contains the word that is written. The function should return 0 for a successful access or 1 for a failed access.

```
int (*add_ahb_slave)(uint32 base, uint32 size, int cacheable, void *priv,
uint8 *(*get_mem_ptr)(void *priv, uint32 base, uint32 size), int (*read)
(void *priv, struct ahb_access *access), int (*write)(void *priv, struct
ahb_access *access))
```

Registers an AHB slave. Here, *base* is the start address of the area, *size* is the size of the area (in bytes). Both needs to be divisible by 0x100. The *cacheable* parameter indicates if the area is cacheable or not. The *priv* parameter can be set freely by the user and is provided to calls to the *read* and *write* functions. Details on the *read*, *write* and *get_mem_ptr* functions are described in Section 5.3.1.3.

```
void (*add_apb_pp)(uint32 vendor, uint32 device, uint32 version, uint32
irq, uint32 absolute_address, uint32 absolute_mask)
```

Add APB plug&play entry. Here, *vendor* is the vendor ID of the device, *device* is the device ID and *version* is the device version. The *irq* parameter is the registered device IRQ. The *absolute_address* parameter is the base address of the area. *absolute_mask* is the address mask, usually taking the size of the area in bytes - 1 and bit-invert that.

```
uint32 (*build_ahb_id)(uint32 vendor, uint32 device, uint32 version, uint32
irq)
```

Helper function to build a plug&play ID.

```
uint32 (*build_ahb_membar)(uint32 start, uint32 size, int cacheable, int
prefetchable)
```

Helper function to build a plug&play bar. Here, *start* is the beginning of the area, *size* is the size. *cacheable* indicates if the area is cacheable and *prefetchable* indicates if the area is prefetchable. Calling with both *start* and *size* set to 0 will produce an all zero bar and can be used as a disabled bar. Otherwise a return value of 0 indicates failure to build an I/O bar, and an accompanying error printout will be made.

```
uint32 (*build_ahb_iobar)(uint32 start, uint32 size, int cacheable, int
prefetchable)
```

Helper function to build a plug&play bar. *start* is the beginning of the area, *size* is the size. *cacheable* indicates if the area is cacheable and *prefetchable* indicates if the area is prefetchable. Calling with both *start* and *size* set to 0 will produce an all zero bar and can be used as a disabled bar. Otherwise a return value of 0 indicates failure to build an I/O bar, and an accompanying error printout will be made.

```
int (*add_ahb_pp)(int master, uint32 id, uint32 bar0, uint32 bar1, uint32
bar2, uint32 bar3)
```

Register an AHB plug&play entry. Above helper build_* functions can be used to construct the different bars. The *master* argument should be 1 when registering an entry for an AHB master and 0 when registering an AHB slave. The *build_ahb_id* helper function can be used for building the *id*, and the *build_ahb_iobar* and *build_ahb_membar* helper functions can be used for building the different bars. Returns 0 on success, and non-zero on error. See also add_ahb_pp_full that also supports choosing a bus to add the entry to and adding user data fields.

```
int (*add_ahb_pp_full)(int busid, int master, uint32 id, uint32 user0,
uint32 user1, uint32 user2, uint32 bar0, uint32 bar1, uint32 bar2, uint32
bar3);
```

Register an AHB plug&play entry. Works just like add_ahb_pp, but with the added possibility to choosing a bus to add the entry to and adding user data fields. Above helper build_* functions can be used to constructthe different bars. The *master* argument should be 1 when registering an entry for an AHB master and 0 when registering an AHB slave. The *build_ahb_id* helper function can be used for build-

ing the *id*, and the *build_ahb_iobar* and *build_ahb_membar* helper functions can be used for building the different bars. Returns 0 on success, and non-zero on error.

5.3.1.3. Callbacks for AHB module AHB slaves

For AHB slaves, read and write callback functions is registered using add_ahb_slave to handle reads and writes from and to the registered memory area. It is also possible to register a get_mem_ptr to allow access to emulated memory. That is required for e.g. load to work against user emulated memory. Note that for APB slaves, a slightly different interface is used.

```
struct ahb_access {
   uint32 address;
   uint32 *data;
   uint32 ws;
   uint32 rnum;
   uint32 wsize;
};

/* Callbacks */
int (*read)(void *priv, struct ahb_access *access)
int (*write)(void *priv, struct ahb_access *access)
uint8 *(*get_mem_ptr)(void *priv, uint32 base, uint32 size)
```

AMBA slave read function. The registered read function is called on bus reads from the registered memory area. The *priv* argument is a pointer to the private data used when the area was registered. A read is always treated as a read of one or more 32-bit words. The *access->addr* field contains the address of the first word to read. The *access->data* field points to a buffer that should be filled in with the read data on a successful read. The *access->ws* field should be set by the module to the number of cycles for the complete access. The *access->rnum* field contains the number of words to be read. The function should return 0 for a successful access or 1 for a failed access. The *access->wsize* field is not used for reads.

AMBA slave write function. The registered write function is called on bus writes to the registered memory area. The *priv* argument is a pointer to the private data used when the area was registered. The *access->addr* field contains the address of the write. The *access->data* field points to the data to write; either one word for a byte, half word or word write, or two words for double-word writes. The *access->wsize* field defines write size as follows: 0 = byte, 1 = half-word, 2 = word, 3 = double-word (no other sizes are valid). The *access->ws* field should be set by the module to the number of cycles for the complete access.The function should return 0 for a successful access and 1 for failed access. The *access->rnum* field is not used for writes.

AHB slave get_mem_ptr function. During file load operations, TSIM will access emulated memory through a memory pointer. Such a pointer can be returned from user emulated memory via the get_mem_ptr function. Without such a pointer, loads can not be performed to user emulated memory. When this function is available it can also be used by TSIM for other non-simulation accesses like when displaying memory contents. The *priv* argument is the private data pointer used when the area was registered. The *base* parameter is the base address of the area and and *size* parameter is the size of the area requested (in bytes). The function should return a character pointer to the emulated memory array if the address and size is totally within the range of the emulated memory. If outside the range, NULL should be returned. Set this callback to NULL if not used.

5.3.2. Big versus little endianess

SPARC conforms to the big endian byte ordering. This means that the most significant byte of a (half) word has lowest address. To execute efficiently on little-endian hosts (such as Intel x86 PCs), emulated memory is organised on word basis with the bytes within a word arranged according the endianess of the host. Read cycles can then be performed without any conversion since SPARC always reads a full 32-bit word. During byte and half word writes, care must be taken to insert the written data properly into the emulated memory. On a byte-write to address 0, the written byte should be inserted at address 3, since this is the most significant byte according to little endian. Similarly, on a half-word write to bytes 0/1, bytes 2/3 should be written.

5.3.3. AHB module example

See the ahb.c example pointed out in Section 5.8.

5.3.4. AHB module override

Adding a user modele that overlaps an bus address area that is already modelled by TSIM or another module by default results in an error printout on startup. However, when using the `-bus_ov` option, AHB modules can override internal TSIM models on the bus, using the the regular `add_ahb_slave` and/or `add_apb_slave` functions, replacing the callbacks of the internal model with the callbacks of the module. When using `-v`, TSIM will inform about areas that are overridden.

AHB2AHB bridges themselves can in general not be overridden, but what is behind them can be. The same applies to the IO area of an L2 cache, but overriding the memory or register areas of an L2 cache overrides the L2 cache itself. It is not possible to override the memory area of the memory controller behind the L2 cache. APB slaves can be overridden by an APB slave without overriding the APB bridge which it is placed behind. APB bridges (or parts thereof) can be overridden by an AHB slave. It is possible to override address areas that do not adhere to the regular alignment and size limitations of 0x100 for registering AHB/APB slaves. This however is only intended and supported for overriding uncacheable areas where reads and writes are only done using 32-bit word accesses, e.g. registers.

5.4. I/O module interface

The AHB module system is the primary way to add user models for bus devices. The I/O device interface can be used to add a module to the I/O bus behind the memory controller (when present in the system) or to act as a fallback taking care of accesses for areas that are not modelled by anyone. If neither TSIM or any AHB module handles a memory access it will be forwarded to an I/O module if present. To register an I/O module, call `tsim_register_io_module(iosystem)` from the init function of a loadable_module struct, see Section 5.1.2. There can be only one I/O module.

The io_subsystem struct is described below.

```
struct io_subsystem {
   void (*io_init)(void); /* start-up */
   void (*io_exit)();      /* called once on exit */
   void (*io_reset)();     /* called on processor reset */
   void (*io_restart)();   /* called on simulator restart */
   int (*io_read)(unsigned int addr, int *data, int *ws);
   int (*io_write)(unsigned int addr, int *data, int *ws, int size);
   char *(*get_io_ptr)(unsigned int addr, int size);
};
```

The elements of the structure have the following meanings:

`void (*io_init)(void);`
: Called once on simulator startup. Set to NULL if unused.

`void (*io_exit)();`
: Called once on simulator exit. Set to NULL if unused.

`void (*io_reset)();`
: Called every time the system is reset, including at startup and restart. Set to NULL if unused.

`void (*io_restart)();`
: Called every time the simulator is restarted (simtime set to zero). Set to NULL if unused.

`int (*io_read)(unsigned int addr, int *data, int *ws);`
: Processor read call. The processor always reads one full 32-bit word from addr. The data should be returned in *data, the number of waitstates should be returned in *ws. If the access would fail (illegal address etc.), 1 should be returned, on success 0.

`int (*io_write)(unsigned int addr, int *data, int *ws, int size);`
: Processor write call. The size of the written data is indicated in size: 0 = byte, 1 = half-word, 2 = word, 3 = double-word. The address is provided in addr, and is always aligned with respect to the size of the written data. The number of waitstates should be returned in *ws. If the access would fail (illegal address etc.), 1 should be returned, on success 0.

`char * (*get_io_ptr)(unsigned int addr, int size);`
: TSIM can access emulated memory in the I/O device in two ways: either through the `io_read`/`io_write` functions or directly through a memory pointer. `get_io_ptr()` is called with the target address and transfer size (in bytes), and should return a character pointer to the emulated memory array if the address and size is within the range of the emulated memory. If outside the range, NULL should be returned. Set to NULL if not used.

5.5. Adding startup options

A module can register a startup option by filling in a `struct user_option` and calling the `tsim_register_user_option` function which will return 0 on success and 1 on failure to register the option.

```
struct user_option {
    /* User defined private pointer*/
    void *arg;
    /* Called when the option is parsed */
    int (*option_execute)(void *arg, int argc, const char **argv);
    const char *name;         /* Name of the option */
    const char *help_oneline; /* One line description */
    const char *help_full;    /* Comlete description */
    const char *help_syntax;  /* Description of option syntax */
};

int tsim_register_user_option(struct user_option *user_option);
```

The *name* pointer must be set to a unique option name, and the *option_execute* pointer to a callback function. The *option_execute* callback will be called when the option is parsed at simulator startup and will get the registered *arg* as first parameter with the number of startup arguments in *argc* and the arguments in the *argv* array. The option name itself is included in the count and is the first entry of the array. The return value from *option_execute* should be how many arguments the option parsed, e.g. 1 if no arguments other than the option itself, or 2 if another parameter was parsed.

The *help_oneline*, *help_full* and *help_syntax* pointers can be set to a oneline description of the option, a full documentation of the option and if the option takes any arguments the syntax can be set, in order for the options to be supported by the `-help` option.

5.6. Adding user commands

A module can register a user command by filling in a `struct user_cmd` and calling the `tsim_register_user_cmd` function which will return 0 on success and 1 on failure to register the command.

```
struct user_cmd {
    /* User defined private pointer*/
    void *arg;
    /* Called when the command is executed */
    int (*cmd_execute)(void *arg, int argc, char **argv);
    /* Called on unregistration of commands */
    void (*cmd_unregister)(void *arg);
    const char *name;         /* Name of the command */
    const char *help_oneline; /* One line description */
    const char *help_full;    /* Comlete description */
    const char *help_syntax;  /* Description of command syntax */
};

int tsim_register_user_cmd(struct user_cmd *user_cmd);
```

The *name* pointer must be set to a unique command name, and the *cmd_execute* pointer to a callback function. The *cmd_execute* callback will be called when the command is evaluated and will get the registered *arg* as first parameter with the number of command arguments in *argv* and the arguments in the *argv* array. The command name itself is included in the count and is the first entry of the array. The return value from *cmd_execute* becomes a signed integer Tcl return value.

The *help_oneline*, *help_full* and *help_syntax* pointers can be set to a oneline description of the command, a full documentation of the command and if the command takes any arguments the syntax can be set, in order for the command to be supported by the **help** command. The *cmd_unregister* pointer can optionally be set to be called when TSIM exits, e.g. if cleanup needs to be done.

5.7. Check-pointing module state

Modules can setup up callbacks for the `save` and `restore` module callbacks can be used to support saving and restoring simulation state in user modules.

```
struct checkpoint; /* Type for pointers passed back to TSIM */

struct user_checkpoint {
    void *priv; /* For the module to use during a call to its save/restore */
    struct checkpoint *check; /* To pass to read/write functions */
```

```
    char *filename; /* File name for checkpoint */
    int (*write_uint32)(struct checkpoint *check, uint32 data);
    int (*write_uint64)(struct checkpoint *check, uint64 data);
    int (*write_buf)(struct checkpoint *check, void *data, uint64 size);
    int (*write_sparse_buf)(struct checkpoint *check, void *data, uint64 size);
    int (*read_uint32)(struct checkpoint *check, uint32 *data);
    int (*read_uint64)(struct checkpoint *check, uint64 *data);
    int (*read_buf)(struct checkpoint *check, void *data, uint64 size);
    int (*read_sparse_buf)(struct checkpoint *check, void *data, uint64 size);
};

struct loadable_module {
    ...
    int (*save)(struct loadable_module *module, struct user_checkpoint *ucp);
    int (*restore)(struct loadable_module *module, struct user_checkpoint *ucp);
};
```

The `struct user_checkpoint` passed to `save` and `restore` functions contains the pointers and functions for a module to save state to or restore state from the file that is specified with the **save** *`file`* and **restore** *`file`* commands.

The `write_` functions can be used in the module's `save` callback to write data to the current checkpoint file and the `read_` functions can be used in the module's `restore` callback to read data to the current checkpoint file. Care needs to be taken to exactly match the `write_` function calls during save with the corresponding `read_` between the save and restore call during restore. The `_uint32` and `_uint64` functions handles a single integer. The `_buf` functions handles any binary data of a given number of bytes. The `_sparse_buf` functions can be used for data that potentially contains a lot of repeated areas with zeroes, such as the backing buffer for emulated memory, to keep the checkpoint file size down. The buffer passed to the `_sparse_buf` functions must be 32-bit word aligned and the size (in bytes) must be divisible by 4. The read and write functions returns 0 on success or non-zero on fail. When getting a non-zero return value, return that upwards in order to get the correct error printouts. In case the module wishes to return an error for its own reasons, return `1` as the error code.

When a module is using events it is important that it uses restorable events for check-pointing to work. Pre-register events from any of the `init` callbacks to get an ID using `reg_revent` and/or `reg_revent_prearg`. The latter should be used when the user argument is some pointer that might differ in address between sessions. Events are then added to the event queue with `revent` and `revent_prearg` respectively, and removed with `stop_revent`, using the ID from the pre-registration.

The `examples/modules/ahb.c` example module distributed with TSIM contains an example of using this interface. In case one rather wants to save to and restore from a separate file, the filename of the current file is also available to be able to base a separate file name on it.

5.8. Loadable modules distributed with TSIM

The following table shows which loadable modules are distributed with which TSIM versions.

Table 5.1. Loadable modules distributed with TSIM

Module	File	For TSIM versions
AHB module example	examples/modules/ahb.c	All
I/O module example	examples/modules/io.c	All
Walltime synchronisation example	examples/modules/walltimesync.c	All
CAN node example	examples/input/can_node.c	LEON3/4
GPIO input example	examples/input/gpio.c	LEON3/4
SPI slave example	examples/input/spi.c	LEON3/4
SPI memory example	examples/input/spim.c	LEON3
PCI target example	examples/input/pci_target.c	LEON3
TPS VxWorks 6.x AHB Module	tps/linux-x64/tps.so tps/win64/tps.dll	LEON3/4

The example modules that are provided in source also comes with makefiles to build them. The example modules in `examples/input` also has usage examples in the `examples/input/README.txt`.

6. TSIM library (TLIB)

6.1. Introduction

TSIM is also available as a library, allowing the simulator to be integrated in a larger simulation frame-work. All options, commands and Tcl possibilities of standalone TSIM are accessible through a simple function interface. Both builtin and external user models can be added, using the same interfaces as for standalone TSIM.

6.2. Function interface

The following functions are provided to access TSIM features. Note that a lot of additional functions accessed via the callback structs returned by the `tsim_get_simif`, `tsim_get_procif` and `tsim_get_ioif` functions are available as well. They are described in Section 5.2.

`int tsim_init(char *option);`
: Initialise TSIM. This must be called before any other TSIM function (except `tsim_set_diag` and `tsim_register_module`) are used. The options string can contain any valid TSIM startup options. The `tsim_init` function will return 0 on success or 1 on failure. Quotation with " and escaping with \ is supported in order to pass arguments that contains whitespace. The `tsim_init` or `tsim_argv_init` function may only be called once. Commands like **run**, **boot** and **reset** can restart simulation without restarting the process.

`int tsim_argv_init(int argc, char **argv);`
: Initialise TSIM. This must be called before any other TSIM function (except `tsim_set_diag` and `tsim_register_module`) are used. The options array can contain any valid TSIM startup options. The `tsim_argv_init` function will return 0 on success or 1 on failure. This version can be advantageous over `tsim_init` e.g. to avoid having to escape whitespace characters for arguments with whitespace in them. The `tsim_init` or `tsim_argv_init` function may only be called once. Commands like **run**, **boot** and **reset** can restart simulation without restarting the process.

`int tsim_cmd(char *cmd);`
: Execute TSIM command. Any valid TSIM command-line or Tcl command may be given. If the command was executed successfully the result can be retrieved with the helper functions `tsim_get_result_*()`. Returns 0 on success and non-zero on failure.

`int tsim_cmdf(const char *format, ...) FMT_PRINTF(1, 2);`
`int tsim_vcmdf(const char *format, va_list ap);`
: Execute TSIM command built up as with a printf format string. Any valid TSIM command-line command or Tcl expression may be built. If the command was executed successfully the result can be retrieved with the helper functions `tsim_get_result_*()`. Returns 0 on success and non-zero on failure.

`int tsim_get_result_int32(int32 *result);`
`int tsim_get_result_uint32(uint32 *result);`
`int tsim_get_result_int64(int64 *result);`
`int tsim_get_result_uint64(uint64 *result);`
: After a executed TSIM command. The result of the command can be retrieved as integer. The result will be returned in *`*result`*. Return 0 on success or non-zero if result could not be interpreted as an integer.

`int tsim_get_result_double(double *result);`
: After a executed TSIM command. The result of the command can be retrieved as double. The result will be returned in *`*result`*. Return 0 on success or non-zero if result could not be interpreted as a double.

`int tsim_get_result_str(char **result);`
: After a executed TSIM command. The result of the command can be retrieved as string. The result will be returned in *`**result`*. Return 0 on success or non-zero if result could not be interpreted as a string. The caller is responsible to free the string.

`void tsim_exit(int val);`
: Should be called to cleanup TSIM internal state before main program exits.

`int tsim_cont(struct tsim_duration *duration, int *sig, int *cpuid)`
: Continues simulation for duration specified in the tsim_duration struct, described in tsim.h. If the *`sig`* is not NULL, the reason for why simulation stopped is returned in *`*sig`*. If the *`cpuid`* is not NULL, the CPU resposible for why simulation stopped is returned in *`*cpuid`*. Returns 0 if simulation could be started or non-zero on error starting simulation. Before and after one or more calls to `tsim_cont`, the `tsim_cont_setup` and `tsim_cont_restore` respectively should be called once. See Section 3.4 and Table 3.2 for how to interpret *`*sig`* and *`*cpuid`*.

`int tsim_cont_setup(void);`
Prepares for having tsim_cont drive the simulation. Only one setup call is needed before any number or calls to tsim_cont. This is needed for e.g. UART forwarding. This returns 0 on success, non-zero on failure.

`int tsim_cont_restore(void);`
Restores things after tsim_cont driving the simulation. Only one restorecall is enough after any number or calls to tsim_cont. This is needed e.g. for the terminal to be restored to a normal state after UART forwarding. This returns 0 on success, non-zero on failure.

`int tsim_get_stopreason(int *sig, int *cpuid)`
Returns reason for stopping, and ID of the CPU that was responsible for stopping the last simulation execution. If the *sig* is not NULL, the reason for why simulation stopped is returned in **sig*. If the *cpuid* is not NULL, the CPU resposible for why simulation stopped is returned in **cpuid*. See Section 3.4 and Table 3.2 for how to interpret **sig* and **cpuid*. This returns 0 on success, non-zero on failure.

`int tsim_get_reg(int cpuid, int regid, uint32 *value)`
Get single SPARC register. `cpuid` is an index of the CPU to get the register value from, `regid` is an index of the register to get, as per `enum regnames` in `tsim.h`, and `value` is a pointer to where the register value will be returned. Returns 0 on success, non-zero on failure.

`int tsim_set_reg(int cpuid, int regid, uint32 value)`
Set single SPARC register. `cpuid` is an index of the CPU to set the register value on, `regid` is an index of the register to set, as per `enum regnames` in `tsim.h`, and `value` is the value to set the register to. Returns 0 on success, non-zero on failure.

`int tsim_read(unsigned int addr, unsigned int *data);`
Performs a read from *addr*, returning the value in **data*. Only for diagnostic use. Returns 0 on success else 1.

`int tsim_write(unsigned int addr, unsigned int data);`
Performs a write to *addr*, with value *data*. Only for diagnostic use. Returns the number of bytes written.

`int tsim_set_diag(int (*cfunc)(void *priv, const char *buf, int len), void *priv);`
Set output forwarding function. By default, TSIM writes all output to stdout and stderr. This function can be used to direct all output to a user defined routine. The *cfunc* callback function will be called for all TSIM output. It should make sure to handle the entire buffer and return number for bytes handled. The *priv* parameter gets passed to each call. The *buf* and *len* arguments contains the buffer with text to be handled and the length of it. The `tsim_set_diag` function returns 0 on success 1 or failure.

`int tsim_set_callback(int cpuid, void (*cfunc)(int cpuid, uint32 pc));`
Set the debug callback function for a given CPU. Calling `tsim_set_callback()` with a function pointer will cause TSIM to call the callback function just before each executed instruction on the given CPU. Returns 0 on success else non-zero.

`void tsim_trap(int (*trap)(int cpuid, int tt), void (*rett)(int cpuid));`
`tsim_trap()` is used to install callback functions that are called every time the processor takes a trap or returns from a trap (RETT instruction). The `trap()` function is called with the CPU ID of the trapping CPU, *cpuid*, and the SPARC trap number, *tt*. If the `trap()` function returns 0, execution will continue. A non-zero return value will stop simulation with the program counter pointing to the instruction that will cause the trap. The `rett()` function is called when the program counter points to the RETT instruction but before the instruction is executed. The *cpuid* parameter contains the CPU ID of the CPU that is about to return from trap. It is possible to install only one callback function by setting the other one to NULL. The callbacks can be removed by calling `tsim_trap()` with NULL arguments.

`void tsim_cov_get(int cpuid, int start, int end, char *ptr);`
`tsim_cov_get()` will return the coverage data for the address range from *start* (inclusive) to *end* (exlusive) from the specified CPU. The coverage data will be written to a char array pointed to by **ptr*, starting at ptr[0]. One character per 32-bit word in the address range will be written. The user must assure that the char array is large enough to hold the coverage data. Note that changing coverage modes will reset the coverage data.

`void tsim_cov_set(int cpuid, int start, int end, char val);`
`tsim_cov_set()` will fill the coverage data in the address range limited by *start* and *end* (see above for definition), for the specified CPU, with the value of *val*.

`int tsim_lastbp(int *addr, int *cpu, int *bp)`
When simulation stopped due to breakpoint or watchpoint hit (SIGTRAP), this function will return the address of the break/watchpoint in **addr*. The index of the CPU in **cpu* and the index of the break/watchpoint in **bp*. The function return value indicates the break cause; 0 = breakpoint, 1 = watchpoint.

```
void tsim_ext_ins(int (*func)(void *priv, int cpuid, uint32 inst, uint32
**window, uint32 *icnt), void *priv);
```

Installs a handler, *func*, for custom instructions. The installed function gets to emulate instructions as described in Section 4.1.7. Calling with `func` as a a NULL pointer will remove the handler.

```
typedef int (*gdb_send_func)(void *priv, const void *buf, int len, int
*bytes);
typedef int (*gdb_recv_func)(void *priv, void *buf, int len, int *bytes);
int tsim_gdb(gdb_send_func send, gdb_recv_func recv, void *priv);
```

Starts a GDB session with custom send and receive functions handling the GDB remote protocol input and output streams. Returns 0 on success, non-zero of error. The *send* callback is a function pointer to a function that is called by TSIMs GDB server to send. The *recv* callback is a function pointer to a function that is called by TSIMs GDB server to receive. The *priv* pointer is registered by the user to be passed to the send and recv callbacks.

The callback functions should return 0 on success, non-zero on error. Error return here stops the GDB session. The callback parameters works as follows. The *priv* parameter is the private pointer registered by user. The *buf* parameter is a buffer to send from or to receive into. The *len* parameter is the buffer size for receive, and length of data to be sent. The *bytes* parameter is used to set the number of bytes actually sent or received should be set in **bytes*.

Note that in order for a Ctrl-C in a connected GDB to work, execution should be interrupted using `struct sim_interface.sim_stop` when GDB want to send, i.e. the *recv* callback has data to return, during ongoing simulation.

```
struct sim_interface *tsim_get_simif(void);
```

Get TSIM simif interface. See Section 5.2.

```
struct proc_interface *tsim_get_procif(void);
```

Get TSIM procif interface. See Section 5.2.

```
struct io_interface *tsim_get_ioif(void);
```

Get TSIM ioif interface. See Section 5.2.

```
int tsim_register_module(struct loadable_module *module);
```

Registers a custom module. Must be called before `tsim_init()`. See Chapter 5 for more information. Return 0 on success else non-zero.

6.3. Builtin and external modules and user models

Builtin modules can be loaded when using TLIB by registering a module with the `tsim_register_module` function. See Chapter 5 for further information. It is also possible to use external modules as with standalone TSIM using `-mod` *`module`* when calling `tsim_init()`

6.4. Linking a TLIB application

The library versions of TSIM are provided as dynamic shared objects, as .so files on Linux and DLL files on Windows. Sample applications are provided, demonstrating different TLIB functionalities, together with a Makefile showing how to build and link them with TLIB.

6.5. TLIB constraints and workarounds

There can be only one TSIM instance ever per process. In other words, `tsim_init` can only be called once per process and not be called again, even after calling `tsim_exit`. Simulation can be restarted by the **run**, **reset** and go **boot** TSIM commands within one TSIM instance.

TSIM commands are Tcl commands and all calls to the `tsim_cmd` family of functions go through Tcl. This includes `tsim_cmd`, `tsim_cmdf` and `tsim_vcmd`. Tcl evaluations can only be done in the same thread as where the Tcl interpreter was created. By default this is the thread that called tsim_init. If there is a need to be able to call e.g. `tsim_cmd` from a different thread than the thread that called `tsim_init`, TLIB needs to be e started with the `-tclwt` option, by passing it to `tsim_init`. That will make sure that a Tcl worker thread is started. It will automatically handle all Tcl evaluations under the hood of e.g. `tsim_cmd`. When `-tclwt` is in use, it is especially advisable to use `tsim_cont` rather than e.g. `tsim_cmd` with **cont** if progressing simulated time in small time slices. This avoids an extra thread switch to the worker thread for each such time slice. Note

that the Tcl and multiple threads constraint can also include usage of `tsim_gdb` as the GDB monitor command will result in Tcl evaluation in the TSIM end.

The functions in the TSIM/TLIB C API are not thread safe, unless specifically noted. Even with the usage of the `-tclwt` option, it is up to the TLIB application to make sure that no concurrent calls are ever done to multiple (non-exempt) TSIM/TLIB C functions.

See also the constraints in Section 5.2.1. For TLIB, calls can be done to most of the various TSIM/TLIB functions can also be done when the control is in the hands of the TLIB application.

6.6. Files and Examples

The `tlib` directory contains a subdirectory for each architecture with the TLIB `tsimleon*.so` so-file or `tsimleon*.dll` DLL together with needed external libraries as well as example TLIB applications.

The `app3.c` example contains an example simulation loop using `tsim_cont` between `tsim_cont_setup` and `tsim_cont_restore` calls, which is the suggested way (rather than using the **cont** command) to have a loop progressing TSIM a timeslice at a time.

7. GR712RC emulation

To emulate the GR712RC chip the `-gr712rc` option should be used.

The following table lists which cores in the GR712RC are modelled by TSIM or not. The table contains some notes of some unsupported features for otherwise supported cores, but is not necessarily exhaustive in this respect.

Table 7.1. Simulation models for GR712RC

Core	Status	Notes
LEON3FT	Supported by core TSIM3	Both CPUs are modelled. No FT features are modelled.
GRFPU	Supported by core TSIM3	Does not simulate the possibility of multiple outstanding floating point operations.
AHBSTAT	Supported by core TSIM3	
APBCTRL	Supported by core TSIM3	
APBUART	Supported by core TSIM3	
FTMCTRL	Supported by core TSIM3	No FT features are modelled
GPTIMER	Supported by core TSIM3	No Watchdog support.
GRTIMER	Supported by core TSIM3	
IRQMP	Supported by core TSIM3	
CAN_OC	Supported by core TSIM3	See Chapter 16. Checkpointing currently not supported.
FTAHBRAM	Supported by core TSIM3	No FT features are modelled.
GRETH	Supported by core TSIM3	See Chapter 17. Checkpointing currently not supported.
GRGPIO	Supported by core TSIM3	See Chapter 18.
GRSPW2	Supported by core TSIM3	See Chapter 21. Checkpointing currenty not supported.
SPICTRL	Supported by core TSIM3	See Chapter 23.
CANMUX	Dummy in TSIM3	Functionality-less registers only
CLKGATE	Dummy in TSIM3	Functionality-less registers only
GRGPREG	Dummy in TSIM3	Functionality-less registers only
B1553BRM	Not supported	
GRASCS	Not supported	
GRSLINK	Not supported	
GRTC	Not supported	
GRTM	Not supported	
I2CMST	Not supported	
AHBJTAG	Not supported	Debug link
DSU3	Not supported	Debug unit

TSIM supports running user defined models for unsupported cores.

7.1. Clock Gating Unit, CANMUX and GRGPREG

The Clock Gate Unit, CANMUX and GRGPREG I/O registers and AMBA Plug & Play area are present in the GR712RC module. Some of the logic to control which bits are implemented, readable and writable etc. is implemented. However the register bits has no functionality. The current register values can be used by custom I/O modules in SW validation. For example checking that accessing a specific address are has not been clock gate disabled or that the SpW clock PLL match with the expect value after initialisation.

8. GR716A emulation

To emulate the GR716A chip the `-gr716a` option should be used. When using the GR716 only release, TSIM only simulates GR716A and GR716B.

Table 8.1. Simulation models for GR716A

Core	Notes and model limitations
AHBROM	GR716A Boot ROM. See Section 8.1.
AHBSTAT	Only the AHB Status Register for the main AMBA bus supported
APBCTRL	Atomic operations supported.
APBUART	Transmitter shift register empty interrupt, delayed interrupt and using two stop bits currently not supported.
LRAM	Atomic operations and DMA accesses supported. Configuration registers are implemented as dummy registers, see Section 8.2.
FTMCTRL	EDAC not supported.
GPTIMER	No Watchdog support. Control register bits EV, ES and EE not supported.
GRCAN	See Chapter 15 for details about the CAN bus. Checkpointing currently not supported.
GRGPIO	See Chapter 18. Pulse sampler and Pulse sequencer are not currently supported. Atomic operations supported.
GRGPREG	Only bootstrap register implemented.
GRSPW2	See Chapter 21 for details and limitations. Checkpointing currently not supported.
IRQMP	Watchdog control and Error mode status register currently not implemented.
LEON3FT	No FT features are modelled. ZeroJitter, Alternative Window Pointer and REX ISA not currently supported. Register window partitioning is supported.
SPICTRL	See Chapter 23. Automatic Slave select, ThreeWire mode and Slave mode are not currently supported.
SPIMCTRL	See Chapter 24. EDAC not supported.
DAC	See Section 8.3. Checkpointing currently not supported.

GR716A has tightly coupled dual-port local data and instruction RAM. GR716A does not have cache memories. Some register areas for devices that are not emulated are implemented as dummy registers. See Section 8.2. TSIM does not emulate the DSU, L3STAT and AHBTRACE cores but provide a lot of corresponding functionality and information via TSIM commands instead.

8.1. GR716A Boot ROM

In addition to running RAM images directly from memory using **load** and **run**, TSIM can simulate a cold start going through the bootloader in the GR716A Boot ROM, with its different boot possibilities, or bypassing the Boot ROM, booting directly from a different source. The **boot** command is used to start simulating a cold start.

The bootloader in the GR716A Boot ROM supports multiple boot sources. Booting from external SRAM, external PROM and external SPI memory is supported. There is currently no built-in model for the I^2C controller. Therefore, to support booting an image read over from I^2C, a user model for the I^2C controller and bus is needed. The bootloader can also set up the GR716A for remote access. However, the remote access mode is currently not supported in TSIM. The bootloader can also be bypassed altogether to boot the GR716A directly from external SPI memory, SRAM or PROM (without going through the bootloader first).

GR716A samples various signals on reset and populates the bootstrap register with the result. TSIM does not simulate this sampling. Instead the user can set the value of the bootstrap register with the `-bootstrap` option. For example, to boot using an application software (ASW) image residing in external PROM, start TSIM with `-bootstrap 0x0000c00a`, load the software image with **load** *`image`*, and simulation with **boot** to start execution from the Boot ROM from a reset state. The default bootstrap value is 0x0000c004, which makes execution continue in SRAM, assuming there is already something there to be executed.

The following are examples of different bootstrap values that can be used. This is not an exhaustive list. See the GR716A Data Sheet and User's manual for details.

Table 8.2. Boot methods and example bootstrap values

Boot type	Bootstrap value	Notes
External SPI memory boot	0x0000c000	Execution continues directly in SPI memory
External SRAM boot	0x0000c004	Execution continues directly in SRAM
External PROM boot	0x0000c008	Execution continues directly in PROM
External SPI memory ASW boot	0x0000c002	Extracts ASW image from SPI memory
External SRAM ASW boot	0x0000c006	Extracts ASW image from SRAM
External PROM ASW boot	0x0000c00a	Extracts ASW image from PROM
External I^2C memory ASW boot	0x0000c00e	Not supported without user model of I^2C.
Bypass directly to SPI memory	0x1000c000	Execution starts from SPI memory
Bypass directly to SRAM	0x1000c004	Execution starts from SRAM
Bypass directly to PROM	0x1000c008	Execution starts from PROM
Remote access		Currently not supported by TSIM

Note: if the boot sequence fails the boot software will potentially get stuck in a loop and never reach the main application.

8.2. Dummy registers

The following GR716A register areas are in TSIM currently implemented as dummy registers. They can be written to without effect and read from with value 0.

Table 8.3. Dummy register areas in the GR716A model

Address	Name
0x80001000 - 0x800010ff	DLRAM config
0x80006000 - 0x800060ff	Clock gating unit 0
0x80007000 - 0x800070ff	Clock gating unit 1
0x8000b000 - 0x8000b0ff	ILRAM config
0x8000d000 - 0x8000d0ff	IOMUX config
0x8010c000 - 0x8010c0ff	Brown-Out detection control registers
0x8010d000 - 0x8010d0ff	PLL control registers
0x80307000 - 0x803070ff	NVRAM config

8.3. DAC

TSIM GR716A provides a DAC interface. To connect to TSIM's internal DAC model use `tsim_register_dac_model(dac_input, index)` where `dac_input` is a pointer to a `struct dac_input` (see below), and `index` is the index of the DAC controller to connect to. The `struct dac_input` can be found in `dac_input.h`.

See Chapter 5 for further details on how to connect the user model.

```
struct dac_input {
    void (*dac_output)(double value);
};
```

Table 8.4. struct dac_input members

Parameter	Description
dac_output	Callback set by the user. Will be called each time the DAC controllers output value is changed. `value` is the DAC output value.

9. GR716B emulation

To emulate the GR716B chip the `-gr716b` option should be used. When using the GR716 only release, TSIM only simulates GR716B and GR716A.

Table 9.1. Simulation models for GR716B

Core	Notes and model limitations
AHBROM	GR716B Boot ROM. See Section 9.1.
AHBSTAT	Only the first AHB Status Register for the main AMBA bus supported. Status register bits ME, FW, CF and AF not supported.
APBUART	Transmitter shift register empty interrupt, delayed interrupt and using two stop bits currently not supported.
LRAM	Configuration registers are implemented as dummy registers, see Section 9.2.
FTMCTRL	EDAC not supported.
GPTIMER	No Watchdog support. Control register bits EV, ES and EE not supported.
GRCANFD	See Chapter 15 for details about the CAN bus. CANOpen not supported. Checkpointing currently not supported. ABORT bit in the config register is always 1.
GRGPIO	See Chapter 18. Pulse sampler and Pulse sequencer are not currently supported. Atomic operations supported.
GRGPREG	Only bootstrap register implemented.
SpaceWire router	See Chapter 22 and Section 9.5.
IRQMP	Watchdog control and Error mode status register currently not implemented.
LEON3FT	No FT features are modelled. ZeroJitter, Alternative Window Pointer and REX ISA not currently supported. Register window partitioning is supported.
SPICTRL	See Chapter 23. Automatic Slave select, ThreeWire mode and Slave mode are not currently supported.
SPIMCTRL	GR716A version. See Chapter 24. EDAC not supported.
GRETH	Currently modelled as a GRETH core, not a GRETH_GBIT core. See Chapter 17. Checkpointing currently not supported.
DAC	GR716A version. See Section 9.3. Checkpointing currently not supported.
Real-Time Accelerator (RTA)	See Section 9.4.

GR716B has tightly coupled dual-port local data and instruction RAM. GR716B does not have cache memories. Some register areas for devices that are not emulated are implemented as dummy registers. See Section 9.2. TSIM does not emulate the DSU, L3STAT and AHBTRACE cores but provide a lot of corresponding functionality and information via TSIM commands instead.

9.1. GR716B Boot ROM

In addition to running RAM images directly from memory using **load** and **run**, TSIM can simulate a cold start going through the bootloader in the GR716B Boot ROM, with its different boot possibilities, or bypassing the Boot ROM, booting directly from a different source. The **boot** command is used to start simulating a cold start.

The bootloader in the GR716B Boot ROM supports multiple boot sources. Booting from external SRAM, external PROM and external SPI memory is supported. There is currently no built-in model for the I^2C controller. Therefore, to support booting an image read over from I^2C, a user model for the I^2C controller and bus is needed. The bootloader can also set up the GR716B for remote access. However, the remote access mode is currently not supported in TSIM. The bootloader can also be bypassed altogether to boot the GR716B directly from external SPI memory, SRAM or PROM (without going through the bootloader first).

GR716B samples various signals on reset and populates the bootstrap register with the result. TSIM does not simulate this sampling. Instead the user can set the value of the bootstrap register with the `-bootstrap` option.

For example, to boot using an application software (ASW) image residing in external PROM, start TSIM with `-bootstrap 0x0000c00a`, load the software image with **load** *`image`*, and simulation with **boot** to start execution from the Boot ROM from a reset state. The default bootstrap value is 0x0000c004, which makes execution continue in SRAM, assuming there is already something there to be executed.

The following are examples of different bootstrap values that can be used. This is not an exhaustive list. See the GR716B Data Sheet and User's manual for details.

Table 9.2. Boot methods and example bootstrap values

Boot type	Bootstrap value	Notes
External SPI memory boot	0x0000c000	Execution continues directly in SPI memory
External SRAM boot	0x0000c004	Execution continues directly in SRAM
External PROM boot	0x0000c008	Execution continues directly in PROM
External SPI memory ASW boot	0x0000c002	Extracts ASW image from SPI memory
External SRAM ASW boot	0x0000c006	Extracts ASW image from SRAM
External PROM ASW boot	0x0000c00a	Extracts ASW image from PROM
External I^2C memory ASW boot	0x0000c00e	Not supported without user model of I^2C.
Bypass directly to SPI memory	0x1000c000	Execution starts from SPI memory
Bypass directly to SRAM	0x1000c004	Execution starts from SRAM
Bypass directly to PROM	0x1000c008	Execution starts from PROM
Remote access		Currently not supported by TSIM

Note: if the boot sequence fails the boot software will potentially get stuck in a loop and never reach the main application.

9.2. Dummy registers

The following GR716B register areas are in TSIM currently implemented as dummy registers. They can be written to without effect and read from with value 0.

Table 9.3. Dummy register areas in the GR716B model

Address	Name
0x80001000 - 0x800010ff	DLRAM config
0x80006000 - 0x800060ff	Clock gating unit 0
0x80007000 - 0x800070ff	Clock gating unit 1
0x8000b000 - 0x8000b0ff	ILRAM config
0x8000d000 - 0x8000d0ff	IOMUX config
0x8010c000 - 0x8010c0ff	Brown-Out detection control registers
0x8010d000 - 0x8010d0ff	PLL control registers
0x80307000 - 0x803070ff	NVRAM config

9.3. DAC

TSIM GR716B provides a DAC interface. To connect to TSIM's internal DAC model use `tsim_register_dac_model(dac_input, index)` where `dac_input` is a pointer to a `struct dac_input` (see below), and `index` is the index of the DAC controller to connect to. The `struct dac_input` can be found in `dac_input.h`.

See Chapter 5 for further details on how to connect the user model.

```
struct dac_input {
    void (*dac_output)(double value);
```

```
};
```

Table 9.4. struct dac_input members

Parameter	Description
dac_output	Callback set by the user. Will be called each time the DAC controllers output value is changed. `value` is the DAC output value.

9.4. Real-Time Accelerator (RTA)

The two RTA present in the GR716B can be emulated by the user who can make their own loadable module to emulate some of the RTA functionality, such as the mailbox communication. TSIM GR716B comes with a simple example on how this can be done as a starting point. The example module is meant to be used with the RTA example bundled with BCC2.

See Chapter 5 for further details on how to connect the user model.

9.5. SpaceWire Router

For GR716B TSIM emulates the GR716B scaled back version of the GRLIB core SpaceWire Router. Having only one AMBA port and two SpaceWire ports and limited routing possibility. See See Chapter 22. for more information.

10. GR740 emulation

When starting tsim3-leon4 TSIM emulates the GR740 by default.

Table 10.1. Simulation models for GR740

Core	Notes and model limitations
LEON4	No FT features are modelled. Dynamically configurable L1 cache replacement policy not yet supported.
GRFPU	Does not simulate the possibility of multiple outstanding floating point operations.
AHBSTAT	Only the AHB Status Register for the main AMBA bus supported
APBUART	Transmitter shift register empty interrupt, delayed interrupt and using two stop bits currently not supported.
FTMCTRL	PROM and I/O controller. EDAC and write lead out cycles not supported.
GPTIMER	No Watchdog support. Control register bits EV, ES and EE not supported.
GRCAN	See Chapter 15 for details about the CAN bus. Checkpointing currently not supported.
GRETH	Currently modelled as a GRETH core, not a GRETH_GBIT core. See Chapter 17. Checkpointing currently not supported.
GRGPIO	See Chapter 18. Pulse sampler and Pulse sequencer are not currently supported.
IRQ(A)MP	Watchdog control and error mode status registers are currently not implemented.
L2 Cache	See Section 4.4.3.
SDCTRL	Timings based on a CPU frequency of 250 or 50 MHz and a memory frequency of either 50 or 100 MHz. No EDAC is supported.
SpaceWire router	See Chapter 22. For backwards compatibility it is possible to disable the router with the `-gr740_no_spwrtr` option. This will disable the router leaving only 4x GRSPW2 cores as AMBA ports
GRIOMMU	See Section 4.4.7.
SPICTRL	See Chapter 23. Automatic Slave select, ThreeWire mode and Slave mode are not currently supported.

10.1. Dummy registers

The following GR740 register areas are in TSIM currently implemented as dummy registers. They can be written to without effect and read from with value 0.

Table 10.2. Dummy register areas in the GR740 model

Address	Name
0xffa04000 - 0xffa040ff	Clock gating unit
0xffa09000 - 0xffa090ff	Register for bootstrap signals
0xffa0b000 - 0xffa0b0ff	General purpose register bank

11. UT699 emulation

To emulate the UT699 chip the `-ut699` option should be used. That sets up parameters for core TSIM to match UT699 and sets snooping as non-functional.

The following table lists which cores in the UT699 are modelled by TSIM or not. The table contains some notes of some unsupported features for otherwise supported cores, but is not necessarily exhaustive in this respect.

Table 11.1. Simulation models for UT699

Core	Status	Notes
LEON3FT	Supported by core TSIM3	No FT features are modelled.
GRFPU	Supported by core TSIM3	Does not simulate the possibility of multiple outstanding floating point operations.
AHBSTAT	Supported by core TSIM3	
APBCTRL	Supported by core TSIM3	
APBUART	Supported by core TSIM3	
FTMCTRL	Supported by core TSIM3	No FT features are modelled
GPTIMER	Supported by core TSIM3	No Watchdog support.
IRQMP	Supported by core TSIM3	
CAN_OC	Supported by core TSIM3	See Chapter 16. Checkpointing currently not supported.
GRETH	Supported by core TSIM3	See Chapter 17. Checkpointing currently not supported.
GRGPIO	Supported by core TSIM3	See Chapter 18.
GRPCI	Supported by core TSIM3	Including DMA controller. See Chapter 19. Checkpointing currently not supported.
GRSPW	Supported by core TSIM3	See Chapter 20. Checkpointing currently not supported.
CLKGATE	Not supported	
AHBJTAG	Not supported	Debug link
AHBUART	Not supported	Debug link
DSU3	Not supported	Debug unit

TSIM supports running user defined models for unsupported cores.

12. UT699E emulation

To emulate the UT699E chip the `-ut699e` option should be used. That sets up parameters for core TSIM to match UT699E.

The following table lists which cores in the UT699E are modelled by TSIM or not. The table contains some notes of some unsupported features for otherwise supported cores, but is not necessarily exhaustive in this respect.

Table 12.1. Simulation models for UT699E

Core	Status	Notes
LEON3FT	Supported by core TSIM3	No FT features are modelled.
GRFPU	Supported by core TSIM3	Does not simulate the possibility of multiple outstanding floating point operations.
AHBSTAT	Supported by core TSIM3	
APBCTRL	Supported by core TSIM3	
APBUART	Supported by core TSIM3	
FTMCTRL	Supported by core TSIM3	No FT features are modelled
GPTIMER	Supported by core TSIM3	No Watchdog support.
IRQMP	Supported by core TSIM3	
CAN_OC	Supported by core TSIM3	See Chapter 16. Checkpointing currently not supported.
GRETH	Supported by core TSIM3	See Chapter 17. Checkpointing currently not supported.
GRGPIO	Supported by core TSIM3	See Chapter 18.
GRPCI	Supported by core TSIM3	Including DMA controller. See Chapter 19. Checkpointing currently not supported.
GRSPW2	Supported by core TSIM3	See Chapter 21. Checkpointing currently not supported.
CLKGATE	Not supported	
AHBJTAG	Not supported	Debug link
AHBUART	Not supported	Debug link
DSU3	Not supported	Debug unit

TSIM supports running user defined models for unsupported cores.

13. UT700 emulation

To emulate the UT700 chip the `-ut700` option should be used. That sets up parameters for core TSIM to match UT700.

The following table lists which cores in the UT700 are modelled by TSIM or not. The table contains some notes of some unsupported features for otherwise supported cores, but is not necessarily exhaustive in this respect.

Table 13.1. Simulation models for UT700

Core	Status	Notes
LEON3FT	Supported by core TSIM3	No FT features are modelled.
GRFPU	Supported by core TSIM3	Does not simulate the possibility of multiple outstanding floating point operations.
AHBSTAT	Supported by core TSIM3	
APBCTRL	Supported by core TSIM3	
APBUART	Supported by core TSIM3	
FTMCTRL	Supported by core TSIM3	No FT features are modelled
GPTIMER	Supported by core TSIM3	No Watchdog support.
IRQMP	Supported by core TSIM3	
CAN_OC	Supported by core TSIM3	See Chapter 16. Checkpointing currently not supported.
GRETH	Supported by core TSIM3	See Chapter 17. Checkpointing currently not supported.
GRGPIO	Supported by core TSIM3	See Chapter 18.
GRPCI	Supported by core TSIM3	Including DMA controller. See Chapter 19. Checkpointing currently not supported.
GRSPW2	Supported by core TSIM3	See Chapter 21. Checkpointing currently not supported.
SPICTRL	Supported by core TSIM3	See Chapter 23.
CLKGATE	Not supported	
GR1553B	Not supported	
GRTC	Not supported	
GRTM	Not supported	
AHBJTAG	Not supported	Debug link
AHBUART	Not supported	Debug link
DSU3	Not supported	Debug unit

TSIM supports running user defined models for unsupported cores.

14. AT697 emulation

To emulate the AT697E chip the `-at697e` option should be used. That sets up parameters for core TSIM3 to match AT697E and enables simulation of the AT697 PCI interface.

The following table lists which cores in the AT697 are modelled by TSIM or not. The table contains some notes of some unsupported features for otherwise supported cores, but is not necessarily exhaustive in this respect. See Chapter 4 for details on the builtin simulation models and Chapter 25 for the PCI model.

Table 14.1. Simulation models for AT697

Core	Status	Notes
LEON2FT	Supported by core TSIM3	No FT features are modelled.
FPU	Supported by core TSIM3	
LEON2 system registers	Supported by core TSIM3	
Interrupt controller	Supported by core TSIM3	
Memory controller	Supported by core TSIM3	No FT features are modelled
UART	Supported by core TSIM3	
PCI	Supported by core TSIM3	See Chapter 25. Checkpointing currently not supported.
I/O port	Not supported	Easily modelled in user module
JTAG	Not supported	Debug link
Debug UART	Not supported	Debug link
DSU	Not supported	Debug unit

15. GRCAN and GRCANFD

Each GRCAN or GRCANFD core is connected to two CAN buses (with the possibility to choose which bus to be active on for each GRCAN core). Default values for the bus index is 0 for the main bus and 1 for the secondary bus, making several GRCAN cores connected together on bus 0 and 1 for the main and secondary bus respectively. TSIM models these buses according to the Section 15.4 API.

See the `examples/test` directory for an example GRCAN test program. This test program can be used together with the example can node found in `examples/input`.

15.1. Start up options

GRCAN core start up options

`-grcanX_bus0 index`
: Sets the index of the main CAN bus for GRCAN core X.

`-grcanX_bus1 index`
: Sets the index of the secondary CAN bus for GRCAN core X.

X in the above commands is the index of the core.

15.2. Commands

GRCAN Commands

grcanX_dbg [*flag*|**all**|**clean**|**list**]
: Toggle specific flag, set all, clear all, or list debug flags for the given GRCAN core. See Section 15.3 for a list of debug flags.

X in the above commands is the index of the core.

15.3. Debug flags

The following debug flags and debug subcommands are available for the GRCAN cores. The *CAN_** flags can be used with the **grcanX_dbg** command to toggle individual flags for individual GRCAN cores. The subcommands can be used with the **grcanX_dbg** command to change and list the settings of all flags for individual GRCAN cores.

Table 15.1. GRCAN debug flags

Flag	Trace
CAN_ACC	GRCAN register accesses
CAN_RX	GRCAN received messages
CAN_TX	GRCAN transmitted messages
CAN_IRQ	GRCAN interrupts
all	Set all debug flags for the core
clean	Set none of the debug flags for the core
list	List the current setting of the debug flags for the core

15.4. CAN interface

Currently, this CAN bus model is only used for GRCAN. For emulation of CAN OC in GR712RC, UT699, UT699E and UT700, see Chapter 16.

15.4.1. Connecting a user CAN model

To connect a custom CAN node to TSIM's internal CAN bus use `tsim_register_can_node(node, canbus_index)` where `node` is a pointer to a `struct can_node` (see below), and `canbus_index` is the index of the bus to connect to. Both `struct can_node` and `struct can_msg` can be found in `canbus_input.h`.

See Chapter 5 for further details on how to connect the user model.

15.4.2. CAN model API

The internal CAN bus is available through `struct canbus_interface *canbus` provided by the init function of `struct can_node` that is called during simulation startup if it has been registered by using the `tsim_register_can_node` function.

The different structs are described below.

```
struct canbus_interface {
    int (*update)(uint32 bus_id);
 };
```

Table 15.2. struct canbus_interface members

Parameter	Description
update	Used to update a bus after a change in node status.

```
struct can_node {
    unsigned int id;
    void (*init)(struct can_node *node, struct canbus_interface *canbus);
    int (*rx_callback)(uint32 bus_id, uint32 sender_id, struct can_node *node, struct can_msg msg);
    void (*tx_callback)(uint32 bus_id, struct can_node *node, uint32 error_flags, int num_acks);
    struct can_msg (*get_message)(struct can_node *node);
    void (*status)(struct can_node *node);
    uint32 *wants_to_send;
    uint32 *bus_off;
    uint32 *error_passive;
    uint32 invisible;
    uint32 disconnect;
    void *priv;
};
```

Table 15.3. struct can_node members

Parameter	Description
id	The nodes id, each node needs an unique id.
init	Callback called by the CAN bus during simulation start up. Provides the module with a canbus_interface, see above. `node` is a pointer to the node being initiated.
rx_callback	Callback called by the CAN bus each time a new message is available. Input parameters described below.
tx_callback	Callback called by the CAN bus each time this node has sent a message. Input parameters described below.
get_message	Callback called by the CAN bus when the bus is free and the node wants to send. Should return a can_msg struct
status	Callback called by the CAN bus when printing status. Can optionally be set to print the nodes status at the same time.
wants_to_send	Pointer set by the user indicating if the node wants to send a message or not. If set get_message will be called each time the CAN bus is free
bus_off	Pointer set by the user indicating if the node is in bus off state.
error_passive	Pointer set by the user indicating if the node is in error passive state. Otherwise it is in error active state
invisible	If set the node is invisible on the bus. It will receive all messages via the rx_callback. But cannot acknowledge or flag errors.
disconnect	If set the node is disconnected from the CAN bus. It will receive no messages and have no impact on the bus.
priv	Pointer to private data. Can be set freely by the user.

If a node wants to send a message it should set `*wants_to_send` to non-zero and call `canbus.update`. Next time the bus is free `get_message` is called and the node should return a can message of type `struct can_msg`. The internal bus model will then collect messages from each node that wants to send and perform arbitration, the winning message will then be sent.

Table 15.4. tx_callback parameters

Parameter	Description
bus_id	Index of the CAN bus which the message was sent on.
node	Pointer to the `struct can_node` that sent the message.
error_flags	If any receiving node forced an error this flag is set.
num_acks	Number of nodes correctly acked the message.

When a message is sent the `rx_callback` is called for each connected node. To acknowledge this message normally this function should return 0, if an error is detected return an error flag. The `msg.flags` property can be used to check if an error should be forced.

Table 15.5. rx_callback parameters

Parameter	Description
bus_id	Index of the CAN bus which the message was received on.
sender_id	ID of the sender node.
node	Pointer to the `struct can_node` that is receiving the message.
msg	A `struct can_msg` containing the message.

If a node enters bus off mode or error passive mode, the corresponding property should be set by the user. While the node is in bus off mode it will not be able to send messages but `rx_callback` will still be called to receive messages, it is up to the user model if it wants to discard the message or not, note that if in bus off mode the return value will be ignored.

When a node is done sending messages `*wants_to_send` should be set to zero.
If arbitration is won and all nodes have received the message, the winning nodes `tx_callback` is called. If any error was detected the `error_flags` is set. If arbitration is won the node will not receive it's own message through `rx_callback`. The `priv` pointer is unused by the CAN bus and can be set freely by the user. It can, for example, be used to differentiate between different nodes using the same callback functions or used to add extra properties to the node.

```
struct can_msg {
   uint32 *data;

   uint32 flags;
   uint32 nominal_bitrate;
   uint32 fd_bitrate;
};
```

Table 15.6. struct can_msg members

Parameter	Description
data	Pointer to the CAN message data.
flags	When transmitting, errors can be forced by this flag. When receiving this indicates if errors has been found
nominal_bitrate	The nominal bit-rate which the message will be sent. Measured in clock cycles per bit.
fd_bitrate	The CAN-FD data bit-rate. Should be left as zero when transmitting ordinary CAN-Messages. Measured in clock cycles per bit.

See the `examples/input` directory for an example can node implementation. The example demonstrates how to set up a basic can node that will receive and acknowledge messages.

15.4.3. Error injections

Errors can be injected by the `uint32 flags` property of `struct can_msg`. When transmitting a message the flags can be set to let a receiver know that it should force an error. When receiving the `rx_callback` can return non-zero to let the transmitter know an error has occurred.

Bit 0-4 and bit 31 is reserved and defined below. Bit 0-4 indicates one of the standard errors, defined by the ISO standard 11898-1:2015 (2nd edition), has occurred. Bit 31 indicates if the error was flagged by an error passive node. Remaining bits can be used freely for system specific flags by the users.

Table 15.7. Error flag definitions

Bit	Description
0	Ack error.
1	Form error.
2	CRC error.
3	Stuff error.
4	Bit error
31	Error flagged by error passive node.

15.4.4. Commands

CAN bus Commands

canbusX_status
: Prints the status information on the given CAN bus. Note that this is only used for systems with one or more GRCAN devices, not for CAN_OC.

X in the above commands is the index of the bus.

15.4.5. Debug status

To display the status of an internal CAN bus use the **canbusX_status** command. This command will print the status of each connected node, as well as call the optional status command that can be provided by a user model.

15.4.6. Current limitations

Arbitration loss is not reported to the node, it has to check it manually when receiving by either comparing its own message with the received one.

Each node has to have an unique ID.

16. CAN_OC interface

The UT699, UT699E, UT700 and GR712RC chips contains CAN_OC cores which models the CAN_OC cores available in the chip. For core details and register specification please see the manual for each emulated chip.

16.1. Start up options

CAN_OC core start up options

`-can_ocX_connect host:port`
: Connect CAN_OC core X to packet server to specified server and TCP port.

`-can_ocX_server port`
: Open a packet server for CAN_OC core X on specified TCP port.

`-can_ocX_ack [0|1]`
: Enables waiting for an acknowledgement packet on transmission for CAN_OC core X.

X in the above commands is the index of the core.

16.2. Commands

CAN OC Commands

can_ocX_connect *`host:[port]`*
: Connect CAN_OC core X to packet server to specified server and TCP port.

can_ocX_server *`port`*
: Open a packet server for CAN_OC core X on specified TCP port.

can_ocX_ack *<0|1>*
: Specifies whether the CAN_OC core will wait for a acknowledgement packet on transmission. This command should only be issued after a connection has been established.

can_ocX_status
: Prints out status information for the CAN_OC core.

can_ocX_dbg [*`flag`*|**all**|**clean**|**list**]
: Toggle, set, clear, list debug flags for the CAN_OC core.

X in the above commands is the index of the core.

16.3. Debug flags

The following debug flags and debug subcommands are available for CAN interfaces. The *`GAISLER_CAN_OC_*`* flags can be used with the **can_ocX_dbg** command to toggle individual flags for individual CAN_OC cores and with the **dbgon** command to toggle individual flags for all CAN_OC cores. The subcommands can be used with the **can_ocX_dbg** command to change and list the settings of all flags for individual CAN_OC cores.

Table 16.1. CAN debug flags

Flag	Trace
GAISLER_CAN_OC_ACC	CAN_OC register accesses
GAISLER_CAN_OC_RXPACKET	CAN_OC received messages
GAISLER_CAN_OC_TXPACKET	CAN_OC transmitted messages
GAISLER_CAN_OC_ACK	CAN_OC acknowledgements
GAISLER_CAN_OC_IRQ	CAN_OC interrupts
all	Set all debug flags for the core
clean	Set none of the debug flags for the core
list	List the current setting of the debug flags for the core

16.4. Packet server

Each CAN_OC core can be configured independently as a packet server or client using either `-can_ocX_server` or `-can_ocX_connect`. When acting as a server the core can only accept a single connection.

A connection should be set up before starting simulation for the first time, and must not be broken after that. Restarting the simulation will not tear down the connection, nor emptying any socket buffers. The socket based interface does not support any signalling of restart of the simulation. To ensure a clean restart of simulation when using this interface, restarting TSIM entirely and reconnecting socket interfaces is advisable.

16.5. CAN packet server protocol

The protocol used to communicate with the packet server is described below. Four different types of packets are defined according to the table below.

Table 16.2. CAN packet types

Type	Value
Message	0x00
Error counter	0xFD
Acknowledge	0xFE
Acknowledge config	0xFF

16.5.1. CAN message packet format

Used to send and receive CAN messages.

31 0

0x0	LENGTH
31:0	LENGTH, specifies the length of the rest of the packet

CAN message

Byte #	Description	Bits (MSB-LSB)							
		7	6	5	4	3	2	1	0
4	Protocol ID = 0	Prot ID 7-0							
5	Control	FF	RTR	-	-	DLC (max 8 bytes)			
6-9	ID (32 bit word in network byte order)	ID 10-0 (bits 31 - 11 ignored for standard frame format) ID 28-0 (bits 31-29 ignored for extended frame format)							
10-17	Data byte 1 - DLC	Data byte *n* 7-0							

Figure 16.1. CAN message packet format

16.5.2. Error counter packet format

Used to write the RX and TX error counter of the modelled CAN interface.

31 0

0x0	LENGTH
31:0	LENGTH, specifies the length of the rest of the packet

Error counter packet

Byte #	Field	Description
4	Packet type	Type of packet, 0xFD for error counter packets
5	Register	0 - RX error counter, 1 - TX error counter
6	Value	Value to write to error counter

Figure 16.2. Error counter packet format

16.5.3. Acknowledge packet format

If the acknowledge function has been enabled through the start up option or command the CAN interface will wait for an acknowledge packet each time it transmits a message. To enable the CAN receiver to send acknowledge packets (either NAK or ACK) an acknowledge configuration packet must be sent. This is done automatically by the CAN interface when **can_ocX_ack** is issued.

31	0
0x0	LENGTH
31:0	LENGTH, specifies the length of the rest of the packet

Acknowledge packet

Byte #	Field	Description
4	Packet type	Type of packet, 0xFE for acknowledge packets
5	Ack payload	0 - No acknowledge, 1 - Acknowledge

Figure 16.3. Acknowledge packet format

16.5.4. Acknowledge packet format

This packet is used for enabling/disabling the transmission of acknowledge packets and their payload (ACK or NAK) by the CAN receiver. The CAN transmitter will always wait for an acknowledge if started with `can_ocX_ack` or if the **can_ocX_ack** command has been issued.

31	0
0x0	LENGTH
31:0	LENGTH, specifies the length of the rest of the packet

Acknowledge configuration packet

Byte #	Field	Description	
4	Packet type	Type of packet, 0xFF for acknowledge configuration packets	
5	Ack configuration	bit 0	Unused
		bit 1	Ack packet enable, 1 - enabled, 0 - disabled
		bit 2	Set ack packet payload, 1 - ACK, 0 - NAK

Figure 16.4. Acknowledge configuration packet format

17. 10/100 Mbps Ethernet Media Access Controller interface

The Ethernet core simulation model is designed to functionally model the 10/100 Ethernet MAC available in UT699, UT699E, UT700 and GR712RC. For core details and register specification please see the chip manual.

The following features are supported:

- Direct Memory Access
- Interrupts

17.1. Start up options

Ethernet core start up options

`-grethX_connect host[:port]`
: Connect Ethernet core to a packet server at the specified host and port. Default port is 2224.

`-grethX_mac X:X:X:X:X:X`
: Set MAC address of the Ethernet core.

17.2. Commands

GRETH Commands

grethX_dbg [*flag*|**all**|**clean**|**list**]
: Toggle specific flag, set all, clear all, or list debug flags for the given GRETH core. See Section 17.3 for a list of debug flags.

grethX_status
: Prints the status of greth core X.

grethX_connect *[ip[:port]]*
: Connect to packet server at given IP address and optional port. Default port is 2224. If no IP address is specified, the default is localhost.

grethX_ping [*ip*]
: Send an ARP request followed by a ping packet (regarding/addressed to the given IP address) to the GRETH and present any ping reply send by the GRETH. If no IP address is specified the address 192.168.0.80 is used by default.

grethX_dump *file*
: Dump packets to Ethereal readable *file*.

grethX_reconnect *<0|1>*
: Turn GRETH autoreconnect on or off.

X in the above commands is the index of the core.

17.3. Debug flags

The following debug flags are available for the Ethernet interface. Use the them in conjunction with the **dbgon** command to enable different levels of debug information.

Table 17.1. Ethernet debug flags

Flag	Trace
GAISLER_GRETH_ACC	GRETH accesses
GAISLER_GRETH_L1	GRETH accesses verbose
GAISLER_GRETH_TX	GRETH transmissions
GAISLER_GRETH_RX	GRETH reception
GAISLER_GRETH_RXPACKET	GRETH received packets
GAISLER_GRETH_RXCTRL	GRETH RX packet server protocol
GAISLER_GRETH_RXBDCTRL	GRETH RX buffer descriptors DMA
GAISLER_GRETH_TXCTRL	GRETH TX packet server protocol

Flag	Trace
GAISLER_GRETH_TXPACKET	GRETH transmitted packets
GAISLER_GRETH_IRQ	GRETH interrupts

17.4. Ethernet packet server

The simulation model relies on a packet server to receive and transmit the Ethernet packets. The packet server should open a TCP socket which the module can connect to. The Ethernet core is connected to a packet server using the `-grethX_connect` start-up parameter or using the **grethX_connect** command.

A connection should be set up before starting simulation for the first time, and must not be broken after that. Restarting the simulation will not tear down the connection, nor emptying any socket buffers. The socket based interface does not support any signalling of restart of the simulation. To ensure a clean restart of simulation when using this interface, restarting TSIM entirely and reconnecting socket interfaces is advisable.

An example implementation of a packet server, named `greth_config`, is included in TSIM distribution. It uses the TUN/TAP interface in Linux, or the WinPcap library on Windows, to connect the GRETH core to a physical Ethernet LAN. This makes it easy to connect the simulated GRETH core to real hardware. It can provide a throughput in the order of magnitude of 500 to 1000 KiB/sec. See its distributed README for usage instructions.

17.5. Ethernet packet server protocol

Ethernet data packets have the following format. Note that each packet is prepended with a one word length field indicating the length of the packet to come (including its header).

Packet length at offset 0x0:

31 0

LEN

31:0	LEN	Length of rest of packet: 4 + number of data bytes

Header at offset 0x4:

31 16 15 8 7 5 4 0

R	IPID	TYPE	*R*

31:16	*R*	Reserved for future use. Must be set to 0.
15:8	IPID	IP core ID: 1 for Ethernet
7:5	TYPE	Packet type: 0 for data packets
4:0	*R*	Reserved for future use. Must be set to 0.

Offset 0x8: The rest of the packet is the encapsulated Ethernet packet

Figure 17.1. Ethernet data packet

18. GPIO interface

18.1. Connecting a user GPIO model

To register a GPIO user module, call `tsim_register_gpio_module(gpio_input, index)` from an input modules init function. Here *`gpio_input`* is a pointer to a gpio_input struct, and *`index`* is the index of the GPIO core to register on. See Chapter 5 for further details on how to connect the user model.

18.2. GPIO model API

The structure `struct gpio_input` models the GPIO pins. It is defined as:

```
/* GPIO input provider */
struct gpio_input {
    int index;
    int (*gpioout)(struct gpio_input *ctrl, unsigned int dir, unsigned int output);
    int (*gpioin) (struct gpio_input *ctrl, unsigned int in);
    void (*gpioinit)(struct gpio_input *ctrl);
    void (*gpioreset)(struct gpio_input *ctrl);
    void (*print_status)(struct gpio_input *ctrl);

    void* priv;
};
```

The `gpioout` callback should be set by the user module at startup. The `gpioin` callback is set by tsim. The `gpioout` callback is called by the module whenever a GPIO output pin changes. The `gpioin` callback is called by the user module when the input pins should change. Typically the user module would register an event handler at a certain time offset and call `gpioin` from within the event handler. The `gpioinit` callback is called during simulator startup and the `gpioreset` is called each time TSIM resets. Optionally the `print_status` callback can be set to print user model status. The `priv` parameter can be set freely by the user.

Table 18.1. gpioout callback parameters

Parameter	Description
dir	Bit x of dir indicates that the grgpio core drives output on line x when 1 and that it does not when it is 0.
out	The values of the output pins

Table 18.2. gpioin callback parameters

Parameter	Description
in	The input pin values

The return value of `gpioin/gpioout` is ignored.

See the `examples/input` directory for an example module implementation. See the `examples/test` directory for an example test program.

18.3. Commands

GPIO Commands

gpioX_status
: Print status for the GPIO core.

gpioX_dbg [*`flag`*|**all**|**clean**|**list**]
: Toggle specific flag, set all, clear all, or list debug flags for the given GPIO core. See Section 18.4 for a list of debug flags.

X in the above commands is the index of the core.

18.4. Debug flags

The following debug flags and debug subcommands are available for GPIO interfaces. The *`GAISLER_GPIO_*`* flags can be used with the **gpioX_dbg** command to toggle individual flags for individual GPIO cores and with the

dbgon command to toggle individual flags for all GPIO cores. The subcommands can be used with the **gpioX_dbg** command to change and list the settings of all flags for individual GPIO cores.

Table 18.3. GPIO debug flags

Flag/subcommand	Trace
GAISLER_GPIO_ACC	GPIO register accesses
GAISLER_GPIO_IRQ	GPIO interrupts
all	Set all GPIO debug flags for the core
clean	Set none of the GPIO debug flags for the core
list	List the current setting of the debug flags for the core

19. GRPCI, PCI initiator/target interface

TSIM models the GRPCI cores available in UT699, UT699E and UT700. For core details and register specification please see the manual for each chip.

19.1. Commands

PCI Commands

grpciX_status
: Print status for PCI core X

grpciX_dbg
: Toggle specific flag, set all, clear all, or list debug flags for the given grpci core. See Section 19.2 for a list of debug flags.

X in the above commands is the index of the core.

19.2. Debug flags

The following debug flags are available for the PCI interface. Use them in conjunction with the **grpciX_dbg** command to enable different levels of debug information.

Table 19.1. PCI interface debug flags

Flag	Trace
GAISLER_GRPCI_ACC	AHB accesses to/from PCI core
GAISLER_GRPCI_REGACC	GRPCI APB register accesses
GAISLER_GRPCI_DMA_REGACC	PCIDMA APB register accesses
GAISLER_GRPCI_DMA	GRPCI DMA accesses on the AHB bus
GAISLER_GRPCI_TARGET_ACC	GRPCI target accesses
GAISLER_GRPCI_INIT	Print summary on startup

19.3. PCI bus model API

To register a GRPCI module call `tsim_register_grpci_module(struct grpci_input *inp, int index);` from an input modules init function. Here *`inp`* is a pointer to a grpci_input struct and *`index`* is the index of the GRPCI controller to register on. See Chapter 5 for further details on how to connect the user model. The `struct grpci_input` is defined in `grpci_input.h` as:

```
struct grpci_input {
    int (*acc)(struct grpci_input *ctrl,
               int cmd,
               unsigned int addr,
               unsigned int wsize,
               unsigned int *data,
               unsigned int *abort,
               unsigned int *ws);

    void (*grpci_init)(struct grpci_interface *grpciif);
};
```

The acc callback should be set by the PCI user module at startup. It is called by the the model whenever the PCI core reads/writes as a PCI bus master.

Table 19.2. acc callback parameters

Parameter	Description
cmd	Command to execute, see Section 19.3.1 details.
addr	PCI address.
wsize	0: 8-bit access 1: 16-bit access, 2: 32-bit access. Is always 2 for read accesses.

Parameter	Description
data	Data buffer. The user module should return the read data in `*data` for read commands or write the data in `*data` for write commands.
abort	Set `*abort` to 1 to generate target abort, or 0 otherwise.
ws	Set `*ws` to the number of PCI clocks it takes to complete the transaction.

The return value of acc determines if the transaction terminates successfully (0 or GRPCI_ACC_OK) or with master abort (1 or GRPCI_ACC_MASTER_ABORT).

The grpci_init callback should be set by the PCI user module at startup. It is called by TSIM at simulator startup.

Table 19.3. grpci_init callback parameters

Parameter	Description
grpciif	Pointer to a `struct grpci_interface`. Should be saved by the module to interface with TSIM's GRPCI model.

The `struct grpci_interface` is defined in `grpci_input.h` as:

```
struct grpci_interface {
    int (*target_acc)(int index,
                      int cmd,
                      unsigned int addr,
                      unsigned int wsize,
                      unsigned int *data,
                      unsigned int *mexc);
};
```

The callback target_acc is installed by the TSIM. The PCI user dynamic library can call this function to initiate an access to the PCI target.

Table 19.4. target_acc parameters

Parameter	Description
index	Index of GRPCI core of the system. Typically, 0 is the only valid index.
cmd	Command to execute, see Section 19.3.1 for details. I/O cycles are not supported by the target.
addr	PCI address. Should always be word aligned for read accesses.
wsize	0: 8-bit access 1: 16-bit access, 2: 32-bit access. Should always be 2 for read accesses.
data	Data buffer. The read data is returned in `*data` for read commands or the data in `*data` is written for write commands.
mexc	The model sets *mexc to 0 if access is successful, or 1 in case of target abort.

If a supported command hits MEMBAR0, MEMBAR1 or CONFIG, target_acc() will return 0. For unsupported commands or other areas, it will return 1.

19.3.1. PCI command table

Table 19.5. PCI command table

Command	Description
0	IRQ acknowledge
1	Special cycle
2	I/O Read
3	I/O Write
4	Reserved

Command	Description
5	Reserved
6	Memory Read
7	Memory Write
8	Reserved
9	Reserved
10	Configuration Read
11	Configuration Write
12	Memory Read Multiple
13	Dual Address Cycle
14	Memory Read Line
15	Memory Write And Invalidate

19.4. Examples

See the PCI files in `examples/input` for header files and an example PCI user module. See example usage in `examples/test`.

20. GRSPW1, SpaceWire interface with RMAP support

The UT699 chip contains 4 GRSPW cores which are modelled in TSIM. For core details and register specification please see the UT699 manual.

The UT699E chip has GRSPW2 cores instead of GRSPW cores. So, for UT699E see Chapter 21 instead.

The following features are supported:

- Transmission and reception of SpaceWire packets
- Interrupts
- RMAP

20.1. Start up options

SpaceWire core start up options

`-grspwX_connect host:port`
: Connect GRSPW core X to packet server at specified server and port.

`-grspwX_server port`
: Open a packet server for core X on specified port.

`-grspw_rxfreq freq`
: Set the RX frequency which is used to calculate receive performance.

`-grspw_txfreq freq`
: Set the TX frequency which is used to calculate transmission performance.

X in the above commands is the index of the core.

20.2. Commands

GRSPW SpaceWire core TSIM commands

grspwX_connect *`host:[port]`*
: Connect GRSPW core *X* to packet server at specified server and TCP port.

grspwX_server *`port`*
: Open a packet server for GRSPW core *X* on specified TCP port.

grspwX_dbg [*`flag`*|**all**|**clean**|**list**]
: Toggle specific flag, set all, clear all, or list debug flags for the given GRSPW core. See Section 20.3 for a list of debug flags.

X in the above commands is the index of the core.

20.3. Debug flags

The following debug flags are available for the SpaceWire interfaces. Use the them in conjunction with the **dbgon** command to enable different levels of debug information.

Table 20.1. SpaceWire debug flags

Flag	Trace
GAISLER_GRSPW_ACC	GRSPW accesses
GAISLER_GRSPW_RXPACKET	GRSPW received packets
GAISLER_GRSPW_RXCTRL	GRSPW rx protocol
GAISLER_GRSPW_TXPACKET	GRSPW transmitted packets
GAISLER_GRSPW_TXCTRL	GRSPW tx protocol

20.4. SpaceWire packet server

Each SpaceWire core can be configured independently as a packet server or client using either `-grspwX_server` or `-grspwX_connect`. TCP sockets are used for establishing the connections. When acting as a server the core can only accept a single connection.

A connection should be set up before starting simulation for the first time, and must not be broken after that. Restarting the simulation will not tear down the connection, nor emptying any socket buffers. The socket based interface does not support any signalling of restart of the simulation. To ensure a clean restart of simulation when using this interface, restarting TSIM entirely and reconnecting socket interfaces is advisable.

For more flexibility, such as custom routing, an external packet server can be implemented using the protocol specified in the following sections. Each core should then be connected to that server.

20.5. SpaceWire packet server protocol

The protocol used to communicate with the packet server is described below. Three different types of packets are defined according to the table below.

Table 20.2. Packet types

Type	Value
Data	0
Time code	1

Note that all packets are prepended by a one word length field which specified the length of the coming packet including the header.

20.5.1. Data packet format

Packet length at offset 0x0:

31	0
LEN	

31:0	LEN	Length of rest of packet: 4 + number of data bytes

Header at offset 0x4:

31	16	15	8	7	5	4	1	0
R		IPID		TYPE		*R*		EE

31:16	*R*	Reserved for future use. Must be set to 0.
15:8	IPID	IP core ID: 0 for SpaceWire
7:5	TYPE	Packet type: 0 for data packets
4:1	*R*	Reserved for future use. Must be set to 0.
0	EE	Error End of Packet. Set when the packet is truncated and terminated by an EEP.

Offset 0x8: The rest of the packet is the encapsulated SpaceWire packet

Figure 20.1. SpaceWire data packet

20.5.2. Time code packet format

Packet length at offset 0x0:

31	0
LEN	

31:0	LEN	Length of rest of packet: 8

Header at offset 0x4:

31	16	15	8	7	5	4	0
R		IPID		TYPE		*R*	

31:16	*R*	Reserved for future use. Must be set to 0.
15:8	IPID	IP core ID: 0 for SpaceWire
7:5	TYPE	Packet type: 1 for time code packets
4:0	*R*	Reserved for future use. Must be set to 0.

Payload at offset 0x8:

31	8	7	6	5	0
R		CT		CN	

31:8	*R*	Reserved for future use. Must be set to 0.
7:6	CT	Time control flags
5:0	CN	Value of time counter

Figure 20.2. SpaceWire time code packet

21. GRSPW2, SpaceWire interface with RMAP support

TSIM models the GRSPW2 cores available in UT699E, UT700, GR712RC and GR716. For core details and register specification please see the manual for each chip.

Supported features include:

- Transmission and reception of SpaceWire packets
- Transmission and reception of Time codes
- RMAP
- Server side link state model
- Link errors
- Link error injection

All GRSPW2 register fields with underlying functionality in UT699E, UT700, GR712RC and GR716 are supported except for:

- The link model is only in `error reset` state or `run` state.
- The RMAP buffer disable (RD) bit in the control register with underlying functionality is not modelled.
- The limitations of the No spill (NS) DMA control register bit as noted in the section on Flow control limitations below.
- No support for RX/TX of SpW interrupt codes (GR716).
- No support for SpaceWire Plug & Play via RMAP access (GR716).
- The port loopback (Loop) bit in the control register with underlying functionality is not modelled (UT700).

21.1. Start up options

SpaceWire core start up options

`-grspwX_connect` *`host:port`*
: Connect GRSPW2 core *X* to packet server at specified server and port.

`-grspwX_server` *`port`*
: Open a packet server for core *X* on specified port.

`-grspw_spwfreq` *`freq`*
: Sets the SpaceWire input clock frequency. Combined with the clock divisor register, this determines the startup frequency and TX frequency.

`-grspw_clkdiv` *`value`*
: Sets the reset value for the clock divisor register for all GRSPW2 cores.

`-grspw_max_tx_part_len` *`len`*
: Sets up all GRSPW2 cores to transmit any SpaceWire packet longer than *`len`* in data part packets with no more than *`len`* bytes of data.

`-grspw_endpacket [0|1]`
: Enable (or disable with 0 argument) end marking data part packets. When enabled, the last data part packet of a simulated SpaceWire packet will always be a data part packet with no data and an end marker. This is the default unless simple mode is enabled. When disabled the last data part packet can contain both data and an end marker. This is the default when simple mode is enabled.

`-grspw_simple [0|1]`
: Enable “simple mode” for all GRSPW2 cores. This can be used for backward compatibility with TSIM 2.0.44 and backwards. See the separate section on simple mode for details.

`-grspw_simple_rxfreq` *`freq`*
: Sets the RX frequency in MHz for all GRSPW2 cores to *`freq`*. This is only valid together with the `-grspw_simple` option.

X in the above commands is the index of the core.

21.2. Commands

GRSPW2 SpaceWire core TSIM commands

grspwX_connect *`host:[port]`*
: Connect GRSPW2 core *X* to packet server at specified server and TCP port.

grspwX_server *port*
Open a packet server for GRSPW2 core *X* on specified TCP port.

grspwX_dbg [*flag*|**all**|**clean**|**list**]
Toggle specific flag, set all, clear all, or list debug flags for the given GRSPW2 core. See Section 21.3 for a list of debug flags.

grspwX_status
Print status for GRSPW2 core *X*.

X in the above commands is the index of the core.

21.3. Debug flags

The following debug flags and debug subcommands are available for SpaceWire interfaces. The *GAISLER_GRSPW_** flags can be used with the **grspwX_dbg** command to toggle individual flags for individual SpaceWire cores and with the **dbgon** command to toggle individual flags for all GRSPW2 cores. The subcommands can be used with the **grspwX_dbg** command to change and list the settings of all flags for individual SpaceWire cores.

Table 21.1. SpaceWire debug flags

Flag/subcommand	Trace
GAISLER_GRSPW_ACC	GRSPW accesses
GAISLER_GRSPW_RXPACKET	GRSPW received packets
GAISLER_GRSPW_RXCTRL	GRSPW rx protocol
GAISLER_GRSPW_TXPACKET	GRSPW transmitted packets
GAISLER_GRSPW_TXCTRL	GRSPW tx protocol
GAISLER_GRSPW_RMAP	GRSPW RMAP accesses
GAISLER_GRSPW_RMAPPACKET	GRSPW RMAP packet dumps
GAISLER_GRSPW_RMAPPACKDE	GRSPW RMAP packet decoding
GAISLER_GRSPW_DMAERR	GRSPW DMA errors
GAISLER_GRSPW_LINK	Link changes
GAISLER_GRSPW_PART	TX/RX of GRSPW data part packets
all	Set all debug flags for the core
clean	Set none of the debug flags for the core
list	List the current setting of the debug flags for the core

21.4. SpaceWire packet server

Each SpaceWire core can be configured independently as a packet server or client using either `-grspwX_server` or `-grspwX_connect`. TCP sockets are used for establishing the connections. When acting as a server the core can only accept a single connection.

A connection should be set up before starting simulation for the first time, and must not be broken after that. Restarting the simulation will not tear down the connection, nor emptying any socket buffers. The socket based interface does not support any signalling of restart of the simulation. To ensure a clean restart of simulation when using this interface, restarting TSIM entirely and reconnecting socket interfaces is advisable.

The server side contains a link model that gets control information from the models at each end of the link, determines the link state and communicates frequencies and link errors to the two models at each ends of the link. It also supports error injection via two error injection packet types that can be sent from a custom client. See the the following sections for details.

For more flexibility, such as custom routing, an external packet server can be implemented using the protocol specified in the following sections. Each core should then be connected to that server. That server would also

have to implement a link model that at least reacts to link control packets and sends out link state packets and RX frequency packets.

21.5. SpaceWire packet server protocol

The protocol used to communicate with the packet server is described below. The different types of packets are defined according to the table below.

Table 21.2. Packet types

Type	Value	Direction	Notes
Data part	0	Both	Only when in run state
Time code	1	Both	Only when in run state
Link state	2	Server to client	
Link control	3	Client to server	Must be sent for model to reach run state
RX frequency	4	Server to client	
Error injection	5	Client to server	Optional
Packet error request	6	Client to server	Optional

All packets begin with a 32-bit big endian word length field which specifies the length of the rest of the packet, including header and other fixed fields. For most packet types this length is fixed for the particular type. Apart from the data part packet type, where data follows the header byte-wise, all fields are 32-bit big endian words if not otherwise specified.

All packets received by the GRSPW2 model are handled sequentially, and all packets sent by the GRSPW2 model and the server side link model are supposed to be handled sequentially by the client. SpaceWire packets can be split into multiple data parts, transferred in data part packets. Between such parts other packets such as for time codes, link state changes, link control changes, etc., can be handled. During the simulated time span for the reception of a data part, the receiver will not/should not handle any other packet types. Use the `-grspw_max_tx_part_len` option to set up GRSPW2 model to split up SpaceWire packets into data parts in order for such events to be able to happen during the data stream.

21.5.1. Flow control limitations

Flow control in terms of credit is not modelled between two ends of a link. A transmitter gets no notice if the other end cannot give more credit. If the no-spill bit in the GRSPW2 core is set and an enabled receiving DMA channel has no enabled descriptors, the data part packet will be held on the receiving side until a descriptor is available. Due to the sequential nature of the packet model a link error, time code, etc. will not be handled at all by the GRSPW2 model during this time.

21.5.2. Data part packet format

A SpaceWire packet is represented by one or more data parts. A data part packet represents one such a part. For the data parts of a multi part SpaceWire packet, only the last data part should have an EOP or EEP end marker, i.e. the ***END*** field set to 0 or 1. The other parts should have no end marker, i.e. the ***END*** field set to 2. Each data part is delivered in its entirety or not at all. A link error occurring between data parts on the other hand cuts the SpaceWire packet short and is considered the end of that SpaceWire packet.

A data part packet is sent at the beginning of transmission of the corresponding part of the SpaceWire packet, so that the receiver can start reacting to it as soon as the transmission starts. The GRSPW2 model by default sends a SpaceWire packet in the form of two data part packets. The first data part packet is sent at the beginning of transmission and contains all data but has no end marker. The second data part packet is sent when transmission is completed and has the appropriate end marker set but contains no data. If a user model is not waiting for the end marker packet before responding, the response could arrive before transmission is considered done by the GRSPW2 model. Generation of separate end marker packets can be turned off using the `-grspw_endpacket` option. Splitting up SpaceWire packets into several data containing data part packets can be enabled with the `-grspw_max_tx_part_len` option.

Packet length at offset 0x0:

31	0
LEN	

31:0	LEN	Length of rest of packet: 4 + number of data bytes in the part

Header at offset 0x4:

31	16	15	8	7	5	4	2	1	0
R		IPID		TYPE		*R*		END	

31:16	*R*	Reserved for future use. Must be set to 0.
15:8	IPID	IP core ID: 0 for SpaceWire
7:5	TYPE	Packet type: 0 for data part packets
4:2	*R*	Reserved for future use. Must be set to 0.
1:0	END	End marker: 0: Normal End of Packet, 1: Error End of Packet, 2: No end marker

Offset 0x8: The data bytes of the part starts here

Figure 21.1. SpaceWire data part packet

21.5.3. Time code packet format

Packet length at offset 0x0:

31	0
LEN	

31:0	LEN	Length of rest of packet: 8

Header at offset 0x4:

31	16	15	8	7	5	4	0
R		IPID		TYPE		*R*	

31:16	*R*	Reserved for future use. Must be set to 0.
15:8	IPID	IP core ID: 0 for SpaceWire
7:5	TYPE	Packet type: 1 for time code packets
4:0	*R*	Reserved for future use. Must be set to 0.

Payload at offset 0x8:

31	8	7	6	5	0
R		CT		CN	

31:8	*R*	Reserved for future use. Must be set to 0.
7:6	CT	Time control flags
5:0	CN	Value of time counter

Figure 21.2. SpaceWire time code packet

21.5.4. Link state packet format

Link state packets are sent out by the server side link model when the link state changes. The only states currently simulated are `Error Reset` and `Run`. A link state packet with state `Error Reset` can have the ***ERROR*** field set to an error seen at the receiver. Other link state packets has only `None` in the ***ERROR*** field.

Packet length at offset 0x0:

31	0
LEN	

31:0	LEN	Length of rest of packet: 4

Header at offset 0x4:

31	19	18	16	15	8	7	5	4	3	2	0
R		ERROR		IPID		TYPE		*R*		LS	

31:19	*R*	Reserved for future use. Must be set to 0.
18:16	ERROR	Link error: 0: None, 1: Disconnect, 2: Parity, 3: Escape, 4:Credit
15:8	IPID	IP core ID: 0 for SpaceWire
7:5	TYPE	Packet type: 2 for link state packets
4:3	*R*	Reserved for future use. Must be set to 0.
2:0	LS	Link State: 0: Error reset, 1: Error wait, 2: Ready, 3: Started, 4: Connecting, 5: Run

Figure 21.3. SpaceWire link state packet

21.5.5. Link control packet format

A link control packet must be sent from a client to the server side link model to inform if that side of the link is in start mode, autostart mode, and/or has the link disabled. In addition, the control packet contains information on the startup frequency and the TX frequency that will be used once run state is reached. A new link control packet should be sent from a client any time any of these parameters change.

If the startup frequencies of the two ends differ by more than a factor 1.1/0.9, the link model will reach run state. This limit is chosen based on the limits on timeout periods in the SpaceWire standard that must be within 10% up or down from nominal frequency. So even though the startup frequency should be 10 MHz, run state can be reached if startup frequencies are close enough.

Packet length at offset 0x0:

31	0
LEN	

31:0	LEN	Length of rest of packet: 12

Header at offset 0x4:

31	16	15	8	7	5	4	3	2	1	0
R		IPID		TYPE		*R*		AS	LS	LD

31:16	*R*	Reserved for future use. Must be set to 0.
15:8	IPID	IP core ID: 0 for SpaceWire
7:5	TYPE	Packet type: 3 for link control packets
4:3	*R*	Reserved for future use. Must be set to 0.
2	AS	Link autostart.
1	LS	Link start.
0	LD	Link disable.

Startup frequency in MHz at offset 0x8:

31	0
SFREQ	

31:0	SFREQ	Startup frequency in MHz, big endian IEEE-754 32-bit float

TX frequency in MHz at offset 0xc:

31	0
TFREQ	

31:0	TFREQ	TX frequency in MHz, big endian in IEEE-754 32-bit float

Figure 21.4. SpaceWire link control packet

21.5.6. RX frequency packet format

The server side link model sends out this packet type to inform the client of with what frequency the other side transmits with when in run state. A new packet of this type is sent any time the transmitter on the other side changes its TX frequency.

Packet length at offset 0x0:

31	0
LEN	

31:0	LEN	Length of rest of packet: 8

Header at offset 0x4:

31	16	15	8	7	5	4	0
R		IPID		TYPE		*R*	

31:16	*R*	Reserved for future use. Must be set to 0.
15:8	IPID	IP core ID: 0 for SpaceWire
7:5	TYPE	Packet type: 4 for rx frequency packets
4:0	*R*	Reserved for future use. Must be set to 0.

RX frequency in MHz at offset 0x8:

31	0
RFREQ	

31:0	RFREQ	RX frequency in MHz, big endian IEEE-754 32-bit float

Figure 21.5. SpaceWire rx frequency packet

21.5.7. Link error injection packet format

A client can send a packet of this kind to the server side link model to request that a link error will occur. The error specified is the link error that is seen at the targeted end. The *OE* bit determines which end of the link is the targeted end, i.e. will see the error.

If the *OE* bit is set to 1, the error will be seen at the receiver of the simulation model on the other end. The simulation model on the client side will see a disconnect error via a link state packet. In order for this error to happen during reception of a SpaceWire packet at the other end, the client can sent a data part packet with no end marker followed by a link error injection packet.

If the *OE* bit is set to 0, the error will be seen at the receiver on the client end. The simulation model at the client end will see the requested error via a link state packet. The simulation model at the other end will see a disconnect error. Note that due to the nature of the model, this cannot in general be relied upon to inject an error during the reception of a SpaceWire packet, even if split up in multiple data parts. The link state packet will not be sent by the server side link model until all previous packets have been handled, and the client should handle all other packets queued up before that link state packet can be handled. To inject a link error during the reception of a SpaceWire packet at the client side, the packet error request packet should be used instead.

Packet length at offset 0x0:

31	0
LEN	

31:0	LEN	Length of rest of packet: 4

Header at offset 0x4:

31	21	20	19	18	16	15	8	7	5	4	0
R		OE	*R*	ERROR		IPID		TYPE		*R*	

31:21	*R*	Reserved for future use. Must be set to 0.
20	OE	Other end: 1: other end gets the error, 0: my end gets error
19	*R*	Reserved for future use. Must be set to 0.
18:16	ERROR	Link error: 1: Disconnect, 2: Parity, 3: Escape, 4:Credit
15:8	IPID	IP core ID: 0 for SpaceWire
7:5	TYPE	Packet type: 5 for link error injection packets
4:0	*R*	Reserved for future use. Must be set to 0.

Figure 21.6. SpaceWire link error injection packet

21.5.8. Packet error request packet format

A client can send a packet of this kind to the server side link model to request that a link error will occur during reception of a certain data packet by the client. The error specified is the link error that is seen, via a link state packet, by the client as a result. The other side will see a disconnect error. A 64-bit packet number, counting from 0, indicates during which SpaceWire packet sent from the other side to the client the link error should happen. Note that this number is indexing SpaceWire packets, not individual data part packets, and does not take SpaceWire packets sent from the client to the server side into account in the numbering. There can only be one outstanding packet error request per targeted GRSPW2 core at a time.

The **grspwX_status** command can be issued for the targeted GRSPW2 core to see how many SpaceWire packets have currently been sent by that core. This includes started but aborted SpaceWire packets, due to link error, core reset or active aborting using the Abort TX (AT) bit in the DMA control register of the GRSPW2 core.

Packet length at offset 0x0:

31	0
LEN	

31:0	LEN	Length of rest of packet: 16

Header at offset 0x4:

31	19	18	16	15	8	7	5	4	0
R		ERROR		IPID		TYPE		*R*	

31:19	*R*	Reserved for future use. Must be set to 0.
18:16	ERROR	Link error: 1: Disconnect, 2: Parity, 3: Escape, 4:Credit
15:8	IPID	IP core ID: 0 for SpaceWire
7:5	TYPE	Packet type: 6 for packet error request packets
4:0	*R*	Reserved for future use. Must be set to 0.

Packet number to request error for, most significant word at offset 0x8:

31	0
MSW	

31:0	MSW	Bits 63:32 of unsigned 64-bit big endian integer

Packet number to request error for, least significant word at offset 0xc:

31	0
LSW	

31:0	LSW	Bits 31:0 of unsigned 64-bit big endian integer

Reserved field at offset 0x10:

31	0
R	

31:0	*R*	Reserved for future use. Must be set to 0.

Figure 21.7. SpaceWire packet error request packet

21.6. Simple Mode

For backwards compatibility with TSIM 2.0.44 and older, the GRSPW2 models can be set up in “simple mode” with the `-grspw_simple` option. This makes the following changes to the simulation model for all GRSPW2 cores:

- The *only* supported packet types are data part packets and time code packets. The model sends out no other packet types and accepts no other packet types.
- In simple mode a SpaceWire packet is by default sent as a single data part packet *with* an end marker. Generation of separate end packets can be enabled with the `-grspw_endpacket` option. Simple mode *does* support all kinds of data part packets. However, if one needs to be compatible with the older protocol, each data part packet should contain a full SpaceWire packet with an end marker and the `-grspw_max_tx_part_len` option should not be used.
- The link state that a GRSPW2 core perceives is solely determined by its own link control setting. The other end is assumed to try to start the link. In other words, run state is achieved once the GRSPW2 is set to start or autostart without having link disable set. Moreover, startup frequencies are ignored and run state is achieved without any delay.

- The RX frequency is determined primarily by the `-grspw_simple_rxfreq` option. If that is not used, the RX frequency is taken by the `-grspw_spwfreq` option. If none of those options are set the CPU frequency is used. No cases take any clock divisors info account. The TX frequency is determined in the usual way as when not in simple mode, which includes taking the clock divisor register into account.

22. SpaceWire router

TSIM models the GRSPWROUTER core available in GR740 and GR716B. For core details and register specification please see the manual for each chip.

Supported features include:

- Transmission and reception of SpaceWire packets
- Transmission and reception of Time codes
- RMAP
- Server side link state model
- Link errors
- Link error injection
- Packet routing using both physical and logical addresses
- Static packet routing
- Packet distribution
- Both SpaceWire port and AMBA ports

Most GRSPWROUTER register fields with underlying functionality in GR740 and GR716B are supported except for:

- The SpW port link model is only in `error reset` state or `run` state.
- The AMBA ports RMAP buffer disable (RD) bit in the control register with underlying functionality is not modelled.
- The AMBA ports limitations of the No spill (NS) DMA control register bit as noted in the section on Flow control limitations below.
- No support for RX/TX of SpW interrupt codes.
- No support for SpaceWire Plug & Play via RMAP access.
- No support for Port0 (config port). Use AHB interface instead.
- No IRQ/DC/DATA timer functionality.
- No Credit/Packet counters.
- No FIFO effects modelled.

Note that some of the above limitations are due to the current socket interface. Some of these limitations will be removed after the release of the C-API.

22.1. Start up options

SpaceWire core start up options

`-grspwrtr_portX_connect` *`host:port`*
: Connect GRSPWROUTER SpW port *X* to packet server at specified server and port. Note that the X is the SpaceWire port index, i.e. use 0 for the first SpaceWire port.

`-grspwrtr_portX_server` *`port`*
: Open a packet server for SpW port *X* on specified port. Note that the X is the SpaceWire port index, i.e. use 0 for the first SpaceWire port.

`-grspw_spwfreq` *`freq`*
: Sets the SpaceWire input clock frequency. Combined with the clock divisor register, this determines the startup frequency and TX frequency.

`-grspw_max_tx_part_len` *`len`*
: Sets up all GRSPWROUTER ports to transmit any SpaceWire packet longer than *`len`* in data part packets with no more than *`len`* bytes of data.

`-grspw_endpacket [0|1]`
: Enable (or disable with 0 argument) end marking data part packets. When enabled, the last data part packet of a simulated SpaceWire packet will always be a data part packet with no data and an end marker. This is the default unless simple mode is enabled. When disabled the last data part packet can contain both data and an end marker. This is the default when simple mode is enabled.

`-grspw_simple [0|1]`
: Enable “simple mode” for all GRSPWROUTER ports. This can be used for backward compatibility with TSIM 2.0.44 and backwards. See the separate section on simple mode for details.

`-grspw_simple_rxfreq` *`freq`*
: Sets the RX frequency in MHz for all GRSPWROUTER ports to *`freq`*. This is only valid together with the `-grspw_simple` option.

`-gr740_no_spwrtr`
: This option completely removes the Router leaving only 4x GRSPW2 cores where the AMBA ports should be. This is for backwards compatibility reasons. This option also changes the PnP entries, and the IOMMU entry.

`-grspw_interconnect` *`X Y`*
: Interconnects SpaceWire port X to SpaceWire Y creating virtual loopback cable between the two.

Note that the X and Y is the SpaceWire port index in the above commands, i.e. use 0 for the first SpaceWire port.

22.2. Commands

SpaceWire Router core TSIM commands

grspwrtr_portX_connect *`host:[port]`*
: Connect SpaceWire routers SpaceWire port *`X`* to packet server at specified server and TCP port.

grspwrtr_portX_server *`port`*
: Open a packet server for SpaceWire routers SpaceWire port *`X`* on specified TCP port.

spwrtr_dbg [*`flag`*|**all**|**clean**|**list**]
: Toggle specific flag, set all, clear all, or list debug flags for the given GRSPWROUTER core. See Section 22.3 for a list of debug flags.

Note that the X is the SpaceWire port index in the above commands, i.e. use 0 for the first SpaceWire port.

For the AMBA ports the following commands can be used.

SpaceWire Router AMBA ports TSIM commands

grspwX_dbg [*`flag`*|**all**|**clean**|**list**]
: Toggle specific flag, set all, clear all, or list debug flags for the given GRSPWRTR_AMBA_PORT core. See Section 22.3 for a list of debug flags.

grspwX_status
: Print status for GRSPWRTR_AMBA_PORT *`X`*.

Note that the X is the SpaceWire Router AMBA port index in the above commands, i.e. use 0 for the first SpaceWire Router AMBA port.

22.3. Debug flags

The following debug flags and debug subcommands are available for the SpaceWire Router interface. The *`GRSPWRTR_*`*/*`GAISLER_GRSPW_*`* flags can be used with the **spwrtr_dbg**/**grspwX_dbg** command to toggle individual flags for the SpaceWire Router core. The subcommands can be used with the **spwrtr_dbg**/**grspwX_dbg** command to change and list the settings of all flags for the SpaceWire Router core. The X in **grspwX_dbg** is the index of a AMBA port, i.e. use 0 for the first AMBA port.

Table 22.1. SpaceWire Router debug flags

Flag/subcommand	Trace
GRSPWRTR_ACC	Router register accesses
GRSPWRTR_IRQ	Router interrupts
GRSPWRTR_ROUTE	Display the route created for packets
GRSPWRTR_AMBAPKT	Display packets on AMBA ports
GRSPWRTR_SPWPKT	Display packets on SPW ports
GRSPWRTR_ERR	Display info when errors occur
all	Set all debug flags for the core
clean	Set none of the debug flags for the core
list	List the current setting of the debug flags for the core

Table 22.2. SpaceWire Router AMBA port debug flags

Flag/subcommand	Trace
GAISLER_GRSPW_ACC	GRSPW accesses
GAISLER_GRSPW_RXPACKET	GRSPW received packets
GAISLER_GRSPW_RXCTRL	GRSPW rx protocol
GAISLER_GRSPW_TXPACKET	GRSPW transmitted packets
GAISLER_GRSPW_TXCTRL	GRSPW tx protocol
GAISLER_GRSPW_RMAP	GRSPW RMAP accesses
GAISLER_GRSPW_RMAPPACKET	GRSPW RMAP packet dumps
GAISLER_GRSPW_RMAPPACKDE	GRSPW RMAP packet decoding
GAISLER_GRSPW_DMAERR	GRSPW DMA errors
GAISLER_GRSPW_PART	TX/RX of GRSPW data part packets
all	Set all debug flags for the core
clean	Set none of the debug flags for the core
list	List the current setting of the debug flags for the core

22.4. SpaceWire packet server

Each SpaceWire port can be configured independently as a packet server or client using either `-grspwrtr_portX_server` or `-grspwrtr_portX_connect`. TCP sockets are used for establishing the connections. When acting as a server the core can only accept a single connection.

A connection should be set up before starting simulation for the first time, and must not be broken after that. Restarting the simulation will not tear down the connection, nor emptying any socket buffers. The socket based interface does not support any signalling of restart of the simulation. To ensure a clean restart of simulation when using this interface, restarting TSIM entirely and reconnecting socket interfaces is advisable.

The server side contains a link model that gets control information from the models at each end of the link, determines the link state and communicates frequencies and link errors to the two models at each ends of the link. It also supports error injection via two error injection packet types that can be sent from a custom client. See the the following sections for details.

For more flexibility, an external packet server can be implemented using the protocol specified in the following sections. Each core should then be connected to that server. That server would also have to implement a link model that at least reacts to link control packets and sends out link state packets and RX frequency packets.

22.5. SpaceWire packet server protocol

The protocol used to communicate with the packet server is described below. The different types of packets are defined according to the table below.

Table 22.3. Packet types

Type	Value	Direction	Notes
Data part	0	Both	Only when in run state
Time code	1	Both	Only when in run state
Link state	2	Server to client	
Link control	3	Client to server	Must be sent for model to reach run state
RX frequency	4	Server to client	
Error injection	5	Client to server	Optional
Packet error request	6	Client to server	Optional

All packets begin with a 32-bit big endian word length field which specifies the length of the rest of the packet, including header and other fixed fields. For most packet types this length is fixed for the particular type. Apart

from the data part packet type, where data follows the header byte-wise, all fields are 32-bit big endian words if not otherwise specified.

All packets received by the GRSPWROUTER model are handled sequentially, and all packets sent by the GRSPWROUTER model and the server side link model are supposed to be handled sequentially by the client. SpaceWire packets can be split into multiple data parts, transferred in data part packets. Between such parts other packets such as for time codes, link state changes, link control changes, etc., can be handled. During the simulated time span for the reception of a data part, the receiver will not/should not handle any other packet types. Use the `-grspw_max_tx_part_len` option to set up GRSPWROUTER model to split up SpaceWire packets into data parts in order for such events to be able to happen during the data stream.

22.5.1. Flow control limitations

Flow control in terms of credit is not modelled between two ends of a link. A transmitter gets no notice if the other end cannot give more credit. If the no-spill bit in the GRSPWROUTER core is set and an enabled receiving DMA channel has no enabled descriptors, the data part packet will be held on the receiving side until a descriptor is available. Due to the sequential nature of the packet model a link error, time code, etc. will not be handled at all by the GRSPWROUTER model during this time.

22.5.2. Data part packet format

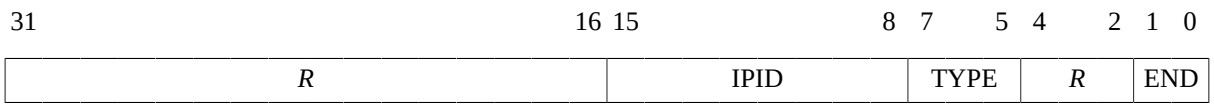
A SpaceWire packet is represented by one or more data parts. A data part packet represents one such a part. For the data parts of a multi part SpaceWire packet, only the last data part should have an EOP or EEP end marker, i.e. the *END* field set to 0 or 1. The other parts should have no end marker, i.e. the *END* field set to 2. Each data part is delivered in its entirety or not at all. A link error occurring between data parts on the other hand cuts the SpaceWire packet short and is considered the end of that SpaceWire packet.

A data part packet is sent at the beginning of transmission of the corresponding part of the SpaceWire packet, so that the receiver can start reacting to it as soon as the transmission starts. The GRSPWROUTER model by default sends a SpaceWire packet in the form of two data part packets. The first data part packet is sent at the beginning of transmission and contains all data but has no end marker. The second data part packet is sent when transmission is completed and has the appropriate end marker set but contains no data. If a user model is not waiting for the end marker packet before responding, the response could arrive before transmission is considered done by the GRSPWROUTER model. Generation of separate end marker packets can be turned off using the `-grspw_endpacket` option. Splitting up SpaceWire packets into several data containing data part packets can be enabled with the `-grspw_max_tx_part_len` option.

Packet length at offset 0x0:

31:0	LEN	Length of rest of packet: 4 + number of data bytes in the part

Header at offset 0x4:



31:16	*R*	Reserved for future use. Must be set to 0.
15:8	IPID	IP core ID: 0 for SpaceWire
7:5	TYPE	Packet type: 0 for data part packets
4:2	*R*	Reserved for future use. Must be set to 0.
1:0	END	End marker: 0: Normal End of Packet, 1: Error End of Packet, 2: No end marker

Offset 0x8: The data bytes of the part starts here

Figure 22.1. SpaceWire data part packet

22.5.3. Time code packet format

Packet length at offset 0x0:

31	0
LEN	

31:0	LEN	Length of rest of packet: 8

Header at offset 0x4:

31	16	15	8	7	5	4	0
R		IPID		TYPE		*R*	

31:16	*R*	Reserved for future use. Must be set to 0.
15:8	IPID	IP core ID: 0 for SpaceWire
7:5	TYPE	Packet type: 1 for time code packets
4:0	*R*	Reserved for future use. Must be set to 0.

Payload at offset 0x8:

31	8	7	6	5	0
R		CT		CN	

31:8	*R*	Reserved for future use. Must be set to 0.
7:6	CT	Time control flags
5:0	CN	Value of time counter

Figure 22.2. SpaceWire time code packet

22.5.4. Link state packet format

Link state packets are sent out by the server side link model when the link state changes. The only states currently simulated are `Error Reset` and `Run`. A link state packet with state `Error Reset` can have the ***ERROR*** field set to an error seen at the receiver. Other link state packets has only `None` in the ***ERROR*** field.

Packet length at offset 0x0:

31	0
LEN	

31:0	LEN	Length of rest of packet: 4

Header at offset 0x4:

31	19	18	16	15	8	7	5	4	3	2	0
R		ERROR		IPID		TYPE		*R*		LS	

31:19	*R*	Reserved for future use. Must be set to 0.
18:16	ERROR	Link error: 0: None, 1: Disconnect, 2: Parity, 3: Escape, 4:Credit
15:8	IPID	IP core ID: 0 for SpaceWire
7:5	TYPE	Packet type: 2 for link state packets
4:3	*R*	Reserved for future use. Must be set to 0.
2:0	LS	Link State: 0: Error reset, 1: Error wait, 2: Ready, 3: Started, 4: Connecting, 5: Run

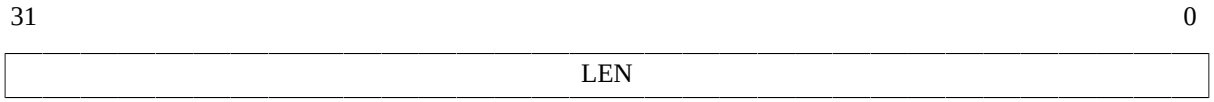
Figure 22.3. SpaceWire link state packet

22.5.5. Link control packet format

A link control packet must be sent from a client to the server side link model to inform if that side of the link is in start mode, autostart mode, and/or has the link disabled. In addition, the control packet contains information on the startup frequency and the TX frequency that will be used once run state is reached. A new link control packet should be sent from a client any time any of these parameters change.

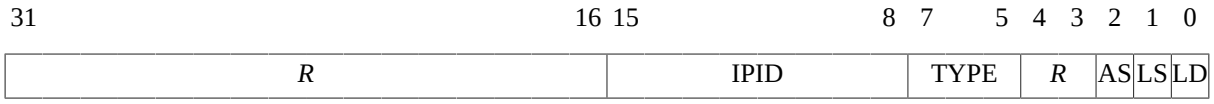
If the startup frequencies of the two ends differ by more than a factor 1.1/0.9, the link model will reach run state. This limit is chosen based on the limits on timeout periods in the SpaceWire standard that must be within 10% up or down from nominal frequency. So even though the startup frequency should be 10 MHz, run state can be reached if startup frequencies are close enough.

Packet length at offset 0x0:



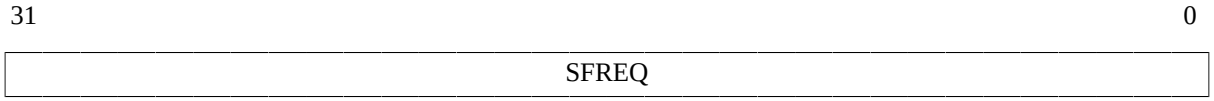
31:0	LEN	Length of rest of packet: 12

Header at offset 0x4:



31:16	*R*	Reserved for future use. Must be set to 0.
15:8	IPID	IP core ID: 0 for SpaceWire
7:5	TYPE	Packet type: 3 for link control packets
4:3	*R*	Reserved for future use. Must be set to 0.
2	AS	Link autostart.
1	LS	Link start.
0	LD	Link disable.

Startup frequency in MHz at offset 0x8:



31:0	SFREQ	Startup frequency in MHz, big endian IEEE-754 32-bit float

TX frequency in MHz at offset 0xc:

31:0	TFREQ	TX frequency in MHz, big endian in IEEE-754 32-bit float

Figure 22.4. SpaceWire link control packet

22.5.6. RX frequency packet format

The server side link model sends out this packet type to inform the client of with what frequency the other side transmits with when in run state. A new packet of this type is sent any time the transmitter on the other side changes its TX frequency.

Packet length at offset 0x0:

31	0
LEN	

31:0	LEN	Length of rest of packet: 8

Header at offset 0x4:

31	16	15	8	7	5	4	0
R		IPID		TYPE		*R*	

31:16	*R*	Reserved for future use. Must be set to 0.
15:8	IPID	IP core ID: 0 for SpaceWire
7:5	TYPE	Packet type: 4 for rx frequency packets
4:0	*R*	Reserved for future use. Must be set to 0.

RX frequency in MHz at offset 0x8:

31	0
RFREQ	

31:0	RFREQ	RX frequency in MHz, big endian IEEE-754 32-bit float

Figure 22.5. SpaceWire rx frequency packet

22.5.7. Link error injection packet format

A client can send a packet of this kind to the server side link model to request that a link error will occur. The error specified is the link error that is seen at the targeted end. The *OE* bit determines which end of the link is the targeted end, i.e. will see the error.

If the *OE* bit is set to 1, the error will be seen at the receiver of the simulation model on the other end. The simulation model on the client side will see a disconnect error via a link state packet. In order for this error to happen during reception of a SpaceWire packet at the other end, the client can sent a data part packet with no end marker followed by a link error injection packet.

If the *OE* bit is set to 0, the error will be seen at the receiver on the client end. The simulation model at the client end will see the requested error via a link state packet. The simulation model at the other end will see a disconnect error. Note that due to the nature of the model, this cannot in general be relied upon to inject an error during the reception of a SpaceWire packet, even if split up in multiple data parts. The link state packet will not be sent by the server side link model until all previous packets have been handled, and the client should handle all other packets queued up before that link state packet can be handled. To inject a link error during the reception of a SpaceWire packet at the client side, the packet error request packet should be used instead.

Packet length at offset 0x0:

31	0
LEN	

31:0	LEN	Length of rest of packet: 4

Header at offset 0x4:

31	21	20	19	18	16	15	8	7	5	4	0
R		OE	*R*	ERROR		IPID		TYPE		*R*	

31:21	*R*	Reserved for future use. Must be set to 0.
20	OE	Other end: 1: other end gets the error, 0: my end gets error
19	*R*	Reserved for future use. Must be set to 0.
18:16	ERROR	Link error: 1: Disconnect, 2: Parity, 3: Escape, 4:Credit
15:8	IPID	IP core ID: 0 for SpaceWire
7:5	TYPE	Packet type: 5 for link error injection packets
4:0	*R*	Reserved for future use. Must be set to 0.

Figure 22.6. SpaceWire link error injection packet

22.5.8. Packet error request packet format

A client can send a packet of this kind to the server side link model to request that a link error will occur during reception of a certain data packet by the client. The error specified is the link error that is seen, via a link state packet, by the client as a result. The other side will see a disconnect error. A 64-bit packet number, counting from 0, indicates during which SpaceWire packet sent from the other side to the client the link error should happen. Note that this number is indexing SpaceWire packets, not individual data part packets, and does not take SpaceWire packets sent from the client to the server side into account in the numbering. There can only be one outstanding packet error request per targeted GRSPWROUTER core at a time.

The **grspwX_status** command can be issued for the targeted GRSPWROUTER core to see how many SpaceWire packets have currently been sent by that core. This includes started but aborted SpaceWire packets, due to link error, core reset or active aborting using the Abort TX (AT) bit in the DMA control register of the GRSPWROUTER core.

Packet length at offset 0x0:

31	0
LEN	

31:0	LEN	Length of rest of packet: 16

Header at offset 0x4:

31	19	18	16	15	8	7	5	4	0
R		ERROR		IPID		TYPE		*R*	

31:19	*R*	Reserved for future use. Must be set to 0.
18:16	ERROR	Link error: 1: Disconnect, 2: Parity, 3: Escape, 4:Credit
15:8	IPID	IP core ID: 0 for SpaceWire
7:5	TYPE	Packet type: 6 for packet error request packets
4:0	*R*	Reserved for future use. Must be set to 0.

Packet number to request error for, most significant word at offset 0x8:

31	0
MSW	

31:0	MSW	Bits 63:32 of unsigned 64-bit big endian integer

Packet number to request error for, least significant word at offset 0xc:

31	0
LSW	

31:0	LSW	Bits 31:0 of unsigned 64-bit big endian integer

Reserved field at offset 0x10:

31	0
R	

31:0	*R*	Reserved for future use. Must be set to 0.

Figure 22.7. SpaceWire packet error request packet

22.6. Simple Mode

For backwards compatibility with TSIM 2.0.44 and older, the GRSPWROUTER models can be set up in “simple mode” with the `-grspw_simple` option. This makes the following changes to the simulation model for all GRSPWROUTER cores:

- The *only* supported packet types are data part packets and time code packets. The model sends out no other packet types and accepts no other packet types.
- In simple mode a SpaceWire packet is by default sent as a single data part packet *with* an end marker. Generation of separate end packets can be enabled with the `-grspw_endpacket` option. Simple mode *does* support all kinds of data part packets. However, if one needs to be compatible with the older protocol, each data part packet should contain a full SpaceWire packet with an end marker and the `-grspw_max_tx_part_len` option should not be used.
- The link state that a GRSPWROUTER link perceives is solely determined by its own link control setting. The other end is assumed to try to start the link. In other words, run state is achieved once the GRSPWROUTER link is set to start or autostart without having link disable set. Moreover, startup frequencies are ignored and run state is achieved without any delay.

- The RX frequency is determined primarily by the `-grspw_simple_rxfreq` option. If that is not used, the RX frequency is taken by the `-grspw_spwfreq` option. If none of those options are set the CPU frequency is used. No cases take any clock divisors info account. The TX frequency is determined in the usual way as when not in simple mode, which includes taking the clock divisor register into account.

22.7. SpaceWire C interface

This C-API is currently only available for the SpaceWire router. For standalone SpaceWire cores use the packet server protocol, see Chapter 21.

22.7.1. Connecting a user SpaceWire model

To connect a custom SpaceWire node to TSIM's SpaceWire router port use `tsim_register_spw_node(node, spw_index)` where `node` is a pointer to a `struct spw_node` (see below), and `spw_index` is the index of the SpaceWire port to connect to. All related structs can be found in `spwrtr_input.h`.

See Chapter 5 for further details on how to connect the user model.

22.7.2. SpaceWire C-API

The main functionality of the API is available through `struct spw_node` described below.

```
struct spw_node {
    unsigned int id;
    int (*init)(struct spw_node *node);
    int (*rx_packet)(struct spw_node *node, struct spw_packet *packet);
    int (*rx_done)(struct spw_node *node);
    int (*rx_timecode)(struct spw_node *node, unsigned int value);
    int (*tx_timecode)(struct spw_node *node, unsigned int value);
    int (*tx_packet)(struct spw_node *node, struct spw_packet *packet);
    int (*tx_done)(struct spw_node *node);
    int (*update_link)(struct spw_node *node, struct spw_link_control *ctrls);
    int (*set_link_state)(struct spw_node *node, struct spw_link_state_change *change);
    void *priv;
    struct user_node_priv *node_priv;
};
```

Table 22.4. struct spw_node members

Parameter	Description
id	Node ID can be set freely by the user to keep track of different nodes.
init	User provided init function that will be called at simulator start-up.
rx_packet	User provided function that will be called when the node receives a packet.
rx_done	TSIM provided function that should be called when a packet is done receiving.
rx_timecode	User provided function that will be called when node receives a time code.
tx_timecode	TSIM provided function that can be called to send a time code.
tx_packet	TSIM provided function that can be called to send a packet.
tx_done	User provided function that will be called when node has finished transmitting a packet.
update_link	TSIM provided function that should be called to update a nodes link-state.
set_link_state	User provided function that will be called when node link-state has changed.
priv	Pointer to private data. Can be set freely by the user.
node_priv	Pointer to private data used internally by TSIM. Should not be touched by the user.

When a node has been registered using `tsim_register_spw_node` the first thing that will happen on simulator start-up it that the `init` function will be called, this can be used to set-up states etc for the user model.

Table 22.5. init function parameters

Parameter	Description
node	Pointer to the `struct spw_node` node that called this function.

22.7.2.1. Link state

To be able to communicate the SpaceWire link controls should be set to autostart or start and the start and transmitting frequencies should be set to match the SpaceWire port the node is connected to. This can be done by calling the nodes `update_link` function that has been installed by TSIM during node registration. It takes a `struct spw_link_control` as argument described below.

```
struct spw_link_control {
    int autostart;
    int start;
    int disable;
    double start_freq;
    unsigned int tx_freq;
};
```

Table 22.6. struct spw_link_control members

Parameter	Description
autostart	Set this to non-zero if the link should be in autostart mode.
start	Set this to non-zero if the link should be in start mode.
disable	Set this to non-zero if the link is disabled.
start_freq	Set the links start frequency in MHz.
tx_freq	Set the links transmitting frequency in MHz.

The arguments to `update_link` are

Table 22.7. update_link function parameters

Parameter	Description
node	Pointer to the `struct spw_node` node that is calling this function.
ctrls	Pointer to a `struct spw_link_control` describing the desired link-state.

If the SpaceWire port simulated by TSIM should change the link state it will call the nodes `set_link_state` function that should be installed by the user. The `spw_link_state_change` described below will contain information about the link-state change.

```
enum spw_link_state {
    SPW_LINK_STATE_ERROR_RESET = 0,
    SPW_LINK_STATE_ERROR_WAIT = 1,
    SPW_LINK_STATE_READY = 2,
    SPW_LINK_STATE_STARTED = 3,
    SPW_LINK_STATE_CONNECTING = 4,
    SPW_LINK_STATE_RUN = 5,
};
enum spw_link_error {
    SPW_LINK_ERROR_NONE = 0,
    SPW_LINK_ERROR_DISCONNECT,
    SPW_LINK_ERROR_PARITY,
    SPW_LINK_ERROR_ESCAPE,
    SPW_LINK_ERROR_CREDIT,
};
struct spw_link_state_change
{
    enum spw_link_state state;
    enum spw_link_error error;
};
```

This information can then be used for the user to update their own SpaceWire node implementation and react to a state change.

22.7.2.2. Transmitting and receiving packages

When transmitting a message `tx_packet` should be called. It takes a pointer to the node that wants to send as well as a `struct spw_packet` containg the message, described below.

```
enum spw_end {
    SPW_NOEND = 0,
    SPW_EOP,
    SPW_EEP,
};

struct spw_packet {
    unsigned char *data;
    uint32 datalen;
    enum spw_end end;

    struct spw_node *node;
};
```

Table 22.8. struct spw_packet members

Parameter	Description
*data	Pointer to the packet data.
datalen	Length of the data to transmitt.
end	Packet ending, `SPW_NOEND`: the packet is not yet done i.e. sent in many parts. `SPW_EOP`: packet is done with no problem and `SPW_EEP`: error end.
node	Pointer to the SpaceWire node that is the sender when transmitting or the receiving node when receiving.

After `tx_packet` has been called TSIM will then send the packet to the router. When done TSIM will call the user provided function `tx_done` letting the node know that the packet has been handled and that it is safe to clean up the packet or to send the next part if part of a series. If the link is busy `tx_packet` will return non-zero and the user will have to try again later, or return 0 if the packet was successfully sent.

Similarly to transmitting packets, when receiving the nodes `rx_packet` callback will be called. With the node receiving the packet and the packet as parameters. When the module has received/handled the packet it should call the `rx_done` function to let TSIM know the packet has been received and it is safe to continue with the next part/new packets.

22.7.2.3. Time codes

Sending and receiving time codes are done through the functions `tx_timecode` and `rx_timecode`. They take the receiving/sending node as parameter as well as the new timecode as an unsigned integer. The main difference between time codes and normal packages are that no `tx/rx_done` needs to be called for time codes.

22.7.2.4. Error injection

Error injection can be performed by setting the `end` property of a `struct spw_packet` to `SPW_EEP` before sending it with `tx_packet`. Similarly when receiving an error TSIM will set the end to `SPW_EEP` of a packet before giving it to the module via `rx_packet`. To inject errors in the middle of a packet the option `-grspw_max_tx_part_len` can be used to split a large packet into smaller parts, then set the `end` property on one of the smaller parts.

23. SPI interface

23.1. Connecting a user SPI model

To register a SPI user module, call `tsim_register_spi_module(spi_input, index)` from an input modules init function. Here *`spi_input`* is a pointer to a spi_input struct, and *`index`* is the index of the SPI core to register on. See Chapter 5 for further details on how to connect the user model.

23.2. SPI bus model API

The structure `struct spi_input` models the SPI bus. It is defined as:

```
/* Spi input provider */

struct spi_input {
   int (*spishift)(struct spi_input *ctrl, uint32 select, uint32 bitcnt,
                   uint32 out, uint32 *in);
   void *priv;
};
```

The spishift callback should be set by the SPI user module at startup. It is called by the SPI core whenever it shifts a word through the SPI bus.

Table 23.1. spishift callback parameters

Parameter	Description
select	Slave select register bits when such a register is present. Zero otherwise.
bitcnt	Number of bits per word (as per the MODE register) for actual shifts. If bitcnt is -1 then the operation is not an actual shift and the call is merely to indicate a change in the slave select register (when such a register is present).
out	Shift out (tx) data
in	Shift in (rx) data

The `priv` parameter is a pointer to private data and be set freely by the user.

The return value of spishift is ignored.

See the `examples/input` directory for an example module implementation. See the `examples/test` directory for an example test program.

23.3. Commands

SPI Commands

spiX_dbg [*`flag`*|**all**|**clean**|**list**]
: Toggle specific flag, set all, clear all, or list debug flags for the given SPI core. See Section 23.4 for a list of debug flags.

23.4. Debug flags

The following debug flags and debug subcommands are available for SPI interfaces. The *`GAISLER_SPI_*`* flags can be used with the **spiX_dbg** command to toggle individual flags for individual SPI cores and with the **dbgon** command to toggle individual flags for all SPI cores. The subcommands can be used with the **spiX_dbg** command to change and list the settings of all flags for individual SPI cores.

Table 23.2. SPI debug flags

Flag/subcommand	Trace
GAISLER_SPI_ACC	SPI register accesses
GAISLER_SPI_XFER	SPI bus transfers
GAISLER_SPI_IRQ	SPI interrupts

Flag/subcommand	Trace
all	Set all SPI debug flags for the core
clean	Set none of the SPI debug flags for the core
list	List the current setting of the debug flags for the core

24. SPIM interface

24.1. Connecting a user SPIM model to TSIM

To register a user module on a SPIM controller call `tsim_register_spim_module(spim_subsystem, index)` from an input modules init function. Here *`spim_subsystem`* is a pointer to a spim_subsystem struct and *`index`* is the index of the SPIM controller to register on. See Chapter 5 for further details on how to connect the user model. The `struct spim_subsystem` is defined in `spim_input.h` as:

```
struct spim_subsystem {
    struct spim_input *inp;
};
```

24.2. SPIM model API

The structure `struct spim_input` models the SPIM bus. It is defined as:

```
/* Spim input provider */

struct spim_input {
    int (*spishift)(struct spim_input * ctrl, unsigned int select,
                    unsigned int bitcnt, unsigned int timing_scaler,
                    unsigned int out, unsigned int *in);
    void (*spim_init)(struct spim_interface *spimif);
    void *priv;
};
```

The `spishift` callback should be set by the SPIM user module at startup. It is called by TSIM3 whenever a new byte is written to the TX register.

Table 24.1. spishift callback parameters

Parameter	Description
select	Slave select bits
bitcnt	Number of bits the user model will receive. This will always be set to 8.
timing_scaler	The relation between the SPIM core SCK and the system clk.
out	Shift out (tx) data
in	Shift in (rx) data

The `priv` is a pointer to private data and can be set freely by the user. The `spim_init` is called at startup and provides the user model with a SPIM interface struct. The SPIM interface struct is defined as:

```
struct spim_interface {
    int (*spim_get_flashb)(unsigned int index, void *data);
};
```

Table 24.2. spim_get_flashb parameters

Parameter	Description
index	Index of the SPIM controller to access.
data	Pointer to a `spim_flash_data` struct to be filled in.

The spimif struct allows access to TSIM3s internal SPIM memory representation with `spim_get_flashb`. `index` is the index of the SPIM controller to access. The `data` parameter should be a pointer to a `spim_flash_data` struct which will be updated with the necessary data to access the internal memory representation. The `spim_flash_data` struct is defined as:

```
struct spim_flash_data {
    unsigned int flash_size;
    unsigned char *flashb;
    unsigned int flash_mask;
};
```

Table 24.3. struct spim_flash_data members

Parameter	Description
flash_size	Size of the flash memory.
flashb	Pointer to the flash memory.
flash_mask	Flash memory mask.

See the `examples/input` directory for an example module implementation. The example demonstrates how to set up a basic model, get access to TSIM's internal memory representation and updates the RX register. See the `examples/test` directory for an example test program.

25. AT697 PCI interface

TSIM models the PCI interface in AT697, in TSIM called “ESAPCI”. For core details and register specification please see the manual for the chip.

The PCI model will process all accesses to memory region 0xa0000000 - 0xf0000000 (AHB slave mode) and the APB registers starting at 0x80000100. The AT697 PCI model implements all registers of the PCI core. It will in turn load the PCI user modules that will implement the devices. The AT697 model is supposed to be the PCI host. Both PCI Initiator mode and PCI Target mode are supported. The interface to the PCI user modules is implemented on bus level. Two callbacks model the PCI bus.

25.1. Commands

ESAPCI TSIM Commands

esapci0_dbg
: Toggle specific flag, set all, clear all, or list debug flags for the given PCI core. See Section 25.2 for a list of debug flags.

25.2. Debug flags

The following debug flags are available for the ESAPCI interface. Use them in conjunction with the **esapci0_dbg** command to enable different levels of debug information.

Table 25.1. ESAPCI interface debug flags

ESAPCI_REGACC	Trace accesses to the PCI registers
ESAPCI_ACC	Trace accesses to the PCI AHB-slave address space
ESAPCI_DMA	Trace DMA
ESAPCI_IRQ	Trace PCI IRQ

25.3. Registers

Table 25.2 contains a list of implemented and not implemented fields of the AT697F PCI Registers. Only register fields that are relevant for the emulated PCI module is implemented.

Table 25.2. PCI register support

Register	Implemented	Not implemented
PCIID1	device id, vendor id	
PCISC	stat 13, stat 12, stat 11, stat 7, stat 6 stat 5, stat 4, com2, com 1, com1	stat15 stat14 stat10_9 stat8 com10 com9 com8 com7 com6 com5 com4 com3
PCIID2	class code, revision id	
PCIBHDLC	[bist, header type, latency timer, cache size] config-space only	
PCIMBAR1	base address, pref, type, msi	
PCIMBAR2	base address, pref, type, msi	
PCIIOBAR3	io base address, ms	
PCISID	subsystem id, svi	
PCICP	pointer	
PCILI	[max_lat min_gnt int_pin int_line] con-fig-space-only	
PCIRT	[retry trdy] config-space-only	
PCICW		ben

Register	Implemented	Not implemented
PCISA	start address	
PCIIW		ben
PCIDMA	wdcnt, com	b2b
PCIIS	act, xff, xfe, rfe	dmas, ss
PCIIC	mod, commsb	dwr, dww, perr
PCITPA	tpa1, tpa2	
PCITSC		errmem, xff, xfe, rfe, tms
PCIITE	dmaer,imier, tier	cmfer, imper, tbeer, tper, syser
PCIITP	dmaer,imier, tier	cmfer, imper, tbeer, tper, syser
PCIITF	dmaer,imier, tier, cmfer, imper, tbeer, tper, syser	
PCID	dat	
PCIBE	dat	
PCIDMAA	addr	
PCIA		p0, p1, p2, p3

25.4. ESAPCI bus model API

To register a ESAPCI module call `tsim_register_esa_pci_module(struct esa_pci_input *inp, int index);` from an input modules init function. Here *`inp`* is a pointer to a esa_pci_input struct and *`index`* is the index of the ESAPCI controller to register on. See Chapter 5 for further details on how to connect the user model. The `struct esa_pci_input` is defined in `esa_pci_input.h` as:

```
struct esa_pci_input {
    int (*acc)(struct esa_pci_input *ctrl,
               int cmd,
               unsigned int addr,
               unsigned int wsize,
               unsigned int *data,
               unsigned int *abort,
               unsigned int *ws);

    void (*esapci_init)(struct esa_pci_interface *esapciif);
};
```

The acc callback should be set by the PCI user module at startup. It is called by the the model whenever the PCI core reads/writes as a PCI bus master.

Table 25.3. acc callback parameters

Parameter	Description
cmd	Command to execute, see Section 25.4.1 details.
addr	PCI address.
wsize	0: 8-bit access 1: 16-bit access, 2: 32-bit access. Is always 2 for read accesses.
data	Data buffer. The user module should return the read data in *`*data`* for read commands or write the data in *`*data`* for write commands.
abort	Set *`*abort`* to 1 to generate target abort, or 0 otherwise.
ws	Set *`*ws`* to the number of PCI clocks it takes to complete the transaction.

The return value of acc determines if the transaction terminates successfully (0 or ESA_PCI_ACC_OK) or with master abort (1 or ESA_PCI_ACC_MASTER_ABORT).

The esapci_init callback should be set by the PCI user module at startup. It is called by TSIM at simulator startup.

Table 25.4. esapci_init callback parameters

Parameter	Description
esapciif	Pointer to a `struct esa_pci_interface`. Should be saved by the module to interface with TSIM's ESAPCI model.

The `struct esa_pci_interface` is defined in `esapci_input.h` as:

```
struct esa_pci_interface {
    int (*target_acc)(int index,
                      int cmd,
                      unsigned int addr,
                      unsigned int wsize,
                      unsigned int *data,
                      unsigned int *mexc);
};
```

The callback target_acc is installed by the TSIM. The PCI user dynamic library can call this function to initiate an access to the PCI target.

Table 25.5. target_acc parameters

Parameter	Description
index	Index of ESAPCI core of the system. This should be 0.
cmd	Command to execute, see Section 25.4.1 for details. Configuration cycles are not supported. ESAPCI is supposed to be the host.
addr	PCI address. Should always be word aligned for read accesses.
wsize	0: 8-bit access 1: 16-bit access, 2: 32-bit access. Should always be 2 for read accesses.
data	Data buffer. The read data is returned in `*data` for read commands or the data in `*data` is written for write commands.
mexc	The model sets *mexc to 0 if access is successful, or 1 in case of target abort.

If a supported command hits MEMBAR0, MEMBAR1 or IOBAR, target_acc() will return 0. For unsupported commands or other areas, it will return 1.

25.4.1. PCI command table

Table 25.6. PCI command table

Command	Description
0	IRQ acknowledge
1	Special cycle
2	I/O Read
3	I/O Write
4	Reserved
5	Reserved
6	Memory Read
7	Memory Write
8	Reserved
9	Reserved
10	Configuration Read
11	Configuration Write
12	Memory Read Multiple
13	Dual Address Cycle

Command	Description
14	Memory Read Line
15	Memory Write And Invalidate

25.5. Examples

See the PCI files in `examples/input` for header files and an example PCI user module. See example usage in `examples/test`.

26. TPS VxWorks 6.x AHB Module

26.1. Overview

The TPS VxWorks Module is a loadable module that simplifies communication between TSIM and the VxWorks Workbench for VxWorks 6.7 and 6.9. It provides a virtual core that acts similar to a basic Ethernet controller, but does not require a packet server.

The module is only useful in conjunction with VxWorks 6.7 and 6.9.

Table 26.1. Files delivered with the TPS VxWorks TSIM module

File	Description
tps/linux/tps-vxworks.so	TPS VxWorks module for Linux
tps/win64/tps-vxworks.dll	TPS VxWorks module for Windows

26.2. Loading the module

The module is loaded using the TSIM3 option `-mod`. It can be used in conjunction with other modules, such as the UT699 and GR712RC modules.

On Linux (together with the UT699 design):

```
tsim-leon3 -ut699 -mod ./tps/linux/tps-vxworks.so
```

On Windows (together with the GR712RC design):

```
tsim-leon3 -gr712rc -mod ./tps/win64/tps-vxworks.dll
```

26.3. Configuration

By default the module has registers starting at 0x80002000, uses IRQ 5 and UDP port 0x4321. This can be changed by using the following command line arguments:

-tps_vxworks_addr addr
: Uses address `addr` as start address of the register area instead of the default. Must be put where there is an APB controller and aligned to 0x100.

-tps_vxworks_irq irq
: Uses IRQ `irq` instead of the default.

-tps_vxworks_port port
: Uses UDP port `port` instead of the default.

Use the following command line to make the TPS module use IRQ 10 and port 5000 on Linux together with the UT699 design:

```
tsim-leon3 -ut699 -mod ./tps/linux/tps-vxworks.so
         -tps_vxworks_port 5000 -tps_vxworks_irq 10
```

27. Support

For support contact the support team at support@gaisler.com.

When contacting support, please identify yourself in full, including company affiliation and site name and address. Please identify exactly what product that is used, specifying if it is an IP core (with full name of the library distribution archive file), component, software version, compiler version, operating system version, debug tool version, simulator tool version, board version, etc.

The support service is only for paying customers with a support contract.

Frontgrade Gaisler AB
Kungsgatan 12
411 19 Göteborg
Sweden
frontgrade.com/gaisler
sales@gaisler.com
T: +46 31 7758650
F: +46 31 421407