TSIM2 Simulator User's Manual
# Table of Contents

1. Introduction .................................................................................................................. 7  
   1.1. General ...................................................................................................................... 7  
   1.2. Supported platforms and system requirements ......................................................... 7  
   1.3. Obtaining TSIM ......................................................................................................... 7  
   1.4. License ..................................................................................................................... 7  
   1.5. Evaluation version .................................................................................................... 7  
   1.6. Problem reports ....................................................................................................... 7  
2. Installation ...................................................................................................................... 9  
   2.1. General ..................................................................................................................... 9  
   2.2. License installation .................................................................................................. 9  
      2.2.1. Floating keys ....................................................................................................... 9  
      2.2.2. LDK type license keys ....................................................................................... 9  
3. Operation ....................................................................................................................... 10  
   3.1. Overview .................................................................................................................. 10  
   3.2. Starting TSIM ........................................................................................................... 10  
   3.3. Standalone mode commands .................................................................................. 14  
      3.3.1. Time specification for commands ...................................................................... 17  
   3.4. Symbolic debug information .................................................................................... 17  
   3.5. Breakpoints and watchpoints .................................................................................. 18  
   3.6. Profiling .................................................................................................................. 18  
   3.7. Code coverage ......................................................................................................... 18  
   3.8. Check-pointing ....................................................................................................... 20  
   3.9. Performance ........................................................................................................... 20  
   3.10. Backtrace ............................................................................................................... 20  
   3.11. Connecting to gdb .................................................................................................. 20  
   3.12. Thread support ....................................................................................................... 21  
      3.12.1. TSIM thread commands .................................................................................... 21  
      3.12.2. GDB thread commands ................................................................................... 22  
   3.13. Synchronising TSIM time to external time ............................................................. 23  
4. Emulation characteristics ............................................................................................... 25  
   4.1. Common behaviour ................................................................................................. 25  
      4.1.1. Timing ................................................................................................................. 25  
      4.1.2. UARTs ................................................................................................................ 25  
      4.1.3. Floating point unit (FPU) .................................................................................... 25  
      4.1.4. Delayed write to special registers ..................................................................... 26  
      4.1.5. Idle-loop optimisation ....................................................................................... 26  
      4.1.6. Custom instruction emulation ............................................................................ 26  
      4.1.7. Chip-specific errata ........................................................................................... 27  
   4.2. ERC32 specific emulation ......................................................................................... 27  
      4.2.1. Processor emulation .......................................................................................... 27  
      4.2.2. MEC emulation .................................................................................................. 27  
      4.2.3. Interrupt controller ............................................................................................ 28  
      4.2.4. Watchdog .......................................................................................................... 28  
      4.2.5. Power-down mode ............................................................................................. 28  
      4.2.6. Memory emulation ............................................................................................. 28  
      4.2.7. EDAC operation ................................................................................................. 28  
      4.2.8. Extended RAM and I/O areas ........................................................................... 29  
      4.2.9. SYSAV signal ...................................................................................................... 29  
      4.2.10. EXTINTACK signal ........................................................................................ 29  
      4.2.11. IWDE signal .................................................................................................... 29  
   4.3. LEON2 specific emulation ....................................................................................... 29  
      4.3.1. Processor ............................................................................................................ 29  
      4.3.2. Cache memories ................................................................................................. 29  
      4.3.3. LEON peripherals registers .............................................................................. 30  
      4.3.4. Interrupt controller ........................................................................................... 30  
   4.3.5. Chip-specific errata ............................................................................................. 30
5. Loadable modules ......................................................... 34
  5.1. TSIM I/O emulation interface ........................................ 34
    5.1.1. simif structure .................................................. 34
    5.1.2. ioif structure .................................................. 36
    5.1.3. Structure to be provided by I/O device .......................... 36
    5.1.4. Cygwin specific io_init() ...................................... 37
  5.2. LEON AHB emulation interface ...................................... 37
    5.2.1. procif structure ................................................ 38
    5.2.2. Structure to be provided by AHB module ......................... 38
    5.2.3. Big versus little endianess .................................... 41
  5.3. TSIM/LEON co-processor emulation ................................ 41
    5.3.1. FPU/CP interface ............................................... 41
    5.3.2. Structure elements ............................................. 42
    5.3.3. Attaching the FPU and CP ...................................... 42
    5.3.4. Big versus little endianess .................................... 43
    5.3.5. Additional TSIM commands ...................................... 43
    5.3.6. Example FPU .................................................... 43
  5.4. Loadable modules distributed with TSIM ............................ 43
    5.4.1. General AHB module limitations ................................ 43
  6. TSIM library (TLIB) ................................................... 45
    6.1. Introduction ..................................................... 45
    6.2. Function interface ............................................... 45
    6.3. External I/O and AHB modules .................................... 46
    6.4. Built-in I/O module and/or AHB module ........................... 46
    6.5. UART handling ................................................... 47
<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>6.6. Linking a TLIB application</td>
<td>47</td>
</tr>
<tr>
<td>6.7. Limitations</td>
<td>47</td>
</tr>
<tr>
<td>7. Cobham UT699 emulation</td>
<td>48</td>
</tr>
<tr>
<td>7.1. Overview of the UT699 AHB module</td>
<td>48</td>
</tr>
<tr>
<td>7.2. Loading the module</td>
<td>49</td>
</tr>
<tr>
<td>7.2.1. User input module interface</td>
<td>49</td>
</tr>
<tr>
<td>7.3. Debugging</td>
<td>51</td>
</tr>
<tr>
<td>7.4. 10/100 Mbps Ethernet Media Access Controller interface</td>
<td>51</td>
</tr>
<tr>
<td>7.4.1. Start up options</td>
<td>51</td>
</tr>
<tr>
<td>7.4.2. Commands</td>
<td>51</td>
</tr>
<tr>
<td>7.4.3. Debug flags</td>
<td>51</td>
</tr>
<tr>
<td>7.4.4. Ethernet packet server</td>
<td>52</td>
</tr>
<tr>
<td>7.4.5. Ethernet packet server protocol</td>
<td>52</td>
</tr>
<tr>
<td>7.5. SpaceWire interface with RMAP support</td>
<td>52</td>
</tr>
<tr>
<td>7.5.1. Start up options</td>
<td>53</td>
</tr>
<tr>
<td>7.5.2. Commands</td>
<td>53</td>
</tr>
<tr>
<td>7.5.3. Debug flags</td>
<td>53</td>
</tr>
<tr>
<td>7.5.4. SpaceWire packet server</td>
<td>53</td>
</tr>
<tr>
<td>7.5.5. SpaceWire packet server protocol</td>
<td>53</td>
</tr>
<tr>
<td>7.6. PCI initiator/target interface</td>
<td>55</td>
</tr>
<tr>
<td>7.6.1. Connecting a user PCI model with the UT699 module</td>
<td>55</td>
</tr>
<tr>
<td>7.6.2. Commands</td>
<td>55</td>
</tr>
<tr>
<td>7.6.3. Debug flags</td>
<td>55</td>
</tr>
<tr>
<td>7.6.4. PCI bus model API</td>
<td>55</td>
</tr>
<tr>
<td>7.7. GPIO interface</td>
<td>56</td>
</tr>
<tr>
<td>7.7.1. Connecting a user GPIO model with the UT699 module</td>
<td>56</td>
</tr>
<tr>
<td>7.7.2. Commands</td>
<td>56</td>
</tr>
<tr>
<td>7.7.3. Debug flags</td>
<td>56</td>
</tr>
<tr>
<td>7.7.4. GPIO model API</td>
<td>56</td>
</tr>
<tr>
<td>7.8. CAN interface</td>
<td>57</td>
</tr>
<tr>
<td>7.8.1. Start up options</td>
<td>57</td>
</tr>
<tr>
<td>7.8.2. Commands</td>
<td>57</td>
</tr>
<tr>
<td>7.8.3. Debug flags</td>
<td>57</td>
</tr>
<tr>
<td>7.8.4. Packet server</td>
<td>58</td>
</tr>
<tr>
<td>7.8.5. CAN packet server protocol</td>
<td>58</td>
</tr>
<tr>
<td>8. Cobham UT699E emulation</td>
<td>60</td>
</tr>
<tr>
<td>8.1. Overview of the UT699E AHB module</td>
<td>60</td>
</tr>
<tr>
<td>8.2. Loading the module</td>
<td>61</td>
</tr>
<tr>
<td>8.2.1. User input module interface</td>
<td>61</td>
</tr>
<tr>
<td>8.3. Debugging</td>
<td>63</td>
</tr>
<tr>
<td>8.4. 10/100 Mbps Ethernet Media Access Controller interface</td>
<td>63</td>
</tr>
<tr>
<td>8.4.1. Start up options</td>
<td>63</td>
</tr>
<tr>
<td>8.4.2. Commands</td>
<td>63</td>
</tr>
<tr>
<td>8.4.3. Debug flags</td>
<td>63</td>
</tr>
<tr>
<td>8.4.4. Ethernet packet server</td>
<td>64</td>
</tr>
<tr>
<td>8.4.5. Ethernet packet server protocol</td>
<td>64</td>
</tr>
<tr>
<td>8.5. SpaceWire interface with RMAP support</td>
<td>64</td>
</tr>
<tr>
<td>8.5.1. Start up options</td>
<td>65</td>
</tr>
<tr>
<td>8.5.2. Commands</td>
<td>65</td>
</tr>
<tr>
<td>8.5.3. Debug flags</td>
<td>65</td>
</tr>
<tr>
<td>8.5.4. SpaceWire packet server</td>
<td>66</td>
</tr>
<tr>
<td>8.5.5. SpaceWire packet server protocol</td>
<td>66</td>
</tr>
<tr>
<td>8.5.6. Simple Mode</td>
<td>73</td>
</tr>
<tr>
<td>8.6. PCI initiator/target interface</td>
<td>74</td>
</tr>
<tr>
<td>8.6.1. Connecting a user PCI model with the UT699E module</td>
<td>74</td>
</tr>
<tr>
<td>8.6.2. Commands</td>
<td>74</td>
</tr>
<tr>
<td>8.6.3. Debug flags</td>
<td>74</td>
</tr>
<tr>
<td>8.6.4. PCI bus model API</td>
<td>74</td>
</tr>
</tbody>
</table>
8.7. GPIO interface .............................................................................................................. 75
  8.7.1. Connecting a user GPIO model with the UT699E module ........................................ 75
  8.7.2. Commands .............................................................................................................. 75
  8.7.3. Debug flags ............................................................................................................ 75
  8.7.4. GPIO model API ...................................................................................................... 76
8.8. CAN interface ................................................................................................................ 76
  8.8.1. Start up options ....................................................................................................... 76
  8.8.2. Commands ............................................................................................................... 76
  8.8.3. Debug flags ............................................................................................................. 77
  8.8.4. Packet server .......................................................................................................... 77
  8.8.5. CAN packet server protocol ..................................................................................... 77
9. Cobham UT700 emulation ................................................................................................. 80
  9.1. Overview of the UT700 AHB module ......................................................................... 80
  9.2. Loading the module .................................................................................................... 81
    9.2.1. User input module interface .................................................................................. 81
  9.3. Debugging .................................................................................................................... 83
  9.4. 10/100 Mbps Ethernet Media Access Controller interface ........................................ 83
    9.4.1. Start up options ..................................................................................................... 83
    9.4.2. Commands ............................................................................................................ 83
    9.4.3. Debug flags ......................................................................................................... 83
    9.4.4. Ethernet packet server ......................................................................................... 84
    9.4.5. Ethernet packet server protocol ............................................................................ 84
  9.5. SpaceWire interface with RMAP support ................................................................... 84
    9.5.1. Start up options ..................................................................................................... 85
    9.5.2. Commands ............................................................................................................ 85
    9.5.3. Debug flags ......................................................................................................... 86
    9.5.4. SpaceWire packet server ...................................................................................... 86
    9.5.5. SpaceWire packet server protocol ......................................................................... 86
    9.5.6. Simple Mode ......................................................................................................... 93
  9.6. PCI initiator/target interface ....................................................................................... 94
    9.6.1. Connecting a user PCI model with the UT700 module .......................................... 94
    9.6.2. Commands .......................................................................................................... 94
    9.6.3. Debug flags ....................................................................................................... 94
    9.6.4. PCI bus model API ............................................................................................. 94
  9.7. GPIO interface ............................................................................................................. 95
    9.7.1. Connecting a user GPIO model with the UT700 module ....................................... 95
    9.7.2. Commands .......................................................................................................... 95
    9.7.3. Debug flags ....................................................................................................... 95
    9.7.4. GPIO model API ................................................................................................. 96
  9.8. CAN interface ............................................................................................................. 96
    9.8.1. Start up options .................................................................................................... 96
    9.8.2. Commands .......................................................................................................... 96
    9.8.3. Debug flags ....................................................................................................... 97
    9.8.4. Packet server ..................................................................................................... 97
    9.8.5. CAN packet server protocol ............................................................................... 97
  9.9. SPI interface ............................................................................................................... 99
    9.9.1. Connecting a user SPI model with the UT700 module ......................................... 99
    9.9.2. Commands ......................................................................................................... 99
    9.9.3. Debug flags ..................................................................................................... 99
    9.9.4. SPI bus model API ............................................................................................. 99
10. Cobham Gaisler GR712RC emulation ........................................................................... 101
  10.1. Overview of the GR712RC AHB module ................................................................. 101
  10.2. Loading the module ................................................................................................ 102
    10.2.1. User input module interface .............................................................................. 102
  10.3. Debugging ................................................................................................................. 103
  10.4. CAN interface .......................................................................................................... 104
    10.4.1. Start up options .................................................................................................. 104
    10.4.2. Commands ....................................................................................................... 104
10.4.3. Debug flags ............................................................... 104
10.4.4. Packet server .......................................................... 105
10.4.5. CAN packet server protocol ...................................... 105
10.5. 10/100 Mbps Ethernet Media Access Controller interface ........ 106
10.5.1. Start up options ....................................................... 106
10.5.2. Commands ............................................................. 107
10.5.3. Debug flags ........................................................... 107
10.5.4. Ethernet packet server ............................................. 107
10.5.5. Ethernet packet server protocol .................................. 107
10.6. SpaceWire interface with RMAP support ............................. 108
10.6.1. Start up options ....................................................... 108
10.6.2. Commands ............................................................. 109
10.6.3. Debug flags ........................................................... 109
10.6.4. SpaceWire packet server ............................................ 110
10.6.5. SpaceWire packet server protocol .................................. 110
10.6.6. Simple Mode ......................................................... 117
10.7. SPI interface .............................................................. 118
10.7.1. Connecting a user SPI model with the GR712RC module .......... 118
10.7.2. Commands ............................................................. 118
10.7.3. Debug flags ........................................................... 118
10.7.4. SPI bus model API .................................................... 118
10.8. GPIO interface ........................................................... 119
10.8.1. Connecting a user GPIO model with the GR712RC module .......... 119
10.8.2. Commands ............................................................. 119
10.8.3. Debug flags ........................................................... 119
10.8.4. GPIO model API ....................................................... 119
10.9. Clock Gating Unit, CANMUX and GRGPREG ......................... 120
11. Atmel AT697 emulation .................................................... 121
11.1. Overview of the AT697 AHB module ................................ 121
11.2. Loading the module ..................................................... 122
11.3. AT697 initiator mode ................................................... 122
11.4. AT697 target mode ...................................................... 122
11.5. Definitions .............................................................. 122
11.5.1. PCI command table .................................................. 123
11.6. Read/write function installed by PCI module ......................... 123
11.7. Read/write function installed by AT697 module .................... 123
11.8. Registers ............................................................... 124
11.9. Debug flags ............................................................. 125
11.10. Commands ............................................................. 125
12. TPS VxWorks 6.x AHB Module .......................................... 126
12.1. Overview ............................................................... 126
12.2. Loading the module ..................................................... 126
12.3. Configuration ........................................................... 126
13. Support ........................................................................... 127
1. Introduction

1.1. General

TSIM is a generic SPARC architecture simulator capable of emulating ERC32- and LEON-based computer systems.

TSIM provides several unique features:

• Emulation of ERC32 and LEON2/3/4 processors (in single processor systems)
• Superior performance: up to 60 MIPS on high-end PC (Intel i7-2600K @3.4GHz)
• Accelerated processor standby mode, allowing faster-than-realtime simulation speeds
• Standalone operation or remote connection to GNU debugger (gdb)
• Also provided as library to be included in larger simulator frameworks
• 64-bit time for practically unlimited simulation periods
• Instruction trace buffer
• EDAC emulation (ERC32)
• MMU emulation (LEON2/3/4)
• SRAM emulation and functional emulation of SDRAM (with SRAM timing) (LEON2/3/4)
• Local scratch-pad RAM (LEON3/4)
• Loadable modules to include user-defined I/O devices
• Non-intrusive execution time profiling
• Code coverage monitoring
• Instruction trace buffer
• Stack backtrace with symbolic information
• Check-pointing capability to save and restore complete simulator state
• Unlimited number of breakpoints and watchpoints
• Pre-defined functional simulation modules for GR712RC, UT699, UT700 and AT697

1.2. Supported platforms and system requirements

TSIM supports the following platforms: Solaris 2.8, Linux, Linux-x64, Windows XP/7, and Windows XP/7 with Cygwin Unix emulation.

1.3. Obtaining TSIM

The primary site for TSIM is the Cobham Gaisler website [http://www.gaisler.com] where the latest version of TSIM can be ordered and evaluation versions downloaded.

1.4. License

TSIM2 ERC32, TSIM2 LEON2, TSIM2 LEON3 and TSIM2 LEON4 are licensed separately as separate products. The license can be found in license.txt in the top directory after installation. See Section 2.2 for details on the license solution.

1.5. Evaluation version

An evaluation version of TSIM2 LEON3 is available from the Cobham Gaisler website [http://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 is limited to 32-bit time. It does not support check-pointing, loadable modules, library interface, code coverage, configuration of caches, configuration of memory or chip flags such as -gr712rc or -ut700.

1.6. Problem reports

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

---

1SPARC is a registered trademark of SPARC International
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 `-logfile filename`. Try to include as much details as possible from commands such as `reg`, `hist/inst` (enable history with `hist len`), `bt` and with relevant debug options turned on. See also Chapter 13.
2. Installation

2.1. General

TSIM is distributed as a tar-file (e.g. tsim-erc32-2.0.66.tar.gz) with the following contents:

<table>
<thead>
<tr>
<th>Directory</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>coverage</td>
<td>Source level coverage helper scripts</td>
</tr>
<tr>
<td>doc</td>
<td>TSIM documentation</td>
</tr>
<tr>
<td>samples</td>
<td>Sample programs</td>
</tr>
<tr>
<td>iomod</td>
<td>Example loadable modules</td>
</tr>
<tr>
<td>tsim/cygwin</td>
<td>TSIM binary for cygwin</td>
</tr>
<tr>
<td>tsim/linux</td>
<td>TSIM binary for linux</td>
</tr>
<tr>
<td>tsim/linux-x64</td>
<td>TSIM binary for linux-x64</td>
</tr>
<tr>
<td>tsim/solaris</td>
<td>TSIM binary for solaris</td>
</tr>
<tr>
<td>tsim/win32</td>
<td>TSIM binary for native Windows</td>
</tr>
<tr>
<td>tlib/cygwin</td>
<td>TSIM library for cygwin</td>
</tr>
<tr>
<td>tlib/linux</td>
<td>TSIM library for linux</td>
</tr>
<tr>
<td>tlib/linux-x64</td>
<td>TSIM library for linux-x64</td>
</tr>
<tr>
<td>tlib/solaris</td>
<td>TSIM library for solaris</td>
</tr>
<tr>
<td>tlib/win32</td>
<td>TSIM library for native Windows</td>
</tr>
</tbody>
</table>

The tar-file can be installed at any location with the following command:

```
gunzip -c tsim-erc32-2.0.66.tar.gz | tar xf -
```

2.2. License installation

TSIM is licensed using a HASP USB hardware key. Before use, the Sentinel HASP runtime must be installed. See the simulator download page at the Cobham Gaisler website [http://www.gaisler.com] for download links.

2.2.1. Floating keys

Floating keys can be indentified by having a red housing (instead of blue/purple).

Floating license keys requires that the runtime is installed in both client and server. In addition the server also need to have a license manager installed. The license manager software can be also be downloaded from the simulator download page. Note that this is a separate license manager from the license manager that is included with the HASP runtime.

The Linux install script is outdated and will fail on many modern distributions. The following workaround works for Ubuntu 16.04 and 18.04:

```
$ sudo RUNLEVELDIR=/etc/rc2.d bash ./dinst .
```

The license manager can also be started manually by running the hasplm executable.

Note that a limitation of the floating keys normally used for TSIM2, is that only one such key can be used per license server. If more than one floating key is needed for the same server, there is an option to switch to a LDK type key option (that can be accommodated by the same HASP USB hardware key), see below.

2.2.2. LDK type license keys

Some situations warrants an LDK type license key. Note that these keys does not support running TSIM under Cygwin, and does not support running TSIM versions prior to 2.0.66.
3. Operation

3.1. Overview

TSIM can operate in two modes: standalone and attached to gdb. In standalone mode, ERC32 or LEON applications can be loaded and simulated using a command line interface. A number of commands are available to examine data, insert breakpoints and advance simulation. When attached to gdb, TSIM acts as a remote gdb target, and applications are loaded and debugged through gdb (or a gdb front-end such as ddd).

3.2. Starting TSIM

TSIM is started as follows on a command line:

- `tsim-erc32 [options] [input_files]`
- `tsim-leon [options] [input_files]`
- `tsim-leon3 [options] [input_files]`
- `tsim-leon4 [options] [input_files]`

The following command line options are supported by TSIM:

- `-ahbm ahb_module`
  Use `ahb_module` as loadable AHB module rather than the default `ahb.so` (LEON only). If multiple `-ahbm` switches are specified, up to 16 AHB modules can be loaded. The environmental variable `TSIM_MODULE_PATH` can be set to a `:` separated (`;` in WIN32) list of search paths.

- `-ahbstatus`
  Adds AHB status register support.

- `-asilnoallocate`
  Makes ASI 1 reads not allocate cache lines (LEON3/4 only).

- `-at697e`
  Configure caches according to the Atmel AT697E device (LEON2 only). See Chapter 11 for details on AT697 emulation.

- `-banks ram_banks`
  Sets how many RAM banks the SRAM is divided on. Supported values are 1, 2 or 4. Default is 1. (LEON only).

- `-bopt`
  Enables idle-loop optimisation (see Section 4.1.5).

- `-bp`
  Enables emulation of LEON3/4 branch prediction

- `-bz`
  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 the -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.

- `-c file`
  Reads commands from `file` and executes them at startup.

- `-cfg file`
  Reads extra configuration options from `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`.

- `-covtrans`
  Enable MMU translations for the coverage system. Needed when MMU is enabled and not mapping 1-to-1.

- `-cpm cp_module`
  Use `cp_module` as loadable co-processor module file name (LEON). The environmental variable `TSIM_MODULE_PATH` can be set to a `:` separated (`;` in WIN32) list of search paths.

- `-cas (-nocas)`
  When running a VXWORKS SMP image the SPARC V9 “casa” instruction is used. The option `-cas` enables this instruction (LEON3/4 only). The `-nocas` option can disable CAS support when otherwise already enabled.
-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.

-d lock
  Enable data cache line locking. Default is disabled. (LEON only).

-dram addr size
  Allocates size KiB of local dcache scratchpad memory at address addr. (LEON3/4)

-dls size
  Sets the line size of the LEON data cache (in bytes). Allowed values are 16 or 32. Default is 16.

-drepl repl
  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.

-eclipse
  Enable some special handling of the GDB protocol when connecting with Eclipse.

-exc2b
  Issue 0x2b memory exception on memory write store error (LEON2 only)

-ext nr
  Enable extended irq ctrl with extended irq number nr (LEON3/4 only)

-fast_uart
  Run UARTs at infinite speed, rather than with correct baud rate.

-fpm fp_module
  Use fp_module as loadable FPU module rather than a built in FPU model or looking for the default fp.so/dll module (LEON only). The environmental variable TSIM_MODULE_PATH can be set to a ':' separated (";" in WIN32) list of search paths.

-freq system_clock
  Sets the simulated system clock (MHz). Will affect UART timing and performance statistics. Default is 14 for ERC32 and 50 for LEON.

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

-gdbuartfwd
  Forward output from first UART to GDB.

-gr702rc
  Set cache parameters to emulate the GR702RC device.

-gr712rc
  Set parameters to emulate the GR712RC device (albeit as a single processor device). Must be used when using the GR712RC AHB module. This also sets up TSIM to simulate 6 APBUART cores and the GR-TIMER core. See Chapter 10 for details on GR712RC emulation.

-grfpu
  Emulate the GRFPU floating point unit, rather then Meiko or GRFPU-lite (LEON only).

-hwbp
  Use TSIM hardware breakpoints for gdb breakpoints.

-stack addr
  Set initial stack pointer.

-i
  Forces bootloader-like initialization even when running from ROM. See the run and go commands for details. See also the -ni option.

-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.

-ift
  Generate illegal instruction trap on IFLUSH. Emulates the IFT input on the ERC32 processor.

-ilock
  Enable instruction cache line locking. Default is disabled.
-ilram addr size
  Allocates size bytes of local icache scratchpad memory at address addr. (LEON3/4)

-ilsize size
  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.

-lom io_module
  Use io_module as loadable I/O module rather than the default io.so. The environmental variable TSIM_MODULE_PATH can be set to a ':' separated (';' in WIN32) list of search paths.

-irepl repl
  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.

-i sets sets
  Defines the number of sets in the LEON instruction cache. Allowed values are 1(default) - 4.

-iwde
  Set the IWDE input to 1. Default is 0. (TSC695E only)

-l2wsize size
  Enable emulation of L2 cache (LEON4 only) with size KiB. The size must be binary aligned (e.g. 16, 32, 64 ...).

-logfile filename
  Logs the console output to filename. If filename is preceded by '+' output is append.

-mcfgX value
  Set the reset value of memory configuration register X, where X can be 1, 2 or 3 (LEON only).

-mfailok
  Do not fail on startup even if explicitly requested io/ahb modules fails to load.

-mflat
  This switch should be used when the application software has been compiled with the gcc -mflat option, and debugging with gdb is done.

-mmum / -nommu
  Enables or disables MMU support (LEON only). By default LEON2 and LEON3 does not have MMU support, and LEON4 has MMU support. Chip options, e.g. -gr712rc, enables MMU support when the corresponding chip has it.

-nb
  Do not break on error exceptions when debugging through GDB. This also enables the -hwbp option.

-nfp
  Disables the FPU to emulate system without FP hardware. Any FP instruction will generate an FP disabled trap.

-ni
  Prevents bootloader-like initialization even when not running from ROM. See the run and go commands for details. See also the -i option. This option also affect restarting execution via GDB. See gdb reset for details.

-nomac
  Disable LEON MAC instruction. (LEON only).

-noeditline
  Disable use of editline for history and tab completion.

-nosram
  Disable SRAM on startup. SDRAM will appear at 0x40000000 (LEON only).

-nothreads
  Disable threads support.

-nouart
  Disable emulation of UARTs. All access to UART registers will be routed to the I/O module.

-nov8
  Disable SPARC V8 MUL/DIV instructions (LEON only).

-nrtimers val
  Adds support for more than 2 timers. Value val can be in the range of 2 - 7 (LEON3/4 only). Default: 2. See also the -sametimerirq and -timerirqbase number switches.
-numbp num
Sets the upper limit on number of possible breakpoints.

-numwp num
Sets the upper limit on number of possible watchpoints.

-nwin win
Defines the number of register windows in the processor. The default is 8. Only applicable to LEON3/4.

-port portnum
Use portnum for gdb communication (port 1234 is default)

-pr
Enable profiling.

-ram ram_size
Sets the amount of simulated RAM (KiB). Default is 4096.

-rest file_name
Restore saved state from file_name.tss. See Section 3.8.

-rom rom_size
Sets the amount of simulated ROM (KiB). Default is 2048.

-rom8, -rom16
By default, the PROM area at reset time is considered to be 32-bit. Specifying -rom8 or -rom16 will initialise the memory width field in the memory configuration register to 8- or 16-bits. The only visible difference is in the instruction timing.

-rtems ver
Override autodetected RTEMS version for thread support. ver should be 46, 48, 48-edisoft or 410.

-sametimerirq
Force the irq number to be the same for all timers. Default: separate increasing irqs for each timer. (LEON3/4 only). See also the -nrtimers val and -timerirqbase number switches.

-sdram sdram_size
Sets the amount of simulated SDRAM (MiB). Default is 32. (LEON only)

-sdbanks <1|2>
Sets the SDRAM banks. This parameter is used to calculate the used SDRAM in conjunction with the mcfg2.sdramsize field. The actually used SDRAM at runtime is sdbanks*mcfg2.sdramsize. Default:1 (LEON only)

-sym file
Read symbols from file. Useful for self-extracting applications

-timer32
Use 32 bit timers instead of 24 bit. (LEON2 only)

-timerirqbase number
Set the irq number of the first timer to interrupt number number (LEON3/4 only). Default: 8. See also the -nrtimers val and -sametimerirq switches.

-tsc691
Emulate the TSC691 device, rather than TSC695

-tsc695e
Obsolete. TSIM/ERC32 now always emulates the TSC695 device rather that the early ERC32 chip-set.

-uartX device
This option connects the chosen UART to a serial device. Here, X can be in the range 1 up to the number of uarts. By default, UART1 is connected to stdin/stdout and all others are disconnected.

On Linux, e.g. connecting the first uart to /dev/ttyUSB0 can be done with “-uart1 /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>
UART FIFO depth in characters (LEON3/4 only). This setting affects all APBUARTs in the system. Valid configurations are 1 (default), 2, 4, 8, 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.

\textbf{-ut699}\n
Set parameters to emulate the UT699 device. Must be used when using the UT699 AHB module. Note that when \textbf{-ut699} is given, snooping will be set as non-functional. This also sets up TSIM to simulate only one APBUART core. See Chapter 7 for details on UT699 emulation.

\textbf{-ut699e}\n
Set parameters to emulate the UT699E device. Must be used when using the UT699E AHB module. This also sets up TSIM to simulate only one APBUART core. See Chapter 8 for details on UT699E emulation.

\textbf{-ut700}\n
Set parameters to emulate the UT700 device. Must be used when using the UT700 AHB module. This also sets up TSIM to simulate only one APBUART core. See Chapter 9 for details on UT700 emulation.

\textbf{-wdfreq freq}\n
Specify the frequency of the watchdog clock. (ERC32 only)

\textbf{input_files}\n
Executable files to be loaded into memory. The input file is loaded into the emulated memory according to the entry point for each segment. Recognized formats are elf32, aout and srecords.

Command line options can also be specified in the file .tsimcfg in the home directory. This file will be read at startup and the contents will be appended to the command line.

### 3.3. Standalone mode commands

If the file .tsimrc exists in the home directory, it will be used as a batch file and the commands in it will be executed at startup.

Below is a description of commands that are recognized by the simulator when used in standalone mode:

- **batch file**
  Execute a batch file of TSIM commands.

- **load file[startaddr]**
  Load the binary file \textit{file} into memory starting at \textit{startaddr}. The default \textit{startaddr} is the start of RAM memory.

- **+bp, break address**
  Adds an breakpoint at \textit{address}.

- **bp, break**
  Prints all breakpoints and watchpoints.

- **-bp, del [num]**
  Deletes breakpoint/watchpoint \textit{num}. If \textit{num} is omitted, all breakpoints and watchpoints are deleted.

- **bt**
  Print backtrace.

- **cont [instructions|amount timeunit]**
  Continue execution at present position, optionally for a number of instructions or an amount of time. See Section 3.3.1 for the syntax for specifying time.

- **coverage <enable | disable | save [file_name] | clear | print address [len]>**
  Code coverage control. Coverage can be enabled, disabled, cleared, saved to a file or printed to the console.

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

- **dis [addr][count]**
  Disassemble [count] instructions at address [addr]. Default values for count is 16 and \textit{addr} is the program counter address.

- **echo string**
  Print \textit{string} to the simulator window.

- **edac [clear | cerr | merr address]**
  Insert EDAC errors, or clear EDAC checksums (ERC32 only)

- **ep [clear][address]**
  Show, clear or set entry point for execution. Setting the entry point overrides the default start of execution address for \texttt{run} and \texttt{go} commands.
event
Print events in the event queue. Only user-inserted events are printed.

flush [all | icache | dcache | addr]
Flush the LEON caches. Specifying all will flush both the icache and dcache. Specifying icache or dcache will flush the respective cache. Specifying addr will flush the corresponding line in both caches.

float [-v]
Prints the FPU registers. With the optional -v argument, the fields of the fsr registers are listed and denormalized numbers are marked.

gdb
Listen for gdb connection.

gdb reset
Prepares TSIM for a new run via GDB. This is needed before loading an image from GDB (or via GDB e.g. from Eclipse). An exception to this need is when using the extended-remote target type with remote exec-file and starting each new execution via the GDB “run” command. Unless the -ni option is used, bootloader-like initialisation is always performed.

go [address/symbol | -t] [instructions | amount timeunit]
The go command will continue execution without restarting the simulation. If an address or symbol is given, execution starts from there. Otherwise, if an entry point has been set with the ep command, execution starts from that entry point. Otherwise execution starts from the entry point of the last loaded image.

If the simulation time is at 0 (e.g. due to TSIM having just been started or due to the reset command) and the address execution is being started from is not the reset address (i.e. address 0), TSIM sets up memory controllers, timers and the like normally done by a bootloader in order to be able to run memory images without a bootloader. Starting TSIM with the -i option forces such an initialization even when starting from the reset address (but still only when the time is 0) and starting TSIM with the -ni option prevents TSIM from doing such initialization in any case.

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 3.3.1 for the syntax for specifying time.

help
Print a small help menu for the TSIM commands.

hist [length|-f file]
hist length enables (or disables) the instruction trace buffer. The length last executed instructions will be placed in the trace buffer. Specifying a zero trace length will disable the trace buffer.

hist with no arguments will display the trace buffer. See the inst [length] command for displaying only a part of the instruction trace buffer.

hist -f filename will write the history trace to file.

icache, dcache
Dumps the contents of tag and data cache memories (LEON only).

inc cycles | amount timeunit
Increment simulator time without executing instructions for a number of cycles or an amount of time. The event queue is evaluated during the advancement of time. See Section 3.3.1 for the syntax for specifying time.

inst [length]
Display the latest length (default 30) instructions in the instruction trace buffer. See the hist [length]-f file] command for how to enable the instruction trace buffer.

leon
Display LEON peripherals registers.

load files
Load files into simulator memory.

l2cache
Display contents of L2 cache. (LEON4 only)

mcfgX [value]
Set or show the user defined reset value of memory configuration register X, where X can be 1, 2 or 3 (LEON only).
mec
Display ERC32 MEC registers.

mem [addr] [count]
Display memory at addr for count bytes. Same default values as for dis. Unimplemented registers are displayed as zero.

vmem [vaddr] [count]
Same as mem but does a MMU translation on vaddr first (LEON only).

mmu
Display the MMU registers (LEON only).

quit
Exits the simulator.

perf [reset]
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. The run and reset command also resets the statistic information.

prof [0|1] [stime]
Enable (‘prof 1’) or disable (‘prof 0’) profiling. Without parameters, profiling information is printed. Default sampling period is 1000 clock cycles, but can be changed by specifying stime.

reg [reg_name] [value] | window
Prints and sets the IU registers in the current register window, prints and sets individual registers and prints other register windows. reg without parameters prints the IU registers of the current register window. reg reg_name shows the value of the corresponding register. Valid register names are psr, tbr, wim, y, pc, npc, fsr, gl-g7, o0-o7, i0-i7, f0-f31. reg reg_name value sets the corresponding register to value. To view a certain register window, use reg wn, where n is the index of the register window.

reset
Performs a power-on reset without starting any execution.

restore file
Restore simulator state from file.

run [address/symbol | -t ] [instructions | amount timeunit]
Resets the simulator and starts simulation from time 0. The event queue is emptied but any set breakpoints remain. If an address or symbol is given, execution starts from there. Otherwise, if an entry point has been set with the ep command, execution starts from that entry point. Otherwise execution starts from the reset address (i.e. address 0) if a binary is loaded there or from the entry point from the last loaded image if the memory at the reset address is zero.

If execution starts from the reset address, no bootloader-like initialization of the system is done. Otherwise TSIM sets up memory controllers, timers and the like normally done by a bootloader in order to be able to run memory images without a bootloader. Starting TSIM with the –i option forces such an initialization even when starting from the reset address and starting TSIM with the –ni option prevents TSIM from doing such initialization in any case.

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 3.3.1 for the syntax for specifying time.

save file
Save simulator state to file.

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

stack [clear] address
Show, clear or set initial stack pointer. 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
Execute and disassemble one instruction. See also trace [instructions | amount timeunit].

sym [file]
Load symbol table from file. If file is omitted, prints current (.text) symbols.
### trace \([instructions] [amount timeunit]\)
Executes and disassembles instruction(s). Equivalent to the `cont` but with each instruction disassembled as it is executed. An optional number of instructions or amount of time to stop after can also be specified. See Section 3.3.1 for the syntax for specifying time.

### version
Prints the TSIM version and build date.

### walk address iswrite isid issu*
If the MMU is enabled printout a table walk for the given address. The flags iswrite, isid and issu are specifying the context: iswrite for a write access (default read), isid for an icache access (default dcache), issu for a supervisor access (default user).

### watch address
Adds a watchpoint at `address`.

### wmem, wmemh, wmemb address value
Write a word, half-word or byte directly to simulated memory.

### xwmem asi address value
Write a word to simulated memory using ASI=insi. Applicable to LEON3/4.

Typing a ‘Ctrl-C’ will interrupt a running simulator. Short forms of the commands are allowed, e.g. `c`, `co`, or `con`, are all interpreted as `cont`.

### 3.3.1. Time specification for commands

Commands such as `run`, `cont`, `go`, `trace` and `int` 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 (except for the `inc` command that will stop after that many cycles). 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>
<thead>
<tr>
<th>Argument</th>
<th>Unit</th>
</tr>
</thead>
<tbody>
<tr>
<td>c</td>
<td>cycles</td>
</tr>
<tr>
<td>us</td>
<td>microseconds</td>
</tr>
<tr>
<td>ms</td>
<td>milliseconds</td>
</tr>
<tr>
<td>s</td>
<td>seconds</td>
</tr>
<tr>
<td>min</td>
<td>minutes</td>
</tr>
<tr>
<td>h</td>
<td>hours</td>
</tr>
<tr>
<td>d</td>
<td>days</td>
</tr>
</tbody>
</table>

### 3.4. Symbolic debug information

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

```
tsim> break main
breakpoint 3 at 0x020012f0: main
```
```
tsim> dis strcmp 5
02002c04 84120009 or %o0, %o1, %g2
02002c08 8088a003 andcc %g2, 0x3, %g0
02002c0c 3280001a bne,a 0x02002c74
02002c10 c64a0000 ldsb [%o0], %g3
02002c14 c6020000 ld [%o0], %g3
```

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

```
tsim> sym /opt/rtems/src/examples/samples/dhry
read 234 symbols
```
```
tsim> sym
```
Reading symbols from alternate files is necessary when debugging self-extracting applications, such as bootpoms created with mkprom or linux/uClinux.

### 3.5. Breakpoints and watchpoints

TSIM supports execution breakpoints and write data watchpoints. In standalone mode, hardware breakpoints are always used and no instrumentation of memory is made. When using the gdb interface, the gdb ‘break’ command normally uses software breakpoints by overwriting the breakpoint address with a ‘ta 1’ instruction. Hardware breakpoints can be inserted by using the gdb ‘hbreak’ command or by starting tsim with -hwbp, which will force the use of hardware breakpoints also for the gdb ‘break’ command. Data write watchpoints are inserted using the ‘watch’ command. A watchpoint can only cover one word address, block watchpoints are not available.

### 3.6. Profiling

The profiling function calculates the amount of execution time spent in each subroutine of the simulated program. This is made without intervention or instrumentation of the code by periodically sample the execution point and the associated call tree. 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. Profiling is enabled through the `prof 1` 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 the `prof 1 n` command. Profiling can be disabled through the `prof 0` command. Below is an example profiling the dhrystone benchmark:

```bash
bash$tsim-erc32 /opt/rtems/src/examples/samples/dhry
tsim> pro 1
profiling enabled, sample period 1000
resuming at 0x02000000 runs through Dhrystone
Stopped at time 23375862 (1.670e+00 s)
function      samples      ratio(%)
start         36144        100.00
 _start        36144        100.00
main          36134         99.97
 Proc_1        10476         28.98
 Func_2         9885         27.34
 strcmp         8161         22.57
 Proc_8         2641          7.30
 .div          2097          5.80
 Proc_6         1412          3.90
 Proc_3         1321          3.65
 Proc_2         1187          3.28
 .umul         1092          3.02
 Func_1          777          2.14
 Proc_7          772          2.13
 Proc_4          731          2.02
 Proc_5          453          1.25
 Func_3          227          0.62
 printf          8          0.02
 vfprintf         8          0.02
 _vfprintf_r     8          0.02

```

### 3.7. Code coverage

To aid software verification, the professional version of 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. The coverage function is controlled by the coverage command:
coverage enable enable coverage
coverage disable disable coverage
coverage save [filename] write coverage data to file (filename optional)
coverage print address [len] print coverage data to console, starting at address
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 start
02000000 : 1 1 1 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 1 1 1 1 1 0 0 0 0
02000080 : 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
02000100 : 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
02000180 : 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
```

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.

NOTE: Only the internally emulated memory (PROM, SRAM and SDRAM) are subject for code coverage. Any memory emulated in the user’s I/O module must be handled by a user-defined coverage function.

The address ranges that are monitored are based on TSIM’s startup parameters. For instance, the range corresponding to the SDRAM for LEON will begin at address 0x40000000 if TSIM was started with -nosram or -ram 0, or will begin at 0x60000000 otherwise. Reconfiguration of the memory controller during execution will not be taken into account for monitored address ranges. Coverage information on memory reads will be recorded even for cache hits.

NOTE on MMU and coverage: The TSIM coverage system does no address translations by default, for performance reasons. To get physical address coverage when the MMU is is enabled and not mapping 1-to-1, use the -covtrans option. The monitored address ranges are based on the physical addresses where TSIM emulates ROM, SRAM or SDRAM. There is 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> di start
02000000 1 a0100000 clr $10
02000004 1 29008004 sethi $hi(0x2001000), $14
02000008 1 81c52000 jmp $14
0200000c 1 01000000 nop
02000010 0 91d02000 ta 0x0
02000014 0 01000000 nop
02000018 0 01000000 nop
```

The coverage data is not saved or restored during check-pointing operations. When enabled, the coverage function reduces the simulation performance of about 30%. When disabled, the coverage function does not impact simulation performance. Individual coverage fields can be read and written using the TSIM function interface using the tsim_coverage() call (see Section 6.2). Enabling and disabling the coverage functionality from the function interface should be done using tsim_cmd().

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

The professional version of TSIM can save and restore its complete state, allowing to resume simulation from a saved check-point. Saving the state is done with the `save` command:

    tsim> save file_name

The state is saved to `file_name.tss`. To restore the state, use the `restore` command:

    tsim> restore file_name

The state will be restored from `file_name.tss`. Restore directly at startup can be performed with the `-rest file_name` command line switch.

NOTE: TSIM command line options are not stored (such as alternate UART devices, etc.).

NOTE: AT697, UT699, UT700 and GR712RC simulation modules do not support check-pointing.

3.9. Performance

TSIM is highly optimised, and capable of simulating ERC32 systems faster than realtime. On high-end Athlon processors, TSIM achieves more than 1 MIPS / 100 MHz (CPU frequency of host). Enabling various debugging features such as watchpoints, profiling and code coverage can however reduce the simulation performance.

3.10. Backtrace

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

    tsim> bt
    %pc          %sp
    #0   0x0200190c   0x023ffcc8  Proc_1 + 0xf0
    #1   0x02001520   0x023ffd38  main + 0x230
    #2   0x02001208   0x023ffe00  _start + 0x60
    #3   0x02001014   0x023ffe40  start + 0x1014

3.11. Connecting to gdb

TSIM can act as a remote target for gdb, allowing symbolic debugging of target applications. To initiate gdb communication, start the simulator with the `-gdb` switch or use the TSIM `gdb` command:

    bash-2.04$ ./tsim -gdb

TSIM/LEON - remote SPARC simulator, build 2001.01.10 (demo version)
serial port A on stdin/stdout
allocated 4096 K RAM memory
allocated 2048 K ROM memory
gdb interface: using port 1234

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

    bash-2.04$ sparcrem-gdb t4.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 to set up TSIM for a new run and reload the program image into the simulator:

    (gdb) monitor gdb reset
    (gdb) load
    Loading section .text, size 0x14e50 lma 0x40000000
    Loading section .data, size 0x640 lma 0x400014e50
    Start address 0x40000000 , load size 87184
    Transfer rate: 697472 bits/sec, 278 bytes/write.
    (gdb) run
    The program being debugged has been started already.
    Start it from the beginning? (y or n) y
    Starting program: /home/jgais/src/gnc/t4.exe
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.

TSIM translates SPARC traps into (Unix) signals which are properly 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` 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 sym t4.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.

### 3.12. Thread support

TSIM has thread support for the RTEMS 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.

#### 3.12.1. 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.)

```plaintext

tsim> thread info

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

```

... | PC       | State      |
... | 0x40044bec _Thread_Dispatch + 0xd8 | READY    |
... | 0x40044bec _Thread_Dispatch + 0xd8 | SUSP     |
... | 0x40044bec _Thread_Dispatch + 0xd8 | READY    |
... | 0x40044bec _Thread_Dispatch + 0xd8 | Wevnt    |
... | 0x40006e28 printf + 0x4        | READY    |
... | 0x400044bec _Thread_Dispatch + 0xd8 | DELAY    |
... | 0x40044bec _Thread_Dispatch + 0xd8 | DELAY    |
```
thread bt  id prints a backtrace of a thread.

tsim> thread bt 0x0a010007

%pc %sp
#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 0x400366c 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

3.12.2 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 sym 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 Wenvt) 0x4002f760 in _Thread_Dispatch () at ../../../../../../rtems-4.6.5/cpukit/score/src/threaddispatch.c:109
7 Thread 167837701 (FTPa Wenvt) 0x4002f760 in _Thread_Dispatch () at ../../../../../../rtems-4.6.5/cpukit/score/src/threaddispatch.c:109
6 Thread 167837700 (DCtx Wenvt) 0x4002f760 in _Thread_Dispatch () at ../../../../../../rtems-4.6.5/cpukit/score/src/threaddispatch.c:109
5 Thread 16783699 (DCrx Wenvt) 0x4002f760 in _Thread_Dispatch () at ../../../../../../rtems-4.6.5/cpukit/score/src/threaddispatch.c:109
4 Thread 16783698 (ntwk ready) 0x4002f760 in _Thread_Dispatch () at ../../../../../../rtems-4.6.5/cpukit/score/src/threaddispatch.c:109
3 Thread 16783697 (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
  _Context_Switch( &amp;executing-&gt;Registers, &amp;heir-&gt;Registers );

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

(gdb) bt

#0 0x4002f760 in _Thread_Dispatch () at ../../../../../../rtems-dispatch.c:109
#1 0x40013e00 in rtems_event_receive (event_in=33554432, option_set=0, ticks=0, event_out=0x43fec14)
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, 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/ftpdc.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 0x43fecf0  1140772080
o2 0x43fececc  1140772076
o3 0x0  0
o4 0x34000e4  -213909276
o5 0x4007cc00  1074252800
sp 0x43fecd88  0x43fecd88
o7 0x40004020  1073758240
10 0x4007ce88  1074253448
11 0x4000e688  1074253448
12 0x400048fc  1073760508
13 0x43fecz0  1140772080
14 0x3  3
15 0x1  1
16 0x0  0
17 0x0  0
10 0x0  0
11 0x40003f94  1073758100
12 0x0  0
13 0x43fecz8  1140830152
14 0x0  0
15 0x4007ce40  1074253120
fp 0x43fecc0  0x43fecc0
17 0x40053380  1074082688
y 0x0  0
psr 0x34000e0  -213909280
vim 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.

### 3.13. 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 AHB module in the `iomod` 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.
4. Emulation characteristics

4.1. Common behaviour

4.1.1. Timing

The simulator time is maintained and incremented according to the IU and FPU instruction timing. 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 ERC32 and LEON MEIKO 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` or `hist` command will display the current simulator time in the left column. This time indicates when the instruction is fetched. 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, correct timing and fast mode. In the correct timing 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 with correct UART timing, it is possible that the last character of a program is not displayed on the console. This can happen if the program forces the 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 should 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.

4.1.2.1. APBUART model (LEON3/4 only)

The APBUART model used on LEON3 and LEON4 systems has support for receiver and transmitter FIFO mode also. In this mode the additional FIFO flags and level interrupts are also modelled like the APBUART IP. FIFO mode is enabled by setting the FIFO depth to a larger value than 1 with the `-uart_fs` switch. 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.

4.1.2.2. UART model (ERC32/LEON2 only)

The UART model of ERC32/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 simulator maps floating-point operations on the hosts floating point capabilities. This means that accuracy and generation of IEEE exceptions is sometimes host dependent and will not always be identical to the actual ERC32/LEON hardware. For GRFPU we have seen no discrepancies for any calculations or IEEE exceptions on
any host. On Windows and Linux hosts, the only known discrepancies for calculations or IEEE exceptions for the Meiko on LEON2 and GRFPU-lite are that NaN:s can differ in significand bits. No discrepancies has been seen in the signalling/quiet bit.

The models for the ERC32 FPU, GRFPU-lite and GRFPU models supports parallel IU and FPU execution, deferred floating point traps and the floating point deferred trap queue. The GRFPU model does not however simulate the possibility of multiple outstanding floating point operations. 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 simulator implements (to some extent) data-dependent execution timing for the ERC32 FPU, the Meiko FPU and GRFPU-lite. The complex timing of the GRFPU is not modelled in detail.

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 ERC32 and 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. Idle-loop optimisation

To minimise power consumption, LEON and ERC32 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 without 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.

4.1.6. 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 opcode and all processor registers in a pointer, allowing it to decode and emulate a custom instruction, and update the processor state.

The definition for the custom instruction handler is:

```c
int myhandler (struct ins_interface *r);
```

The pointer *r is a structure containing the current instruction opcode and processor state:

```c
struct ins_interface {
    uint32 psr;     /* Processor status registers */
    uint32 tbr;     /* Trap base register */
    uint32 wim;     /* Window mafs register */
    uint32 g[8];    /* Global registers */
    uint32 r[128];  /* Windowed register file */
    uint32 y;       /* Y register */
    uint32 pc;      /* Program counter */
    uint32 npc;     /* Next program counter */
    uint32 inst;    /* Current instruction */
    uint32 icnt;    /* Clock cycles in curr inst */
    uint32 asr17;
    uint32 asr18;
};
```

SPARC uses an overlapping windowed register file, and accessing registers must be done using the current window pointer (%psr[4:0]). To access registers %r8 - %r31 in the current window, use:
cwp = r->psr & 7;
regval = r->r[((cwp << 4) + RS1) % (nwindows * 16)];

Note that global registers (%r0 - %r7) should always be accessed by r->g[RS1].

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 r->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.

4.1.7. Chip-specific errata

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

4.2. ERC32 specific emulation

4.2.1. Processor emulation

TSIM/ERC32 emulates the behaviour of the TSC695 processor from Atmel by default. The parallel execution between the IU and FPU is modelled, as well as stalls due to operand dependencies (IU & FPU). Starting TSIM with the -tsc691 will enable TSC691 emulation (3-chip ERC32).

4.2.2. MEC emulation

The following table outlines the implemented MEC registers:

<table>
<thead>
<tr>
<th>Register</th>
<th>Address</th>
<th>Status</th>
</tr>
</thead>
<tbody>
<tr>
<td>MEC control register</td>
<td>0x01f80000</td>
<td>implemented</td>
</tr>
<tr>
<td>Software reset register</td>
<td>0x01f80004</td>
<td>implemented</td>
</tr>
<tr>
<td>Power-down register</td>
<td>0x01f80008</td>
<td>implemented</td>
</tr>
<tr>
<td>Memory configuration register</td>
<td>0x01f80010</td>
<td>partly implemented</td>
</tr>
<tr>
<td>IO configuration register</td>
<td>0x01f80014</td>
<td>implemented</td>
</tr>
<tr>
<td>Waitstate configuration register</td>
<td>0x01f80018</td>
<td>implemented</td>
</tr>
<tr>
<td>Access protection base register 1</td>
<td>0x01f80020</td>
<td>implemented</td>
</tr>
<tr>
<td>Access protection end register 1</td>
<td>0x01f80024</td>
<td>implemented</td>
</tr>
<tr>
<td>Access protection base register 2</td>
<td>0x01f80028</td>
<td>implemented</td>
</tr>
<tr>
<td>Access protection end register 2</td>
<td>0x01f8002c</td>
<td>implemented</td>
</tr>
<tr>
<td>Interrupt shape register</td>
<td>0x01f80044</td>
<td>implemented</td>
</tr>
<tr>
<td>Interrupt pending register</td>
<td>0x01f80048</td>
<td>implemented</td>
</tr>
<tr>
<td>Interrupt mask register</td>
<td>0x01f8004c</td>
<td>implemented</td>
</tr>
<tr>
<td>Interrupt clear register</td>
<td>0x01f80050</td>
<td>implemented</td>
</tr>
<tr>
<td>Interrupt force register</td>
<td>0x01f80054</td>
<td>implemented</td>
</tr>
<tr>
<td>Watchdog acknowledge register</td>
<td>0x01f80060</td>
<td>implemented</td>
</tr>
<tr>
<td>Watchdog trap door register</td>
<td>0x01f80064</td>
<td>implemented</td>
</tr>
<tr>
<td>RTC counter register</td>
<td>0x01f80080</td>
<td>implemented</td>
</tr>
<tr>
<td>RTC counter program register</td>
<td>0x01f80080</td>
<td>implemented</td>
</tr>
<tr>
<td>RTC scaler register</td>
<td>0x01f80084</td>
<td>implemented</td>
</tr>
<tr>
<td>Register</td>
<td>Address</td>
<td>Status</td>
</tr>
<tr>
<td>--------------------------------</td>
<td>------------</td>
<td>-----------------</td>
</tr>
<tr>
<td>RTC scaler program register</td>
<td>0x01f80084</td>
<td>implemented</td>
</tr>
<tr>
<td>GPT counter register</td>
<td>0x01f80088</td>
<td>implemented</td>
</tr>
<tr>
<td>GPT counter program register</td>
<td>0x01f80088</td>
<td>implemented</td>
</tr>
<tr>
<td>GPT scaler register</td>
<td>0x01f8008c</td>
<td>implemented</td>
</tr>
<tr>
<td>GPT scaler program register</td>
<td>0x01f8008c</td>
<td>implemented</td>
</tr>
<tr>
<td>Timer control register</td>
<td>0x01f80098</td>
<td>implemented</td>
</tr>
<tr>
<td>System fault status register</td>
<td>0x01f800A0</td>
<td>implemented</td>
</tr>
<tr>
<td>First failing address register</td>
<td>0x01f800A4</td>
<td>implemented</td>
</tr>
<tr>
<td>GPI configuration register</td>
<td>0x01f800A8</td>
<td>I/O module callback</td>
</tr>
<tr>
<td>GPI data register</td>
<td>0x01f800AC</td>
<td>I/O module callback</td>
</tr>
<tr>
<td>Error and reset status register</td>
<td>0x01f800B0</td>
<td>implemented</td>
</tr>
<tr>
<td>Test control register</td>
<td>0x01f800D0</td>
<td>implemented</td>
</tr>
<tr>
<td>UART A RX/TX register</td>
<td>0x01f800E0</td>
<td>implemented</td>
</tr>
<tr>
<td>UART B RX/TX register</td>
<td>0x01f800E4</td>
<td>implemented</td>
</tr>
<tr>
<td>UART status register</td>
<td>0x01f800E8</td>
<td>implemented</td>
</tr>
</tbody>
</table>

The MEC registers can be displayed with the `mec` command, or using `mem` (‘mem 0x1f80000 256’). The registers can also be written using `wmem` (e.g., ‘wmem 0x1f80000 0x1234’). When written, care has to be taken not to write an unimplemented register bit with ‘1’, or a MEC parity error will occur.

### 4.2.3. Interrupt controller

Internal interrupts are generated as defined in the MEC specification. All 15 interrupts can be tested via the interrupt force register. External interrupts can be generated through loadable modules.

### 4.2.4. Watchdog

The watchdog timer operate as defined in the MEC specification. The frequency of the watchdog clock can be specified using the `-wdfreq` switch. The frequency is specified in MHz.

### 4.2.5. Power-down mode

The power-down register (0x01f80008) is implemented as in the specification. A Ctrl-C in the simulator window will exit the power-down mode. In power-down mode, the simulator skips time until the next event in the event queue, thereby significantly increasing the simulation speed.

### 4.2.6. Memory emulation

The amount of simulated memory is configured through the `-ram` and `-rom` switches. The RAM size can be between 256 KiB and 32 MiB, the ROM size between 128 KiB and 4 MiB. Access to unimplemented MEC registers or non-existing memory will result in a memory exception trap.

The memory configuration register is used to decode the simulated memory. The fields RSIZ and PSIZ are used to set RAM and ROM size, the remaining fields are not used.

NOTE: After reset, the MEC is set to decode 128 KiB of ROM and 256 KiB of RAM. The memory configuration register has to be updated to reflect the available memory. The waitstate configuration register is used to generate waitstates. This register must also be updated with the correct configuration after reset.

### 4.2.7. EDAC operation

The EDAC operation of ERC32 is implemented on the simulated RAM area (0x2000000 - 0x2FFFFFF). The ERC32 Test Control Register can be used to enable the EDAC test mode and insert EDAC errors to test the
operation of the EDAC. The edac command can be used to monitor the number of errors in the memory, to insert new errors, or clear all errors. To see the current memory status, use the edac command without parameters:

```
tsim> edac
RAM error count : 2
0x20000000 : MERR
0x20000040 : CERR
```

TSIM keeps track of the number of errors currently present, and reports the total error count, the address of each error, and its type. The errors can either be correctable (CERR) or non-correctable (MERR). To insert an error using the edac command, do 'edac cerr addr' or 'edac merr addr':

```
tsim> edac cerr 0x2000000
correctable error at 0x02000000
```

To remove all injected errors, do edac clear. When accessing a location with an EDAC error, the behaviour of TSIM is identical to the real hardware. A correctable error will trigger interrupt 1, while un-correctable errors will cause a memory exception. The operation of the FSFR and FAR registers are fully implemented.

NOTE: The EDAC operation affect simulator performance when there are inserted errors in the memory. To obtain maximum simulation performance, any diagnostic software should remove all inserted errors after having performed an EDAC test.

4.2.8. Extended RAM and I/O areas

TSIM allows emulation of user defined I/O devices through loadable modules. EDAC emulation of extended RAM areas is not supported.

4.2.9. SYSAV signal

TSIM emulates changes in the SYSAV output by calling the command() callback in the I/O module with either “sysav 0” or “sysav 1” on each changes of SYSAV.

4.2.10. EXTINTACK signal

TSIM emulates assertion of the EXTINTACK output by calling the command() callback in the I/O module with “extintack” on each assertion. Note that EXTINTACK is only asserted for one external interrupt as programmed in the MEC interrupt shape register.

4.2.11. IWDE signal

The TSC695E processor input signal can be controlled by the -iwde switch at start-up. If the switch is given, the IWDE signal will be high, and the internal watchdog enabled. If -iwde is not given, IWDE will be low and the internal watchdog will be disabled. Note that the simulator must started in TSC695E-mode using the -tsc695e switch, for this option to take effect.

4.3. LEON2 specific emulation

4.3.1. Processor

The LEON2 version of TSIM emulates the behavior of the LEON2 VHDL model. The (optional) MMU can be emulated using the -mmu switch.

4.3.2. 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.
4.3.3. LEON peripherals registers

The LEON peripherals registers can be displayed with the `leon` command, or using `mem` (e.g. `mem 0x80000000 256`). The registers can also be written using `wmem` (e.g. `wmem 0x80000000 0x1234`).

4.3.4. 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.

4.3.5. Power-down mode

The power-down register (0x80000018) is implemented as in the specification. A Ctrl-C in the simulator window will exit the power-down mode. In power-down mode, the simulator skips time until the next event in the event queue, thereby significantly increasing the simulation speed.

4.3.6. 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.

4.3.7. SPARC V8 MUL/DIV/MAC instructions

TSIM/LEON optionally supports the SPARC V8 multiply, divide and MAC instruction. To correctly emulate LEON systems which do not implement these instructions, use the `-nomac` to disable the MAC instruction and/or `-nov8` to disable multiply and divide instructions.

4.3.8. 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.

4.3.9. DSU and hardware breakpoints

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

4.4. LEON3 specific emulation

4.4.1. General

The LEON3 version of TSIM emulates the behavior of the LEON3MP template VHDL model distributed in the GRLIB-1.0 IP library. The system includes the following modules: LEON3 processor, APB bridge, IRQMP interrupt controller, LEON2 memory controller, GPTIMER timer unit with two 32-bit timers, two APBUART uarts (unless a chip option changes this number). The AHB/APB plug&play information is provided at address 0xFFFFF000 - 0xFFFFFFFF (AHB) and 0x800FF000 - 0x800FFFFF (APB).

4.4.2. Processor

The instruction timing of the emulated LEON3 processor is modelled after LEON3 VHDL model in GRLIB IP library. The processor can be configured with 2 - 32 register windows using the `-nwin` switch. The MMU can be emulated using the `-mmu` switch. Local scratch pad RAM can be added with the `-ilram` and `-dlram` switches.
4.4.3. Cache memories

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

4.4.4. Power-down mode

The LEON3 power-down function is implemented as in the specification. A Ctrl-C in the simulator window will exit the power-down mode. In power-down mode, the simulator skips time until the next event in the event queue, thereby significantly increasing the simulation speed.

4.4.5. LEON3 peripherals registers

The LEON3 peripherals registers can be displayed with the leon command, or using mem (‘mem 0x80000000 256’). The registers can also be written using wmem (e.g. ‘wmem 0x80000000 0x1234’).

4.4.6. Interrupt controller

The IRQMP interrupt controller is fully emulated as described in the GRLIB IP Manual. The IRQMP registers are mapped at address 0x80000200. All 15 interrupts can also be generated from the user-defined I/O module using the set_irq() callback. When having extended interrupts enabled, all 31 interrupts can be generated. Extended interrupts can be enabled by the -ext option or a chip option for a chip that has extended interrupts (e.g. -gr712rc, -ut700 and -ut699e).

4.4.7. Memory emulation

The LEON2 memory controller is emulated in the LEON3 version of TSIM. 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 LEON3-FT is not implemented.

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

4.4.8. CASA instruction

The SPARCv9 “casa” instruction is implemented if the -cas switch is given. The “casa” instruction is used in VXWORKS SMP multiprocessing to synchronize using a lock free protocol.

4.4.9. SPARC V8 MUL/DIV/MAC instructions

TSIM/LEON3 optionally supports the SPARC V8 multiply, divide and MAC instruction. To correctly emulate LEON systems which do not implement these instructions, use the -nomac to disable the MAC instruction and/or -nov8 to disable multiply and divide instructions.

4.4.10. FPU emulation

By default, TSIM/LEON3 emulates the GRFPU-lite FPU. The -grfpu command line option enables the GRFPU model. 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

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. Furthermore, the HMASTER field is set to 0 when the CPU caused the error and 1 when any other master caused the error.

4.4.13. GRTIMER emulation

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

4.5. LEON4 specific emulation

4.5.1. General

The LEON4 version of TSIM emulates the behavior of the LEON4MP template VHDL model distributed in the GRLIB-1.0.x IP library. The system includes the following modules: LEON4 processor, APB bridge, IRQMP interrupt controller, LEON2 memory controller, L2 cache, GPTIMER timer unit with two 32-bit timers, two AP-BUART uarts. The AHB/APB plug&play information is provided at address 0xFFFFF000 - 0xFFFFFFFF (AHB) and 0x800FF000 - 0x800FFFFFF (APB).

4.5.2. Processor

The instruction timing of the emulated LEON4 processor is modelled after LEON4 VHDL model in GRLIB IP library. The processor can be configured with 2 - 32 register windows using the `-nwin` switch. The MMU can be emulated using the `-mmu` switch. Local scratch pad RAM can be added with the `-ilram` and `-dlram` switches.

4.5.3. 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.

4.5.4. L2 Cache memory

The LEON4 L2 cache is emulated, and placed between the memory controller and AHB bus. Both the PROM and SRAM/SDRAM areas are cached in the L2. The size of the L2 cache is default 64 KiB, but can be configured to any (binary aligned) size using the `-l2wsize` switch at start-up. Setting the size to 0 will disable the L2 cache. The L2 cache is implemented with one way and 32 bytes/line. The contents of the L2 cache can be displayed with the `l2cache` command.

4.5.5. Power-down mode

The LEON4 power-down function is implemented as in the specification. A Ctrl-C in the simulator window will exit the power-down mode. In power-down mode, the simulator skips time until the next event in the event queue, thereby significantly increasing the simulation speed.

4.5.6. LEON4 peripherals registers

The LEON4 peripherals registers can be displayed with the `leon` command, or using `mem` (‘mem 0x80000000 256’). The registers can also be written using `wmem` (e.g. `wmem 0x80000000 0x1234`).

4.5.7. Interrupt controller

The IRQMP interrupt controller is fully emulated as described in the GRLIB IP Manual. The IRQMP registers are mapped at address 0x80000200. All 15 interrupts can also be generated from the user-defined I/O module using
the `set_irq()` callback. When having extended interrupts enabled, all 31 interrupts can be generated. Extended interrupts can be enabled by the `-ext` option.

### 4.5.8. Memory emulation

The LEON2 memory controller is emulated in the LEON4 version of TSIM. 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 LEON4-FT is not implemented.

### 4.5.9. CASA instruction

The SPARCv9 “casa” instruction is implemented if the `-cas` switch is given. The “casa” instruction is used in VXWORKS SMP multiprocessing to synchronize using a lock free protocol.

### 4.5.10. SPARC V8 MUL/DIV/MAC instructions

TSIM/LEON4 optionally supports the SPARC V8 multiply, divide and MAC instruction. To correctly emulate LEON systems which do not implement these instructions, use the `-nomac` to disable the MAC instruction and/or `-nov8` to disable multiply and divide instructions.

### 4.5.11. FPU emulation

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

### 4.5.12. DSU and hardware breakpoints

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

### 4.5.13. AHB status registers

When using `-ahbstatus`, AHB status registers are enabled. As TSIM/LEON4 does not emulate FT, the CE bit will never be set. Furthermore, the HMASTER field is set to 0 when the CPU caused the error and 1 when any other master caused the error.
5. Loadable modules

5.1. TSIM I/O emulation interface

User-defined I/O devices can be loaded into the simulator through the use of loadable modules. 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. This is implemented through the module interface described below. The interface is made up of two parts; one that is exported by TSIM and defines TSIM functions and data structures that can be used by the I/O device; and one that is exported by the I/O device and allows TSIM to access the I/O device. Address decoding of the I/O devices is straight-forward: All access that do not map on any AHB module (LEON only) or the internally emulated memory and control registers are forwarded to the I/O module.

TSIM exports two structures: simif and ioif. The simif structure defines functions and data structures belonging to the simulator core, while ioif defines functions provided by system (ERC32/LEON) emulation. At startup, TSIM searches for `io.so` in the current directory, but the location of the module can be specified using the `-iom` switch.

Note that the module must be compiled to be position-independent, i.e. with the `-fPIC` switch. The win32 version of TSIM loads io.dll instead of io.so. See the iomod directory in the TSIM distribution for an example io.c and how to build the .so and .dll modules. The environmental variable TSIM_MODULE_PATH can be set to a `:` separated (`;` in WIN32) list of search paths.

5.1.1. simif structure

The simif structure is defined in tsim.h:

```c
struct simif {
    struct sim_options *options; /* tsim command-line options */
    uint64 *simtime;              /* current simulator time */
    void (*event)(void (*cfunc)(), uint32 arg, uint64 offset);
    void (*stop_event)(void (*cfunc)());
    int *irl;                     /* interrup request level */
    void (*sys_reset)();         /* reset processor */
    void (*sim_stop)();          /* stop simulation */
    char *args;                  /* concaterated argv */
    /* Restorable events */
    unsigned short (*reg_revent)(void (*cfunc) (unsigned long arg));
    unsigned short (*reg_revent_prearg)(void (*cfunc) (unsigned long arg),
                                        unsigned long arg);
    int (*revent)(unsigned short index, unsigned long arg, uint64 offset);
    int (*revent_prearg)(unsigned short index, uint64 offset);
    void (*stop_revent)(unsigned short index);
    int (*lprintf)(const char *format, ...); /* logged formatted output */
    int (*vlprintf)(const char *format, va_list ap); /* logged formatted output */
};
```

The elements in the structure has the following meaning:

- `struct simif simif;` /* exported simulator functions */

    - `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;` Contains 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 (*event)(void (*cfunc)(), int 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 NOT be called from a signal handler installed by the I/O module, but only from event callbacks or at start of simulation. The event queue can hold a maximum of 2048 events.

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

```c
void (*stop_event)(void (*cfunc)());
```

`stop_event()` will remove all events from the event queue which has the calling function equal to `cfunc()`.

NOTE: The `stop_event()` function may NOT be called from a signal handler installed by the I/O module.

```c
int *irl;
```

Current IU interrupt level. Should not be used by I/O functions unless they explicitly monitor these lines.

```c
void (*sys_reset)();
```

Performs a system reset. Should only be used if the I/O device is capable of driving the reset input.

```c
void (*sim_stop)();
```

Stops current simulation. Can be used for debugging purposes if manual intervention is needed after a certain event.

```c
char *args;
```

Arguments supplied when starting tsim. The arguments are concatenated as a single string.

```c
void (*stop_event_arg)(void (*cfunc)(), int arg, int op);
```

Similar to `stop_event()` but differentiates between 2 events with same `cfunc` but with different `arg` given when inserted into the event queue via `event()`. Used when simulating multiple instances of an entity. Parameter `op` should be 1 to enable the `arg` check.

```c
unsigned short (*reg_revent)(void (*cfunc) (unsigned long 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 I/O or AHB module initialization.

```c
unsigned short (*reg_revent_prearg)(void (*cfunc) (unsigned long arg), unsigned long 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 I/O or AHB module initialization.

```c
int (*revent)(unsigned short index, unsigned long 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.

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

```c
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.

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

```c
void (*stop_revent)(unsigned short index);
```

This removes all events from the event queue that has been entered by `revent()` or `revent_prearg()` using the given `index`.

NOTE: The `stop_revent()` function may not be called from a signal handler installed by the I/O module.

```c
int (*lprintf)(const char *format, ...)
```

Function for logged formatted output. The function interface works like for `printf`.

```c
int (*vlprintf)(const char *format, va_list ap)
```

Function for logged formatted output. The function interface works like for `vprintf`.
5.1.2. ioif structure

ioif is defined in tsim.h:

```c
struct io_interface {
    void (*set_irq)(int irq, int level);
    int (*dma_read)(uint32 addr, uint32 *data, int num);
    int (*dma_write)(uint32 addr, uint32 *data, int num);
    int (*dma_write_sub)(uint32 addr, uint32 *data, int sz);
};
extern struct io_interface ioif; /* exported processor interface */
```

The elements of the structure have the following meaning:

- `void (*set_irq)(int irq, int level);`
  - ERC32 use: drive the external MEC interrupt signal. Valid interrupts are 0 - 5 (corresponding to MEC external interrupt 0 - 4, and EWDINT) and valid levels are 0 or 1. Note that the MEC interrupt shape register controls how and when processor interrupts are actually generated. When `-nouart` has been used, MEC interrupts 4, 5 and 7 can be generated by calling `set_irq()` with irq 6, 7 and 9 (level is not used in this case.

- `LEON use: set the interrupt pending bit for interrupt irq. Valid values on irq is 1 - 15 for systems without extended interrupts and 1-31 for systems with extended interrupts. Care should be taken not to set interrupts used by the LEON emulated peripherals. Note that the LEON interrupt control register controls how and when processor interrupts are actually generated. Note that the level parameter is not used for LEON.

- `int (*dma_read)(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. For ERC32, the DMA transfer uses the external DMA interface. For LEON, DMA takes place on the AMBA AHB bus.

- `int (*dma_write)(uint32 addr, uint32 *data, int num);`
  - Performs DMA transactions to/from the emulated processor memory on the AMBA AHB bus. Available for LEON only. 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.

5.1.3. Structure to be provided by I/O device

```c
struct io_subsystem {
    void (*io_init)(struct sim_interface sif, struct io_interface iif); /* 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);
    void (*command)(char * cmd); /* I/O specific commands */
    void (*sigio)();/* Not used */
    void (*save)(char *fname);/* save simulation state */
    void (*restore)(char *fname);/* restore simulation state */
};
extern struct io_subsystem *iosystem; /* imported I/O emulation functions */
```

The elements of the structure have the following meanings:

- `void (*io_init)(struct sim_interface sif, struct io_interface iif);`
  - 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 processor is reset (i.e also startup). 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 = doubleword. 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, -1 should be returned. Set to NULL if not used.

int (*command)(char * cmd);
The I/O module can optionally receive command-line commands. A command is first sent to the AHB and I/O modules, and if not recognised, the to TSIM. command() is called with the full command string in *cmd. Should return 1 if the command is recognized, otherwise 0. TSIM/ERC32 also calls this callback when the SYSAV bit in the ERSR register changes. The commands “sysav 0” and “sysav 1” are then issued. When TSIM commands are issued through the gdb ‘monitor’ command, a return value of 0 or 1 will result in an ‘OK’ response to the gdb command. A return value > 1 will send the value itself as the gdb response. A return value %lt; 1 will truncate the lsb 8 bits and send them back as a gdb error response: ‘Err’.

void (*sigio)();
Not used as of tsim-1.2, kept for compatibility reasons.

void (*save)(char *fname);
The save() function is called when save command is issued in the simulator. The I/O module should save any required state which is needed to completely restore the state at a later stage. *fname points to the base file name which is used by TSIM. TSIM saves its internal state to fname.tss. It is suggested that the I/O module save its state to fname.ios. Note that any events placed in the event queue by the I/O module will be saved (and restored) by TSIM.

void (*restore)(char *fname);
The restore() function is called when restore command is issued in the simulator. The I/O module should restore any required state to resume operation from a saved check-point. *fname points to the base file name which is used by TSIM. TSIM restores its internal state from fname.tss.

5.1.4. Cygwin specific io_init()
Due to problems of resolving cross-referenced symbols in the module loading when using Cygwin, the io_init() routine in the I/O module must initialise a local copy of simif and ioif. This is done by providing the following io_init() routine:

static void io_init(struct sim_interface sif, struct io_interface iif)
{
#ifdef __CYGWIN32__
    /* Do not remove, needed when compiling on Cygwin! */
    simif = sif;
    ioif = iif;
#endif
    /* additional init code goes here */
}

The same method is also used in the AHB and FPU/CP modules.

5.2. LEON AHB emulation interface
In addition to the above described I/O modules, TSIM also allows loading AHB modules that can add simulation models, override built-in simulation models, and even emulating the LEON2/3/4 processor core with a completely user-defined memory and I/O architecture. This is not applicable to ERC32. The emulated processor core communicates with an AHB module using an interface similar to the AHB master interface in the real LEON VHDL model. A single AHB module have the possibility to emulate the complete AHB bus and all attached units, or to just emulate some cores on the AHB or APB bus. An AHB module that emulates memory can disable the internal memory emulation.
TSIM supports up to 16 AHB modules, loaded using multiple `-ahbm` options on TSIM startup. The order they are specified corresponds to the precedence order in which the modules handles bus accesses. For each bus access, one module at a time, in the same order as the `-ahbm` options, gets the chance to handle an access or to let it get passed to the next module. If an access is unhandled by all AHB modules, core TSIM will handle the access, or if applicable pass it on to an I/O module.

The AHB module interface is made up of two parts; one that is exported by TSIM and defines TSIM functions and data structures that can be used by the AHB module; and one that is exported by the AHB module and allows TSIM to access the emulated AHB devices.

At start-up, TSIM searches for `ahb.so` in the current directory, but the location of the module can be specified using the `-ahbm` switch. Note that the module must be compiled to be position-independent, i.e. with the `-fPIC` switch (gcc). The win32 version of TSIM loads ahb.dll instead of ahb.so. See the iomod directory in the TSIM distribution for an example ahb.c and how to build the .so / .dll modules. The environmental variable TSIM_MODULE_PATH can be set to a ':' separated (';' in WIN32) list of search paths.

### 5.2.1. procif structure

TSIM exports one structure for AHB emulation: procif. The procif structure defines a few functions giving access to the processor emulation and cache behaviour. The procif structure is defined in tsim.h:

```c
struct proc_interface {
    void (*set_irl)(int level); /* generate external interrupt */
    void (*cache_snoop)(uint32 addr);
    void (*cctrl)(uint32 *data, uint32 read);
    void (*power_down)();
    void (*set_irq_level)(int level, int set);
    void (*set_irq)(uint32 irq, uint32 level); /* generate external interrupt */
};
extern struct proc_interface procif;
```

The elements in the structure have the following meaning:

- `void (*set_irl)(int level);`
  
  Set the current interrupt level (iui.irl in VHDL model). 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.

- `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)(uint32 *data, uint32 read);`
  
  Read and write the cache control register (CCR). The CCR is attached to the APB bus in the LEON2 VHDL model, and this function can be called by the AHB module to read and write the register. If read = 1, the CCR value is returned in `*data`, else the value of `*data` is written to the CCR. The `cctrl()` function is only needed for LEON2 emulation, since LEON3/4 accesses the cache controller through a separate ASI load/store instruction.

- `void (*power_down)();`
  
  The LEON processor enters power down-mode when called.

- `void (*set_irq_level)(int level, int set);`
  
  Callback `set_irq_level` can be used to emulate level triggered interrupts. Parameter `set` should be 1 to activate the interrupt level specified in parameter `level` or 0 to reset it. The interrupt level will remain active after it is set until it is reset again. Multiple calls can be made with different `level` parameters in which case the highest level is used.

- `void (*set_irq)(uint32 irq, uint32 level);`
  
  Set the interrupt pending bit for interrupt irq. Valid values on irq is 1 - 15. Care should be taken not to set interrupts used by the LEON emulated peripherals. Note that the LEON interrupt control register controls how and when processor interrupts are actually generated.

### 5.2.2. Structure to be provided by AHB module

tsim.h defines the structure to be provided by the emulated AHB module:
The elements of the structure have the following meanings:

void (*init)(struct proc_interface procif); /* called once on startup */
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 on processor reset */
void (*restart)();/* called on simulator restart */
int (*read)(struct ahb_access *access);
int (*write)(struct ahb_access *access);
char *(*get_io_ptr)(unsigned int addr, int size);
int (*command)(char * cmd); /* I/O specific commands */
int (*sigio)(); /* Not used */
void (*save)(char * fname); /* save state */
void (*restore)(char * fname); /* restore state */
int (*intack)(int level); /* interrupt acknowledge */
int (*plugandplay)(struct pp_amba **); /* LEON3/4: get plug & play information */
void (*intpend)(unsigned int pend); /* LEON3/4 only: interrupt pending change */
int meminit; /* tell tsim weather to initialize mem */
void (*start)(void); /* Called each time simulation starts (again) (run, go, cont) */
void (*stop)(void); /* Called each time simulation stops, (Ctrl-C, breakpoints, etc.) */
int (*correctable_error)(uint32 addr, uint32 master, uint32 size, int write); /* call correctable error */

void (*get_mem_ptr_ws)(); /* initialized if meminit was set */
void (*snoop)(uint32 addr); /* initialized with cache snoop routine */

The elements of the structure have the following meanings:

The elements of the structure have the following meanings:
ahbacc.wsize field defines write size as follows: 0 = byte, 1 = half-word, 2 = word, 3 = double-word. The function should return 0 for a successful access, 1 for failed access and -1 for an area not handled by the module. The ahbacc.num field is not used during write cycles. The ahbacc.cache field is no longer in use (use struct ahb_subsystem.cacheable instead).

char * (*get_io_ptr)(unsigned int addr, int size);
During file load operations and displaying of memory contents, TSIM will access emulated memory 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, -1 should be returned. Set to NULL if not used.

int (*command)(char * cmd);
The AHB module can optionally receive command-line commands. A command is first sent to the AHB and I/O modules, and if not recognised, then to TSIM. command() is called with the full command string in *cmd. Should return 1 if the command is recognized, otherwise 0. When TSIM commands are issued through the gdb ‘monitor’ command, a return value of 0 or 1 will result in an ‘OK’ response to the gdb command. A return value > 1 will send the value itself as the gdb response. A return value < 1 will truncate the lsb 8 bits and send them back as a gdb error response: ‘Err’.

void (*save)(char *fname);
The save() function is called when save command is issued in the simulator. The AHB module should save any required state which is needed to completely restore the state at a later stage. *fname points to the base file name which is used by TSIM. TSIM save its internal state to fname.tss. It is suggested that the AHB module save its state to fname.ahs. Note that any events placed in the event queue by the AHB module will be saved (and restored) by TSIM.

void (*restore)(char * fname);
The restore() function is called when restore command is issued in the simulator. The AHB module should restore any required state to resume operation from a saved check-point. *fname points to the base file name which is used by TSIM. TSIM restores its internal state from fname.tss.

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.

int (*plugandplay)(struct pp_amba **p);
Leon3/4 only: The plugandplay() function is called at startup. plugandplay() should return in p a static pointer to an array with elements of type struct pp_amba and return the number of entries in the array. The callback plugandplay() is used to add entries in the AHB and APB configuration space. Each struct pp_amba entry specifies an entry: If is_apb is set to 1 the entry will appear in the APB configuration space and only member bars[0] will be used. If is_apb is 0 then the entry will appear in the AHB slave configuration space and bars[0-3] will be used. If is_apb is 2 then the entry will appear in the AHB master configuration space and bars[0-3] will be used. The members of the struct resemble the fields in a configuration space entries. The entry is mapped to the first free slot. When using the -gr712rc or -ut700 option, if is_apb is 3 the entry will appear under a second ABPCTRL core.

void (*intpend)(unsigned int pend);
Leon3/4 only: The intpend() function is called when the set of pending interrupts changes. The pend argument is a bitmask with the bits of pending interrupts set to 1.

int meminit;
If all loaded AHB modules sets meminit to 1, TSIM will initialize and emulate initialize and emulate SRAM/SDRAM/PROM memory. Thus, the AHB module should initialize meminit with 1 if TSIM (or another AHB module) should handle memory simulation. Calls to read and write should return -1 (undecoded area) for the memory regions in which case TSIM (or possibly some other AHB module) will handle them. If meminit is set to 0 the AHB module itself should emulate the memory address regions.

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

unsigned char *(*get_mem_ptr_ws) (unsigned int addr, int size, int *wws, int *rws);
If meminit was set to 1 tsim will initialize get_mem_ptr_ws with a callback that can be used to query a pointer to the host memory emulating the LEON’s memory, along with waitstate information. Note that the host memory pointer returned is in the hosts byte order (normally little endian on a PC). The size
parameter should be the length of the access (1 for byte, 2 for short, 4 for word and 8 for double word access). The \texttt{wss} and \texttt{rws} parameters will return the calculated write and read waitstates for a possible access. See also \texttt{snoop} below for responsibilities when DMA writes are done via pointers from this function.

\begin{verbatim}
void (*snoop) (unsigned int addr)

The callback \texttt{snoop} is initialized by \texttt{tsim}. If data cache snooping is enabled (and functioning, i.e. not \texttt{ut699}) it flushes (i.e. invalidates) data cache lines corresponding to physical address \texttt{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.
\end{verbatim}

\begin{verbatim}
int (*cacheable)(uint32 addr, uint32 size)

The \texttt{cacheable} callback is initialized by the module to NULL or a function returning cacheability for a memory area. The function should return 1 if the range \{addr,addr+size\} is cacheable, 0 if it is un-cacheable or -1 if the memory area it is not handled by the module. If all modules return -1 and/or lack the \texttt{cacheable} callback, the area will be considered cacheable for memory areas \{0x00000000,0x20000000\} and \{0x40000000-0x80000000\} and non-cacheable for all other areas. NOTE: For any (correspondingly aligned) area as large as the largest data cache or instruction cache line size in the system, the \texttt{cacheable} callback may not return different results for different words in the area.
\end{verbatim}

\begin{verbatim}
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 \texttt{rw} field in LEON2 FAILSR corresponding to !write).
\end{verbatim}

5.2.3. Big versus little endianess

\texttt{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 \texttt{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. For a complete example, see the prom emulation function in \texttt{io.c}.

5.3. TSIM/LEON co-processor emulation

5.3.1. FPU/CP interface

The professional version of TSIM/LEON can emulate a user-defined floating-point unit (FPU) and co-processor (CP). The FPU and CP are included into the simulator using loadable modules. To access the module, use the
structure ‘cp_interface’ defined in tsim.h. The structure contains a number of functions and variables that must be provided by the emulated FPU/CP:

```c
/* structure of function to be provided by an external co-processor */
struct cp_interface {
    void (*cp_init)(); /* called once on start-up */
    void (*cp_exit)(); /* called once on exit */
    void (*cp_reset)(); /* called on processor reset */
    void (*cp_restart)(); /* called on simulator restart */
    uint32 (*cp_reg)(int reg, uint32 data, int read);
    int (*cp_load)(int reg, uint32 data, int *hold);
    int (*cp_store)(int reg, uint32 *data, int *hold);
    int (*cp_exec)(uint32 pc, uint32 inst, int *hold);
    int (*cp_cc)(int *cc, int *hold); /* get condition codes */
    int *cp_status; /* unit status */
    void (*cp_print)(); /* print registers */
    int (*command)(char * cmd); /* CP specific commands */
    int (*set_fsr)(uint32 fsr); /* initialized by tsim */
};
extern struct cp_interface *cp; /* imported co-processor emulation functions */
```

5.3.2. Structure elements

- **void (*cp_init)(struct sim_interface sif, struct io_interface iif);**
  Called once on simulator startup. Set to NULL if not used.

- **void (*cp_exit)();**
  Called once on simulator exit. Set to NULL if not used.

- **void (*cp_reset)();**
  Called every time the processor is reset. Set to NULL if not used.

- **void (*cp_restart)();**
  Called every time the simulator is restarted. Set to NULL if not used.

- **uint32 (*cp_reg)(int reg, uint32 data, int read);**
  Used by the simulator to perform diagnostics read and write to the FPU/CP registers. Calling `cp_reg()` should not have any side-effects on the FPU/CP status. `reg` indicates which register to access: 0-31 indicates %f0-%f31/%c0-%c31, 34 indicates %fsr/%csr. `read` indicates read or write access: read==0 indicates write access, read!=0 indicates read access. Written data is passed in `data`, the return value contains the read value on read accesses.

- **int (*cp_exec)(uint32 pc, uint32 inst, int *hold);**
  Execute FPU/CP instruction. The `%pc` is passed in `pc` and the instruction opcode in `inst`. If data dependency is emulated, the number of stall cycles should be return in `*hold`. The return value should be zero if no trap occurred or the trap number if a trap did occur (0x8 for the FPU, 0x28 for CP). A trap can occur if the FPU/CP is in exception_pending mode when a new FPU/CP instruction is executed.

- **int (*cp_cc)(int *cc, int *hold); /* get condition codes */**
  Read condition codes. Used by FBCC/CBCC instructions. The condition codes (0 - 3) should be returned in `*cc`. If data dependency is emulated, the number of stall cycles should be return in `*hold`. The return value should be zero if no trap occurred or the trap number if a trap did occur (0x8 for the FPU, 0x28 for CP). A trap can occur if the FPU/CP is in exception_pending mode when a FBCC/CBCC instruction is executed.

- **int *cp_status; /* unit status */**
  Should contain the FPU/CP execution status: 0 = execute_mode, 1 = exception_pending, 2 = exception_mode.

- **void (*cp_print)();/* print registers */**
  Should print the FPU/CP registers to stdio.

- **int (*command)(char * cmd); /* CP specific commands */**
  User defined FPU/CP control commands. NOT YET IMPLEMENTED.

- **int (*set_fsr)(char * cmd); /* initialized by tsim */**
  This callback is initialized by tsim and can be called to set the FPU status register.

5.3.3. Attaching the FPU and CP

At startup the simulator tries to load two dynamic link libraries containing an external FPU or CP. The simulator looks for the file fp.so and cp.so in the current directory and in the search path defined by ldconfig. The location of the modules can also be defined using -cpm and -fpm switches. The environmental variable
TSIM_MODULE_PATH can be set to a ‘:’ separated (‘;’ in WIN32) list of search paths. Each library is searched for a pointer ‘cp’ that points to a cp_interface structure describing the co-processor. Below is an example from fp.c:

```c
struct cp_interface test_fpu = {
  cp_init,           /* cp_init */
  NULL,              /* cp_exit */
  cp_init,           /* cp_reset */
  cp_reg,            /* cp_reg */
  cp_reg,            /* cp_reg */
  cp_load,           /* cp_load */
  cp_load,           /* cp_load */
  fpinitko,          /* cp_exec */
  &fpregs.fpstate,   /* cp_status */
  cp_cc,             /* cp_cc */
  &fpregs.fpstate,   /* cp_status */
  cp_print,          /* cp_print */
  NULL               /* cp_command */
};
struct cp_interface *cp = &test_fpu; /* Attach pointer!! */
```

5.3.4. Big versus little endianess

SPARC is conform 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 register-file is organised on word basis with the bytes within a word arranged according the endianess of the host. Double words are also in host order, and the read/write register functions must therefore invert the lsb of the register address when performing word access on little-endian hosts. See the file fp.c for examples (cp_load(), cp_store()).

5.3.5. Additional TSIM commands

- `float`: Shows the registers of the FPU
- `cp`: Shows the registers of the co-processor.

5.3.6. Example FPU

The file fp.c contains a complete SPARC FPU using the co-processor interface. It can be used as a template for implementation of other co-processors. Note that data-dependency checking for correct timing is not implemented in this version (it is however implemented in the built-in version of TSIM).

5.4. Loadable modules distributed with TSIM

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

<table>
<thead>
<tr>
<th>Module</th>
<th>For TSIM versions</th>
</tr>
</thead>
<tbody>
<tr>
<td>Atmel AT697 emulation</td>
<td>LEON2</td>
</tr>
<tr>
<td>Cobham UT699 emulation</td>
<td>LEON3</td>
</tr>
<tr>
<td>Cobham UT700 emulation</td>
<td>LEON3</td>
</tr>
<tr>
<td>Cobham Gaisler GR712RC emulation</td>
<td>LEON3</td>
</tr>
<tr>
<td>TPS VxWorks 6.x AHB Module</td>
<td>LEON3/4</td>
</tr>
<tr>
<td>Example IO module</td>
<td>ERC32, LEON2/3/4</td>
</tr>
<tr>
<td>Example AHB module</td>
<td>LEON2/3/4</td>
</tr>
<tr>
<td>Example FPU/coprocessor module</td>
<td>LEON2/3/4</td>
</tr>
</tbody>
</table>

5.4.1. General AHB module limitations

The general AHB module interface allows for the possibility to support checkpointing and to support system reset during simulation. However, the modules distributed with TSIM does not support these features unless otherwise noted.
The socket base interfaces for the simulation models for cores such as GRETH, GRSPW1, GRSPW2 and CAN_OC does not support any signalling of restart of the simulation. To ensure a clean restart of simulation when using these cores, restarting TSIM and reconnecting all such socket interfaces is advisable.
6. TSIM library (TLIB)

6.1. Introduction

The professional version of TSIM is also available as a library, allowing the simulator to be integrated in a larger simulation frame-work. The various TSIM commands and options are accessible through a simple function interface. I/O functions can be added, and use a similar interface to the loadable I/O modules described earlier.

6.2. Function interface

The following functions are provided to access TSIM features:

- `int tsim_init (char *option);` /* initialise tsim with optional params. */
  Initialize TSIM - must be called before any other TSIM function (except tsim_set_diag()) are used. The options string can contain any valid TSIM startup option (as used for the standalone simulator), with the exception that no filenames for files to be loaded into memory may be given. tsim_init() may only be called once, use the TSIM reset command to reset the simulator without exiting. tsim_init() will return 1 on success or 0 on failure.

- `int tsim_cmd (char *cmd);` /* execute tsim command */
  Execute TSIM command. Any valid TSIM command-line command may be given. The following return values are defined:
  - SIGINT: Simulation stopped due to interrupt
  - SIGHUP: Simulation stopped normally
  - SIGTRAP: Simulation stopped due to breakpoint hit
  - SIGSEGV: Simulation stopped due to processor in error mode
  - SIGTERM: Simulation stopped due to program termination

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

- `void tsim_get_regs (unsigned int *regs);`
  Get SPARC registers. regs is a pointer to an array of integers, see tsim.h for how the various registers are indexed.

- `void tsim_set_regs (unsigned int *regs);`
  Set SPARC registers. *regs is a pointer to an array of integers, see tsim.h for how the various registers are indexed.

- `void tsim_disas(unsigned int addr, int num);`
  Disassemble memory. addr indicates which address to disassemble, num indicates how many instructions.

- `void tsim_set_diag (void (*cfunc)(char *));`
  Set console output function. By default, TSIM writes all diagnostics and console messages on stdout. tsim_set_diag() can be used to direct all output to a user defined routine. The user function is called with a single string parameter containing the message to be written.

- `void tsim_set_callback (void (*cfunc)(void));`
  Set the debug callback function. Calling tsim_set_callback() with a function pointer will cause TSIM to call the callback function just before each executed instruction, when the history is enabled. At this point the instruction to be executed can be seen as the last entry in the history. History can be enabled with the tsim_cmd() function.

- `void tsim_gdb (unsigned char (*inchar)(), void (*outchar)(unsigned char c));`
  Controls the simulator using the gdb ‘extended-remote’ protocol. The inchar parameter is a pointer to a function that when called, returns next character from the gdb link. The outchar parameter is a pointer to a function that sends one character to the gdb link.

- `void tsim_read(unsigned int addr, unsigned int *data);`
  Performs a read from addr, returning the value in *data. Only for diagnostic use.

- `void tsim_write(unsigned int addr, unsigned int data);`
  Performs a write to addr, with value data. Only for diagnostic use.
void tsim_stop_event(void (*cfunc)(), int arg, int op);

*tsim_stop_event*() can remove certain event depending on the setting of *arg* and *op*. If op = 0, all instance of the callback function *cfunc* will be removed. If op = 1, events with the argument = *arg* will be removed. If op = 2, only the first (earliest) of the events with the argument = *arg* will be removed.

NOTE: The stop_event() function may NOT be called from a signal handler installed by the I/O module.

void tsim_inc_time(uint64);

*tsim_inc_time*() will increment the simulator time without executing any instructions. The event queue is evaluated during the advancement of time and the event callbacks are properly called. Can not be called from event handlers.

int tsim_trap(int (*trap)(int tt), void (*rett)());

*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 one argument (*tt*) that contains the SPARC trap number. If *tsim_trap*() returns with 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 callbacks are removed by calling *tsim_trap*() with a NULL arguments.

int tsim_cov_get(int start, int end, char *ptr);

*tsim_cov_get*() will return the coverage data for the address range >= *start* and < *end*. 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.

int tsim_cov_set(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) with the value of *val*.

int tsim_ext_ins (int (*func) (struct ins_interface *r));

*tsim_ext_ins*() installs a handler for custom instructions. func is a pointer to an instruction emulation function as described in Section 4.1.6. Calling *tsim_ext_ins*() with a NULL pointer will remove the handler.

int tsim_lastbp (int *addr);

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

void tsim_set_iosystem (struct io_subsystem *ioarg)

On win32 (excluding cygwin), this installs an I/O module that is built into the TLIB application. It must be called before calling *tsim_init*(). See also Section 6.4 for more details.

void tsim_set_ahbsystem (struct ahb_subsystem *ahbarg)

On win32 (excluding cygwin), this installs an AHB module that is built into the TLIB application. It must be called before calling *tsim_init*(). See also Section 6.4 for more details.

### 6.3. External I/O and AHB modules

External I/O and AHB modules can be loaded when using TLIB, just as with standalone TSIM, by adding the -iom name and -ahbm name switches to the *tsim_init*() argument string when starting. See Chapter 5 for further information.

### 6.4. Builtin I/O module and/or AHB module

The TSIM library can use a builtin I/O module and or a builtin AHB module that is part of the TLIB application that uses the same I/O interface and AHB interface as the standalone simulator. Instead of loading a shared library containing the module, the module is linked with the main program (for non-win32, including cygwin) or installed using the *tsim_set_iosystem*() and/or *tsim_set_ahbsystem*() functions (for win32). The I/O and AHB functions (and the main program) has the same access to the exported simulator interface (*simif* and *ioif*) as described in the loadable module interface.

For non-win32 (including cygwin) the TSIM library imports the I/O structure pointer, *struct io_subsystem *iosystem*, and AHB structure pointer *struct ahb_subsystem *ahbsystem*. These pointers must exist, but can be set to NULL if no builtin I/O and/or AHB module is present.
For win32 (but not cygwin) the `tsim_set_iosystem()` and `tsim_set_ahbsystem()` functions instead used used to install a builtin I/O module and/or a builtin AHB module. If used, these functions must be called before calling `tsim_init()`. Calling these functions is optional (unlike the direct linking case where the pointers must exist).

An example I/O module and an example AHB module, prepared to be builtin, are provided in `simple_io.c` and `simple_ahb.c`. These modules provides a simpler interface to attach I/O functions and AHB functions. The `app1.c` example shows how the builtin modules are connected for both the win32 case and for other hosts. By default the builtin I/O module is the one that is actively used, but by defining `USEIO` to 0 the builtin AHB module is instead the one that is actively used.

The following interface is provided by `simple_io.c`:

```c
void simple_io_set_ioread (void (*cfunc)(int address, int *data, int *ws));
```

This function is used to pass a pointer to a user function which is to be called by TSIM when an I/O read access is made. The user function is called with the address of the access, a pointer to where the read data should be returned, and a pointer to a waitstate variable that should be set to the number of waitstates that the access took.

```c
void simple_io_set_iowrite (void (*cfunc)(int address, int *data, int *ws, int size));
```

This function is used to pass a pointer to a user function which is to be called by TSIM when an I/O write access is made. The user function is called with the address of the access, a pointer to the data to be written, a pointer to a waitstate variable that should be set to the number of waitstates that the access took, and the size of the access (0=byte, 1=half-word, 2=word, 3=double-word).

The interface in `simple_ahb.c` follows the same principles as the one in `simple_io.c`, but uses the AHB interface.

### 6.5. UART handling

By default, the library is using the same UART handling as the standalone simulator. This means that the UARTs can be connected to the console, or any Unix device (pseudo-ttys, pipes, fifos). If the UARTs are to be handled by the user’s I/O emulation routines, `tsim_init()` should be called with `'-nouart'`, which will disable all internal UART emulation. Any access to the UART register by an application will then be routed to the I/O module read/write functions.

### 6.6. Linking a TLIB application

Three sample application are provided, one that uses the simplified I/O (or simplified AHB interface) interface (`app1.c`), and two that uses the standard loadable module interface (`app2` and `app3`). They are built by doing a ‘make all’ in the tlib directory. The win32 version of TSIM provides the library as a DLL, for all other platform a static library is provided (.a). Support for dynamic libraries on Linux or Solaris is not available.

### 6.7. Limitations

On Windows/Cygwin hosts a TLIB application is not capable of reading UART A/B from the console, only writing is possible. If reading of UART A/B is necessary, the simulator should be started with `-nouart`, and emulation of the UARTs should be handled by the I/O module.
7. Cobham UT699 emulation

To emulate the UT699 chip the `-ut699` option should be used. That sets up parameters for core TSIM2 to match UT699 and sets snooping as non-functional. In addition, the UT699 loadable AHB module implements emulation models for additional cores of the UT699. Apart from an overview of what TSIM supports for UT699 emulation, this chapter is mainly about the UT699 AHB module.

The following table lists which cores in the UT699 are modelled by TSIM or not. Some supported cores are modelled by the core TSIM2 and some in the UT699 AHB module. 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 built-in simulation models and the rest of the chapter for the AHB module models.

Table 7.1. Simulation models for UT699

<table>
<thead>
<tr>
<th>Core</th>
<th>Status</th>
<th>Notes</th>
</tr>
</thead>
<tbody>
<tr>
<td>LEON3FT</td>
<td>Supported by core TSIM2</td>
<td>Only one CPU is modelled. No FT features are modelled.</td>
</tr>
<tr>
<td>GRFPU</td>
<td>Supported by core TSIM2</td>
<td></td>
</tr>
<tr>
<td>AHBSTAT</td>
<td>Supported by core TSIM2</td>
<td></td>
</tr>
<tr>
<td>APBCTRL</td>
<td>Supported by core TSIM2</td>
<td></td>
</tr>
<tr>
<td>APBUART</td>
<td>Supported by core TSIM2</td>
<td></td>
</tr>
<tr>
<td>FTMCTRL</td>
<td>Supported by core TSIM2</td>
<td>No FT features are modelled</td>
</tr>
<tr>
<td>GPTIMER</td>
<td>Supported by core TSIM2</td>
<td></td>
</tr>
<tr>
<td>IRQMP</td>
<td>Supported by core TSIM2</td>
<td></td>
</tr>
<tr>
<td>CAN_OC</td>
<td>Supported by AHB module</td>
<td></td>
</tr>
<tr>
<td>GRETH</td>
<td>Supported by AHB module</td>
<td></td>
</tr>
<tr>
<td>GRGPIO</td>
<td>Supported by AHB module</td>
<td></td>
</tr>
<tr>
<td>GRPCI</td>
<td>Supported by AHB module</td>
<td>Including DMA controller</td>
</tr>
<tr>
<td>GRSPW</td>
<td>Supported by AHB module</td>
<td></td>
</tr>
<tr>
<td>CLKGATE</td>
<td>Not supported</td>
<td></td>
</tr>
<tr>
<td>AHBJTAG</td>
<td>Not supported</td>
<td>Debug link</td>
</tr>
<tr>
<td>AHBUART</td>
<td>Not supported</td>
<td>Debug link</td>
</tr>
<tr>
<td>DSU3</td>
<td>Not supported</td>
<td>Debug unit</td>
</tr>
</tbody>
</table>

TSIM supports running several AHB modules. User defined models for unsupported cores can be used in parallel with the UT699 AHB module. User defined models can also override the simulation models in the UT699 AHB module.

7.1. Overview of the UT699 AHB module

The UT699 AHB module is a loadable AHB module that implements UT699 peripherals in addition to what is emulated by core TSIM as listed in Table 7.1. The interfaces are modelled at packet/transaction/message level and provides an easy way to connect the simulated UT699 to a larger simulation framework. See also Section 5.4.1 on some limitations of some features when using this module. The UT699 AHB module is only supported for TSIM2 LEON3.

The following files are delivered with the UT699 TSIM module:

Table 7.2. Files delivered with the UT699 TSIM module

<table>
<thead>
<tr>
<th>File</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>ut699/linux/ut699.so</td>
<td>UT699 AHB module for Linux</td>
</tr>
<tr>
<td>File</td>
<td>Description</td>
</tr>
<tr>
<td>------</td>
<td>-------------</td>
</tr>
<tr>
<td>ut699/win32/ut699.dll</td>
<td>UT699 AHB module for Windows</td>
</tr>
<tr>
<td>ut699/examples/input</td>
<td>The input directory contains two examples of PCI user modules</td>
</tr>
<tr>
<td>ut699/examples/input/README.txt</td>
<td>Description of the user module examples</td>
</tr>
<tr>
<td>ut699/examples/input/pci.c</td>
<td>PCI user module example that makes UT699 PCI initiator accesses</td>
</tr>
<tr>
<td>ut699/examples/input/pci_target.c</td>
<td>PCI user module example that makes UT699 PCI target accesses</td>
</tr>
<tr>
<td>ut699/examples/input/gpio.c</td>
<td>GPIO user module example</td>
</tr>
<tr>
<td>ut699/examples/input/ut699inputprovider.h</td>
<td>Interface between the UT699 module and the user defined PCI module</td>
</tr>
<tr>
<td>ut699/examples/input/pci_input.h</td>
<td>UT699 PCI input provider definitions</td>
</tr>
<tr>
<td>ut699/examples/input/input.h</td>
<td>Generic input provider definitions</td>
</tr>
<tr>
<td>ut699/examples/input/tsim.h</td>
<td>TSIM interface definitions</td>
</tr>
<tr>
<td>ut699/examples/input/end.h</td>
<td>Defines the endian of the local machine</td>
</tr>
<tr>
<td>ut699/examples/test</td>
<td>The test directory contains tests that can be executed in TSIM</td>
</tr>
<tr>
<td>ut699/examples/test/README.txt</td>
<td>Description of the tests</td>
</tr>
<tr>
<td>ut699/examples/test/Makefile</td>
<td>Makefile for building the tests</td>
</tr>
<tr>
<td>ut699/examples/test/cansend.c</td>
<td>CAN transmission test</td>
</tr>
<tr>
<td>ut699/examples/test/canrec.c</td>
<td>CAN reception test</td>
</tr>
<tr>
<td>ut699/examples/test/pci.c</td>
<td>PCI interface test</td>
</tr>
<tr>
<td>ut699/examples/test/pcitest.h</td>
<td>Header file for PCI test</td>
</tr>
</tbody>
</table>

### 7.2. Loading the module

The module is loaded using the TSIM2 option `-ahbm`. A user input module for SPI and PCI can optionally be declared, between `-designinput` and `-designinputend` options. For example:

**On Linux:**

```
tsim-leon3 -ut699 -ahbm ut699/linux/ut699.so
  -designinput ./input.so -designinputend
```

**On Windows:**

```
tsim-leon3 -ut699 -ahbm ut699/win32/ut699.dll
  -designinput input.dll -designinputend
```

The option `-ut699` needs to be given to TSIM to enable the UT699 processor configuration. The above line loads the UT699 AHB module `ut699.so` which in turn loads the user user input module `./input.so`. The user input module `./input.so` communicates with `ut699.so` using the user module interface described in `ut699inputprovider.h`, while `ut699.so` communicates with TSIM via the AHB interface.

Example user input modules can be found in `ut699/examples/input/`.

### 7.2.1. User input module interface

The SPI and PCI models in the UT699 module use a user supplied user input module, in the form of a dynamic loadable library, that models the outside world. This section describes the general interface for hooking up the user module to the UT699 module. The details on the interfaces to the particular cores, see their respective sections.
A user supplied dynamic library should expose a public symbol `ut699inputsystem` of type `struct ut699_subsystem *`. The `struct ut699_subsystem` is defined in `ut699inputprovider.h` as:

```c
struct ut699_subsystem {
    void (*ut699_inp_setup) (int id,
                            struct ut699_inp_layout * l,
                            char **argv, int argc);
    void (*ut699_inp_restart) (int id,
                                struct ut699_inp_layout * l);
    struct sim_interface *simif;
};
```

The callback `ut699_inp_restart` will be called every time the simulator restarts. At initialization the callback `ut699_inp_setup` will be called once, supplied with a pointer to structure `struct ut699_inp_layout` defined in `ut699inputprovider.h`.

```c
struct ut699_inp_layout {
    struct grpci_input grpci;
    struct gpio_input gpio;
};
```

The user module can access the global TSIM `struct sim_interface` structure through the `simif` member. See Chapter 5 for more details.

The user supplied dynamic library should, in its `ut699_inp_setup` function, “claim” the input structs it uses using the `INPUT_CLAIM` macro. For example `INPUT_CLAIM(l->gpio)` as in the example below.

A user supplied dynamic library that only sets up a model for GPIO could look like this:

```c
#include <stdio.h>
#include <string.h>
#include "tsim.h"
#include "ut699inputprovider.h"
extern struct ut699_subsystem *ut699inputsystem;
static struct ut699_inp_layout *lay = 0;
static void Change(struct gpio_input *ctrl) {
    ...}
int gpioout(struct gpio_input *ctrl, unsigned int out) {
    ...}
static void ut699_inp_setup (int id,
                             struct ut699_inp_layout * l,
                             char **argv, int argc) {
    lay = l;
    printf("User-dll: ut699_inp_setup:Claiming %s\n", l->gpio._b.name);
    INPUT_CLAIM(l->gpio);
    l->gpio.gpioout = gpioout;
    ut699inputsystem->simif->event(Change, (unsigned long)&l->gpio, 10000000);
}
static struct ut699_subsystem ut699_gpio = {
    ut699_inp_setup, 0, 0
};
struct ut699_subsystem *ut699inputsystem = &ut699_gpio;
```

A Makefile that would build a user supplied dynamic library gpio.(dll|so) could look like this:

```
M_DLL_FIX=$(if $(strip $(shell uname|grep MINGW32)),dll,so)
M_LIB=$(if $(strip $(shell uname|grep MINGW32)),-lws2_32 -luser32 -lkernel32 -lwinmm,)
all: gpio.$(M_DLL_FIX)
gpio.$(M_DLL_FIX) : gpio.o
  $(CC) -shared -g gpio.o -o gpio.$(M_DLL_FIX) $(M_LIB)
gpio.o: gpio.c
  $(CC) -fPIC -c -g -o gpio.c -o gpio.o
```
The user can then specify the user module to be loaded by the ut699.so AHB module using the -designinput and -designinputend command line options. The first argument after -designinput is the user module. Arguments after that are passed to the user input module in the call to ut699_inp_setup.

For example: -designinput ut699/examples/input/gpio.so -gpioverbose -designinputend will specify that the example user input module gpio.so should be used and that it should receive the argument -gpioverbose.

### 7.3. Debugging

To enable printout of debug information the -ut699_dbgon flag switch can be used. Alternatively one can issue the ut699_dbgon flag command on the TSIM2 command line to toggle the on/off state of a flag. The debug flags that are available are described for each core in the following sections and can be listed by ut699_dbgon help.

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.

### 7.4. 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 the UT699. For core details and register specification please see the UT699 manual.

The following features are supported:
- Direct Memory Access
- Interrupts

#### 7.4.1. Start up options

**Ethernet core start up options**

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

#### 7.4.2. Commands

**Ethernet core TSIM commands**

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

- `greth_status`
  
  Print Ethernet register status

#### 7.4.3. Debug flags

The following debug flags are available for the Ethernet interface. Use the them in conjunction with the ut699_dbgon command to enable different levels of debug information.

*Table 7.3. Ethernet debug flags*

<table>
<thead>
<tr>
<th>Flag</th>
<th>Trace</th>
</tr>
</thead>
<tbody>
<tr>
<td>GAISSLER_GRETH_ACC</td>
<td>GRETH accesses</td>
</tr>
<tr>
<td>GAISSLER_GRETH_L1</td>
<td>GRETH accesses verbose</td>
</tr>
<tr>
<td>GAISSLER_GRETH_TX</td>
<td>GRETH transmissions</td>
</tr>
<tr>
<td>GAISSLER_GRETH_RX</td>
<td>GRETH reception</td>
</tr>
<tr>
<td>GAISSLER_GRETH_RXPACKET</td>
<td>GRETH received packets</td>
</tr>
</tbody>
</table>
7.4.4. 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 \texttt{-grethconnect} start-up parameter or using the \texttt{greth_connect} command.

An example implementation of a packet server, named \texttt{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.

7.4.5. 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:

\[
\begin{array}{c}
31 \\
\hline
0
\end{array}
\]

\textbf{LEN}

Length of rest of packet: 4 + number of data bytes

Header at offset 0x4:

\[
\begin{array}{cccccc}
31 & 16 & 15 & 8 & 7 & 5 & 4 & 0 \\
\hline
R & IPID & TYPE & R
\end{array}
\]

31:16 \textbf{R} Reserved for future use. Must be set to 0.

15:8 \textbf{IPID} IP core ID: 1 for Ethernet

7:5 \textbf{TYPE} Packet type: 0 for data packets

4:0 \textbf{R} Reserved for future use. Must be set to 0.

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

\textit{Figure 7.1. Ethernet data packet}

7.5. SpaceWire interface with RMAP support

The UT699 AHB module contains 4 GRSPW cores which models the GRSPW cores available in the UT699. For core details and register specification please see the UT699 manual.

The UT699E AHB module has GRSPW2 cores instead of GRSPW cores. So, for UT699E see Section 9.5 instead.

The following features are supported:

- Transmission and reception of SpaceWire packets
- Interrupts
- RMAP
7.5.1. Start up options

*SpaceWire core start up options*

- **-grspwX_connect host:port**
  Connect GRPSW 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 options has the range 1-4.

7.5.2. Commands

*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 core X on specified TCP port.
- **grspw_status**
  Print status for all GRSPW cores.

X in the above commands has the range 1-4.

7.5.3. Debug flags

The following debug flags are available for the SpaceWire interfaces. Use them in conjunction with the `ut699_dbgon` command to enable different levels of debug information.

*Table 7.4. SpaceWire debug flags*

<table>
<thead>
<tr>
<th>Flag</th>
<th>Trace</th>
</tr>
</thead>
<tbody>
<tr>
<td>GAISLER_GRSPW_ACC</td>
<td>GRSPW accesses</td>
</tr>
<tr>
<td>GAISLER_GRSPW_RXPACKET</td>
<td>GRSPW received packets</td>
</tr>
<tr>
<td>GAISLER_GRSPW_RXCTRL</td>
<td>GRSPW rx protocol</td>
</tr>
<tr>
<td>GAISLER_GRSPW_TXPACKET</td>
<td>GRSPW transmitted packets</td>
</tr>
<tr>
<td>GAISLER_GRSPW_TXCTRL</td>
<td>GRSPW tx protocol</td>
</tr>
</tbody>
</table>

7.5.4. 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.

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.

7.5.5. 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 7.5. Packet types*

<table>
<thead>
<tr>
<th>Type</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>Data</td>
<td>0</td>
</tr>
<tr>
<td>Time code</td>
<td>1</td>
</tr>
</tbody>
</table>
Note that all packets are prepended by a one word length field which specified the length of the coming packet including the header.

**7.5.5.1. Data packet format**

Packet length at offset 0x0:

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

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 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 7.2. SpaceWire data packet*

**7.5.5.2. Time code packet format**

Packet length at offset 0x0:

```
31:0  LEN  Length of rest of packet: 8
```

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: 1 for time code packets
4:0    R  Reserved for future use. Must be set to 0.
```

Payload at offset 0x8:

```
31:0  8 7 6 5 0
```

```
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 7.3. SpaceWire time code packet*
7.6. PCI initiator/target interface

The UT699 AHB module models the PCI core available in the UT699 ASIC. For core details and register specification please see the UT699 manual.

7.6.1. Connecting a user PCI model with the UT699 module

See Section 7.2 for details on how to connect the user PCI model to the UT699 module.

7.6.2. Commands

PCI Commands

pci_status

Print status for the PCI core

7.6.3. Debug flags

The following debug flags are available for the PCI interface. Use them in conjunction with the ut699_dbgcon command to enable different levels of debug information.

<table>
<thead>
<tr>
<th>Flag</th>
<th>Trace</th>
</tr>
</thead>
<tbody>
<tr>
<td>GAISLER_GRPCI_ACC</td>
<td>AHB accesses to/from PCI core</td>
</tr>
<tr>
<td>GAISLER_GRPCI_REGACC</td>
<td>GRPCI APB register accesses</td>
</tr>
<tr>
<td>GAISLER_GRPCI_DMA_REGACC</td>
<td>PCI DMA APB register accesses</td>
</tr>
<tr>
<td>GAISLER_GRPCI_DMA</td>
<td>GRPCI DMA accesses on the AHB bus</td>
</tr>
<tr>
<td>GAISLER_GRPCI_TARGET_ACC</td>
<td>GRPCI target accesses</td>
</tr>
<tr>
<td>GAISLER_GRPCI_INIT</td>
<td>Print summary on startup</td>
</tr>
</tbody>
</table>

7.6.4. PCI bus model API

The structure struct grpci_input models the PCI bus. It is defined as:

```c
struct grpci_input {
    struct input_inp _b;
    int (*acc)(struct grpci_input *ctrl, int cmd, unsigned int addr,
               unsigned int *data, unsigned int *abort, unsigned int *ws);
    int (*target_acc)(struct grpci_input *ctrl, int cmd, unsigned int addr,
                      unsigned int *data, unsigned int *mexc);
};
```

The acc callback should be set by the PCI user module at startup. It is called by the UT699 module whenever it reads/writes as a PCI bus master.

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>cmd</td>
<td>Command to execute, see Section 7.6.2 details.</td>
</tr>
<tr>
<td>addr</td>
<td>PCI address.</td>
</tr>
<tr>
<td>data</td>
<td>Data buffer. The user module should return the read data in *data for read commands or write the data in *data for write commands.</td>
</tr>
<tr>
<td>wsize</td>
<td>0: 8-bit access 1: 16-bit access, 2: 32-bit access. Is always 2 for read accesses.</td>
</tr>
<tr>
<td>ws</td>
<td>Set *ws to the number of PCI clocks it takes to complete the transaction.</td>
</tr>
<tr>
<td>abort</td>
<td>Set *abort to 1 to generate target abort, or 0 otherwise.</td>
</tr>
</tbody>
</table>

The return value of acc determines if the transaction terminates successfully (1, GRPCI_ACC_OK) or with master abort (0, GRPCI_ACC_MASTER_ABORT).
The callback target_acc is installed by the UT699 AHB module. The PCI user dynamic library can call this function to initiate an access to the UT699 PCI target.

Table 7.8. target_acc parameters

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>cmd</td>
<td>Command to execute, see Section 7.6.2 for details. I/O cycles are not supported by the UT699 target.</td>
</tr>
<tr>
<td>addr</td>
<td>PCI address. Should always be word aligned for read accesses.</td>
</tr>
<tr>
<td>data</td>
<td>Data buffer. The read data is returned in *data for read commands or the data in *data is written for write commands.</td>
</tr>
<tr>
<td>wsize</td>
<td>0: 8-bit access 1: 16-bit access, 2: 32-bit access. Should always be 2 for read accesses.</td>
</tr>
<tr>
<td>mexc</td>
<td>0 if access is successful, 1 in case of target abort.</td>
</tr>
</tbody>
</table>

If the address matched MEMBAR0, MEMBAR1 or CONFIG target_acc will return 1 otherwise 0.

See the ut699/examples/input for example implementations.

7.7. GPIO interface

7.7.1. Connecting a user GPIO model with the UT699 module

See Section 7.2 for details on how to connect the user GPIO model to the UT699 module.

7.7.2. Commands

GPIO Commands

```
gpio0_status
   Print status for the GPIO core.
gpio0_dbg [flag] subcommand
   Toggle, set, clear, list debug flags for the GPIO core.
```

7.7.3. Debug flags

The following debug flags and debug subcommands are available for GPIO interfaces. The GAISLER_GPIO_* flags can be used with the gpio0_dbg command to toggle individual flags for individual GPIO cores and with the ut699_dbg command to toggle individual flags for all GPIO cores. The subcommands can be used with the gpio0_dbg command to change and list the settings of all flags for individual GPIO cores.

Table 7.9. GPIO debug flags

<table>
<thead>
<tr>
<th>Flag/subcommand</th>
<th>Trace</th>
</tr>
</thead>
<tbody>
<tr>
<td>GAISLER_GPIO_ACC</td>
<td>GPIO register accesses</td>
</tr>
<tr>
<td>GAISLER_GPIO_IRQ</td>
<td>GPIO interrupts</td>
</tr>
<tr>
<td>all</td>
<td>Set all GPIO debug flags for the core</td>
</tr>
<tr>
<td>clean</td>
<td>Set none of the GPIO debug flags for the core</td>
</tr>
<tr>
<td>list</td>
<td>List the current setting of the debug flags for the core</td>
</tr>
</tbody>
</table>

7.7.4. GPIO model API

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

```c
/* GPIO input provider */
struct gpio_input {
    struct input_inp _b;
    int (*gpioout)(struct gpio_input *ctrl, unsigned int out);
    int (*gpioin) (struct gpio_input *ctrl, unsigned int in);
};
```
The `gpioout` callback should be set by the user module at startup. The `gpioin` callback is set by the UT699 AHB module. The `gpioout` callback is called by the UT699 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.

Table 7.10. `gpioout` callback parameters

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>out</td>
<td>The values of the output pins</td>
</tr>
</tbody>
</table>

Table 7.11. `gpioin` callback parameters

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>in</td>
<td>The input pin values</td>
</tr>
</tbody>
</table>

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

See the `ut699/examples/input` for an example implementation.

### 7.8. CAN interface

The UT699 AHB module contains 2 CAN_OC cores which models the CAN_OC cores available in the UT699. For core details and register specification please see the UT699 manual.

#### 7.8.1. Start up options

**CAN 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]`
  
  Specifies whether the CAN_OC core will wait for a acknowledgment packet on transmission. This option must be put after `-can_ocX_connect`.

X in the above options is in the range 1-2.

#### 7.8.2. Commands

**CAN core TSIM 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 acknowledgment 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`
  
  Toggle, set, clear, list debug flags for the CAN_OC core.

X in the above commands is in the range 1-2.

#### 7.8.3. 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 `ut699_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 7.12. CAN debug flags**

<table>
<thead>
<tr>
<th>Flag</th>
<th>Trace</th>
</tr>
</thead>
<tbody>
<tr>
<td>GAISLER_CAN_OC_ACC</td>
<td>CAN_OC register accesses</td>
</tr>
<tr>
<td>GAISLER_CAN_OC_RXPACKET</td>
<td>CAN_OC received messages</td>
</tr>
<tr>
<td>GAISLER_CAN_OC_TXPACKET</td>
<td>CAN_OC transmitted messages</td>
</tr>
<tr>
<td>GAISLER_CAN_OC_ACK</td>
<td>CAN_OC acknowledgements</td>
</tr>
<tr>
<td>GAISLER_CAN_OC_IRQ</td>
<td>CAN_OC interrupts</td>
</tr>
<tr>
<td>all</td>
<td>Set all debug flags for the core</td>
</tr>
<tr>
<td>clean</td>
<td>Set none of the debug flags for the core</td>
</tr>
<tr>
<td>list</td>
<td>List the current setting of the debug flags for the core</td>
</tr>
</tbody>
</table>

### 7.8.4. 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.

### 7.8.5. 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 7.13. CAN packet types**

<table>
<thead>
<tr>
<th>Type</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>Message</td>
<td>0x00</td>
</tr>
<tr>
<td>Error counter</td>
<td>0xFD</td>
</tr>
<tr>
<td>Acknowledge</td>
<td>0xFE</td>
</tr>
<tr>
<td>Acknowledge config</td>
<td>0xFF</td>
</tr>
</tbody>
</table>

#### 7.8.5.1. CAN message packet format

Used to send and receive CAN messages.

```
31 0
```

<table>
<thead>
<tr>
<th>0x0</th>
<th>LENGTH</th>
</tr>
</thead>
<tbody>
<tr>
<td>31:0</td>
<td>LENGTH, specifies the length of the rest of the packet</td>
</tr>
</tbody>
</table>

**Figure 7.4. CAN message packet format**

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

**Figure 7.4. CAN message packet format**
7.8.5.2. Error counter packet format

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

<table>
<thead>
<tr>
<th>Byte #</th>
<th>Field</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>4</td>
<td>Packet type</td>
<td>Type of packet, 0xFD for error counter packets</td>
</tr>
<tr>
<td>5</td>
<td>Register</td>
<td>0 - RX error counter, 1 - TX error counter</td>
</tr>
<tr>
<td>6</td>
<td>Value</td>
<td>Value to write to error counter</td>
</tr>
</tbody>
</table>

![Error counter packet format](image)

7.8.5.3. 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.

<table>
<thead>
<tr>
<th>Byte #</th>
<th>Field</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>4</td>
<td>Packet type</td>
<td>Type of packet, 0xFE for acknowledge packets</td>
</tr>
<tr>
<td>5</td>
<td>Ack payload</td>
<td>0 - No acknowledge, 1 - Acknowledge</td>
</tr>
</tbody>
</table>

![Acknowledge packet format](image)

7.8.5.4. Acknowledge configuration 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.

<table>
<thead>
<tr>
<th>Byte #</th>
<th>Field</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>4</td>
<td>Packet type</td>
<td>Type of packet, 0xFF for acknowledge configuration packets</td>
</tr>
<tr>
<td>5</td>
<td>Ack configuration</td>
<td>bit 0 - Unused, bit 1 - Ack packet enable, 1 - enabled, 0 - disabled, bit 2 - Set ack packet payload, 1 - ACK, 0 - NAK</td>
</tr>
</tbody>
</table>

![Acknowledge configuration packet format](image)
8. Cobham UT699E emulation

To emulate the UT699E chip the \texttt{-ut699e} option should be used. That sets up parameters for core TSIM2 to match UT699E. In addition, the UT699E loadable AHB module implements emulation models for additional cores of the UT699E. Apart from an overview of what TSIM supports for UT699E emulation, this chapter is mainly about the UT699E AHB module.

The following table lists which cores in the UT699E are modelled by TSIM or not. Some supported cores are modelled by the core TSIM2 and some in the UT699E AHB module. 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 the rest of the chapter for the AHB module models.

Table 8.1. Simulation models for UT699E

<table>
<thead>
<tr>
<th>Core</th>
<th>Status</th>
<th>Notes</th>
</tr>
</thead>
<tbody>
<tr>
<td>LEON3FT</td>
<td>Supported by core TSIM2</td>
<td>Only one CPU is modelled. No FT features are modelled.</td>
</tr>
<tr>
<td>GRFPU</td>
<td>Supported by core TSIM2</td>
<td></td>
</tr>
<tr>
<td>AHBSTAT</td>
<td>Supported by core TSIM2</td>
<td></td>
</tr>
<tr>
<td>APBCTRL</td>
<td>Supported by core TSIM2</td>
<td></td>
</tr>
<tr>
<td>APBUART</td>
<td>Supported by core TSIM2</td>
<td></td>
</tr>
<tr>
<td>FTMCTRL</td>
<td>Supported by core TSIM2</td>
<td>No FT features are modelled</td>
</tr>
<tr>
<td>GPTIMER</td>
<td>Supported by core TSIM2</td>
<td></td>
</tr>
<tr>
<td>IRQMP</td>
<td>Supported by core TSIM2</td>
<td></td>
</tr>
<tr>
<td>CAN_OC</td>
<td>Supported by AHB module</td>
<td></td>
</tr>
<tr>
<td>GRETH</td>
<td>Supported by AHB module</td>
<td></td>
</tr>
<tr>
<td>GRGPIO</td>
<td>Supported by AHB module</td>
<td></td>
</tr>
<tr>
<td>GRPCI</td>
<td>Supported by AHB module</td>
<td>Including DMA controller</td>
</tr>
<tr>
<td>GRSPW2</td>
<td>Supported by AHB module</td>
<td></td>
</tr>
<tr>
<td>CLKGATE</td>
<td>Not supported</td>
<td></td>
</tr>
<tr>
<td>AHBJTAG</td>
<td>Not supported</td>
<td>Debug link</td>
</tr>
<tr>
<td>AHBUART</td>
<td>Not supported</td>
<td>Debug link</td>
</tr>
<tr>
<td>DSU3</td>
<td>Not supported</td>
<td>Debug unit</td>
</tr>
</tbody>
</table>

TSIM supports running several AHB modules. User defined models for unsupported cores can be used in parallel with the UT699E AHB module. User defined models can also override the simulation models in the UT699E AHB module.

8.1. Overview of the UT699E AHB module

The UT699E AHB module is a loadable AHB module that implements UT699E peripherals in addition to what is emulated by core TSIM as listed in Table 8.1. The interfaces are modelled at packet/transaction/message level and provides an easy way to connect the simulated UT699E to a larger simulation framework. See also Section 5.4.1 on some limitations of some features when using this module. The UT699E AHB module is only supported for TSIM2 LEON3.

The following files are delivered with the UT699E TSIM module:

Table 8.2. Files delivered with the UT699E TSIM module

<table>
<thead>
<tr>
<th>File</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>ut699/linux/ut699e.so</td>
<td>UT699E AHB module for Linux</td>
</tr>
</tbody>
</table>
8.2. Loading the module

The module is loaded using the TSIM2 option -ahbm. A user input module for SPI and PCI can optionally be declared, between -designinput and -designinputend options. For example:

On Linux:

```bash
tsim-leon3 -ut699e -ahbm ut699/linux/ut699e.so -designinput ./input.so -designinputend
```

On Windows:

```bash
tsim-leon3 -ut699e -ahbm ut699/win32/ut699e.dll -designinput input.dll -designinputend
```

The option -ut699e needs to be given to TSIM to enable the UT699E processor configuration. The above line loads the UT699E AHB module ut699e.so which in turn loads the user user input module ./input.so. The user input module ./input.so communicates with ut699e.so using the user module interface described in ut699inputprovider.h, while ut699e.so communicates with TSIM via the AHB interface.

Example user input modules can be found in ut699/examples/input/.

8.2.1. User input module interface

The SPI and PCI models in the UT699E module uses a user supplied user input module, in the form of a dynamic loadable library, that models the outside world. This section describes the general interface for hooking up the user module to the UT699E module. The details on the interfaces to the particular cores, see their respective sections.

<table>
<thead>
<tr>
<th>File</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>ut699/win32/ut699e.dll</td>
<td>UT699E AHB module for Windows</td>
</tr>
<tr>
<td>ut699/examples/input</td>
<td>The input directory contains two examples of PCI user modules</td>
</tr>
<tr>
<td>ut699/examples/input/README.txt</td>
<td>Description of the user module examples</td>
</tr>
<tr>
<td>ut699/examples/input/pci.c</td>
<td>PCI user module example that makes UT699E PCI initiator accesses</td>
</tr>
<tr>
<td>ut699/examples/input/pci_target.c</td>
<td>PCI user module example that makes UT699E PCI target accesses</td>
</tr>
<tr>
<td>ut699/examples/input/gpio.c</td>
<td>GPIO user module example</td>
</tr>
<tr>
<td>ut699/examples/input/ut699inputprovider.h</td>
<td>Interface between the UT699E module and the user defined PCI module</td>
</tr>
<tr>
<td>ut699/examples/input/pci_input.h</td>
<td>UT699E PCI input provider definitions</td>
</tr>
<tr>
<td>ut699/examples/input/input.h</td>
<td>Generic input provider definitions</td>
</tr>
<tr>
<td>ut699/examples/input/tsim.h</td>
<td>TSIM interface definitions</td>
</tr>
<tr>
<td>ut699/examples/input/end.h</td>
<td>Defines the endian of the local machine</td>
</tr>
<tr>
<td>ut699/examples/test</td>
<td>The test directory contains tests that can be executed in TSIM</td>
</tr>
<tr>
<td>ut699/examples/test/README.txt</td>
<td>Description of the tests</td>
</tr>
<tr>
<td>ut699/examples/test/Makefile</td>
<td>Makefile for building the tests</td>
</tr>
<tr>
<td>ut699/examples/test/cansend.c</td>
<td>CAN transmission test</td>
</tr>
<tr>
<td>ut699/examples/test/canrecv.c</td>
<td>CAN reception test</td>
</tr>
<tr>
<td>ut699/examples/test/pci.c</td>
<td>PCI interface test</td>
</tr>
<tr>
<td>ut699/examples/test/pcitest.h</td>
<td>Header file for PCI test</td>
</tr>
</tbody>
</table>
A user supplied dynamic library should expose a public symbol `ut699inputsystem` of type `struct ut699_subsystem *`. The `struct ut699_subsystem` is defined in `ut699inputprovider.h` as:

```c
struct ut699_subsystem {
    void (*ut699_inp_setup) (int id,
                             struct ut699_inp_layout * l,
                             char **argv, int argc);
    void (*ut699_inp_restart) (int id,
                               struct ut699_inp_layout * l);
    struct sim_interface *simif;
};
```

The callback `ut699_inp_restart` will be called every time the simulator restarts. At initialization the callback `ut699_inp_setup` will be called once, supplied with a pointer to structure `struct ut699_inp_layout` defined in `ut699inputprovider.h`.

The user module can access the global TSIM `struct sim_interface` structure through the `simif` member. See Chapter 5 for more details.

The user supplied dynamic library should, in its `ut699_inp_setup` function, “claim” the input structs it uses using the `INPUT_CLAIM` macro. For example `INPUT_CLAIM(l->gpio)` as in the example below.

A user supplied dynamic library that only sets up a model for GPIO could look like this:

```c
#include <stdio.h>
#include <string.h>
#include "tsim.h"
#include "ut699inputprovider.h"
extern struct ut699_subsystem *ut699inputsystem;
static struct ut699_inp_layout *lay = 0;
static void Change(struct gpio_input *ctrl) {
    ...
}
int gpioout(struct gpio_input *ctrl, unsigned int out) {
    ...
}
static void ut699_inp_setup (int id,
                             struct ut699_inp_layout * l,
                             char **argv, int argc) {
    lay = l;
    printf("User-dll: ut699_inp_setup:Claiming %s\n", l->gpio._b.name);
    INPUT_CLAIM(l->gpio);
    l->gpio.gpioout = gpioout;
    ut699inputsystem->simif->event(Change, (unsigned long)l->gpio, 10000000);
}
static struct ut699_subsystem ut699_gpio = {
    ut699_inp_setup, 0, 0
};
struct ut699_subsystem *ut699inputsystem = &ut699_gpio;
```

A Makefile that would build a user supplied dynamic library gpio.(dll|so) could look like this:

```makefile
M_DLL_FIX=$(if $(shell $(uname)|grep MINGW32),dll,so)
M_LIB=$(if $(shell $(uname)|grep MINGW32),-lws2_32 -luser32 -lkernel32 -lwinmm,)
all: gpio.$(M_DLL_FIX)
gpio.$(M_DLL_FIX) : gpio.o
    $(CC) -shared -g gpio.o -o gpio.$(M_DLL_FIX) $(M_LIB)
    $(CC) -g gpio.o
    $(CC) -g -o gpio.o
    clean:
    -rm -f *.o *.so
```

The user can then specify the user module to be loaded by the `ut699e.so` AHB module using the `-designinput` and `-designinputend` command line options. The first argument after `-designinput` is the user module. Arguments after that are passed to the user input module in the call to `ut699_inp_setup`. 

The user supplied dynamic library should expose a public symbol `ut699inputsystem` of type `struct ut699_subsystem *`. The `struct ut699_subsystem` is defined in `ut699inputprovider.h` as:

```c
struct ut699_subsystem {
    void (*ut699_inp_setup) (int id,
                             struct ut699_inp_layout * l,
                             char **argv, int argc);
    void (*ut699_inp_restart) (int id,
                               struct ut699_inp_layout * l);
    struct sim_interface *simif;
};
```

The callback `ut699_inp_restart` will be called every time the simulator restarts. At initialization the callback `ut699_inp_setup` will be called once, supplied with a pointer to structure `struct ut699_inp_layout` defined in `ut699inputprovider.h`.

The user module can access the global TSIM `struct sim_interface` structure through the `simif` member. See Chapter 5 for more details.

The user supplied dynamic library should, in its `ut699_inp_setup` function, “claim” the input structs it uses using the `INPUT_CLAIM` macro. For example `INPUT_CLAIM(l->gpio)` as in the example below.

A user supplied dynamic library that only sets up a model for GPIO could look like this:

```c
#include <stdio.h>
#include <string.h>
#include "tsim.h"
#include "ut699inputprovider.h"
extern struct ut699_subsystem *ut699inputsystem;
static struct ut699_inp_layout *lay = 0;
static void Change(struct gpio_input *ctrl) {
    ...
}
int gpioout(struct gpio_input *ctrl, unsigned int out) {
    ...
}
static void ut699_inp_setup (int id,
                             struct ut699_inp_layout * l,
                             char **argv, int argc) {
    lay = l;
    printf("User-dll: ut699_inp_setup:Claiming %s\n", l->gpio._b.name);
    INPUT_CLAIM(l->gpio);
    l->gpio.gpioout = gpioout;
    ut699inputsystem->simif->event(Change, (unsigned long)l->gpio, 10000000);
}
static struct ut699_subsystem ut699_gpio = {
    ut699_inp_setup, 0, 0
};
struct ut699_subsystem *ut699inputsystem = &ut699_gpio;
```

A Makefile that would build a user supplied dynamic library gpio.(dll|so) could look like this:

```makefile
M_DLL_FIX=$(if $(shell $(uname)|grep MINGW32),dll,so)
M_LIB=$(if $(shell $(uname)|grep MINGW32),-lws2_32 -luser32 -lkernel32 -lwinmm,)
all: gpio.$(M_DLL_FIX)
gpio.$(M_DLL_FIX) : gpio.o
    $(CC) -shared -g gpio.o -o gpio.$(M_DLL_FIX) $(M_LIB)
    $(CC) -g gpio.o
    $(CC) -g -o gpio.o
    clean:
    -rm -f *.o *.so
```

The user can then specify the user module to be loaded by the `ut699e.so` AHB module using the `-designinput` and `-designinputend` command line options. The first argument after `-designinput` is the user module. Arguments after that are passed to the user input module in the call to `ut699_inp_setup`. 

The user supplied dynamic library should expose a public symbol `ut699inputsystem` of type `struct ut699_subsystem *`. The `struct ut699_subsystem` is defined in `ut699inputprovider.h` as:

```c
struct ut699_subsystem {
    void (*ut699_inp_setup) (int id,
                             struct ut699_inp_layout * l,
                             char **argv, int argc);
    void (*ut699_inp_restart) (int id,
                               struct ut699_inp_layout * l);
    struct sim_interface *simif;
};
```

The callback `ut699_inp_restart` will be called every time the simulator restarts. At initialization the callback `ut699_inp_setup` will be called once, supplied with a pointer to structure `struct ut699_inp_layout` defined in `ut699inputprovider.h`.

The user module can access the global TSIM `struct sim_interface` structure through the `simif` member. See Chapter 5 for more details.

The user supplied dynamic library should, in its `ut699_inp_setup` function, “claim” the input structs it uses using the `INPUT_CLAIM` macro. For example `INPUT_CLAIM(l->gpio)` as in the example below.

A user supplied dynamic library that only sets up a model for GPIO could look like this:

```c
#include <stdio.h>
#include <string.h>
#include "tsim.h"
#include "ut699inputprovider.h"
extern struct ut699_subsystem *ut699inputsystem;
static struct ut699_inp_layout *lay = 0;
static void Change(struct gpio_input *ctrl) {
    ...
}
int gpioout(struct gpio_input *ctrl, unsigned int out) {
    ...
}
static void ut699_inp_setup (int id,
                             struct ut699_inp_layout * l,
                             char **argv, int argc) {
    lay = l;
    printf("User-dll: ut699_inp_setup:Claiming %s\n", l->gpio._b.name);
    INPUT_CLAIM(l->gpio);
    l->gpio.gpioout = gpioout;
    ut699inputsystem->simif->event(Change, (unsigned long)l->gpio, 10000000);
}
static struct ut699_subsystem ut699_gpio = {
    ut699_inp_setup, 0, 0
};
struct ut699_subsystem *ut699inputsystem = &ut699_gpio;
```
For example: -designinput ut699/examples/input/gpio.so -gpioverbose -designinputend will specify that the example user input module gpio.so should be used and that it should receive the argument -gpioverbose.

8.3. Debugging

To enable printout of debug information the -ut699_dbg/on flag switch can be used. Alternatively one can issue the ut699_dbg/on flag command on the TSIM2 command line to toggle the on/off state of a flag. The debug flags that are available are described for each core in the following sections and can be listed by ut699_dbg/help.

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.

8.4. 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 the UT699E. For core details and register specification please see the UT699E manual.

The following features are supported:
- Direct Memory Access
- Interrupts

8.4.1. Start up options

Ethernet core start up options

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

8.4.2. Commands

Ethernet core TSIM commands

greth_connect host[:port]
Connect Ethernet core to a packet server at the specified host and port. Default port is 2224.
greth_status
Print Ethernet register status

8.4.3. Debug flags

The following debug flags are available for the Ethernet interface. Use the them in conjunction with the ut699_dbg/on command to enable different levels of debug information.

Table 8.3. Ethernet debug flags

<table>
<thead>
<tr>
<th>Flag</th>
<th>Trace</th>
</tr>
</thead>
<tbody>
<tr>
<td>GAISLER_GRETH_ACC</td>
<td>GRETH accesses</td>
</tr>
<tr>
<td>GAISLER_GRETH_L1</td>
<td>GRETH accesses verbose</td>
</tr>
<tr>
<td>GAISLER_GRETH_TX</td>
<td>GRETH transmissions</td>
</tr>
<tr>
<td>GAISLER_GRETH_RX</td>
<td>GRETH reception</td>
</tr>
<tr>
<td>GAISLER_GRETH_RXPACKET</td>
<td>GRETH received packets</td>
</tr>
<tr>
<td>GAISLER_GRETH_RXCTRL</td>
<td>GRETH RX packet server protocol</td>
</tr>
<tr>
<td>GAISLER_GRETH_RXBDCTRL</td>
<td>GRETH RX buffer descriptors DMA</td>
</tr>
<tr>
<td>GAISLER_GRETH_RXBDCTRL</td>
<td>GRETH TX packet server protocol</td>
</tr>
<tr>
<td>GAISLER_GRETH_TXPACKET</td>
<td>GRETH transmitted packets</td>
</tr>
<tr>
<td>GAISLER_GRETH_IRQ</td>
<td>GRETH interrupts</td>
</tr>
</tbody>
</table>
8.4.4. 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 `-grethconnect` start-up parameter or using the `greth_connect` command.

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.

8.4.5. 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:

```
<table>
<thead>
<tr>
<th>31:0 LEN</th>
</tr>
</thead>
</table>
```

Length of rest of packet: 4 + number of data bytes

Header at offset 0x4:

```
<table>
<thead>
<tr>
<th>31:16 R</th>
<th>15:8 IPID</th>
<th>7:5 TYPE</th>
<th>4:0 R</th>
</tr>
</thead>
</table>
```

Reserved for future use. Must be set to 0.
IP core ID: 1 for Ethernet
Packet type: 0 for data packets
Reserved for future use. Must be set to 0.

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

*Figure 8.1. Ethernet data packet*

8.5. SpaceWire interface with RMAP support

The UT699E AHB module contains 4 GRSPW2 cores which models the GRSPW2 cores available in the UT699E. For core details and register specification please see the UT699E manual.

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 the UT699E 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.
8.5.1. 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_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_spw_max_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 val`
  When val is 1, 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 val is 0, the last data part packet can contain both data and an end marker. This is the default when simple mode is enabled.
- `grspw_simple 1`
  Set all GRSPW2 cores to “simple mode”. This can be used for backward compatibility with TSIM 2.0.44 and backwards. See the separate section on simple mode for details. Note the needed 1 argument.
- `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 1` option.

X in the above options has the range 1-4.

8.5.2. Commands

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_status`
  Print status for GRSPW2 core X.
- `grspwX_dbg [argument]`
  Sets, clears, lists, toggles debug options for individual GRSPW2 cores. Using `grspwX_dbg` without any arguments will list all available options. The list argument will list current debug option settings. The all argument will turn on all debug options. The clean argument will turn off all debug options. Using one of the available debug options as argument will toggle that debug option. See the section below.

X in the above commands has the range 1-4.

8.5.3. 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 `ut699_dbgon` command to toggle individual flags for all SpaceWire 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>
<thead>
<tr>
<th>Flag</th>
<th>Trace</th>
</tr>
</thead>
<tbody>
<tr>
<td>GAISLER_GRSPW_ACC</td>
<td>GRSPW accesses</td>
</tr>
<tr>
<td>GAISLER_GRSPW_RXPACKET</td>
<td>GRSPW received packets</td>
</tr>
</tbody>
</table>
### 8.5.4. 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 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 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.

### 8.5.5. 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 8.5. Packet types

<table>
<thead>
<tr>
<th>Type</th>
<th>Value</th>
<th>Direction</th>
<th>Notes</th>
</tr>
</thead>
<tbody>
<tr>
<td>Data part</td>
<td>0</td>
<td>Both</td>
<td>Only when in run state</td>
</tr>
<tr>
<td>Time code</td>
<td>1</td>
<td>Both</td>
<td>Only when in run state</td>
</tr>
<tr>
<td>Link state</td>
<td>2</td>
<td>Server to client</td>
<td></td>
</tr>
<tr>
<td>Link control</td>
<td>3</td>
<td>Client to server</td>
<td>Must be sent for model to reach run state</td>
</tr>
<tr>
<td>RX frequency</td>
<td>4</td>
<td>Server to client</td>
<td></td>
</tr>
<tr>
<td>Error injection</td>
<td>5</td>
<td>Client to server</td>
<td>Optional</td>
</tr>
<tr>
<td>Packet error request</td>
<td>6</td>
<td>Client to server</td>
<td>Optional</td>
</tr>
</tbody>
</table>

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 \texttt{-grspw\_tx\_max\_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.

**8.5.5.1. Flow control limitations**

Flow control in terms of credit is not modeled 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.

**8.5.5.2. 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 \texttt{END} field set to 0 or 1. The other parts should have no end marker, i.e. the \texttt{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 \texttt{-grspw\_endpacket} option. Splitting up SpaceWire packets into several data containing data part packets can be enabled with the \texttt{-grspw\_tx\_max\_part\_len} option.

Packet length at offset 0x0:

<table>
<thead>
<tr>
<th>LEN</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
</tr>
</tbody>
</table>

Length of rest of packet: 4 + number of data bytes in the part

Header at offset 0x4:

<table>
<thead>
<tr>
<th>R</th>
<th>IPID</th>
<th>TYPE</th>
<th>R</th>
<th>END</th>
</tr>
</thead>
<tbody>
<tr>
<td>31:16</td>
<td>Reserved for future use. Must be set to 0.</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>15:8</td>
<td>IPID</td>
<td>IP core ID: 0 for SpaceWire</td>
<td></td>
<td></td>
</tr>
<tr>
<td>7:5</td>
<td>TYPE</td>
<td>Packet type: 0 for data part packets</td>
<td></td>
<td></td>
</tr>
<tr>
<td>4:2</td>
<td>R</td>
<td>Reserved for future use. Must be set to 0.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>1:0</td>
<td>END</td>
<td>End marker: 0: Normal End of Packet, 1: Error End of Packet, 2: No end marker</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Offset 0x8: The data bytes of the part starts here

*Figure 8.2. SpaceWire data part packet*
8.5.5.3. Time code packet format

Packet length at offset 0x0:

| 31:0 | LEN | Length of rest of packet: 8 |

Header at offset 0x4:

| 31 | 16 15 | 8 7 5 4 0 |

| 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 |

| 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 8.3. SpaceWire time code packet

8.5.5.4. 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:

<table>
<thead>
<tr>
<th>LEN</th>
</tr>
</thead>
<tbody>
<tr>
<td>31:0 LEN Length of rest of packet: 4</td>
</tr>
</tbody>
</table>

Header at offset 0x4:

<table>
<thead>
<tr>
<th>R</th>
<th>ERROR</th>
<th>IPID</th>
<th>TYPE</th>
<th>R</th>
<th>LS</th>
</tr>
</thead>
<tbody>
<tr>
<td>31:19</td>
<td>R</td>
<td>Reserved for future use. Must be set to 0.</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>18:16</td>
<td>ERROR</td>
<td>Link error: 0: None, 1: Disconnect, 2: Parity, 3: Escape, 4: Credit</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>15:8</td>
<td>IPID</td>
<td>IP core ID: 0 for SpaceWire</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>7:5</td>
<td>TYPE</td>
<td>Packet type: 2 for link state packets</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>4:3</td>
<td>R</td>
<td>Reserved for future use. Must be set to 0.</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>2:0</td>
<td>LS</td>
<td>Link State: 0: Error reset, 1: Error wait, 2: Ready, 3: Started, 4: Connecting, 5: Run</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Figure 8.4. SpaceWire link state packet

8.5.5.5. 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:

<table>
<thead>
<tr>
<th>LEN</th>
</tr>
</thead>
<tbody>
<tr>
<td>31:0 LEN</td>
</tr>
</tbody>
</table>

Header at offset 0x4:

<table>
<thead>
<tr>
<th>R</th>
<th>IPID</th>
<th>TYPE</th>
<th>R</th>
<th>AS</th>
<th>LS</th>
<th>LD</th>
</tr>
</thead>
<tbody>
<tr>
<td>31:16</td>
<td>R</td>
<td>Reserved for future use. Must be set to 0.</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>15:8</td>
<td>IPID</td>
<td>IP core ID: 0 for SpaceWire</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>7:5</td>
<td>TYPE</td>
<td>Packet type: 3 for link control packets</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>4:3</td>
<td>R</td>
<td>Reserved for future use. Must be set to 0.</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>AS</td>
<td>Link autostart.</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>LS</td>
<td>Link start.</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>LD</td>
<td>Link disable.</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Startup frequency in MHz at offset 0x8:

<table>
<thead>
<tr>
<th>SFREQ</th>
</tr>
</thead>
<tbody>
<tr>
<td>31:0 SFREQ</td>
</tr>
</tbody>
</table>

TX frequency in MHz at offset 0xc:

<table>
<thead>
<tr>
<th>TFREQ</th>
</tr>
</thead>
<tbody>
<tr>
<td>31:0 TFREQ</td>
</tr>
</tbody>
</table>

*Figure 8.5. SpaceWire link control packet*

### 8.5.5.6. 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  Length of rest of packet: 8

Header at offset 0x4:

31  16  15  8  7  5  4  0

| 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  RX frequency in MHz, big endian IEEE-754 32-bit float

Figure 8.6. SpaceWire rx frequency packet

8.5.5.7. 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 | Length of rest of packet: 4 |

Header at offset 0x4:

<table>
<thead>
<tr>
<th>31</th>
<th>21</th>
<th>20</th>
<th>19</th>
<th>18</th>
<th>16</th>
<th>15</th>
<th>8</th>
<th>7</th>
<th>5</th>
<th>4</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>R</td>
<td>OE</td>
<td>R</td>
<td>ERROR</td>
<td>IPID</td>
<td>TYPE</td>
<td>R</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

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 8.7. SpaceWire link error injection packet

8.5.5.8. 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 | Length of rest of packet: 16 |

Header at offset 0x4:

<table>
<thead>
<tr>
<th>31</th>
<th>19</th>
<th>18</th>
<th>16</th>
<th>15</th>
<th>8</th>
<th>7</th>
<th>5</th>
<th>4</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>R</td>
<td>ERROR</td>
<td>IPID</td>
<td>TYPE</td>
<td>R</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

- **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:

<table>
<thead>
<tr>
<th>31</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>MSW</td>
<td></td>
</tr>
</tbody>
</table>

- **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:

<table>
<thead>
<tr>
<th>31</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>LSW</td>
<td></td>
</tr>
</tbody>
</table>

- **31:0** **LSW** Bits 31:0 of unsigned 64-bit big endian integer

Reserved field at offset 0x10:

<table>
<thead>
<tr>
<th>31</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>R</td>
<td></td>
</tr>
</tbody>
</table>

- **31:0** **R** Reserved for future use. Must be set to 0.

**Figure 8.8. SpaceWire packet error request packet**

### 8.5.6. 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 1` 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_tx_max_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.

8.6. PCI initiator/target interface

The UT699E AHB module models the PCI core available in the UT699E ASIC. For core details and register specification please see the UT699E manual.

8.6.1. Connecting a user PCI model with the UT699E module

See Section 8.2 for details on how to connect the user PCI model to the UT699E module.

8.6.2. Commands

**PCI Commands**

`pci_status`

Print status for the PCI core

8.6.3. Debug flags

The following debug flags are available for the PCI interface. Use them in conjunction with the ut699_dbgon command to enable different levels of debug information.

<table>
<thead>
<tr>
<th>Flag</th>
<th>Trace</th>
</tr>
</thead>
<tbody>
<tr>
<td>GAISSLER_GRPCI_ACC</td>
<td>AHB accesses to/from PCI core</td>
</tr>
<tr>
<td>GAISSLER_GRPCI_REGACC</td>
<td>GRPCI APB register accesses</td>
</tr>
<tr>
<td>GAISSLER_GRPCI_DMA_REGACC</td>
<td>PCIDMA APB register accesses</td>
</tr>
<tr>
<td>GAISSLER_GRPCI_DMA</td>
<td>GRPCI DMA accesses on the AHB bus</td>
</tr>
<tr>
<td>GAISSLER_GRPCI_TARGET_ACC</td>
<td>GRPCI target accesses</td>
</tr>
<tr>
<td>GAISSLER_GRPCI_INIT</td>
<td>Print summary on startup</td>
</tr>
</tbody>
</table>

8.6.4. PCI bus model API

The structure `struct grpci_input` models the PCI bus. It is defined as:

```c
struct grpci_input {
    struct input_in   b;
    int (*acc)(struct grpci_input *ctrl, int cmd, unsigned int addr,
               unsigned int *data, unsigned int *abort, unsigned int *ws);
    int (*target_acc)(struct grpci_input *ctrl, int cmd, unsigned int addr,
                      unsigned int *data, unsigned int *mexc);
};
```

The acc callback should be set by the PCI user module at startup. It is called by the UT699E module whenever it reads/writes as a PCI bus master.

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>cmd</td>
<td>Command to execute, see Section 8.6.2 details.</td>
</tr>
<tr>
<td>addr</td>
<td>PCI address.</td>
</tr>
</tbody>
</table>
### Table 8.8. target_acc parameters

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>cmd</td>
<td>Command to execute, see Section 8.6.2 for details. I/O cycles are not supported by the UT699E target.</td>
</tr>
<tr>
<td>addr</td>
<td>PCI address. Should always be word aligned for read accesses.</td>
</tr>
<tr>
<td>data</td>
<td>Data buffer. The read data is returned in *data for read commands or the data in *data is written for write commands.</td>
</tr>
<tr>
<td>wsize</td>
<td>0: 8-bit access 1: 16-bit access, 2: 32-bit access. Should always be 2 for read accesses.</td>
</tr>
<tr>
<td>wsuc</td>
<td>Set 0 to the number of PCI clocks it takes to complete the transaction.</td>
</tr>
<tr>
<td>abort</td>
<td>Set *abort to 1 to generate target abort, or 0 otherwise.</td>
</tr>
</tbody>
</table>

The return value of acc determines if the transaction terminates successfully (1, GRPCI_ACC_OK) or with master abort (0, GRPCI_ACC_MASTER_ABORT).

The callback target_acc is installed by the UT699E AHB module. The PCI user dynamic library can call this function to initiate an access to the UT699E PCI target.

### 8.7. GPIO interface

#### 8.7.1. Connecting a user GPIO model with the UT699E module

See Section 8.2 for details on how to connect the user GPIO model to the UT699E module.

#### 8.7.2. Commands

**GPIO Commands**

- `gpio0_status`
  - Print status for the GPIO core.
- `gpio0_dbg [flag|subcommand]`
  - Toggle, set, clear, list debug flags for the GPIO core.

#### 8.7.3. Debug flags

The following debug flags and debug subcommands are available for GPIO interfaces. The `GAISLER_GPIO_*` flags can be used with the `gpio0_dbg` command to toggle individual flags for individual GPIO cores and with the `ut699_dbg` command to toggle individual flags for all GPIO cores. The subcommands can be used with the `gpio0_dbg` command to change and list the settings of all flags for individual GPIO cores.

### Table 8.9. GPIO debug flags

<table>
<thead>
<tr>
<th>Flag/subcommand</th>
<th>Trace</th>
</tr>
</thead>
<tbody>
<tr>
<td>GAISLER_GPIO_ACC</td>
<td>GPIO register accesses</td>
</tr>
<tr>
<td>GAISLER_GPIO_IRQ</td>
<td>GPIO interrupts</td>
</tr>
<tr>
<td>Flag/subcommand</td>
<td>Trace</td>
</tr>
<tr>
<td>-----------------</td>
<td>----------------------------</td>
</tr>
<tr>
<td>all</td>
<td>Set all GPIO debug flags for the core</td>
</tr>
<tr>
<td>clean</td>
<td>Set none of the GPIO debug flags for the core</td>
</tr>
<tr>
<td>list</td>
<td>List the current setting of the debug flags for the core</td>
</tr>
</tbody>
</table>

8.7.4. GPIO model API

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

```c
/* GPIO input provider */
struct gpio_input {
    struct input_inp _b;
    int (*gpioout)(struct gpio_input *ctrl, unsigned int out);
    int (*gpioin) (struct gpio_input *ctrl, unsigned int in);
};
```

The `gpioout` callback should be set by the user module at startup. The `gpioin` callback is set by the UT699E AHB module. The `gpioout` callback is called by the UT699E 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.

Table 8.10. gpioout callback parameters

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>out</td>
<td>The values of the output pins</td>
</tr>
</tbody>
</table>

Table 8.11. gpioin callback parameters

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>in</td>
<td>The input pin values</td>
</tr>
</tbody>
</table>

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

See the `ut699/examples/input` for an example implementation.

8.8. CAN interface

The UT699E AHB module contains 2 CAN_OC cores which models the CAN_OC cores available in the UT699E. For core details and register specification please see the UT699E manual.

8.8.1. Start up options

CAN core start up options

```bash
-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 acknowledgment packet on transmission. This option must be put after `-can_ocX_connect`.
```

X in the above options is in the range 1-2.

8.8.2. Commands

CAN core TSIM commands

```bash
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 acknowledgment 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
Toggle, set, clear, list debug flags for the CAN_OC core.

X in the above commands is in the range 1-2.

8.8.3. 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 ut699_dbugon 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>
<thead>
<tr>
<th>Flag</th>
<th>Trace</th>
</tr>
</thead>
<tbody>
<tr>
<td>GAISLER_CAN_OC_ACC</td>
<td>CAN_OC register accesses</td>
</tr>
<tr>
<td>GAISLER_CAN_OC_RXPACKET</td>
<td>CAN_OC received messages</td>
</tr>
<tr>
<td>GAISLER_CAN_OC_TXPACKET</td>
<td>CAN_OC transmitted messages</td>
</tr>
<tr>
<td>GAISLER_CAN_OC_ACK</td>
<td>CAN_OC acknowledgements</td>
</tr>
<tr>
<td>GAISLER_CAN_OC_IRQ</td>
<td>CAN_OC interrupts</td>
</tr>
<tr>
<td>all</td>
<td>Set all debug flags for the core</td>
</tr>
<tr>
<td>clean</td>
<td>Set none of the debug flags for the core</td>
</tr>
<tr>
<td>list</td>
<td>List the current setting of the debug flags for the core</td>
</tr>
</tbody>
</table>

8.8.4. 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.

8.8.5. 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>
<thead>
<tr>
<th>Type</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>Message</td>
<td>0x00</td>
</tr>
<tr>
<td>Error counter</td>
<td>0xFD</td>
</tr>
<tr>
<td>Acknowledge</td>
<td>0xFE</td>
</tr>
<tr>
<td>Acknowledge config</td>
<td>0xFF</td>
</tr>
</tbody>
</table>

8.8.5.1. CAN message packet format

Used to send and receive CAN messages.
CAN message

<table>
<thead>
<tr>
<th>Byte #</th>
<th>Description</th>
<th>Bits (MSB-LSB)</th>
</tr>
</thead>
<tbody>
<tr>
<td>4</td>
<td>Protocol ID = 0</td>
<td>7 6 5 4 3 2 1 0</td>
</tr>
<tr>
<td>5</td>
<td>Control</td>
<td>FF RTR - -</td>
</tr>
<tr>
<td>6-9</td>
<td>ID (32 bit word in network byte order)</td>
<td>ID 10-0 (bits 31 - 11 ignored for standard frame format)</td>
</tr>
<tr>
<td></td>
<td></td>
<td>ID 28-0 (bits 31-29 ignored for extended frame format)</td>
</tr>
<tr>
<td>10-17</td>
<td>Data byte 1 - DLC</td>
<td>Data byte n 7-0</td>
</tr>
</tbody>
</table>

Figure 8.9. CAN message packet format

8.8.5.2. Error counter packet format

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

<table>
<thead>
<tr>
<th>Byte #</th>
<th>Field</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>4</td>
<td>Packet type</td>
<td>Type of packet, 0xFD for error counter packets</td>
</tr>
<tr>
<td>5</td>
<td>Register</td>
<td>0 - RX error counter, 1 - TX error counter</td>
</tr>
<tr>
<td>6</td>
<td>Value</td>
<td>Value to write to error counter</td>
</tr>
</tbody>
</table>

Figure 8.10. Error counter packet format

8.8.5.3. 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.

<table>
<thead>
<tr>
<th>Byte #</th>
<th>Field</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>4</td>
<td>Packet type</td>
<td>Type of packet, 0xFE for acknowledge packets</td>
</tr>
<tr>
<td>5</td>
<td>Ack payload</td>
<td>0 - No acknowledge, 1 - Acknowledge</td>
</tr>
</tbody>
</table>

Figure 8.11. Acknowledge packet format
8.8.5.4. 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.

<table>
<thead>
<tr>
<th>31</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x0</td>
<td>LENGTH</td>
</tr>
<tr>
<td>31:0</td>
<td>LENGTH, specifies the length of the rest of the packet</td>
</tr>
</tbody>
</table>

Acknowledgment configuration packet

<table>
<thead>
<tr>
<th>Byte #</th>
<th>Field</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>4</td>
<td>Packet type</td>
<td>Type of packet, 0xFF for acknowledge configuration packets</td>
</tr>
<tr>
<td>5</td>
<td>Ack configuration</td>
<td>bit 0</td>
</tr>
<tr>
<td></td>
<td></td>
<td>bit 1</td>
</tr>
<tr>
<td></td>
<td></td>
<td>bit 2</td>
</tr>
</tbody>
</table>

Figure 8.12. Acknowledge configuration packet format
9. Cobham UT700 emulation

To emulate the UT700 chip the -ut700 option should be used. That sets up parameters for core TSIM2 to match UT700. In addition, the UT700 loadable AHB module implements emulation models for additional cores of the UT700. Apart from an overview of what TSIM supports for UT700 emulation, this chapter is mainly about the UT700 AHB module.

The following table lists which cores in the UT700 are modelled by TSIM or not. Some supported cores are modelled by the core TSIM2 and some in the UT700 AHB module. 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 built-in simulation models and the rest of the chapter for the AHB module models.

<table>
<thead>
<tr>
<th>Core</th>
<th>Status</th>
<th>Notes</th>
</tr>
</thead>
<tbody>
<tr>
<td>LEON3FT</td>
<td>Supported by core TSIM2</td>
<td>Only one CPU is modelled. No FT features are modelled.</td>
</tr>
<tr>
<td>GRFPU</td>
<td>Supported by core TSIM2</td>
<td></td>
</tr>
<tr>
<td>AHBSTAT</td>
<td>Supported by core TSIM2</td>
<td></td>
</tr>
<tr>
<td>APBCTRL</td>
<td>Supported by core TSIM2</td>
<td></td>
</tr>
<tr>
<td>APBUART</td>
<td>Supported by core TSIM2</td>
<td></td>
</tr>
<tr>
<td>FTMCTRL</td>
<td>Supported by core TSIM2</td>
<td>No FT features are modelled</td>
</tr>
<tr>
<td>GPTIMER</td>
<td>Supported by core TSIM2</td>
<td></td>
</tr>
<tr>
<td>IRQMP</td>
<td>Supported by core TSIM2</td>
<td></td>
</tr>
<tr>
<td>CAN_OC</td>
<td>Supported by AHB module</td>
<td></td>
</tr>
<tr>
<td>GRETH</td>
<td>Supported by AHB module</td>
<td></td>
</tr>
<tr>
<td>GRGPO</td>
<td>Supported by AHB module</td>
<td></td>
</tr>
<tr>
<td>GRPCI</td>
<td>Supported by AHB module</td>
<td>Including DMA controller</td>
</tr>
<tr>
<td>GRSPW</td>
<td>Supported by AHB module</td>
<td></td>
</tr>
<tr>
<td>SPICTRL</td>
<td>Supported by AHB module</td>
<td></td>
</tr>
<tr>
<td>CLKGATE</td>
<td>Not supported</td>
<td></td>
</tr>
<tr>
<td>GR1553B</td>
<td>Not supported</td>
<td></td>
</tr>
<tr>
<td>GRTC</td>
<td>Not supported</td>
<td></td>
</tr>
<tr>
<td>GRTM</td>
<td>Not supported</td>
<td></td>
</tr>
<tr>
<td>AHBJTAG</td>
<td>Not supported</td>
<td>Debug link</td>
</tr>
<tr>
<td>AHBUART</td>
<td>Not supported</td>
<td>Debug link</td>
</tr>
<tr>
<td>DSU3</td>
<td>Not supported</td>
<td>Debug unit</td>
</tr>
</tbody>
</table>

TSIM supports running several AHB modules. User defined models for unsupported cores can be used in parallel with the UT700 AHB module. User defined models can also override the simulation models in the UT700 AHB module.

9.1. Overview of the UT700 AHB module

The UT700 AHB module is a loadable AHB module that implements UT700 peripherals in addition to what is emulated by core TSIM as listed in Table 9.1. The interfaces are modelled at packet/transaction/message level and provides an easy way to connect the simulated UT700 to a larger simulation framework. See also Section 5.4.1 on some limitations of some features when using this module. The UT700 AHB module is only supported for TSIM2 LEON3.

The following files are delivered with the UT700 TSIM module:
<table>
<thead>
<tr>
<th>File</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>ut700/linux/ut700.so</td>
<td>UT700 AHB module for Linux</td>
</tr>
<tr>
<td>ut700/win32/ut700.dll</td>
<td>UT700 AHB module for Windows</td>
</tr>
<tr>
<td>ut700/examples/input</td>
<td>The input directory contains two examples of PCI user modules</td>
</tr>
<tr>
<td>ut700/examples/input/README.txt</td>
<td>Description of the user module examples</td>
</tr>
<tr>
<td>ut700/examples/input/Makefile</td>
<td>Makefile for building the user modules</td>
</tr>
<tr>
<td>ut700/examples/input/pci.c</td>
<td>PCI user module example that makes UT700 PCI initiator accesses</td>
</tr>
<tr>
<td>ut700/examples/input/pci_target.c</td>
<td>PCI user module example that makes UT700 PCI target accesses</td>
</tr>
<tr>
<td>ut700/examples/input/ut700inputprovider.h</td>
<td>Interface between the UT700 module and the user defined PCI module</td>
</tr>
<tr>
<td>ut700/examples/input/pci_input.h</td>
<td>UT700 PCI input provider definitions</td>
</tr>
<tr>
<td>ut700/examples/input/input.h</td>
<td>Generic input provider definitions</td>
</tr>
<tr>
<td>ut700/examples/input/ut700inputprovider.h</td>
<td>TSIM interface definitions</td>
</tr>
<tr>
<td>ut700/examples/input/end.h</td>
<td>Defines the endian of the local machine</td>
</tr>
<tr>
<td>ut700/examples/test</td>
<td>The test directory contains tests that can be executed in TSIM</td>
</tr>
<tr>
<td>ut700/examples/test/README.txt</td>
<td>Description of the tests</td>
</tr>
<tr>
<td>ut700/examples/test/Makefile</td>
<td>Makefile for building the tests</td>
</tr>
<tr>
<td>ut700/examples/test/cansend.c</td>
<td>CAN transmission test</td>
</tr>
<tr>
<td>ut700/examples/test/canrec.c</td>
<td>CAN reception test</td>
</tr>
<tr>
<td>ut700/examples/test/pci.c</td>
<td>PCI interface test</td>
</tr>
<tr>
<td>ut700/examples/test/pcitest.h</td>
<td>Header file for PCI test</td>
</tr>
</tbody>
</table>

### 9.2. Loading the module

The module is loaded using the TSIM2 option `-ahbm`. A user input module for SPI, GPIO and PCI can optionally be declared, between `-designinput` and `-designinputend` options. For example:

**On Linux:**
```
tsim-leon3 -ut700 -ahbm ut700/linux/ut700.so -designinput ./input.so -designinputend
```

**On Windows:**
```
tsim-leon3 -ut700 -ahbm ut700/win32/ut700.dll -designinput input.dll -designinputend
```

The option `-ut700` needs to be given to TSIM to enable the UT700 processor configuration. The above line loads the UT700 AHB module `ut700.so` which in turn loads the user user input module `./input.so`. The user input module `./input.so` communicates with `ut700.so` using the user module interface described in `ut700inputprovider.h`, while `ut700.so` communicates with TSIM via the AHB interface.

Example user input modules can be found in `ut700/examples/input/`.

#### 9.2.1. User input module interface

The SPI, GPIO and PCI models in the UT700 module uses a user supplied user input module, in the form of a dynamic loadable library, that models the outside world. This section describes the general interface for hooking
up the user module to the UT700 module. The details on the interfaces to the particular cores, see their respective sections.

A user supplied dynamic library should expose a public symbol ut700inputsystem of type struct ut700_subsystem *. The struct ut700_subsystem is defined in ut700inputprovider.h as:

```c
struct ut700_subsystem {
    void (*ut700_inp_setup) (int id, struct ut700_inp_layout * l, char **argv, int argc);
    void (*ut700_inp_restart) (int id, struct ut700_inp_layout * l);
    struct sim_interface *simif;
};
```

The callback ut700_inp_restart will be called every time the simulator restarts. At initialization the callback ut700_inp_setup will be called once, supplied with a pointer to structure struct ut700_inp_layout defined in ut700inputprovider.h.

```c
struct ut700_inp_layout {
    struct grpci_input grpci;
    struct gpio_input gpio;
    struct spi_input spi;
};
```

The user module can access the global TSIM struct sim_interface structure through the simif member. See Chapter 5 for more details.

The user supplied dynamic library should, in its ut700_inp_setup function, “claim” the input structs it uses using the INPUT_CLAIM macro. For example INPUT_CLAIM(l->gpio) as in the example below.

A user supplied dynamic library that only sets up a model for GPIO could look like this:

```c
#include <stdio.h>
#include <string.h>
#include "tsim.h"
#include "ut700inputprovider.h"

extern struct ut700_subsystem *ut700inputsystem;
static struct ut700_inp_layout *lay = 0;

static void Change(struct gpio_input *ctrl) {
    ...
}

int gpioout(struct gpio_input *ctrl, unsigned int out) {
    ...
}

static void ut700_inp_setup (int id, struct ut700_inp_layout * l, char **argv, int argc) {
    lay = l;
    printf("User-dll: ut700_inp_setup: Claiming %s\n", l->gpio._b.name);
    INPUT_CLAIM(l->gpio);
    l->gpio.gpioout = gpioout;
    ut700inputsystem->simif->event(Change,(unsigned long)&l->gpio,10000000);
}

static struct ut700_subsystem ut700_gpio = {
    ut700_inp_setup,0,0
};

struct ut700_subsystem *ut700inputsystem = &ut700_gpio;
```

A Makefile that would build a user supplied dynamic library gpio.(dll|so) could look like this:

```make
M_DLL_FIX=$(if $(strip $(shell uname|grep MINGW32)),dll,so)
M_LIB=$(if $(strip $(shell uname|grep MINGW32)),-lws2_32 -luser32 -lkernel32 -lwinmm,)
all: gpio.$(M_DLL_FIX)
```
The user can then specify the user module to be loaded by the ut700.so AHB module using the -designinput and -designinputend command line options. The first argument after -designinput is the user module. Arguments after that are passed to the user input module in the call to ut700_inp_setup.

For example: -designinput ut700/examples/input/gpio.so -gpioverbose -designinputend will specify that the example user input module gpio.so should be used and that it should receive the argument -gpioverbose.

9.3. Debugging

To enable printout of debug information the -ut700_dbgon flag switch can be used. Alternatively one can issue the ut700_dbgon flag command on the TSIM2 command line to toggle the on/off state of a flag. The debug flags that are available are described for each core in the following sections and can be listed by ut700_dbgon help.

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.

9.4. 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 the UT700. For core details and register specification please see the UT700 manual.

The following features are supported:

- Direct Memory Access
- Interrupts

9.4.1. Start up options

Ethernet core start up options

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

9.4.2. Commands

Ethernet core TSIM commands

  greth_connect host[:port]
      Connect Ethernet core to a packet server at the specified host and port. Default port is 2224.
  greth_status
      Print Ethernet register status

9.4.3. Debug flags

The following debug flags are available for the Ethernet interface. Use the them in conjunction with the ut700_dbgon command to enable different levels of debug information.

Table 9.3. Ethernet debug flags

<table>
<thead>
<tr>
<th>Flag</th>
<th>Trace</th>
</tr>
</thead>
<tbody>
<tr>
<td>GAISLER_GRETH_ACC</td>
<td>GRETH accesses</td>
</tr>
</tbody>
</table>
### 9.4.4. 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 `-grethconnect` start-up parameter or using the `greth_connect` command.

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.

### 9.4.5. 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:

<table>
<thead>
<tr>
<th>31</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>LEN</td>
<td></td>
</tr>
</tbody>
</table>

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

Header at offset 0x4:

<table>
<thead>
<tr>
<th>31</th>
<th>16</th>
<th>15</th>
<th>8</th>
<th>7</th>
<th>5</th>
<th>4</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>R</td>
<td>IPID</td>
<td>TYPE</td>
<td>R</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

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 9.1. Ethernet data packet*

### 9.5. SpaceWire interface with RMAP support

The UT700 AHB module contains 4 GRSPW2 cores which models the GRSPW2 cores available in the UT700. For core details and register specification please see the UT700 manual.
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 the UT700 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 port loopback (Loop) 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.

9.5.1. Start up options

SpaceWire core start up options

-grspw\_connect host:port
    Connect GRSPW core \( X \) to packet server at specified server and port.

-grspw\_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\_tx\_max\_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 val
    When \( val \) is 1, 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 \( val \) is 0, the last data part packet can contain both data and an end marker. This is the default when simple mode is enabled.

-grspw\_simple 1
    Set all GRSPW2 cores to “simple mode”. This can be used for backward compatibility with TSIM 2.0.44 and backwards. See the separate section on simple mode for details. Note the needed 1 argument.

-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 1 option.

\( X \) in the above options has the range 1-4.

9.5.2. Commands

SpaceWire core TSIM commands

-grspw\_connect host:port
    Connect GRSPW2 core \( X \) to packet server at specified server and TCP port.

-grspw\_server port
    Open a packet server for GRSPW2 core \( X \) on specified TCP port.

-grspw\_status
    Print status for GRSPW2 core \( X \).

-grspw\_dbg [argument]
    Sets, clears, lists, toggles debug options for individual GRSPW2 cores. Using grspw\_dbg without any arguments will list all available options. The list argument will list current debug option settings. The all 
The argument will turn on all debug options. The `clean` argument will turn off all debug options. Using one of the available debug options as argument will toggle that debug option. See the section below.

$x$ in the above commands has the range 1-4.

### 9.5.3. 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 `ut700 dbg on` command to toggle individual flags for all SpaceWire 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 9.4. SpaceWire debug flags

<table>
<thead>
<tr>
<th>Flag</th>
<th>Trace</th>
</tr>
</thead>
<tbody>
<tr>
<td>GAISLER_GRSPW_ACC</td>
<td>GRSPW accesses</td>
</tr>
<tr>
<td>GAISLER_GRSPW_RXPACKET</td>
<td>GRSPW received packets</td>
</tr>
<tr>
<td>GAISLER_GRSPW_RXCTRL</td>
<td>GRSPW rx protocol</td>
</tr>
<tr>
<td>GAISLER_GRSPW_TXPACKET</td>
<td>GRSPW transmitted packets</td>
</tr>
<tr>
<td>GAISLER_GRSPW_TXCTRL</td>
<td>GRSPW tx protocol</td>
</tr>
<tr>
<td>GAISLER_GRSPW_RMAP</td>
<td>GRSPW RMAP accesses</td>
</tr>
<tr>
<td>GAISLER_GRSPW_RMAPPACKET</td>
<td>GRSPW RMAP packet dumps</td>
</tr>
<tr>
<td>GAISLER_GRSPW_RMAPPACKDE</td>
<td>GRSPW RMAP packet decoding</td>
</tr>
<tr>
<td>GAISLER_GRSPW_DMAERR</td>
<td>GRSPW DMA errors</td>
</tr>
<tr>
<td>GAISLER_GRSPW_LINK</td>
<td>Link changes</td>
</tr>
<tr>
<td>GAISLER_GRSPW_PART</td>
<td>TX/RX of GRSPW data part packets</td>
</tr>
<tr>
<td>all</td>
<td>Set all debug flags for the core</td>
</tr>
<tr>
<td>clean</td>
<td>Set none of the debug flags for the core</td>
</tr>
<tr>
<td>list</td>
<td>List the current setting of the debug flags for the core</td>
</tr>
</tbody>
</table>

### 9.5.4. 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 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 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.

### 9.5.5. 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 9.5. Packet types

<table>
<thead>
<tr>
<th>Type</th>
<th>Value</th>
<th>Direction</th>
<th>Notes</th>
</tr>
</thead>
<tbody>
<tr>
<td>Data part</td>
<td>0</td>
<td>Both</td>
<td>Only when in run state</td>
</tr>
<tr>
<td>Time code</td>
<td>1</td>
<td>Both</td>
<td>Only when in run state</td>
</tr>
<tr>
<td>Link state</td>
<td>2</td>
<td>Server to client</td>
<td></td>
</tr>
<tr>
<td>Link control</td>
<td>3</td>
<td>Client to server</td>
<td>Must be sent for model to reach run state</td>
</tr>
<tr>
<td>RX frequency</td>
<td>4</td>
<td>Server to client</td>
<td></td>
</tr>
<tr>
<td>Error injection</td>
<td>5</td>
<td>Client to server</td>
<td>Optional</td>
</tr>
<tr>
<td>Packet error request</td>
<td>6</td>
<td>Client to server</td>
<td>Optional</td>
</tr>
</tbody>
</table>

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_tx_max_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.

9.5.5.1. Flow control limitations

Flow control in terms of credit is not modeled 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.

9.5.5.2. 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_tx_max_part_len` option.
Packet length at offset 0x0:

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>31</td>
<td>0</td>
</tr>
</tbody>
</table>

LEN

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

Header at offset 0x4:

<p>| | | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>31</td>
<td>16</td>
<td>15</td>
<td>8</td>
<td>7</td>
<td>5</td>
</tr>
</tbody>
</table>

LEN

31:0 LEN Length of rest of packet: 4

<p>| | | | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>31</td>
<td>16</td>
<td>15</td>
<td>8</td>
<td>7</td>
<td>5</td>
<td>4</td>
</tr>
</tbody>
</table>

Type

31:8 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

Payload at offset 0x8:

<p>| | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>31</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

LEN

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 9.2. SpaceWire data part packet

9.5.5.3. Time code packet format

Packet length at offset 0x0:

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>31</td>
<td>0</td>
</tr>
</tbody>
</table>

LEN

31:0 LEN Length of rest of packet: 8

Header at offset 0x4:

<p>| | | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>31</td>
<td>16</td>
<td>15</td>
<td>8</td>
<td>7</td>
<td>5</td>
</tr>
</tbody>
</table>

LEN

31:0 LEN Length of rest of packet: 8

<p>| | | | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>31</td>
<td>16</td>
<td>15</td>
<td>8</td>
<td>7</td>
<td>5</td>
<td>4</td>
</tr>
</tbody>
</table>

Type

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:

<p>| | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>31</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

LEN

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 9.3. SpaceWire time code packet
9.5.5.4. 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:

<table>
<thead>
<tr>
<th>31</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>LEN</td>
<td></td>
</tr>
</tbody>
</table>

Length of rest of packet: 4

Header at offset 0x4:

<table>
<thead>
<tr>
<th>31</th>
<th>19</th>
<th>18</th>
<th>16</th>
<th>15</th>
<th>8</th>
<th>7</th>
<th>5</th>
<th>4</th>
<th>3</th>
<th>2</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>R</td>
<td>ERROR</td>
<td>IPID</td>
<td>TYPE</td>
<td>R</td>
<td>LS</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

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 9.4. SpaceWire link state packet

9.5.5.5. 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:

<table>
<thead>
<tr>
<th></th>
<th></th>
<th>LEN</th>
<th>Length of rest of packet: 12</th>
</tr>
</thead>
<tbody>
<tr>
<td>31</td>
<td></td>
<td>0</td>
<td></td>
</tr>
</tbody>
</table>

Header at offset 0x4:

<table>
<thead>
<tr>
<th>31</th>
<th>16</th>
<th>15</th>
<th>8</th>
<th>7</th>
<th>5</th>
<th>4</th>
<th>3</th>
<th>2</th>
<th>1</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td>R</td>
<td>IPID</td>
<td>TYPE</td>
<td>R</td>
<td>AS</td>
<td>LS</td>
<td>LD</td>
<td></td>
</tr>
<tr>
<td>31:16</td>
<td>R</td>
<td>Reserved for future use. Must be set to 0.</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>15:8</td>
<td>IPID</td>
<td>IP core ID: 0 for SpaceWire</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>7:5</td>
<td>TYPE</td>
<td>Packet type: 3 for link control packets</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>4:3</td>
<td>R</td>
<td>Reserved for future use. Must be set to 0.</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>AS</td>
<td>Link autostart.</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>LS</td>
<td>Link start.</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>LD</td>
<td>Link disable.</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Startup frequency in MHz at offset 0x8:

<table>
<thead>
<tr>
<th>31</th>
<th></th>
<th>SFREQ</th>
</tr>
</thead>
<tbody>
<tr>
<td>31:0</td>
<td>SFREQ</td>
<td>Startup frequency in MHz, big endian IEEE-754 32-bit float</td>
</tr>
</tbody>
</table>

TX frequency in MHz at offset 0xc:

<table>
<thead>
<tr>
<th>31</th>
<th></th>
<th>TFREQ</th>
</tr>
</thead>
<tbody>
<tr>
<td>31:0</td>
<td>TFREQ</td>
<td>TX frequency in MHz, big endian in IEEE-754 32-bit float</td>
</tr>
</tbody>
</table>

*Figure 9.5. SpaceWire link control packet*

**9.5.5.6. 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 | Length of rest of packet: 8 |

Header at offset 0x4:

<table>
<thead>
<tr>
<th>31</th>
<th>16</th>
<th>15</th>
<th>8</th>
<th>7</th>
<th>5</th>
<th>4</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>R</td>
<td>IPID</td>
<td>TYPE</td>
<td>R</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

| 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 | RX frequency in MHz, big endian IEEE-754 32-bit float |

Figure 9.6. SpaceWire rx frequency packet

9.5.5.7. 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:

<table>
<thead>
<tr>
<th>LEN</th>
</tr>
</thead>
<tbody>
<tr>
<td>31:0 LEN Length of rest of packet: 4</td>
</tr>
</tbody>
</table>

Header at offset 0x4:

<table>
<thead>
<tr>
<th>R</th>
<th>OE</th>
<th>ERROR</th>
<th>IPID</th>
<th>TYPE</th>
<th>R</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>31:21</td>
<td>R</td>
<td>Reserved for future use. Must be set to 0.</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>20</td>
<td>OE</td>
<td>Other end: 1: other end gets the error, 0: my end gets error</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>19</td>
<td>R</td>
<td>Reserved for future use. Must be set to 0.</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>18:16</td>
<td>ERROR</td>
<td>Link error: 1: Disconnect, 2: Parity, 3: Escape, 4: Credit</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>15:8</td>
<td>IPID</td>
<td>IP core ID: 0 for SpaceWire</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>7:5</td>
<td>TYPE</td>
<td>Packet type: 5 for link error injection packets</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>4:0</td>
<td>R</td>
<td>Reserved for future use. Must be set to 0.</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Figure 9.7. SpaceWire link error injection packet

9.5.5.8. 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
LEN
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
MSW
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
LSW
0

LSW
Bits 31:0 of unsigned 64-bit big endian integer

Reserved field at offset 0x10:

31
R
0

R
Reserved for future use. Must be set to 0.

Figure 9.8. SpaceWire packet error request packet

9.5.6. Simple Mode

For backwards compatibility with TSIM 2.0.44 and older, the GRSPW2 models can be set up in “simple mode” with the \(-grspw_simple 1\) 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 \(-grspw_tx_max_part_len\) 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 \texttt{-grspw\_simple\_rxfreq} option. If that is not used, the RX frequency is taken by the \texttt{-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.

9.6. PCI initiator/target interface

The UT700 AHB module models the PCI core available in the UT700 ASIC. For core details and register specification please see the UT700 manual.

9.6.1. Connecting a user PCI model with the UT700 module

See Section 9.2 for details on how to connect the user PCI model to the UT700 module.

9.6.2. Commands

\textit{PCI Commands}

\texttt{pci\_status}

Print status for the PCI core

9.6.3. Debug flags

The following debug flags are available for the PCI interface. Use them in conjunction with the \texttt{ut700\_dbgon} command to enable different levels of debug information.

\textit{Table 9.6. PCI interface debug flags}

<table>
<thead>
<tr>
<th>Flag</th>
<th>Trace</th>
</tr>
</thead>
<tbody>
<tr>
<td>GAISLER_GRPCI_ACC</td>
<td>AHB accesses to/from PCI core</td>
</tr>
<tr>
<td>GAISLER_GRPCI_REGACC</td>
<td>GRPCI APB register accesses</td>
</tr>
<tr>
<td>GAISLER_GRPCI_DMA_REGACC</td>
<td>PCIDMA APB register accesses</td>
</tr>
<tr>
<td>GAISLER_GRPCI_DMA</td>
<td>GRPCI DMA accesses on the AHB bus</td>
</tr>
<tr>
<td>GAISLER_GRPCI_TARGET_ACC</td>
<td>GRPCI target accesses</td>
</tr>
<tr>
<td>GAISLER_GRPCI_INIT</td>
<td>Print summary on startup</td>
</tr>
</tbody>
</table>

9.6.4. PCI bus model API

The structure \texttt{struct grpci\_input} models the PCI bus. It is defined as:

\begin{verbatim}
struct grpci_input {
    struct input_inp _b;
    int (*acc)(struct grpci_input *ctrl, int cmd, unsigned int addr,
                unsigned int *data, unsigned int *abort, unsigned int *ws);
    int (*target_acc)(struct grpci_input *ctrl, int cmd, unsigned int addr,
                      unsigned int *data, unsigned int *mexc);
};
\end{verbatim}

The acc callback should be set by the PCI user module at startup. It is called by the UT700 module whenever it reads/writes as a PCI bus master.

\textit{Table 9.7. acc callback parameters}

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>cmd</td>
<td>Command to execute, see Section 9.6.2 details.</td>
</tr>
<tr>
<td>addr</td>
<td>PCI address.</td>
</tr>
<tr>
<td>Parameter</td>
<td>Description</td>
</tr>
<tr>
<td>-----------</td>
<td>-------------</td>
</tr>
<tr>
<td>data</td>
<td>Data buffer. The user module should return the read data in *data for read commands or write the data in *data for write commands.</td>
</tr>
<tr>
<td>wsize</td>
<td>0: 8-bit access 1: 16-bit access, 2: 32-bit access. Is always 2 for read accesses.</td>
</tr>
<tr>
<td>ws</td>
<td>Set *ws to the number of PCI clocks it takes to complete the transaction.</td>
</tr>
<tr>
<td>abort</td>
<td>Set *abort to 1 to generate target abort, or 0 otherwise.</td>
</tr>
</tbody>
</table>

The return value of acc determines if the transaction terminates successfully (1, GRPCI_ACC_OK) or with master abort (0, GRPCI_ACC_MASTER_ABORT).

The callback target_acc is installed by the UT700 AHB module. The PCI user dynamic library can call this function to initiate an access to the UT700 PCI target.

Table 9.8. target_acc parameters

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>cmd</td>
<td>Command to execute, see Section 9.6.2 for details. I/O cycles are not supported by the UT700 target.</td>
</tr>
<tr>
<td>addr</td>
<td>PCI address. Should always be word aligned for read accesses.</td>
</tr>
<tr>
<td>data</td>
<td>Data buffer. The read data is returned in *data for read commands or the data in *data is written for write commands.</td>
</tr>
<tr>
<td>wsize</td>
<td>0: 8-bit access 1: 16-bit access, 2: 32-bit access. Should always be 2 for read accesses.</td>
</tr>
<tr>
<td>mexc</td>
<td>0 if access is successful, 1 in case of target abort.</td>
</tr>
</tbody>
</table>

If the address matched MEMBAR0, MEMBAR1 or CONFIG target_acc will return 1 otherwise 0.

See the ut700/examples/input for example implementations.

9.7. GPIO interface

9.7.1. Connecting a user GPIO model with the UT700 module

See Section 9.2 for details on how to connect the user GPIO model to the UT700 module.

9.7.2. Commands

GPIO Commands

GPIO Commands

<table>
<thead>
<tr>
<th>subcommand</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>gpio0_status</td>
<td>Print status for the GPIO core.</td>
</tr>
<tr>
<td>gpio0_dbg [flag</td>
<td>subcommand]</td>
</tr>
</tbody>
</table>

9.7.3. Debug flags

The following debug flags and debug subcommands are available for GPIO interfaces. The GAISLER_GPIO_* flags can be used with the gpio0_dbg command to toggle individual flags for individual GPIO cores and with the ut700_dbg command to toggle individual flags for all GPIO cores. The subcommands can be used with the gpio0_dbg command to change and list the settings of all flags for individual GPIO cores.

Table 9.9. GPIO debug flags

<table>
<thead>
<tr>
<th>Flag/subcommand</th>
<th>Trace</th>
</tr>
</thead>
<tbody>
<tr>
<td>GAISLER_GPIO_ACC</td>
<td>GPIO register accesses</td>
</tr>
<tr>
<td>GAISLER_GPIO_IRQ</td>
<td>GPIO interrupts</td>
</tr>
<tr>
<td>Flag/subcommand</td>
<td>Trace</td>
</tr>
<tr>
<td>-----------------</td>
<td>-------</td>
</tr>
<tr>
<td>all</td>
<td>Set all GPIO debug flags for the core</td>
</tr>
<tr>
<td>clean</td>
<td>Set none of the GPIO debug flags for the core</td>
</tr>
<tr>
<td>list</td>
<td>List the current setting of the debug flags for the core</td>
</tr>
</tbody>
</table>

### 9.7.4. GPIO model API

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

```c
/* GPIO input provider */
struct gpio_input {
    struct input_inp b;
    int (*gpioout)(struct gpio_input *ctrl, unsigned int out);
    int (*gpioin) (struct gpio_input *ctrl, unsigned int in);
};
```

The `gpioout` callback should be set by the user module at startup. The `gpioin` callback is set by the UT700 AHB module. The `gpioout` callback is called by the UT700 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.

**Table 9.10. `gpioout` callback parameters**

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>out</td>
<td>The values of the output pins</td>
</tr>
</tbody>
</table>

**Table 9.11. `gpioin` callback parameters**

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>in</td>
<td>The input pin values</td>
</tr>
</tbody>
</table>

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

See the `ut700/examples/input` for an example implementation.

### 9.8. CAN interface

The UT700 AHB module contains 2 CAN_OC cores which models the CAN_OC cores available in the UT700. For core details and register specification please see the UT700 manual.

#### 9.8.1. Start up options

**CAN 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]`
  
  Specifies whether the CAN_OC core will wait for a acknowledgment packet on transmission. This option must be put after `-can_ocX_connect`.

X in the above options is in the range 1-2.

#### 9.8.2. Commands

**CAN core TSIM 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 acknowledgment 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`
Toggle, set, clear, list debug flags for the CAN_OC core.

X in the above commands is in the range 1-2.

### 9.8.3. 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 `ut700_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 9.12. CAN debug flags**

<table>
<thead>
<tr>
<th>Flag</th>
<th>Trace</th>
</tr>
</thead>
<tbody>
<tr>
<td>GAISLER_CAN_OC_ACC</td>
<td>CAN_OC register accesses</td>
</tr>
<tr>
<td>GAISLER_CAN_OC_RXPACKET</td>
<td>CAN_OC received messages</td>
</tr>
<tr>
<td>GAISLER_CAN_OC_TXPACKET</td>
<td>CAN_OC transmitted messages</td>
</tr>
<tr>
<td>GAISLER_CAN_OC_ACK</td>
<td>CAN_OC acknowledgements</td>
</tr>
<tr>
<td>GAISLER_CAN_OC_IRQ</td>
<td>CAN_OC interrupts</td>
</tr>
<tr>
<td>all</td>
<td>Set all debug flags for the core</td>
</tr>
<tr>
<td>clean</td>
<td>Set none of the debug flags for the core</td>
</tr>
<tr>
<td>list</td>
<td>List the current setting of the debug flags for the core</td>
</tr>
</tbody>
</table>

### 9.8.4. 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.

### 9.8.5. 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 9.13. CAN packet types**

<table>
<thead>
<tr>
<th>Type</th>
<th>Value</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Message</td>
<td>0x00</td>
<td></td>
</tr>
<tr>
<td>Error counter</td>
<td>0xFD</td>
<td></td>
</tr>
<tr>
<td>Acknowledge</td>
<td>0xFE</td>
<td></td>
</tr>
<tr>
<td>Acknowledge config</td>
<td>0xFF</td>
<td></td>
</tr>
</tbody>
</table>

#### 9.8.5.1. CAN message packet format

Used to send and receive CAN messages.
| 0x0 | LENGTH |
| 31:0 | LENGTH, specifies the length of the rest of the packet |

**CAN message**

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

*Figure 9.9. CAN message packet format*

### 9.8.5.2. Error counter packet format

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

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

**Error counter packet**

<table>
<thead>
<tr>
<th>Byte #</th>
<th>Field</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>4</td>
<td>Packet type</td>
<td>Type of packet, 0xFD for error counter packets</td>
</tr>
<tr>
<td>5</td>
<td>Register</td>
<td>0 - RX error counter, 1 - TX error counter</td>
</tr>
<tr>
<td>6</td>
<td>Value</td>
<td>Value to write to error counter</td>
</tr>
</tbody>
</table>

*Figure 9.10. Error counter packet format*

### 9.8.5.3. 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.

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

**Acknowledge packet**

<table>
<thead>
<tr>
<th>Byte #</th>
<th>Field</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>4</td>
<td>Packet type</td>
<td>Type of packet, 0xFE for acknowledge packets</td>
</tr>
<tr>
<td>5</td>
<td>Ack payload</td>
<td>0 - No acknowledge, 1 - Acknowledge</td>
</tr>
</tbody>
</table>

*Figure 9.11. Acknowledge packet format*
9.8.5.4. 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 –\texttt{can\_ocX\_ack} or if the \texttt{can\_ocX\_ack} command has been issued.

<table>
<thead>
<tr>
<th>0x0</th>
<th>LENGTH</th>
</tr>
</thead>
<tbody>
<tr>
<td>31:0</td>
<td>LENGTH, specifies the length of the rest of the packet</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Byte #</th>
<th>Field</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>4</td>
<td>Packet type</td>
<td>Type of packet, 0xFF for acknowledge configuration packets</td>
</tr>
<tr>
<td>5</td>
<td>Ack configuration</td>
<td>bit 0 - Unused, bit 1 - Ack packet enable, 1 - enabled, 0 - disabled, bit 2 - Set ack packet payload, 1 - ACK, 0 - NAK</td>
</tr>
</tbody>
</table>

Figure 9.12. Acknowledge configuration packet format

9.9. SPI interface

9.9.1. Connecting a user SPI model with the UT700 module
See Section 9.2 for details on how to connect the user SPI model to the UT700 module.

9.9.2. Commands

\textit{SPI Commands}

\texttt{spi0\_status}

Print status for the SPI core.

\texttt{spi0\_dbg \{flag\|subcommand\}}

Toggle, set, clear, list debug flags for the SPI core.

9.9.3. Debug flags

The following debug flags and debug subcommands are available for SPI interfaces. The \texttt{GAISLER\_SPI\_*} flags can be used with the \texttt{spi0\_dbg} command to toggle individual flags for individual SPI cores and with the \texttt{ut700\_dbgon} command to toggle individual flags for all SPI cores. The subcommands can be used with the \texttt{spi0\_dbg} command to change and list the settings of all flags for individual SPI cores.

<table>
<thead>
<tr>
<th>Flag/subcommand</th>
<th>Trace</th>
</tr>
</thead>
<tbody>
<tr>
<td>\texttt{GAISLER_SPI_ACC}</td>
<td>SPI register accesses</td>
</tr>
<tr>
<td>\texttt{GAISLER_SPI_IRQ}</td>
<td>SPI interrupts</td>
</tr>
<tr>
<td>all</td>
<td>Set all SPI debug flags for the core</td>
</tr>
<tr>
<td>clean</td>
<td>Set none of the SPI debug flags for the core</td>
</tr>
<tr>
<td>list</td>
<td>List the current setting of the debug flags for the core</td>
</tr>
</tbody>
</table>

Table 9.14. SPI debug flags

9.9.4. SPI bus model API

The structure \texttt{struct spi\_input} models the SPI bus. It is defined as:

/* Spi input provider */
The `spishift` callback should be set by the SPI user module at startup. It is called by the UT700 module whenever it shifts a word through the SPI bus.

**Table 9.15. spishift callback parameters**

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>select</td>
<td>Slave select bits</td>
</tr>
<tr>
<td>bitcnt</td>
<td>Number of bits set in the MODE register, if bitcnt is -1 then the operation is not a shift and the call is to indicate a <em>select</em> change, i.e. if the core is disabled.</td>
</tr>
<tr>
<td>out</td>
<td>Shift out (tx) data</td>
</tr>
<tr>
<td>in</td>
<td>Shift in (rx) data</td>
</tr>
</tbody>
</table>

The return value of `spishift` is ignored.

See the `ut700/examples/input` directory for an example implementation.
10. Cobham Gaisler GR712RC emulation

To emulate the GR712RC chip the `-gr712rc` option should be used. That sets up parameters for core TSIM2 to match GR712RC. In addition, the GR712RC loadable AHB module implements emulation models for additional cores of the GR712RC. Apart from an overview of what TSIM supports for GR712RC emulation, this chapter is mainly about the GR712RC AHB module.

The following table lists which cores in the GR712RC are modelled by TSIM or not. Some supported cores are modelled by the core TSIM2 and some in the GR712RC AHB module. 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 built-in simulation models and the rest of the chapter for the AHB module models.

**Table 10.1. Simulation models for GR712RC**

<table>
<thead>
<tr>
<th>Core</th>
<th>Status</th>
<th>Notes</th>
</tr>
</thead>
<tbody>
<tr>
<td>LEON3FT</td>
<td>Supported by core TSIM2</td>
<td>Only one CPU is modelled. No FT features are modelled.</td>
</tr>
<tr>
<td>GRFPU</td>
<td>Supported by core TSIM2</td>
<td></td>
</tr>
<tr>
<td>AHBSTAT</td>
<td>Supported by core TSIM2</td>
<td></td>
</tr>
<tr>
<td>APBCTRL</td>
<td>Supported by core TSIM2</td>
<td></td>
</tr>
<tr>
<td>APBUART</td>
<td>Supported by core TSIM2</td>
<td></td>
</tr>
<tr>
<td>FTMCTRL</td>
<td>Supported by core TSIM2</td>
<td>No FT features are modelled</td>
</tr>
<tr>
<td>GPTIMER</td>
<td>Supported by core TSIM2</td>
<td></td>
</tr>
<tr>
<td>GRTIMER</td>
<td>Supported by core TSIM2</td>
<td></td>
</tr>
<tr>
<td>IRQMP</td>
<td>Supported by core TSIM2</td>
<td></td>
</tr>
<tr>
<td>CAN_OC</td>
<td>Supported by AHB module</td>
<td></td>
</tr>
<tr>
<td>FTAHBGRAM</td>
<td>Supported by AHB module</td>
<td>No FT features are modelled.</td>
</tr>
<tr>
<td>GRETH</td>
<td>Supported by AHB module</td>
<td></td>
</tr>
<tr>
<td>GRGPIO</td>
<td>Supported by AHB module</td>
<td></td>
</tr>
<tr>
<td>GRSPW2</td>
<td>Supported by AHB module</td>
<td></td>
</tr>
<tr>
<td>SPICTRL</td>
<td>Supported by AHB module</td>
<td></td>
</tr>
<tr>
<td>CANMUX</td>
<td>Dummy in AHB module</td>
<td>Functionality-less registers only</td>
</tr>
<tr>
<td>CLKGATE</td>
<td>Dummy in AHB module</td>
<td>Functionality-less registers only</td>
</tr>
<tr>
<td>GRGPREG</td>
<td>Dummy in AHB module</td>
<td>Functionality-less registers only</td>
</tr>
<tr>
<td>B1553BRM</td>
<td>Not supported</td>
<td></td>
</tr>
<tr>
<td>GRASCS</td>
<td>Not supported</td>
<td></td>
</tr>
<tr>
<td>GRSLINK</td>
<td>Not supported</td>
<td></td>
</tr>
<tr>
<td>GRTC</td>
<td>Not supported</td>
<td></td>
</tr>
<tr>
<td>GRTM</td>
<td>Not supported</td>
<td></td>
</tr>
<tr>
<td>I2CMST</td>
<td>Not supported</td>
<td></td>
</tr>
<tr>
<td>AHBJTAG</td>
<td>Not supported</td>
<td>Debug link</td>
</tr>
<tr>
<td>DSU3</td>
<td>Not supported</td>
<td>Debug unit</td>
</tr>
</tbody>
</table>

TSIM supports running several AHB modules. User defined models for unsupported cores can be used in parallel with the GR712RC AHB module. User defined models can also override the simulation models (such as for dummy models) in the GR712RC AHB module.

10.1. Overview of the GR712RC AHB module

The GR712RC AHB module is a loadable AHB module that implements GR712RC peripherals in addition to what is emulated by core TSIM as listed in Table 10.1. The interfaces are modelled at packet/transaction/message...
level and provides an easy way to connect the simulated GR712RC to a larger simulation framework. See also Section 5.4.1 on some limitations of some features when using this module. The GR712RC AHB module is only supported for TSIM2 LEON3.

The following files are delivered with the GR712RC TSIM module:

Table 10.2. Files delivered with the GR712RC TSIM module

<table>
<thead>
<tr>
<th>File</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>gr712/linux/gr712.so</td>
<td>GR712RC AHB module for Linux</td>
</tr>
<tr>
<td>gr712/win32/gr712.dll</td>
<td>GR712RC AHB module for Windows</td>
</tr>
<tr>
<td>gr712/examples/input</td>
<td>The input directory contains two examples of user modules</td>
</tr>
<tr>
<td>gr712/examples/input/README.txt</td>
<td>Description of the user module examples</td>
</tr>
<tr>
<td>gr712/examples/input/Makefile</td>
<td>Makefile for building the user modules</td>
</tr>
<tr>
<td>gr712/examples/input/spi.c</td>
<td>SPI user module example emulating a Intel SPI flash</td>
</tr>
<tr>
<td>gr712/examples/input/gpio.c</td>
<td>GPIO user module emulating GPIO bit toggle</td>
</tr>
<tr>
<td>gr712/examples/input/gr712inputprovider.h</td>
<td>Interface between the GR712RC module and the user module</td>
</tr>
</tbody>
</table>

10.2. Loading the module

The module is loaded using the TSIM2 option \texttt{-ahbm}. A user input module for SPI and GPIO can optionally be declared, between \texttt{-designinput} and \texttt{-designinputend} options. For example:

On Linux:

\begin{verbatim}
tsim-leon3 -gr712rc -ahbm gr712/linux/gr712.so
   -designinput ./input.so -designinputend
\end{verbatim}

On Windows:

\begin{verbatim}
tsim-leon3 -gr712rc -ahbm gr712/win32/gr712.dll
   -designinput input.dll -designinputend
\end{verbatim}

The option \texttt{-gr712rc} needs to be given to TSIM to enable the GR712RC processor configuration. The above line loads the GR712RC AHB module \texttt{gr712.so} which in turn loads the user user input module \texttt{./input.so}. The user input module \texttt{./input.so} communicates with \texttt{gr712.so} using the user module interface described in \texttt{gr712inputprovider.h}, while \texttt{gr712.so} communicates with TSIM via the AHB interface.

Example user input modules can be found in \texttt{gr712/examples/input/}.

10.2.1. User input module interface

The SPI and GPIO models in the GR712RC module uses a user supplied user input module, in the form of a dynamic loadable library, that models the outside world. This section describes the general interface for hooking up the user module to the GR712RC module. The details on the interfaces to the particular cores, see their respective sections.

A user supplied dynamic library should expose a public symbol \texttt{gr712inputsystem} of type \texttt{struct gr712_subsystem *}. The \texttt{struct gr712_subsystem} is defined in \texttt{gr712inputprovider.h} as:

\begin{verbatim}
struct gr712_subsystem {
    void (*gr712_inp_setup) (int id, 
                          struct gr712_inp_layout * l, 
                          char **argv, int argc);
    void (*gr712_inp_restart) (int id, 
                             struct gr712_inp_layout * l);
    struct sim_interface *simif;
};
\end{verbatim}
The callback gr712_inp_restart will be called every time the simulator restarts. At initialization the callback gr712_inp_setup will be called once, supplied with a pointer to structure struct gr712_inp_layout defined in gr712inputprovider.h.

The user module can access the global TSIM struct sim_interface structure through the simif member. See Chapter 5 for more details.

The user supplied dynamic library should, in its gr712_inp_setup function, “claim” the input structs it uses using the INPUT_CLAIM macro. For example INPUT_CLAIM(l->gpio[0]) as in the example below.

A user supplied dynamic library that only sets up a model for GPIO could look like this:

```c
#include <stdio.h>
#include <string.h>
#include "tsim.h"
#include "gr712inputprovider.h"

extern struct gr712_subsystem *gr712inputsystem;
static struct gr712_inp_layout *lay = 0;

static void Change(struct gpio_input *ctrl) {
    ...
}

int gpioout(struct gpio_input *ctrl, unsigned int out) {
    ...
}

static void gr712_inp_setup (int id,
    struct gr712_inp_layout * l,
    char **argv, int argc) {
    lay = l;
    printf("User-dll: gr712_inp_setup:Claiming %s\n", l->gpio[0].b.name);
    INPUT_CLAIM(l->gpio[0]);
    l->gpio[0].gpioout = gpioout;
    gr712inputsystem->simif->event(Change,(unsigned long)&l->gpio[0],10000000);
}

static struct gr712_subsystem gr712_gpio = {
    gr712_inp_setup,0,0
};

struct gr712_subsystem *gr712inputsystem = &gr712_gpio;
```

A Makefile that would build a user supplied dynamic library gpio.(dll|so) could look like this:

```makefile
M_DLL_FIX=$(if $(strip $(shell uname|grep MINGW32)),dll,so)
M_LIB=$(if $(strip $(shell uname|grep MINGW32)),-lws2_32 -luser32 -lkernel32 -lwinmm,)
all: gpio.$(M_DLL_FIX)
gpio.$(M_DLL_FIX) : gpio.o
    $(CC) -shared -g gpio.o -o gpio.$(M_DLL_FIX) $(M_LIB)
gpio.o: gpio.c
    $(CC) -fPIC -c -g -o0 gpio.c -o gpio.o

clean:
    -rm -f *.o *.so
```

The user can then specify the user module to be loaded by the gr712.so AHB module using the -designinput and -designinputend command line options. The first argument after -designinput is the user module. Arguments after that are passed to the user input module in the call to gr712_inp_setup.

For example: -designinput gr712/examples/input/gpio.so -gpioverbose -designinputend will specify that the example user input module gpio.so should be used and that it should receive the argument -gpioverbose.

### 10.3. Debugging

To enable printout of debug information the -gr712_dbgon flag switch can be used. Alternatively one can issue the gr712_dbgon flag command on the TSIM2 command line to toggle the on/off state of a flag. The debug flags that are available are described for each core in the following sections and can be listed by gr712_dbgon help.
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.

10.4. CAN interface

The GR712RC AHB module contains 2 CAN_OCs which models the CAN_OCs available in the GR712RC. For core details and register specification please see the GR712RC manual.

10.4.1. Start up options

**CAN 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]`
  
  Specifies whether the CAN_OC core will wait for a acknowledgment packet on transmission. This option must be put after `-can_ocX_connect`.

X in the above options is in the range 0-1.

10.4.2. Commands

**CAN core TSIM 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 acknowledgment 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`
  
  Toggle, set, clear, list debug flags for the CAN_OC core.

X in the above commands is in the range 0-1.

10.4.3. 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_OCs and with the `gr712_dbg` command to toggle individual flags for all CAN_OCs cores. The subcommands can be used with the `can_ocX_dbg` command to change and list the settings of all flags for individual CAN_OCs cores.

**Table 10.3. CAN debug flags**

<table>
<thead>
<tr>
<th>Flag</th>
<th>Trace</th>
</tr>
</thead>
<tbody>
<tr>
<td>GAISLER_CAN_OC_ACC</td>
<td>CAN_OC register accesses</td>
</tr>
<tr>
<td>GAISLER_CAN_OC_RXPACKET</td>
<td>CAN_OC received messages</td>
</tr>
<tr>
<td>GAISLER_CAN_OC_TXPACKET</td>
<td>CAN_OC transmitted messages</td>
</tr>
<tr>
<td>GAISLER_CAN_OC_ACK</td>
<td>CAN_OC acknowledgements</td>
</tr>
<tr>
<td>GAISLER_CAN_OC_IRQ</td>
<td>CAN_OC interrupts</td>
</tr>
<tr>
<td>all</td>
<td>Set all debug flags for the core</td>
</tr>
<tr>
<td>clean</td>
<td>Set none of the debug flags for the core</td>
</tr>
<tr>
<td>list</td>
<td>List the current setting of the debug flags for the core</td>
</tr>
</tbody>
</table>
10.4.4. 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.

10.4.5. 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>
<thead>
<tr>
<th>Type</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>Message</td>
<td>0x00</td>
</tr>
<tr>
<td>Error counter</td>
<td>0xFD</td>
</tr>
<tr>
<td>Acknowledge</td>
<td>0xFE</td>
</tr>
<tr>
<td>Acknowledge config</td>
<td>0xFF</td>
</tr>
</tbody>
</table>

10.4.5.1. CAN message packet format

Used to send and receive CAN messages.

```
31                0
0x00              LENGTH
31:0              LENGTH, specifies the length of the rest of the packet

CAN message

<table>
<thead>
<tr>
<th>Byte #</th>
<th>Description</th>
<th>Bits (MSB-LSB)</th>
</tr>
</thead>
<tbody>
<tr>
<td>4</td>
<td>Protocol ID = 0</td>
<td>Prot ID 7-0</td>
</tr>
<tr>
<td>5</td>
<td>Control</td>
<td>FF RTR</td>
</tr>
<tr>
<td>6-9</td>
<td>ID (32 bit word in network byte order)</td>
<td>ID 10-0 (bits 31 - 11 ignored for standard frame format)</td>
</tr>
<tr>
<td>10-17</td>
<td>Data byte 1 - DLC</td>
<td>Data byte n 7-0</td>
</tr>
</tbody>
</table>
```

Figure 10.1. CAN message packet format

10.4.5.2. Error counter packet format

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

```
31                0
0x00              LENGTH
31:0              LENGTH, specifies the length of the rest of the packet

Error counter packet

<table>
<thead>
<tr>
<th>Byte #</th>
<th>Field</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>4</td>
<td>Packet type</td>
<td>Type of packet, 0xFD for error counter packets</td>
</tr>
<tr>
<td>5</td>
<td>Register</td>
<td>0 - RX error counter, 1 - TX error counter</td>
</tr>
<tr>
<td>6</td>
<td>Value</td>
<td>Value to write to error counter</td>
</tr>
</tbody>
</table>
```

Figure 10.2. Error counter packet format
10.4.5.3. 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.

<table>
<thead>
<tr>
<th>31</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x0</td>
<td></td>
</tr>
<tr>
<td>31:0</td>
<td>LENGTH, specifies the length of the rest of the packet</td>
</tr>
</tbody>
</table>

Acknowledge packet

<table>
<thead>
<tr>
<th>Byte #</th>
<th>Field</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>4</td>
<td>Packet type</td>
<td>Type of packet, 0xFE for acknowledge packets</td>
</tr>
<tr>
<td>5</td>
<td>Ack payload</td>
<td>0 - No acknowledge, 1 - Acknowledge</td>
</tr>
</tbody>
</table>

Figure 10.3. Acknowledge packet format

10.4.5.4. 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.

<table>
<thead>
<tr>
<th>31</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x0</td>
<td></td>
</tr>
<tr>
<td>31:0</td>
<td>LENGTH, specifies the length of the rest of the packet</td>
</tr>
</tbody>
</table>

Acknowledge configuration packet

<table>
<thead>
<tr>
<th>Byte #</th>
<th>Field</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>4</td>
<td>Packet type</td>
<td>Type of packet, 0xFF for acknowledge configuration packets</td>
</tr>
<tr>
<td>5</td>
<td>Ack configuration</td>
<td>bit 0 - Unused, bit 1 - Ack packet enable, 1 - enabled, 0 - disabled, bit 2 - Set ack packet payload, 1 - ACK, 0 - NAK</td>
</tr>
</tbody>
</table>

Figure 10.4. Acknowledge configuration packet format

10.5. 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 the GR712RC. For core details and register specification please see the GR712RC manual.

The following features are supported:
- Direct Memory Access
- Interrupts

10.5.1. Start up options

*Ethernet core start up options

```
-grethconnect host[:port]
```

Connect Ethernet core to a packet server at the specified host and port. Default port is 2224.
10.5.2. Commands

*Ethernet core TSIM commands*

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

```bash
greth_status
```
Print Ethernet register status

10.5.3. Debug flags

The following debug flags are available for the Ethernet interface. Use the them in conjunction with the `gr712_dbgon` command to enable different levels of debug information.

*Table 10.5. Ethernet debug flags*

<table>
<thead>
<tr>
<th>Flag</th>
<th>Trace</th>
</tr>
</thead>
<tbody>
<tr>
<td>GAISLER_GRETH_ACC</td>
<td>GRETH accesses</td>
</tr>
<tr>
<td>GAISLER_GRETH_L1</td>
<td>GRETH accesses verbose</td>
</tr>
<tr>
<td>GAISLER_GRETH_TX</td>
<td>GRETH transmissions</td>
</tr>
<tr>
<td>GAISLER_GRETH_RX</td>
<td>GRETH reception</td>
</tr>
<tr>
<td>GAISLER_GRETH_RXPACKET</td>
<td>GRETH received packets</td>
</tr>
<tr>
<td>GAISLER_GRETH_RXCTRL</td>
<td>GRETH RX packet server protocol</td>
</tr>
<tr>
<td>GAISLER_GRETH_RXBDCTRL</td>
<td>GRETH TX packet server protocol</td>
</tr>
<tr>
<td>GAISLER_GRETH_TXPACKET</td>
<td>GRETH transmitted packets</td>
</tr>
<tr>
<td>GAISLER_GRETH_IRQ</td>
<td>GRETH interrupts</td>
</tr>
</tbody>
</table>

10.5.4. 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 `-grethconnect` start-up parameter or using the `greth_connect` command.

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.

10.5.5. 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:

<table>
<thead>
<tr>
<th>LEN</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>31:0 LEN</th>
</tr>
</thead>
<tbody>
<tr>
<td>Length of rest of packet: 4 + number of data bytes</td>
</tr>
</tbody>
</table>

Header at offset 0x4:

<table>
<thead>
<tr>
<th>R</th>
<th>IPID</th>
<th>TYPE</th>
<th>R</th>
</tr>
</thead>
<tbody>
<tr>
<td>31:16 R</td>
<td>Reserved for future use. Must be set to 0.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>15:8 IPID</td>
<td>IP core ID: 1 for Ethernet</td>
<td></td>
<td></td>
</tr>
<tr>
<td>7:5 TYPE</td>
<td>Packet type: 0 for data packets</td>
<td></td>
<td></td>
</tr>
<tr>
<td>4:0 R</td>
<td>Reserved for future use. Must be set to 0.</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

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

**Figure 10.5. Ethernet data packet**

### 10.6. SpaceWire interface with RMAP support

The GR712RC AHB module contains 6 GRSPW2 cores which models the GRSPW2 cores available in the GR712RC. For core details and register specification please see the GR712RC manual.

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 the GR712RC 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.

#### 10.6.1. Start up options

**SpaceWire core start up options**

```
-grspwX_connect host:port
   Connect GRPSW 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_tx_max_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 val
  When val is 1, 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 val is 0, the last data part packet can contain both data and an end marker. This is the default when simple mode is enabled.

-grspw_simple 1
  Set all GRSPW2 cores to “simple mode”. This can be used for backward compatibility with TSIM 2.0.44 and backwards. See the separate section on simple mode for details. Note the needed 1 argument.

-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 1 option.

X in the above options has the range 0-5.

### 10.6.2. Commands

**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_status**
  Print status for GRSPW2 core X.

- **grspwX_dbg [argument]**
  Sets, clears, lists, toggles debug options for individual GRSPW2 cores. Using grspwX_dbg without any arguments will list all available options. The list argument will list current debug option settings. The all argument will turn on all debug options. The clean argument will turn off all debug options. Using one of the available debug options as argument will toggle that debug option. See the section below.

X in the above commands has the range 0-5.

### 10.6.3. 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 gr712_dbg command to toggle individual flags for all SpaceWire 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 10.6. SpaceWire debug flags

<table>
<thead>
<tr>
<th>Flag</th>
<th>Trace</th>
</tr>
</thead>
<tbody>
<tr>
<td>GAISLER_GRSPW_ACC</td>
<td>GRSPW accesses</td>
</tr>
<tr>
<td>GAISLER_GRSPW_RXPACKET</td>
<td>GRSPW received packets</td>
</tr>
<tr>
<td>GAISLER_GRSPW_RXCTRL</td>
<td>GRSPW rx protocol</td>
</tr>
<tr>
<td>GAISLER_GRSPW_TXPACKET</td>
<td>GRSPW transmitted packets</td>
</tr>
<tr>
<td>GAISLER_GRSPW_TXCTRL</td>
<td>GRSPW tx protocol</td>
</tr>
<tr>
<td>GAISLER_GRSPW_RMAP</td>
<td>GRSPW RMAP accesses</td>
</tr>
<tr>
<td>GAISLER_GRSPW_RMAPPACKET</td>
<td>GRSPW RMAP packet dumps</td>
</tr>
<tr>
<td>GAISLER_GRSPW_RMAPPACKDE</td>
<td>GRSPW RMAP packet decoding</td>
</tr>
<tr>
<td>GAISLER_GRSPW_DMAERR</td>
<td>GRSPW DMA errors</td>
</tr>
<tr>
<td>GAISLER_GRSPW_LINK</td>
<td>Link changes</td>
</tr>
<tr>
<td>GAISLER_GRSPW_PART</td>
<td>TX/RX of GRSPW data part packets</td>
</tr>
<tr>
<td>all</td>
<td>Set all debug flags for the core</td>
</tr>
</tbody>
</table>
### 10.6.4. SpaceWire packet server

Each SpaceWire core can be configured independently as a packet server or client using either \texttt{-grspw\_server} or \texttt{-grspw\_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 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 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.

### 10.6.5. 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 10.7. Packet types

<table>
<thead>
<tr>
<th>Type</th>
<th>Value</th>
<th>Direction</th>
<th>Notes</th>
</tr>
</thead>
<tbody>
<tr>
<td>Data part</td>
<td>0</td>
<td>Both</td>
<td>Only when in run state</td>
</tr>
<tr>
<td>Time code</td>
<td>1</td>
<td>Both</td>
<td>Only when in run state</td>
</tr>
<tr>
<td>Link state</td>
<td>2</td>
<td>Server to client</td>
<td></td>
</tr>
<tr>
<td>Link control</td>
<td>3</td>
<td>Client to server</td>
<td>Must be sent for model to reach run state</td>
</tr>
<tr>
<td>RX frequency</td>
<td>4</td>
<td>Server to client</td>
<td></td>
</tr>
<tr>
<td>Error injection</td>
<td>5</td>
<td>Client to server</td>
<td>Optional</td>
</tr>
<tr>
<td>Packet error request</td>
<td>6</td>
<td>Client to server</td>
<td>Optional</td>
</tr>
</tbody>
</table>

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 \texttt{-grspw\_tx\_max\_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.

#### 10.6.5.1. Flow control limitations

Flow control in terms of credit is not modeled 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.

### 10.6.5.2. 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 \textit{END} field set to 0 or 1. The other parts should have no end marker, i.e. the \textit{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 \texttt{-grspw\_endpacket} option. Splitting up SpaceWire packets into several data containing data part packets can be enabled with the \texttt{-grspw\_tx\_max\_part\_len} option.

<table>
<thead>
<tr>
<th>Packet length at offset 0x0:</th>
</tr>
</thead>
<tbody>
<tr>
<td>31</td>
</tr>
<tr>
<td>LEN</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Header at offset 0x4:</th>
</tr>
</thead>
<tbody>
<tr>
<td>31:16</td>
</tr>
<tr>
<td>15:8</td>
</tr>
<tr>
<td>7:5</td>
</tr>
<tr>
<td>4:2</td>
</tr>
<tr>
<td>1:0</td>
</tr>
</tbody>
</table>

Offset 0x8: The data bytes of the part starts here

*Figure 10.6. SpaceWire data part packet*
10.6.5.3. Time code packet format

Packet length at offset 0x0:

<table>
<thead>
<tr>
<th>31:0</th>
<th>LEN</th>
<th>Length of rest of packet: 8</th>
</tr>
</thead>
</table>

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: 1 for time code packets |
| 4:0   | R | Reserved for future use. Must be set to 0. |

Payload at offset 0x8:

| 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 10.7. SpaceWire time code packet

10.6.5.4. 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:

<table>
<thead>
<tr>
<th>LEN</th>
</tr>
</thead>
<tbody>
<tr>
<td>31:0 LEN</td>
</tr>
</tbody>
</table>

Header at offset 0x4:

<table>
<thead>
<tr>
<th>R</th>
<th>ERROR</th>
<th>IPID</th>
<th>TYPE</th>
<th>R</th>
<th>LS</th>
</tr>
</thead>
<tbody>
<tr>
<td>31:19</td>
<td>R</td>
<td>Reserved for future use. Must be set to 0.</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>18:16</td>
<td>ERROR</td>
<td>Link error: 0: None, 1: Disconnect, 2: Parity, 3: Escape, 4: Credit</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>15:8</td>
<td>IPID</td>
<td>IP core ID: 0 for SpaceWire</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>7:5</td>
<td>TYPE</td>
<td>Packet type: 2 for link state packets</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>4:3</td>
<td>R</td>
<td>Reserved for future use. Must be set to 0.</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>2:0</td>
<td>LS</td>
<td>Link State: 0: Error reset, 1: Error wait, 2: Ready, 3: Started, 4: Connecting, 5: Run</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

*Figure 10.8. SpaceWire link state packet*

### 10.6.5.5. 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:

<table>
<thead>
<tr>
<th>LEN</th>
</tr>
</thead>
<tbody>
<tr>
<td>31:0</td>
</tr>
</tbody>
</table>

Header at offset 0x4:

<table>
<thead>
<tr>
<th>R</th>
<th>IPID</th>
<th>TYPE</th>
<th>R</th>
<th>AS</th>
<th>LS</th>
<th>LD</th>
</tr>
</thead>
<tbody>
<tr>
<td>31:16</td>
<td>R</td>
<td>Reserved for future use. Must be set to 0.</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>15:8</td>
<td>IPID</td>
<td>IP core ID: 0 for SpaceWire</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>7:5</td>
<td>TYPE</td>
<td>Packet type: 3 for link control packets</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>4:3</td>
<td>R</td>
<td>Reserved for future use. Must be set to 0.</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>AS</td>
<td>Link autostart.</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>LS</td>
<td>Link start.</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>LD</td>
<td>Link disable.</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Startup frequency in MHz at offset 0x8:

<table>
<thead>
<tr>
<th>SFREQ</th>
</tr>
</thead>
<tbody>
<tr>
<td>31:0</td>
</tr>
</tbody>
</table>

TX frequency in MHz at offset 0xc:

<table>
<thead>
<tr>
<th>TFREQ</th>
</tr>
</thead>
<tbody>
<tr>
<td>31:0</td>
</tr>
</tbody>
</table>

Figure 10.9. SpaceWire link control packet

10.6.5.6. 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:

<table>
<thead>
<tr>
<th>LEN</th>
</tr>
</thead>
<tbody>
<tr>
<td>31:0</td>
</tr>
</tbody>
</table>

Header at offset 0x4:

<table>
<thead>
<tr>
<th>R</th>
<th>IPID</th>
<th>TYPE</th>
<th>R</th>
</tr>
</thead>
<tbody>
<tr>
<td>31:16</td>
<td>R</td>
<td>Reserved for future use. Must be set to 0.</td>
<td></td>
</tr>
<tr>
<td>15:8</td>
<td>IPID</td>
<td>IP core ID: 0 for SpaceWire</td>
<td></td>
</tr>
<tr>
<td>7:5</td>
<td>TYPE</td>
<td>Packet type: 4 for rx frequency packets</td>
<td></td>
</tr>
<tr>
<td>4:0</td>
<td>R</td>
<td>Reserved for future use. Must be set to 0.</td>
<td></td>
</tr>
</tbody>
</table>

RX frequency in MHz at offset 0x8:

<table>
<thead>
<tr>
<th>RFREQ</th>
</tr>
</thead>
<tbody>
<tr>
<td>31:0</td>
</tr>
</tbody>
</table>

Figure 10.10. SpaceWire rx frequency packet

10.6.5.7. 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|   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |
|   | 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 |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |
| 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 10.11. SpaceWire link error injection packet

10.6.5.8. 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:

<table>
<thead>
<tr>
<th>LEN</th>
</tr>
</thead>
<tbody>
<tr>
<td>31:0 LEN</td>
</tr>
</tbody>
</table>

Header at offset 0x4:

<table>
<thead>
<tr>
<th>R</th>
<th>ERROR</th>
<th>IPID</th>
<th>TYPE</th>
<th>R</th>
</tr>
</thead>
<tbody>
<tr>
<td>31:19 R</td>
<td>Reserved for future use. Must be set to 0.</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>18:16 ERROR</td>
<td>Link error: 1: Disconnect, 2: Parity, 3: Escape, 4: Credit</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>15:8 IPID</td>
<td>IP core ID: 0 for SpaceWire</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>7:5 TYPE</td>
<td>Packet type: 6 for packet error request packets</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>4:0 R</td>
<td>Reserved for future use. Must be set to 0.</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Packet number to request error for, most significant word at offset 0x8:

<table>
<thead>
<tr>
<th>MSW</th>
</tr>
</thead>
<tbody>
<tr>
<td>31:0 MSW</td>
</tr>
</tbody>
</table>

Packet number to request error for, least significant word at offset 0xc:

<table>
<thead>
<tr>
<th>LSW</th>
</tr>
</thead>
<tbody>
<tr>
<td>31:0 LSW</td>
</tr>
</tbody>
</table>

Reserved field at offset 0x10:

<table>
<thead>
<tr>
<th>R</th>
</tr>
</thead>
<tbody>
<tr>
<td>31:0 R</td>
</tr>
</tbody>
</table>

Figure 10.12. SpaceWire packet error request packet

### 10.6.6. 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_tx_max_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.

10.7. SPI interface

10.7.1. Connecting a user SPI model with the GR712RC module
See Section 10.2 for details on how to connect the user SPI model to the GR712RC module.

10.7.2. Commands

**SPI Commands**

```c
spi0_status
    Print status for the SPI core.

spi0_dbg [flag|subcommand]
    Toggle, set, clear, list debug flags for the SPI core.
```

10.7.3. Debug flags

The following debug flags and debug subcommands are available for SPI interfaces. The `GAISLER_SPI_*` flags can be used with the `spi0_dbg` command to toggle individual flags for individual SPI cores and with the `gr712_dbgon` command to toggle individual flags for all SPI cores. The subcommands can be used with the `spi0_dbg` command to change and list the settings of all flags for individual SPI cores.

**Table 10.8. SPI debug flags**

<table>
<thead>
<tr>
<th>Flag/subcommand</th>
<th>Trace</th>
</tr>
</thead>
<tbody>
<tr>
<td>GAISLER_SPI_ACC</td>
<td>SPI register accesses</td>
</tr>
<tr>
<td>GAISLER_SPI_IRQ</td>
<td>SPI interrupts</td>
</tr>
<tr>
<td>all</td>
<td>Set all SPI debug flags for the core</td>
</tr>
<tr>
<td>clean</td>
<td>Set none of the SPI debug flags for the core</td>
</tr>
<tr>
<td>list</td>
<td>List the current setting of the debug flags for the core</td>
</tr>
</tbody>
</table>

10.7.4. SPI bus model API

The structure `struct spi_input` models the SPI bus. It is defined as:

```c
/* Spi input provider */

struct spi_input {
    struct input_inp _b;
    int (*spishift)(struct spi_input *ctrl, uint32 select, uint32 bitcnt,
                    uint32 out, uint32 *in);
};
```

The spishift callback should be set by the SPI user module at startup. It is called by the GR712RC module whenever it shifts a word through the SPI bus.

**Table 10.9. spishift callback parameters**

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>select</td>
<td>Slave select bits</td>
</tr>
<tr>
<td>bitcnt</td>
<td>Number of bits set in the MODE register, if bitcnt is -1 then the operation is not a shift and the call is to indicate a select change, i.e. if the core is disabled.</td>
</tr>
</tbody>
</table>
### Parameter Description

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>out</td>
<td>Shift out (tx) data</td>
</tr>
<tr>
<td>in</td>
<td>Shift in (rx) data</td>
</tr>
</tbody>
</table>

The return value of `spishift` is ignored.

See the gr712/examples/input directory for an example implementation.

### 10.8. GPIO interface

#### 10.8.1. Connecting a user GPIO model with the GR712RC module

See Section 10.2 for details on how to connect the user GPIO model to the GR712RC module.

#### 10.8.2. Commands

**GPIO Commands**

- `gpioX_status`  
  Print status for the GPIO core.

- `gpioX_dbg [flag|subcommand]`  
  Toggle, set, clear, list debug flags for the GPIO core.

#### 10.8.3. 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 `gr712_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 10.10. GPIO debug flags**

<table>
<thead>
<tr>
<th>Flag/subcommand</th>
<th>Trace</th>
</tr>
</thead>
<tbody>
<tr>
<td>GAISLER_GPIO_ACC</td>
<td>GPIO register accesses</td>
</tr>
<tr>
<td>GAISLER_GPIO_IRQ</td>
<td>GPIO interrupts</td>
</tr>
<tr>
<td>all</td>
<td>Set all GPIO debug flags for the core</td>
</tr>
<tr>
<td>clean</td>
<td>Set none of the GPIO debug flags for the core</td>
</tr>
<tr>
<td>list</td>
<td>List the current setting of the debug flags for the core</td>
</tr>
</tbody>
</table>

#### 10.8.4. GPIO model API

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

```c
/* GPIO input provider */
struct gpio_input {
    struct input_inp _b;
    int (*gpioout)(struct gpio_input *ctrl, unsigned int out);
    int (*gpioin) (struct gpio_input *ctrl, unsigned int in);
};
```

The `gpioout` callback should be set by the user module at startup. The `gpioin` callback is set by the GR712RC AHB module. The `gpioout` callback is called by the GR712RC 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.

**Table 10.11. gpioout callback parameters**

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>out</td>
<td>The values of the output pins</td>
</tr>
</tbody>
</table>
Table 10.12. gpioin callback parameters

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>in</td>
<td>The input pin values</td>
</tr>
</tbody>
</table>

The return value of gpioin/gpioout is ignored.

See the gr712/examples/input for an example implementation.

10.9. 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 initialization.
11. Atmel AT697 emulation

To emulate the AT697E chip the `at697e` option should be used. That sets up parameters for core TSIM2 to match AT697E. In addition, the AT697 loadable AHB module implements emulation models for the PCI core of the AT697E. Apart from an overview of what TSIM supports for AT697 emulation, this chapter is mainly about the AT697 AHB module.

The following table lists which cores in the AT697 are modelled by TSIM or not. Some supported cores are modelled by the core TSIM2 and some in the AT697 AHB module. 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 built-in simulation models and the rest of the chapter for the AHB module models.

<table>
<thead>
<tr>
<th>Core</th>
<th>Status</th>
<th>Notes</th>
</tr>
</thead>
<tbody>
<tr>
<td>LEON2FT</td>
<td>Supported by core TSIM2</td>
<td>No FT features are modelled.</td>
</tr>
<tr>
<td>FPU</td>
<td>Supported by core TSIM2</td>
<td></td>
</tr>
<tr>
<td>LEON2 system registers</td>
<td>Supported by core TSIM2</td>
<td></td>
</tr>
<tr>
<td>Interrupt controller</td>
<td>Supported by core TSIM2</td>
<td></td>
</tr>
<tr>
<td>Memory controller</td>
<td>Supported by core TSIM2</td>
<td>No FT features are modelled</td>
</tr>
<tr>
<td>UART</td>
<td>Supported by core TSIM2</td>
<td></td>
</tr>
<tr>
<td>PCI</td>
<td>Supported by AHB module</td>
<td></td>
</tr>
<tr>
<td>I/O port</td>
<td>Not supported</td>
<td>Easily modelled in user module</td>
</tr>
<tr>
<td>JTAG</td>
<td>Not supported</td>
<td>Debug link</td>
</tr>
<tr>
<td>Debug UART</td>
<td>Not supported</td>
<td>Debug link</td>
</tr>
<tr>
<td>DSU</td>
<td>Not supported</td>
<td>Debug unit</td>
</tr>
</tbody>
</table>

TSIM supports running several AHB modules. User defined models for unsupported cores can be used in parallel with the AT697 AHB module. User defined models can also override the simulation models in the AT697 AHB module.

11.1. Overview of the AT697 AHB module

The PCI emulation is implemented as a AT697 AHB module that will process all accesses to memory region 0xa0000000 - 0xf0000000 (AHB slave mode) and the APB registers starting at 0x80000100. The AT697 AHB module implements all registers of the PCI core. It will in turn load the PCI user modules that will implement the devices. The AT697 AHB module is supposed to be the PCI host. Both PCI Initiator and PCI Target mode are supported. The interface to the PCI user modules is implemented on bus level. Two callbacks model the PCI bus.

See also Section 5.4.1 on some limitations of some features when using this module. The AT697 AHB module is only supported for TSIM2 LEON2.

The following files are delivered with the AT697 TSIM module:

<table>
<thead>
<tr>
<th>File</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>at697/linux/at697.so</td>
<td>AT697 AHB module for Linux</td>
</tr>
<tr>
<td>at697/win32/at697.dll</td>
<td>AT697 AHB module for Windows</td>
</tr>
<tr>
<td>Input</td>
<td>The input directory contains two examples of PCI user modules</td>
</tr>
<tr>
<td>at697/examples/input/README.txt</td>
<td>Description of the user module examples</td>
</tr>
<tr>
<td>at697/examples/input/Makefile</td>
<td>Makefile for building the user modules</td>
</tr>
<tr>
<td>at697/examples/input/pci.c</td>
<td>PCI user module example that makes AT697 PCI initiator accesses</td>
</tr>
</tbody>
</table>
11.2. Loading the module

The module is loaded using the TSIM2 option `--ahbm`. All core specific options described in the following sections need to be surrounded by the options `--designinput` and `--designinputend`, e.g:

On Linux:

```bash
tsimsim -ahbm ./at697/linux/at697.so
  --designinput ./at697/examples/input/pci.so --designinputend
```

On Windows:

```bash
tsimsim -ahbm ./at697/win32/at697.dll
  --designinput ./at697/examples/input/pci.dll --designinputend
```

This loads the AT697 AHB module `/at697.so` which in turn loads the PCI user module `/pci.so`. The PCI user module `/pci.so` communicates with `/at697.so` using the PCI user module interface, while `/at697.so` communicates with TSIM via the AHB interface.

11.3. AT697 initiator mode

The PCI user module should supply the callback function `acc()`. The AT697 AHB module will call this function to emulate AHB slave mode accesses or DMA accesses that are forwarded via `acc()`. The cmd parameter determines which command to use. Configuration cycles have to be handled by the PCI user module.

11.4. AT697 target mode

The AT697 AHB module supplies the callback `target_acc()` to the PCI user modules to implement target mode accesses from the PCI bus to the AHB bus. The PCI user module should trigger access events itself by inserting itself into the event queue.

11.5. Definitions

```c
#define ESA_PCI_SPACE_IO      0
#define ESA_PCI_SPACE_MEM     1
#define ESA_PCI_SPACE_CONFIG  2
#define ESA_PCI_SPACE_MEMLINE 3
#define ESA_PCI_ACC_OK           0
#define ESA_PCI_ACC_MASTER_ABORT 1

struct esa_pci_input {
  struct input_inp _b;
  int (*acc)(struct esa_pci_input *ctrl, int cmd, unsigned int addr, unsigned int wsize, unsigned int *data, unsigned int *abort, unsigned int *ws);
  int (*target_acc)(struct esa_pci_input *ctrl, ...
```
```c
int cmd,
unsigned int addr,
unsigned int wsize,
unsigned int *data,
unsigned int *mexc);
```

### 11.5.1. PCI command table

<table>
<thead>
<tr>
<th>Code</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0000</td>
<td>&quot;IRQ acknowledge&quot;</td>
</tr>
<tr>
<td>0001</td>
<td>&quot;Special cycle&quot;</td>
</tr>
<tr>
<td>0010</td>
<td>&quot;I/O Read&quot;</td>
</tr>
<tr>
<td>0011</td>
<td>&quot;I/O Write&quot;</td>
</tr>
<tr>
<td>0100</td>
<td>&quot;Reserved&quot;</td>
</tr>
<tr>
<td>0101</td>
<td>&quot;Reserved&quot;</td>
</tr>
<tr>
<td>0110</td>
<td>&quot;Memory Read&quot;</td>
</tr>
<tr>
<td>0111</td>
<td>&quot;Memory Write&quot;</td>
</tr>
<tr>
<td>1000</td>
<td>&quot;Reserved&quot;</td>
</tr>
<tr>
<td>1001</td>
<td>&quot;Reserved&quot;</td>
</tr>
<tr>
<td>1010</td>
<td>&quot;Configuration Read&quot;</td>
</tr>
<tr>
<td>1011</td>
<td>&quot;Configuration Write&quot;</td>
</tr>
<tr>
<td>1100</td>
<td>&quot;Memory Read Multiple&quot;</td>
</tr>
<tr>
<td>1101</td>
<td>&quot;Dual Address Cycle&quot;</td>
</tr>
<tr>
<td>1110</td>
<td>&quot;Memory Read Line&quot;</td>
</tr>
<tr>
<td>1111</td>
<td>&quot;Memory Write And Invalidate&quot;</td>
</tr>
</tbody>
</table>

### 11.6. Read/write function installed by PCI module

This function should be set by the PCI user module:

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

If set, this function is called by the AT697 module whenever PCI bus master reads. Called for AHB-slave mapped accesses as well as AHB-Master/APB DMA.

#### Table 11.3. acc callback parameters

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>cmd</td>
<td>Command to execute, see Section 11.5.1.</td>
</tr>
<tr>
<td>addr</td>
<td>PCI address</td>
</tr>
<tr>
<td>wsize</td>
<td>0: 8-bit access 1: 16-bit access, 2: 32-bit access. Is always 2 for read accesses.</td>
</tr>
<tr>
<td>data</td>
<td>Data buffer. The user module should return the read data in *data for read commands or write the data in *data for write commands.</td>
</tr>
<tr>
<td>ws</td>
<td>Set *ws to the number of PCI clocks it takes to complete the transaction.</td>
</tr>
<tr>
<td>abort</td>
<td>Set *abort to 1 to generate target abort, or 0 otherwise.</td>
</tr>
</tbody>
</table>

The return value of acc determines if the transaction terminates with master abort (1, ESA_PCI_ACC_MASTER_ABORT), or not (0, ESA_PCI_ACC_OK).

### 11.7. Read/write function installed by AT697 module

The following function is installed by the AT697 AHB module:

```c
int (*target_acc)(struct esa_pci_input *ctrl,
int cmd,
unsigned int addr,
unsigned int wsize,
unsigned int *data,
unsigned int *mexc);
```
The PCI user module can call this function to emulate a PCI target mode read access to the AT697.

**Table 11.4. target_acc parameters**

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>cmd</td>
<td>Command to execute, see Section 11.5.1. Configuration cycles are not supported, the AT697 module is supposed to be the host.</td>
</tr>
<tr>
<td>addr</td>
<td>PCI address. Should always be word aligned for read accesses.</td>
</tr>
<tr>
<td>wsize</td>
<td>0: 8-bit access 1: 16-bit access, 2: 32-bit access. Should always be 2 for read accesses.</td>
</tr>
<tr>
<td>data</td>
<td>Data buffer. The read data is returned in *data for read commands or the data in *data is written for write commands.</td>
</tr>
<tr>
<td>mexc</td>
<td>The AT697 PCI models sets *mexc to 0 if access is successful, or to 1 in case of target abort.</td>
</tr>
</tbody>
</table>

If a supported command hits MEMBAR0, MEMBAR1 or IOBAR, target_acc() will return 1 otherwise it will return 0.

### 11.8. Registers

Table 11.5 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 11.5. PCI register support**

<table>
<thead>
<tr>
<th>Register</th>
<th>Implemented</th>
<th>Not implemented</th>
</tr>
</thead>
<tbody>
<tr>
<td>PCIID1</td>
<td>device id, vendor id</td>
<td></td>
</tr>
<tr>
<td>PCISC</td>
<td>stat 13, stat 12, stat 11, stat 7, stat 6 stat 5, stat 4, com2, com 1, com1</td>
<td>stat15 stat14 stat10_9 stat8 com10 com9 com8 com7 com6 com5 com4 com3</td>
</tr>
<tr>
<td>PCIID2</td>
<td>class code, revision id</td>
<td></td>
</tr>
<tr>
<td>PCIBHDLC</td>
<td>[bist, header type, latency timer, cache size] config-space only</td>
<td></td>
</tr>
<tr>
<td>PCIMBAR1</td>
<td>base address, pref, type, msi</td>
<td></td>
</tr>
<tr>
<td>PCIMBAR2</td>
<td>base address, pref, type, msi</td>
<td></td>
</tr>
<tr>
<td>PCIIOBAR3</td>
<td>io base address, ms</td>
<td></td>
</tr>
<tr>
<td>PCISID</td>
<td>subsystem id, svi</td>
<td></td>
</tr>
<tr>
<td>PCICP</td>
<td>pointer</td>
<td></td>
</tr>
<tr>
<td>PCILI</td>
<td>[max_lat min_gnt int_pin int_line] config-space-only</td>
<td></td>
</tr>
<tr>
<td>PCIRT</td>
<td>[ retry trdy ] config-space-only</td>
<td></td>
</tr>
<tr>
<td>PCICW</td>
<td></td>
<td>ben</td>
</tr>
<tr>
<td>PCISA</td>
<td>start address</td>
<td></td>
</tr>
<tr>
<td>PCIHW</td>
<td></td>
<td>ben</td>
</tr>
<tr>
<td>PCIDMA</td>
<td>wdcnt, com</td>
<td>b2b</td>
</tr>
<tr>
<td>PCIIS</td>
<td>act, xff, xfe, rfe</td>
<td>dmas, ss</td>
</tr>
<tr>
<td>PCIIC</td>
<td>mod, commsb</td>
<td>dwr, dww, perr</td>
</tr>
<tr>
<td>PCITPA</td>
<td>tpa1, tpa2</td>
<td></td>
</tr>
<tr>
<td>PCITSC</td>
<td></td>
<td>ermem, xff, xfe, rfe, tms</td>
</tr>
<tr>
<td>PCIITE</td>
<td>dmaer,imier, tier</td>
<td>cmfer, imper, tbeer, tper, syser</td>
</tr>
</tbody>
</table>
Register | Implemented | Not implemented
---|---|---
PCIITP | dmaer, imier, tier | cmfer, imper, tbeer, tper, syer
PCIITF | dmaer, imier, tier, cmfer, imper, tbeer, tper, syer |
PCID | dat |
PCIBE | dat |
PCIDMAA | addr |
PCIA | p0, p1, p2, p3 |

11.9. Debug flags

The switch `-designdebug` flags can be used to enable debug output. The possible values for flags are as follows:

Table 11.6. Debug flags

<table>
<thead>
<tr>
<th>Flag</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>ESAPCI_REGACC</td>
<td>Trace accesses to the PCI registers</td>
</tr>
<tr>
<td>ESAPCI_ACC</td>
<td>Trace accesses to the PCI AHB-slave address space</td>
</tr>
<tr>
<td>ESAPCI_DMA</td>
<td>Trace DMA</td>
</tr>
<tr>
<td>ESAPCI_IRQ</td>
<td>Trace PCI IRQ</td>
</tr>
</tbody>
</table>

11.10. Commands

```shell
pci
```
Displays all PCI registers.
12. TPS VxWorks 6.x AHB Module

12.1. 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. See also Section 5.4.1 on some limitations of some features when using this module.

Table 12.1. Files delivered with the TPS VxWorks TSIM module

<table>
<thead>
<tr>
<th>File</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>tps/linux/tps-vxworks.so</td>
<td>TPS VxWorks module for Linux</td>
</tr>
<tr>
<td>tps/win32/tps-vxworks.dll</td>
<td>TPS VxWorks module for Windows</td>
</tr>
</tbody>
</table>

12.2. Loading the module

The module is loaded using the TSIM2 option -ahbm. It can be used in conjunction with other modules, such as the UT699 and GR712RC modules.

On Linux (together with the UT699 design):

```bash
tsim-leon3 -ut699 -ahbm ./tps/linux/tps-vxworks.so -ahbm ./ut699/linux/ut699.so
```

On Windows (together with the GR712RC design):

```bash
tsim-leon3 -gr712rc -ahbm ./tps/win32/tps-vxworks.dll -ahbm ./gr712/win32/gr712.dll
```

12.3. Configuration

By default the module uses IRQ 5 and UDP port 0x4321. This can be changed by using the following command line arguments:

- **-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:

```bash
tsim-leon3 -ut699 -ahbm ./tps/linux/tps-vxworks.so -ahbm ./ut699/linux/ut699.so
   -tps_vxworks_port 5000 -tps_vxworks_irq 10
```
13. Support

For support contact the Cobham Gaisler 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.