Rodos LEON2 Port

Version 0.9

GIT 9f8e517ebdb6bcb0fb48db18a6e61c34762c889b

For Rodos distribution tarball 2010-09-03

AbsInt Angewandte Informatik GmbH
Dr.-Ing. Henrik Theiling

October 8, 2010

Contents

1 Introduction

Rodos is an operating system for embedded systems with a C++ API. One of its main design goals is simplicity.

Rodos is already available for several hardware architectures. This document describes the Rodos LEON2 port, which supports running Rodos on LEON2 hardware. The LEON2 is a SPARC V8 processor that is based on configurable VHDL. The implementation of the Rodos LEON2 port was tested on an Aeroflex Gaisler SPWRTC board which features a fail tolerant variant of a LEON2 processor, the LEON2-FT manufactured by Atmel. The Rodos port for this architecture supports several of the special features of the LEON2-FT, e.g. the fully EDAC-checksum protected busses and memory.

Aeroflex Gaisler's GRMON tool was used for programming and debugging the SPWRTC board.

The following sections will shed light on several architectural details of the Sparc processor and on its LEON2 implementation in the Rodos operating system.

2 Using the Software

2.1 Unpacking

The Rodos LEON2 port is packaged as a tarball containing a set of files needed additional to the Rodos distribution 2010-09-03.

To unpack, change to your Rodos distribution directory:

```
cd rodos-64-distribution
```

There, unpack the Rodos LEON2 port tarball:

```
tar xzvpf \
  $D/rodos-leon2-9f8e517ebdb6bcb0fb48db18a6e61c34762c889b.tgz
```

Replace `$D` with the directory of the tarball file.

2.2 Configuration

The configuration of the Rodos LEON2 port is done via two include files: `params.h`, which contains the general settings, and `hw_config.h`, which contains the hardware dependent settings. These files are documented using Doxygen style comments. Please refer to that documentation for more details.

The configuration options are grouped by topic and can be found in the Doxygen menu "Modules".

Each option is protected with an `#ifndef`, so that the options can be predefined on the compiler's command line.

2.3 Compilation

The Rodos LEON2 port was developed using Aeroflex Gaisler's pre-compiled gcc in the "bcc" configuration, which is the bare compiler without operation system. Any recent version of a cross-compiled gcc for SPARC should do, too, since none of the Gaisler libraries were used.

Once the SPARC compiler is successfully compiled and installed, set up the environment variable `RODOS_GCC_PREFIX_LEON2` to point to the location of the compiler, including the prefix of the executables. E.g.:

```
RODOS_GCC_PREFIX_LEON2=/opt/sparc-elf-4.4.2/bin/sparc-elf
export RODOS_GCC_PREFIX_LEON2
```

Now, enter the "rodos-64-distribution" directory and set up the build environment by typing:

```
source ./make/rodosenvs
source ./make/leon2-vars-envs
```

To compile the Rodos library, use:

```
leon2-lib
```

To compile a specific test, use:

```
leon2-executable tutorials/core/preemptiontest.cpp
```

Substituting the path to the test, of course. The script generates a file `tst` which is an ELF file directly suitable for GRMON for programming the board. To set a different output file name, set the `OUT` environment variable:

```
OUT=out.elf leon2-executable \
    tutorials/core/preemptiontest.cpp
```

To pass configuration options to the compilation (see `hw_config.h` and `params.h`), use the `MORE_CFLAGS` make variable:

```
MORE_CFLAGS='-DHW_CONFIG_CODE_IN_RAM=0'
export MORE_CFLAGS
leon2-lib
leon2-executable tutorial/core/preemptiontest.img
```

To build the Doxygen documentation for LEON2, use:

```
cd doc && doxygen Doxyfile.LEON2
```

3 Implemenation Details

3.1 Timers

The LEON2 architecture supports two hardware timers. The timers are counters that are clocked by a prescaler. The prescaler is a counter that is clocked by the system clock. The Rodos LEON2 port allows the prescaler to be programmed to different values, leading to different time slice granularities. It is usually wise to choose fast prescaler settings, so that the timers are clocked at a high rate. The timers are 24-bits wide, which is sufficient even at high clock rates.

The Rodos LEON2 port uses the first timer to generate preemption interrupts.

The second timer is used to advance the system clock.

The SPWRTC design contains two more timers, which are 32-bits wide each, which are not used by the Rodos LEON2 port, but are free to be used by the application.

3.2 Watchdog

The LEON2 architecture supports an optional watchdog, which is typically present in safety-oriented designs. The watchdog is a simple counter that cannot be disabled, which is clocked from the same prescaler that also clocks the two general-purpose timers.

The watchdog counter can be set in software, but cannot be stopped. It is decremented at each (prescaled) clock tick. If it reaches 0, the counter stops and signals a watchdog timeout on a dedicated CPU pin. Typically, this pin is routed to the reset circuit so the CPU can be rebooted if it becomes unresponsive.

The Rodos LEON2 port supports automatic resetting of the watchdog counter at each task switch. This support is optional and can be switched off if an application wants to use the watchdog for different purposes (e.g. for watching a particularly important thread).

The watchdog can be enabled and disabled with a configuration option (`HW_CONFIG_HAVE_WATCHDOG`).

3.3 Error Mode

The SPARC architecture has a dedicated error mode that is entered when the CPU encounters a catastrophic failure which it cannot handle otherwise. The error mode is also the right way to stop a system if the OS encounters a catastrophic failure.

The Rodos LEON2 port implements a function `hwSysPanic` that can be invoked whenever the OS detects an insoluble problem. It will then print the error message passed to that function and enter error mode.

Like the watchdog trigger, the LEON CPU has a CPU pin that indicates error mode. This pin, too, may be routed to the reset circuitry to cause an immediate reboot as soon as error mode is entered.

Some designs lack this route, but the watchdog will jump in, too: once error mode is entered, the watchdog won't be reset, will trigger, and the watchdog may trigger a reset. This is the way the SPWRTC board is built.

3.4 Delay Slots

The SPARC architecture features delay slots. There are typical examples of control transfer instructions.

`jmp %l1   ; rett %l2`	Return from trap, re-executing the trap-causing instruction, including correct execution of delay slots.
`jmp %l2   ; rett %l2+4`	Return from trap, starting after the trap-causing instruction.
`bCC,a  .+8 ; X`	If CC, execute X, otherwise don't execute X. (CC decides whether the delay slot X is executed or not.)

To implement delay slots, the SPARC architecture has two program counters: the PC (program counter) and the nPC (next program counter). This effects trap handling, because traps can be triggered at any time, and since PC and nPC are independent, the processor must store both program counters to be able to return to the interruption point. The above instruction patterns show this: PC is stored in L1 and nPC is stored in L2, and the return sequence uses both to return to the correct CPU state before the trap.

3.5 Register Windows

The SPARC architecture allows 2–32 register windows with 16 registers each to be implemented in hardware. The LEON2 VHDL can be configured accordingly. The Rodos LEON2 port can also be configured accordingly by setting a configuration option.

Typically, a SPARC processor has 8 register windows. The LEON2-FT used on the SPWRTC board also has 8 register windows.

At any given time, the SPARC processor has direct access to 32 registers. 8 of these registers are global registers not affected by the register window switching. One of these 8 global registers, G0, is hardwired to 0. 8 more registers overlap with the previous register window, 8 with the next, and 8 are local and not shared with any other register window.

The SPARC processor uses two registers to control the register windows: in the program state register (PSR), there is a field called "current window pointer" (CWP) that indicates which register window is currently active.

At the beginning of a routine, user code typically uses the `SAVE` instruction to turn the register window (decrement CWP modulo the number of register windows) in order to use 16 different registers than the caller. The 8 overlapping registers shared with the caller are used for parameter passing and return values. When a routine returns, it typically executes `RESTORE` to turn the register window back to the caller's window (increment CWP modulo the number of register windows).

Since the hardware only has a limited set of register windows, using `SAVE` and `RESTORE` would eventually overwrite registers of previous callers. Therefore, the hardware has another register, the "window invalid mask" (WIM), which contains one bit for each register window. If it is 1, that window is invalid. If a `SAVE` or `RESTORE` instruction turn the register window such that the new window would be marked invalid, a trap is triggered and the Rodos LEON2 port saves or restores registers to or from memory (on the stack), adjusts the WIM register, and re-executes the trapped `SAVE` or `RESTORE`, which will then succeed.

This way of using the register windows is prescribed in the SPARC ABI, and gcc uses `SAVE` and `RESTORE` instructions accordingly. Therefore, the Rodos LEON2 port implements the register windows exactly according to the described methods.

The following sections give some more detail of how to handle register windows in special situations, like interrupts and task switches.

3.6 Interrupts and Traps

The SPARC architecture distinguishes three types of interrupts:

- Software traps invoked like a call with the `TA` instruction,
- Exceptions (or synchronous interrupts) caused by normal instruction execution,
- Interrupts (or asynchronous interrupts) caused by external events

All three are usually called "traps".

The implemented standard SPARC ABI described above reserves one register window for two purposes:

- To detect a register stack overflow/underflow,

- To guarantee that a trap has some registers to use.

When a trap is entered, the register window is turned like in a normal `SAVE` instruction, with the difference that no window overflow interrupt is ever generated.

I.e., if a trap is caused when a `SAVE` would cause a register window overflow, the trap is entered anyway, and may use the window marked invalid in the WIM register. It may then use the 8 local registers, which are invisible to both the previous and the next register window.

When returning from a trap using `RETT`, the register window is turned back just like a normal `RESTORE` would do. If the new register window is marked invalid, the CPU causes error mode.

Unfortunately, this mechanism means that SPARC traps cannot be nested easily. In fact, the SPARC disables traps automatically when entering a trap handler, meaning that if a software trap (`TA` instruction) or an exception occurs, the CPU enters error mode. Interrupts are deferred until traps are re-enabled.

The Rodos LEON2 port runs in the following modes:

User mode with enabled traps (including interrupts): This is used when executing user tasks.

Kernel mode: Supervisor mode with enabled traps (including interrupts): This is used when running the startup code and the scheduler. When the CPU accepts a preemption interrupt in this mode, no thread switch is initiated.

Trap mode: Supervisor mode with disabled traps (including interrupts): This is used during boot time and whenever a trap is taken.

Emergency mode: supervisor mode with enabled traps but disabled interrupts: This is used when trap handlers in C are executed. It allows running C code without interference of external interrupts (except INT 15, the non-maskable interrupt).

3.7 Trap Handlers

As the previous section already mentioned, a SPARC CPU switches off traps when entering a trap handler. The `RETT` instruction switches traps back on.

This means that `SAVE` and `RESTORE` instructions cannot easily be used in a trap handler, because they may in turn trigger window overflow/underflow exceptions. This means that normal C code cannot be executed in trap handlers, because the compiler generated function prologues and epilogues may contain `SAVE` and `RESTORE`.

Since the SPARC architecture does not directly support writing trap handlers in C, the Rodos LEON2 port provides a trampoline function that allows C level trap handlers to be installed. This makes it much easier to add hardware drivers that are highly architecture independent.

The UART and system time interrupts are routed through this trampoline so they can be written in less error prone C instead of Assembler.

3.8 Setting Up a Stack

When initialising a stack for calling an initial function `someStartFunc`, the `PSR.CWP` (current window pointer) is set to 0, and `WIM` (window invalid mask) is set to 2. The frame pointer is set to the address *behind* the end of the stack area. Let that address be called `someStackEnd`. The stack pointer is set up relative to the frame pointer in order to reserve a stack frame.

`PSR.CWP`	$= 0$
`WIM`	$= 2$
`I6`	= `someStackEnd`
`O6`	= `someStackEnd` − size of initial stack frame
`O0 ...O2`	possible parameters of `someStartFunc`

This means that the current window is valid, the frame pointer and stack pointer are set up properly.

Then we call `someStartFunc`.

This pattern can be found multiple times in different places in the implementation, notable in the boot code, in the code entering a previously suspended task, and the code to enter the scheduler. In all these cases, a stack is either switched or initially set up.

3.9 Register Windows and Task Switch

Register windows are used according to the normal SPARC ABI that gcc generates code for. This means the Rodos LEON2 port implements the turning register window with window overflow and underflow exceptions.

The register window exceptions have been reduced to a minimum, so they execute very fast.

The inherent penalty of the SPARC ABI is quite a long time for switching tasks. Freezing a task needs to flush all registers to the stack, which are about 170 32-bit registers for a typical LEON2 processor. There's no way around this: the SPARC architecture does not allow interrupts during this flushing.

The restoring procedure needs to restore 71 registers to enter the new thread.

In detail: saving requires to store PC, nPC, PSR, Y, FSR, F0. . . 31, G1. . . 7, once

O0…7, and once…seven times L0…7 and I0…7. The L* registers of the window marked invalid need not be stored. They are available to the trap handler handling the task switch. Interrupts are disabled for all these stores.

Restauration requires to restore PC, nPC, PSR, Y, FSR, F0…31, G1…7, and once O0…7, L0…7, I0…7. Interrupts are disabled the whole time.

3.10 Global Register Usage

The SPARC ABI defines that of the global registers, `g1` is the only one that is free as a scratch register in user code. The other global registers are either assigned to the application (library code) or the system.

The Rodos LEON2 port does not use any of the global registers, leaving them to the user code. To take advantage of this, the following compile options should be used: `-mapp-regs -fcall-used-g6 -fcall-used-g7`.

3.11 Extensions Of the LEON Port

This section lists extensions of the Rodos LEON2 port compared to the ARM port. Partly this is due to different hardware, and partly to try to provide useful functionality.

Write Protection: The SPWRTC CPU supports extended write protection capabilities, and the Rodos LEON2 port uses it to protect against erroneous write accesses all code and read-only data that was copied to RAM, in order to increase safety.

EDAC support: The SPWRTC hardware has EDAC checksum support for both register file, PROM and RAM. It is supported by the Rodos LEON2 port at all levels: for registers, it is enabled by default. To support it, the register file is initialised with 0 at boot time. For PROM checking, EDAC can be enabled by DIP switch PIO[2]. For RAM checking, EDAC must be enabled in a few hardware registers, and memory must be properly cleared at boot time. Support can be enabled with a configuration option(`HW_CONFIG_ENABLE_EDAC`).

UART Queues: The UART driver provides queues for asynchronous input and output, without which no undelayed communication would be easily possible. It also properly protects hardware access by switching off interrupts.

UART Waiting: The UART driver provides routines to allow non-busy waiting: get and put methods are provided with a timeout parameter, and the driver reschedules during waiting time instead of doing busy-waiting by polling.

Idle: The LEON2 can power down most of the core, waking up automatically at the next interrupt request. This Rodos LEON2 port provides a routine `hwIdle()` to deactivate the core. This is invoked in `sp_partition_yield()` so that the processor sleeps when the idle thread calls that function.

Static Initialisation Guards: GCC protects local static initialisations with a guard, because different threads could try to initialise the data at the same time. The Rodos LEON2 port provides an implementation for the necessary routines (__cxa_guard_acquire, __cxa_guard_release, __cxa_guard_abort).

3.12 Problems With UART Handshaking

The LEON2 contains a serial port (UART) implementation that supports hardware flow control, also called "RTS/CTS handshaking" or "hardware handshaking". Additional to the data signal `RTX` from sender to receiver, flow control uses a signal from receiver to sender, `RTS` *Ready To Send*, to prevent overruns when the receiver is slower than the sender. Using this signal, the receiver indicates that no more data can be received. This will cause the sender to stop sending. For the reverse direction, there is a corresponding signal (`CTS` *Clear To Send*) so that flow control can protect both directions of a full-duplex connection from overrunning.

UART flow control can be switched on and off with the configuration option `HW_CONFIG_UART_FLOW_CONTROL`.

The LEON2 UART flow control is implemented completely in hardware, so that an UART driver cannot influence the `RTS` signal directly.

The LEON2 UART hardware has two registers for receiving data: one is the shift register where the data is composed while being received. There is no software access to this shift register. The other register is the data register where a fully received byte is placed by the hardware for the software driver to get it.

The hardware automatically disables `RTS` to prevent more data to be sent when it receives the start bit of a new byte while the data register is occupied. The hardware will be able to receive that last byte and keep it in the shift register until the software driver empties the data register. The transmission must stop after that byte, because the two hardware registers are full then. If another start bit is encountered while both registers are full, an overrun error will occur.

This effectively means that the LEON2 UART hardware requires `RTS` handshaking to be handled promptly at byte boundaries: when the LEON2 hardware tells the sender to stop, the sender *must* stop sending after the byte currently sent, otherwise an overrun error will occur.

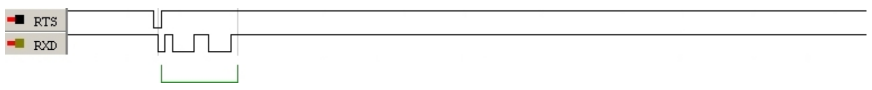

Figure 1: Signals with null-modem cable: the LEON2 hardware stops sending after the current byte when RTS goes high (=disable). Byte boundaries including start, parity and stop bit are indicated in grey.

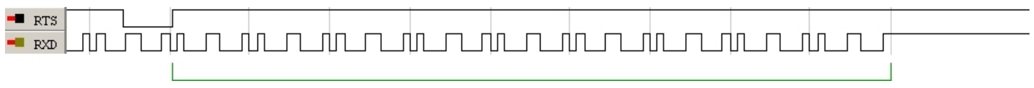

Figure 2: Signals with connected PC: the PC hardware only stops after several more bytes when RTS goes high (=disable).

During our testing we found that when the LEON2 receives data from a connected PC with enabled flow control, the UART still produces overrun errors.

We examined this problem closer in order to exclude a software bug and found that the PC does not stop sending data immediately, but instead may send several more bytes. Since the LEON2 hardware is not able to cope with the additional data, as explained above, this explains the overrun errors despite the enabled flow control.

We used a logic analyser to document this behaviour.

When two LEON2 UARTs are connected, this problem will not occur, because the LEON2 sender hardware behaves as expected and stops sending without any jitter. We validated this by using a null-modem cable as well as the internal loop-back of the UART hardware.

Unfortunately, there is no way to fix this problem in software e.g. by disabling `RTS` earlier, because the LEON2 UART low-level implementation is completely in hardware and the hardware signals cannot directly be influenced, but only by emptying the data register.

Since the PC hardware is industry standard, this means that the LEON2 UART hardware handshaking is incompatible with many other UART implementations, because it is not industry standard compliant.

4 Testing

Most of the Rodos testsuite were tested manually for the Rodos LEON2 port. There usually require manual supervision and cannot be automated. The next section lists all the tutorial programs that have manually been tested with the Rodos LEON2 port. Each test is marked with a GIT identifier for which the test was checked, so that the test result can be reproduced.

Most tests never end, because there is no way to halt the operating system. Therefore, the “runtime” data may also indicate when an successful test was interrupted. For tests that are supposed to enter error mode or let the watchdog fire, the approximate runtime to that program end is given. The runtime given is the total runtime of all tests done.

For stress testing, some tests are done with a very high rate of preemption interrupts.

The following final versions were tested. No modification was made to Rodos or the LEON2 port itself, but the documentation and the tests themselves have changed during testing:

9dc978851ea157901e41454a8dd0a41a686cf66d	The first final Rodos LEON2 port
f76cc0fa5ff32eacd3b2eb4dcdc2164a8ff6b191	Modifications to systime test

4.1 Own Tests

4.1.1 Test: idle

This test is the bare librodos.a without additional object files. The only task is the idle task, so the OS does nothing but task switches. This test verifies that a bare system works. This test includes the functionality of the hwIdle() function that powers down much of the processor until the next interrupt.

Date: `2010-10-04`

GIT Version Tested: `9dc978851ea157901e41454a8dd0a41a686cf66d`

Runtime of Test: >10min

Preemptions/Second: 100

Error Mode Entered: no

Watchdog Fired: no

Observation: The RODOS boot message is correcty sent over UART0.

4.1.2 Test: aitest/uart

Two tasks, one sending “0” on UART0, one sending “1” on UART1, each sending 10 characters per second.

Date: `2010-10-04`

GIT Version Tested: `9dc978851ea157901e41454a8dd0a41a686cf66d`

Runtime of Test: >10min

Preemptions/Second: 100

Error Mode Entered: no

Watchdog Fired: no

Observation: The characters where correctly received by a Linux PC on each of the serial lines.

4.1.3 Test: aitest/regwast

Four identical tasks, each storing random values in all registers (int and float), then waiting for a couple of nops, then performing a recursion deep enough to cause register window overflow and underflow traps (the exact depth is selected by random), and verifying that registers haven't changed (caller save regs before the recursion, callee save regs after the recursion). Due to the recursion, the register window is constantly turned. Furthermore, this installs a user trap with some complex floating point operations, testing that the trap gate to C code works without changing registers. This is a stress test for window overflow/underflow traps, preemptive task switching, task gates, and saving, restauration and switching of registers and stacks. If any register value or PSR.ICC ever changes, a system panic is triggered.

Date: `2010-10-04`

GIT Version Tested: `9dc978851ea157901e41454a8dd0a41a686cf66d`

Runtime of Test: >60min

Preemptions/Second: 40000

Error Mode Entered: no

Watchdog Fired: no

Observation: All tasks continuously report their task ID via the serial line every few seconds, indicating 256 successful recursive tests each time.

4.1.4 Test: aitest/regwastrom1

The same test as `regwast`, with code and rodata in ROM instead of in RAM.

Date: `2010-10-04`

GIT Version Tested: `9dc978851ea157901e41454a8dd0a41a686cf66d`

Runtime of Test: >30min

Preemptions/Second: 40000

Error Mode Entered: no

Watchdog Fired: no

Observation: All tasks continuously report their task ID via the serial line every few seconds, indicating 256 successful recursive tests each time.

4.1.5 Test: aitest/regwastrom2

The same test as `regwast`, with code in RAM and rodata in ROM.

Date: `2010-10-04`

GIT Version Tested: `9dc978851ea157901e41454a8dd0a41a686cf66d`

Runtime of Test: >10min

Preemptions/Second: 40000

Error Mode Entered: no

Watchdog Fired: no

Observation: All tasks continuously report their task ID via the serial line every few seconds, indicating 256 successful recursive tests each time.

4.1.6 Test: aitest/regwastrom3

The same test as `regwast`, with code in ROM and rodata in RAM.

Date: `2010-10-04`

GIT Version Tested: `9dc978851ea157901e41454a8dd0a41a686cf66d`

Runtime of Test: >5min

Preemptions/Second: 40000

Error Mode Entered: no

Watchdog Fired: no

Observation: All tasks continuously report their task ID via the serial line every few seconds, indicating 256 successful recursive tests each time.

4.1.7 Test: aitest/systime

Two tasks each checking that the system time NOW() never runs backwards. In an infinite loop, each task does a few thousand tests and the prints a signature mark. This is an interesting test, because the hardware counter only has 24 bits, meaning an underrun (and interrupt), once every 17 seconds. The test checks that the interrupt-handled software counter plus the hardware counter are read consistently.

Date: `2010-10-04`

GIT Version Tested: `f76cc0fa5ff32eacd3b2eb4dcdc2164a8ff6b191`

Runtime of Test: >16 hours

Preemptions/Second: 100

Error Mode Entered: no

Watchdog Fired: no

Observation: The signature marks of the two tasks were printed correctly and no error was encountered.

4.2 Tutorial Examples

4.2.1 Test: core/preemptiontest

A low priority and a high priority task printing a unique signature byte every now and then. The low priority task does not yield but use busy waiting, while the high priority task uses real waiting. This tests the preemption of the low priority task.

Date: `2010-10-05`

GIT Version Tested: `f76cc0fa5ff32eacd3b2eb4dcdc2164a8ff6b191`

Runtime of Test: >45min

Preemptions/Second: 100

Error Mode Entered: no

Watchdog Fired: no

Observation: The signature marks of both tasks are output correctly.

4.2.2 Test: core/commbufftest

One sender thread and one receiver thread is started. The sender writes every 3s and the receiver reads from the buffer every 2s. An integer counter is used as payload.

Date: `2010-10-06`

GIT Version Tested: `f76cc0fa5ff32eacd3b2eb4dcdc2164a8ff6b191`

Runtime of Test: >10min

Preemptions/Second: 100

Error Mode Entered: no

Watchdog Fired: no

Observation: Receiver and sender report the counter integer being sent/received. Every now and then one counter is reported as received twice due to the difference in timing.

4.2.3 Test: core/commbufftest-struct

Similar to commbufftes, except that 3 counting floats are transmitted as payloads.

Date: `2010-10-06`

GIT Version Tested: `f76cc0fa5ff32eacd3b2eb4dcdc2164a8ff6b191`

Runtime of Test: >10min

Preemptions/Second: 100

Error Mode Entered: no

Watchdog Fired: no

Observation: Receiver and sender report the numbers being sent/received. Every now and then one counter is reported as received twice due to the difference in timing.

4.2.4 Test: core/fifotest

A transmitting thread tries to write 15 bytes consecutively to the fifo every 3 seconds. A receiving thread reads as many bytes as possible and then waits for 1s.

Date: `2010-10-06`

GIT Version Tested: `f76cc0fa5ff32eacd3b2eb4dcdc2164a8ff6b191`

Runtime of Test: >10min

Preemptions/Second: 100

Error Mode Entered: no

Watchdog Fired: no

Observation: Every 3 seconds a block of 9 transmitted bytes, 6 writes to a full buffer and 9 received counter values are reported.

4.2.5 Test: core/getsnowiat

In an infinite loop, a thread tries to receive a string from stdin (the first serial line) and writes a status message every second. Each time it reads a string it prints the string back to stdout together with some explanatory text.

Date: `2010-10-06`

GIT Version Tested: `f76cc0fa5ff32eacd3b2eb4dcdc2164a8ff6b191`

Runtime of Test: some texts were sent and received

Preemptions/Second: 100

Error Mode Entered: no

Watchdog Fired: no

Observation: The idle messages can be seen every second and when a string is written to the serial port it will get echoed back by the test program.

4.2.6 Test: core/resumefromevent

One thread suspends itself after printing a debug message and another thread starts a timer which is then used to resume the suspended thread each time the timer triggers (starting with 5 seconds, then retriggering every 3 seconds).

Date: `2010-10-06`

GIT Version Tested: `f76cc0fa5ff32eacd3b2eb4dcdc2164a8ff6b191`

Runtime of Test: >10min

Preemptions/Second: 100

Error Mode Entered: no

Watchdog Fired: no

Observation: At first after 5 seconds and then every 3 seconds a message of a resumed test-thread is printed.

4.2.7 Test: core/semaphoretest

4 threads cycle through endless loops printing debug lines and calling Thread::yield(). This leads to debug output that is clobbered by other threads and their output, except for one block of debug output which is secured by a semaphore.

Date: `2010-10-06`

GIT Version Tested: `f76cc0fa5ff32eacd3b2eb4dcdc2164a8ff6b191`

Runtime of Test: >1min

Preemptions/Second: 100

Error Mode Entered: no

Watchdog Fired: no

Observation: The debug lines within the protector scope (semaphore) are almost printed in one run consecutiveley. The first line within the protector and any other output (which is not protected) is a mixture of string-fragments of the 4 concurrent threads output. The first protected line still gets clobbered because there are non-protected outputs in other threads that can print at the same time. When they run into the semaphore they will wait until the end of the block (line 4) is reached.

4.2.8 Test: core/syncfifotest

A receiver thread reads big chunks and a sender thread sends small chunks of data. That leads to a blocking receiver when the fifo gets empty. When done receiving a big chunk the receiver pauses for 15s which leads to a blocking sender (due to a full fifo buffer).

Date: `2010-10-06`

GIT Version Tested: `f76cc0fa5ff32eacd3b2eb4dcdc2164a8ff6b191`

Runtime of Test: >5min

Preemptions/Second: 100

Error Mode Entered: no

Watchdog Fired: no

Observation: All sent bytes are being received in the correct order and reported in the debug output.

4.2.9 Test: core/threadchecker

A thread is set up with 2000 bytes of stack and uses increasing amounts of stack (in 100 byte increments). A second thread continuously does stack overflow checks on all threads in the system.

Date: `2010-10-06`

GIT Version Tested: `f76cc0fa5ff32eacd3b2eb4dcdc2164a8ff6b191`

Runtime of Test: ≈10s

Preemptions/Second: 100

Error Mode Entered: no

Watchdog Fired: no

Observation: The first thread prints its incrementing stack-usage. The second thread causes a panic when it detects a stack usage of over 2000 bytes which is the desired behaviour because the stack limit was set to 2000 bytes.

4.2.10 Test: core/timeevent_resumer01

A `TIME_LOOP` in a thread is set up to create debug output every 5s and a separate TimeEvent that activates itself after 17s and then fires every 2s is created (as long as `NOW()` $<$ 50s).

Date: `2010-10-06`

GIT Version Tested: `f76cc0fa5ff32eacd3b2eb4dcdc2164a8ff6b191`

Runtime of Test: >5min

Preemptions/Second: 100

Error Mode Entered: no

Watchdog Fired: no

Observation: The first 3 lines are output from the 5s `TIME_LOOP`. From then until 35s later there are reports of the `TIME_LOOP` every 5s, and also every 2s as well as reports of the TimeEvent resuming the Thread (every 2s).

4.2.11 Test: core/timeevent_resumer02

One thread repeatedly sets up a TimeEvent to trigger a resume 3s later and then suspends itself. The Thread gets resumed after every 3s, then renews the TimeEvent and suspends itself again.

Date: `2010-10-06`

GIT Version Tested: `f76cc0fa5ff32eacd3b2eb4dcdc2164a8ff6b191`

Runtime of Test: >5min

Preemptions/Second: 100

Error Mode Entered: no

Watchdog Fired: no

Observation: Debug lines printing the system time can be observed every 3s.

4.2.12 Test: core/timeevents

Three TimeEvents with different parameters are set up. One single shot and 2 cyclic Events. The single shot fires ≈5s (5125ms) after system start. The 2 cyclic timers fire one every 500ms and the other every 2s.

Date: `2010-10-06`

GIT Version Tested: `f76cc0fa5ff32eacd3b2eb4dcdc2164a8ff6b191`

Runtime of Test: >2min

Preemptions/Second: 100

Error Mode Entered: no

Watchdog Fired: no

Observation: All debug traces printing their according system time can be observed. The single shot timer prints only once.

4.2.13 Test: core/timepoints

An infinite loop creates new entries in a TimePoint object every 100ms and prints the after reaching 20 entries. The TimePoint object clears its buffer when doing the print().

Date: `2010-10-06`

GIT Version Tested: `f76cc0fa5ff32eacd3b2eb4dcdc2164a8ff6b191`

Runtime of Test: >2min

Preemptions/Second: 100

Error Mode Entered: no

Watchdog Fired: no

Observation: At first 2 sample entries and then an empty TimePoints object get printed. Then every 2s 20 lines of TimePoint traces can be seen.

4.2.14 Test: core/timewaitat

A thread uses AT() to wait for a given system time in the future. At first it waits for the 3rd second after system start and then it waits until NOW() + 2s.

Date: `2010-10-06`

GIT Version Tested: `f76cc0fa5ff32eacd3b2eb4dcdc2164a8ff6b191`

Runtime of Test: >5min

Preemptions/Second: 100

Error Mode Entered: no

Watchdog Fired: no

Observation: The first output appears after 3s and prints the system time. After that the system time and an incrementing counter is being printed to the screen every 2s.

4.2.15 Test: core/timewaitbeats

A thread uses setPeriodicBeat() to initialize a cyclic timer to trigger its resume. By calling waitUntilNextBeat() in an endless loop it suspends itself until the timer fires. The initial waits is 3s and the cyclic wait is 2s long.

Date: `2010-10-06`

GIT Version Tested: `f76cc0fa5ff32eacd3b2eb4dcdc2164a8ff6b191`

Runtime of Test: >5min

Preemptions/Second: 100

Error Mode Entered: no

Watchdog Fired: no

Observation: The first output appears after 3s and prints the system time. After that the system time and an incrementing counter is being printed to the screen every 2s.

4.2.16 Test: core/timewaitbeats_shortcut

Initiates a thread and uses `TIME_LOOP` with one initial 3s and one cyclic 2s timer value to loop over a printf statement that prints the system time.

Date: `2010-10-07`
GIT Version Tested: `f76cc0fa5ff32eacd3b2eb4dcdc2164a8ff6b191`
Runtime of Test: >10min
Preemptions/Second: 100
Error Mode Entered: no
Watchdog Fired: no
Observation: After the first 3seconds a debug line printing the system time can be observed every 2 seconds.

4.2.17 Test: core/timewaitfor

An endless loop waits for 2s and then prints a debug line with the system time.

Date: `2010-10-07`
GIT Version Tested: `f76cc0fa5ff32eacd3b2eb4dcdc2164a8ff6b191`
Runtime of Test: >5min
Preemptions/Second: 100
Error Mode Entered: no
Watchdog Fired: no
Observation: Every 2s the system time is printed.

4.2.18 Test: core/utctest

A thread initializes the system to 2008-11-12 17:35:43.21 and then starts a `TIME_LOOP` with 2s delay. Every loop cycle calls a function to convert the local system time to a readable format and print it to the screen.

Date: `2010-10-07`
GIT Version Tested: `f76cc0fa5ff32eacd3b2eb4dcdc2164a8ff6b191`
Runtime of Test: >5min
Preemptions/Second: 100
Error Mode Entered: no
Watchdog Fired: no
Observation: Every 2s the system time is printed.

4.2.19 Test: core/yieldtime

3 threads run an endless loop each and print a debug line after doing

100000 loop cycles. Each loop cycle increments a yield counter after calling Thread::yield().

Date: `2010-10-07`

GIT Version Tested: `f76cc0fa5ff32eacd3b2eb4dcdc2164a8ff6b191`

Runtime of Test: >5min

Preemptions/Second: 100

Error Mode Entered: no

Watchdog Fired: no

Observation: Every now and then 3 debug lines can be observed reporting the system time, their average time/yield and their thread name.

4.2.20 Test: core/yieldtime_extra

20 simple threads are started that only run in loops and call Thread::yield(). Every 100.000 cycles they print a simple debug line.

Date: `2010-10-07`

GIT Version Tested: `f76cc0fa5ff32eacd3b2eb4dcdc2164a8ff6b191`

Runtime of Test: >15min

Preemptions/Second: 100

Error Mode Entered: no

Watchdog Fired: no

Observation: Bulkwise appearance of debug lines of the threads can be observed since they all get started at once they finish their 100.000 cycles almost at the same time. The longer the test runs the more equally the debug messages are being distributed over time.

4.2.21 Test: core/semaphore_worm

Several threads print coordinates to a field on the screen. Each thread randomly repositions itself in the field. The field has a virtual wall (marked by a vertical line of ":"). A sempahore mechanism ensures that only one thread can cross that wall into the right side of the field.

Date: `2010-10-07`

GIT Version Tested: `f76cc0fa5ff32eacd3b2eb4dcdc2164a8ff6b191`

Runtime of Test: >5min

Preemptions/Second: 100

Error Mode Entered: no

Watchdog Fired: no

Observation: The letters of each thread can be observed to move around on the screen. At any time there is maximally one thread on the right side of the wall.