Mock Turtle Documentation

Release 4.0.0

gateware: Tomasz Wlostowski <Tomasz.Wlostowski@cern.ch>,
software: Federico Vaga <federico.vaga@cern.ch>

Jul 03, 2018

CONTENTS:

CHAPTER

ONE

INTRODUCTION

The Mock Turtle is a framework to develop embedded systems on FPGA.

The need for this framework comes from the fact that in some contexts the development of a gateware core is more complex than writing a software application. Software takes more computation time than a custom-designed gateware core; but on the other hand, the development and support efforts are significantly reduced. Mock Turtle is a solution for such problem. The gateware core complexity is moved to the software domain within the Mock Turtle boundaries, without sacrificing determinism.

The Mock Turtle framework provides an infrastructure on which you can build an FPGA-based embedded system. The basic ingredient of this framework is a soft-cpu multi-core environment that can be used to write firmware to control/monitor gateware cores. In other words, you can connect Mock Turtle to your gateware cores and control them with the firmware running on the soft-cpu. In addition, the Mock Turtle framework provides a communication channel between the firmware and the host applications which can be used to configure or control the firmware.

The Mock Turtle framework focuses mainly on the determinism of the firmware running in it. Indeed, Mock Turtle does not support any kind of interrupt or scheduling which might compromise the determinism.

The Mock Turtle framework includes the following components:

- Gateware
  - The Mock Turtle core
    - Shared Memory among soft-CPUs and host system
    - Up to 8 soft-CPUs
    - communication with the host system (input, output)
    - communication with remote systems (input, output)
- Software
  - the Mock Turtle firmware library to access gateware cores from the firmware
  - the Mock Turtle firmware framework to develop firmware

If the Mock Turtle is used within a Linux host system, the user can take advantage of a number of software components which run on the host and support Mock Turtle:

- the Mock Turtle Linux device driver
- the Mock Turtle library that provides uniform access to the driver
- the Mock Turtle Python module to access the library using Python

1.1 Use Cases

The focus on the high determinism of the soft-cpus makes the Mock Turtle a very good candidate to implement rtc systems and control systems (which are, typically, rtc applications).

To make clear the system architecture differences let's take the classical control system from the control theory: the heating system. We have a thermometer sensor, and an actuator to adjust the temperature.

The following figures show the control system architecture with and without Mock Turtle.

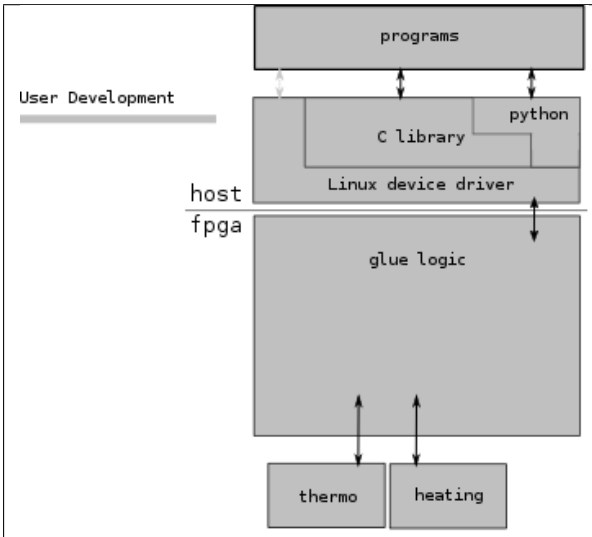

Fig. 1: Control system **without** Mock-Turtle

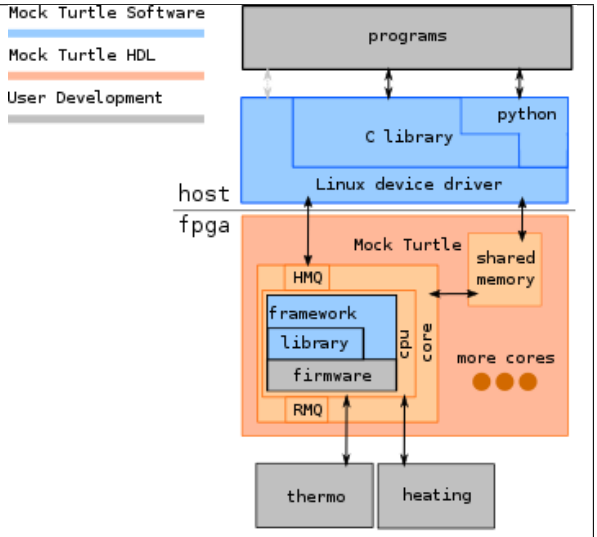

Fig. 2: Control system **with** Mock-Turtle

In the scenario without Mock-Turtle in figure the user is responsible for the entire development. On the other hands, with Mock Turtle, the user will be responsible of the development only of your business logic. This will limit your gateware development to the essential blocks, and move the control logic to the software domain. You do not have to care about the communication with the host system or the external world because it's already part of the Mock Turtle framework.

1.1.1 When Do Not Consider Mock Turtle

The Mock Turtle soft-CPUs have limited computation power, this precludes some applications like: dsp.

If you really want to use Mock Turtle for dsp analysis, please consider the development of a dedicated gateware core to perform the dsp analysis and to use the Mock Turtle as a control system for the dsp gateware core.

CHAPTER

TWO

THE MOCK TURTLE ARCHITECTURE

The Mock Turtle is a framework to develop embedded systems on FPGA. This framework offers a complete and integrated stack from the HDL core to the software application.

The following figure shows an overview over the Mock Turtle architecture.

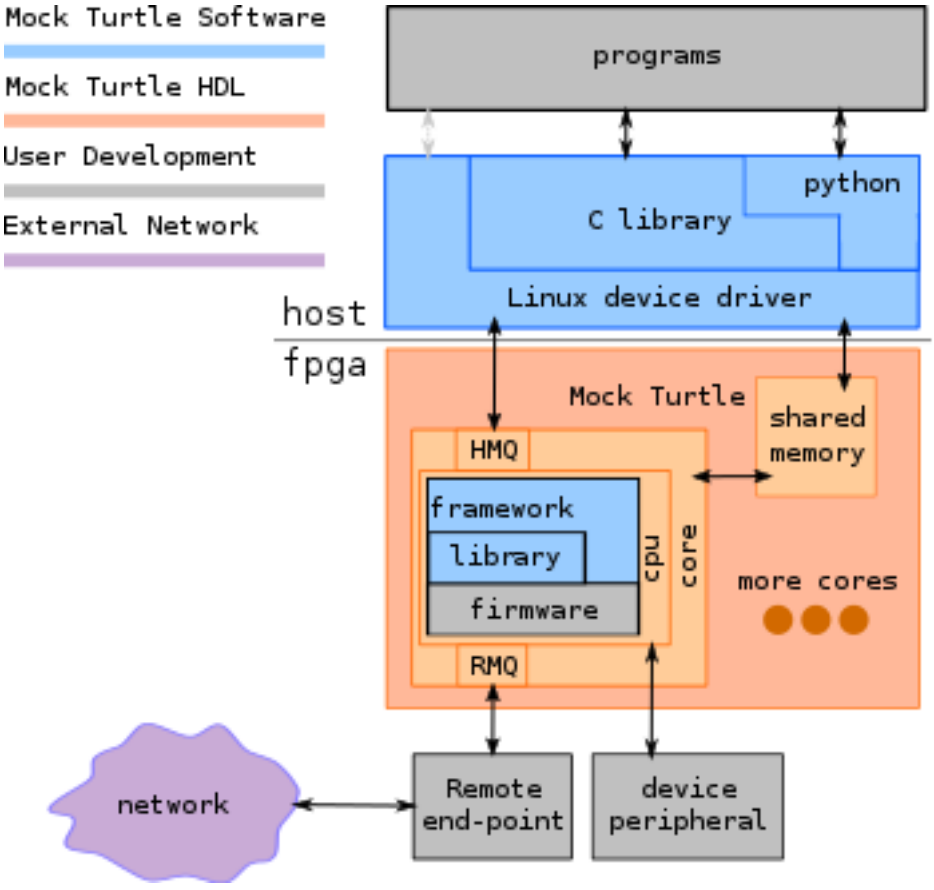

Fig. 1: Mock Turtle Architecture Overview.

Mock Turtle Architecture Overview. The blue and orange blocks are Mock Turtle components (respectively software and gateware cores). Gray blocks are external components (gateware cores or software) developed by the user. In purple any external world communication over the network.

This document tries to provide a big overview of what Mock Turtle offers and what is important to know when designing a Mock Turtle application.

For more information read the dedicated chapts for the different parts

2.1 Mock Turtle Core

Mock Turtle can have one or more core. A single cor eis made of the following components: soft-CPU, Serial console and message queues.

2.1.1 Soft-CPU

Mock Turtle uses the uRV processor, a RISC-V ISA implementation. This is just a CPU without any sort of integrated peripheral. This is where the firmware runs. Any kind of bus controller, or device must be connected externally as a *device peripheral* and driven from the firmware.

Memory size for code and data is *configurable at synthesis time*

For more information about how to handle cores from software read:

- *Linux Library - Cores Management*

2.1.2 Serial Console

Each core has a serial console connected to the host system. This link is unidirectional from core to host. Whenever there is a pending character in a serial buffer, Mock Turtle raises an interrupt for the host.

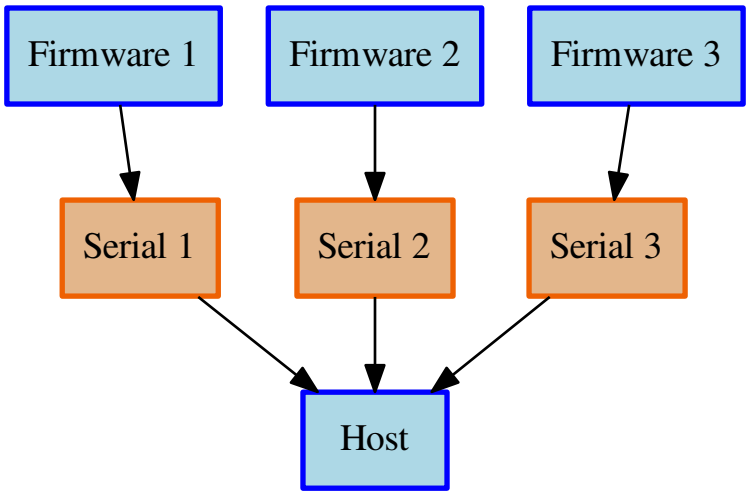

Fig. 2: Example of Mock Turtle Serial Connection.

This is used to send string messages from the running firmware to the host system:

- *Firmware Library - Serial Interface*

2.1.3 Message Queue

Mock Turtle firmwares can communicate with external agents using *message queues*; as the name suggests, this is a message queue with FIFO priority. Each soft-CPU has two sets of private message queues: one set is for host communication (*host message queue*), the other one is for external communication (*remote message queue*). Each message queue is bidirectional (one queue per direction).

A message queue entry is split in two fixed size buffers: header and payload. Header should be used to store a protocol header, while payload should be used to store the message content to be exchanged.

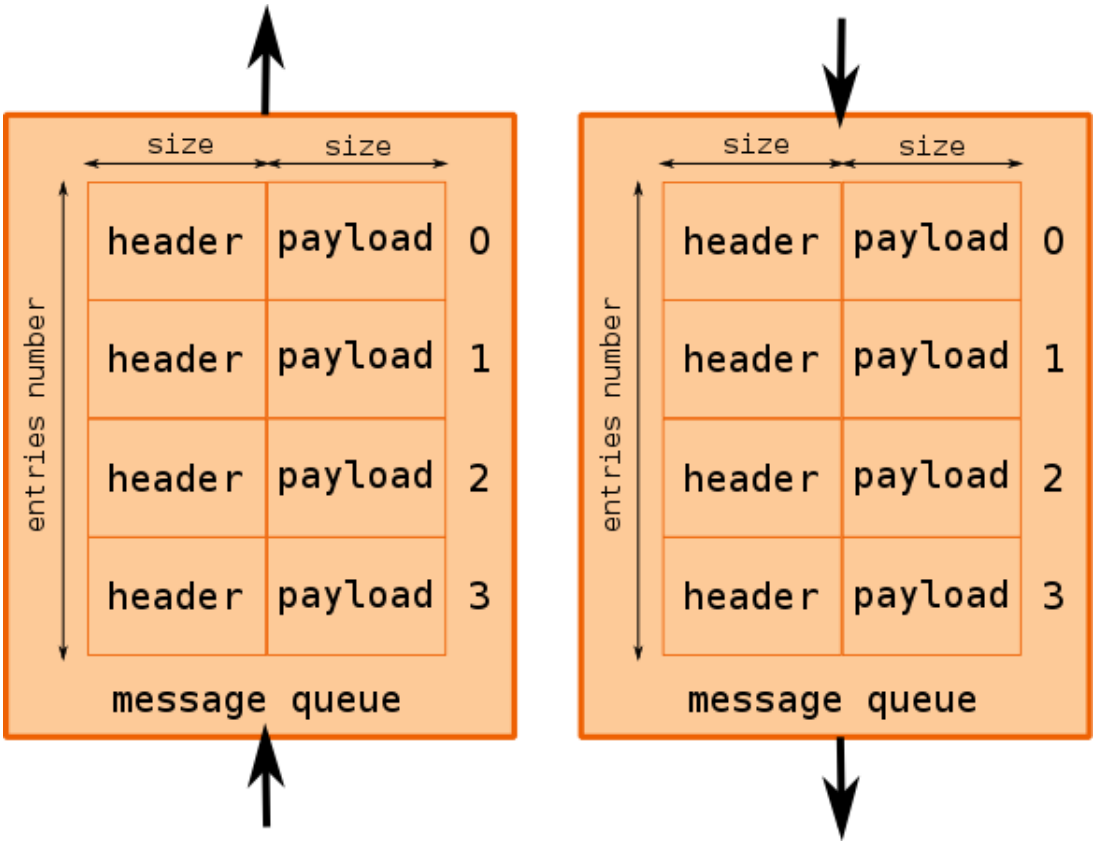

Fig. 3: Mock Turtle Message Queue Overview.

Note: Header and payload are conventions, nothing prevents users from using them otherwise. But remember that this convention is used in the Mock Turtle software APIs as described above.

All Message queue dimensions are fixed and *configured at synthesis time*. These dimensions apply to both input and output queues:

- maximum entries number
- maximum header size
- maximum payload size

Host message queues are connected to the host system. The host receives an interrupt whenever an input queue contains at least one message; while for output it receives an interrupt when a queue has at least one free entry.

Remote message queues do not have interrupt. They must be connected to an *end-point* that provides connection to the external world. Their task is to pack and unpack messages according to the type of network on which are connected. End-point implementation is application specific and outside the Mock Turtle scope.

For more information about how to access it from software read:

- *Firmware Library - Message Queue*
- *Linux Library - Message Queue*

2.2 Shared Memory

Mock Turtle offers a *shared memory* block accessible from the host system as well as from soft-CPU cores. This can be used to share data among all actors. Access to the shared memory is serialized, this means that an intensive use of it can affect the determinism.

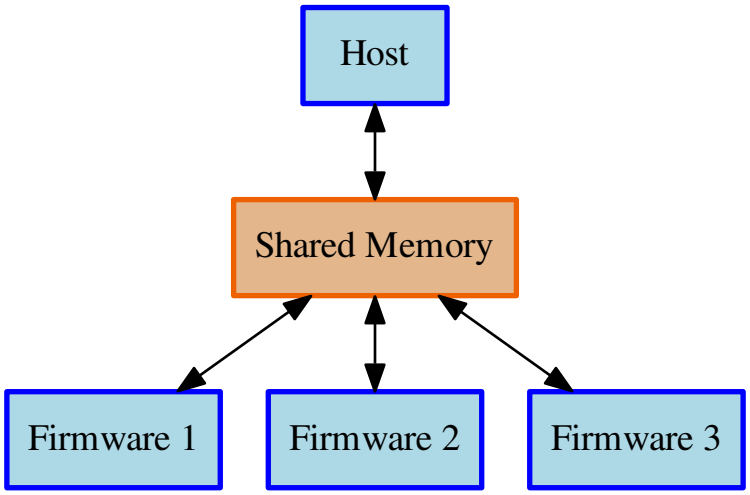

Fig. 4: Mock Turtle Shared Memory.

The shared memory size can be *configured at synthesis time* but it cannot exceed 64KiB. Its address space is mirrored into multiple address ranges (contiguous), each responsible for a single atomic operation; for more information about how to access it from software read:

- *Firmware Library - Shared Memory*
- *Linux Library - Shared Memory*

2.3 Device Peripheral

Device peripherals are external components connected to a Mock Turtle core. What they do, how many they are and how they are connected is application specific: Mock Turtle just offers connections to cores. From the core on which the device is connected, the firmware must be able to address the device.

For more information about how to access it from software read:

- *Firmware Library - Memory Location*

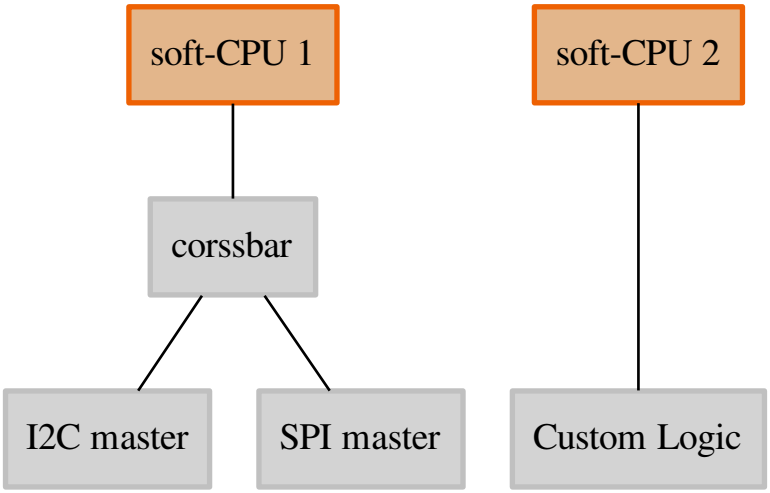

Fig. 5: Example of Mock Turtle Device Peripheral Connection.

CHAPTER

THREE

THE HDL CORE

Contents

The VHDL top-level entity of Mock Turtle (MT), to be included in any HDL design that uses it, can be found under *hdl/rtl/mock_turtle_core.vhd*.

Complete examples of MT instantation are available in the *demo projects*.

3.1 HDL Dependencies

MT depends heavily on OHWR general-cores. Furthermore, if *White Rabbit support* is enabled, MT will also require the White Rabbit PTP Core. Last but not least, each one of the soft-CPUs inside the MT is an instance of a uRV processor.

As an absolute minimum, for *Mock Turtle Configuration* and *Mock Turtle Instantiation*, users will need to use the following packages in their top-level design file:

```
-- from General Cores
use work.gencores_pkg.all;
use work.wishbone_pkg.all;
-- Local MT packages
use work.mock_turtle_pkg.all;
use work.mt_mqueue_pkg.all;
```

Users will probably add other application-specific packages to the above list.

For *demo projects*, where HDL is involved, dependencies are handled by Hdlmake, through the use of Git submodules, located under *hdl/ip_cores/*. To view a list of currently used submodules, along with their versions (Git

SHA hashes), issue from anywhere in the MT folder structure the command: `git submodule`.

The output of this command should look something like this:

```
8e7e01ef56a4af08f29fbe86f0166edd30ab903d hdl/ip_cores/general-cores (wrpc-v4.2-32-
↪g8e7e01e)
e7cd73db41ba056ed4b27731c21a3b2aa53eaa51 hdl/ip_cores/gn4124-core (v2.0-21-
↪ge7cd73d)
134759b20e8fa5241d3a3424393c6fbdfb66c6df hdl/ip_cores/urv-core (v0.9-32-g134759b)
a120e2262e1cb23fa611dddb7fa3727b520a125c hdl/ip_cores/vme64x-core (v2.0-6-ga120e22)
d4b42139d3cf88ebbc3bb78eb718db9f5dcce305 hdl/ip_cores/wr-cores (wrpc-v4.2-2-
↪gd4b4213)
```

Important: Even if you do not intend to follow the same approach in your project, you should use the `git submodule` command above to get a list of working/compatible versions for the various HDL dependencies.

3.2 Mock Turtle Configuration

As seen in *Mock Turtle Instantiation*, an MT instance expects a VHDL record with the complete configuration of the MT. Configurable aspects of the MT include:

- An ID for the instance/application
- Number of soft CPU cores
- Size of shared and per-CPU memories
- Number and dimensions of host and remote message queues per CPU

Details about the meaning of the fields of this VHDL record can also be found in *The Mock Turtle Architecture*.

All MT configuration values are also accessible at run-time from the configuration ROM of the MT.

3.2.1 VHDL Configuration Record

This VHDL record is of type **t_mt_config**, and it is defined in the **mock_turtle_pkg**:

```
type t_mt_config is record
  app_id          : std_logic_vector(31 downto 0);
  cpu_count       : natural range 1 to 8;
  cpu_config      : t_mt_cpu_config_array;
  -- shared memory size, in words
  shared_mem_size : integer range 256 to 65536;
end record t_mt_config;
```

As described in the comments, shared memory size is in 4-byte words. The *cpu_config* field contains information about each CPU core, is of type **t_mt_cpu_config_array** and is defined as:

```
subtype t_maxcpu_range is natural range 0 to 7;

type t_mt_cpu_config_array is array(t_maxcpu_range) of t_mt_cpu_config;
```

It is essentialy an array of 8 **t_mt_cpu_config**, which in turn is defined as:

```
type t_mt_cpu_config is record
  -- CPU memory sizes, in words
  memsize    : natural;
  -- Per CPU message queue config.
  hmq_config : t_mt_mqueue_config;
```

(continues on next page)

(continued from previous page)

```
  rmq_config : t_mt_mqueue_config;
end record t_mt_cpu_config;
```

Per-CPU memory size is in 4-byte words. Yet another VHDL record, of type **t_mt_mqueue_config** (defined in **mt_mqueue_pkg**), is used for the configuration of the host and remote message queues per CPU:

```
type t_mt_mqueue_config is record
  --  IN and OUT slots are always peered.
  slot_count  : integer;
  slot_config : t_mt_mqueue_slot_config_array;
end record t_mt_mqueue_config;
```

In this context, a *slot* is a bidirectional queue. Therefore, *slot_count* is the number of queues for that CPU. *slot_config* is of type **t_mt_mqueue_slot_config_array**, which is defined as:

```
subtype t_maxslot_range is natural range 0 to 7;

type t_mt_mqueue_slot_config_array is
  array(t_maxslot_range) of t_mt_mqueue_slot_config;
```

It is essentialy an array of 8 **t_mt_mqueue_slot_config**, which in turn is defined as:

```
type t_mt_mqueue_slot_config is record
  entries_bits : natural;
  width_bits   : natural;
  header_bits  : positive;
  endpoint_id  : std_logic_vector(31 downto 0);
end record t_mt_mqueue_slot_config;
```

The meaning of these fields is explained in *The Mock Turtle Architecture*. Do note that *entries_bits*, *width_bits* and *header_bits* express number of bits; if *entries_bits* is set to 7, there will be 2^7 = 128 entries, if *width_bits* is set to 2 the width of each entry will be 2^2 = 4, and so on.

3.2.2 Default Configuration

If a **t_mt_config** is not provided, a default one will be used:

```
constant c_DEFAULT_MT_CONFIG : t_mt_config :=
(
  app_id          => x"115790de",
  cpu_count       => 2,
  cpu_config      => (others => (8192,
                                 c_MT_DEFAULT_MQUEUE_CONFIG,
                                 c_MT_DEFAULT_MQUEUE_CONFIG)),
  shared_mem_size => 2048
  );
```

The default configuration instantiates two soft CPUs, each with 32KB of memory, plus 8KB of shared memory. Each CPU will have default host and remote message queue configurations defined as:

```
constant c_MT_DEFAULT_MQUEUE_CONFIG : t_mt_mqueue_config :=
  (1, ((7, 3, 2, x"0000_0000"), others => (c_DUMMY_MT_MQUEUE_SLOT)));
```

The default message queue configuration declares one queue, with 7 *entries_bits*, 3 *width_bits*, 2 *header_bits* and an *endpoint_id* of 0. All other queues will be disabled, as defined in:

```
constant c_DUMMY_MT_MQUEUE_CONFIG : t_mt_mqueue_config := (
  slot_count  => 0,
  slot_config => (others => (c_DUMMY_MT_MQUEUE_SLOT)));
```

(continues on next page)

(continued from previous page)

```
constant c_DUMMY_MT_MQUEUE_SLOT : t_mt_mqueue_slot_config :=
  (0, 0, 1, x"0000_0000");
```

Hint: Users are advised to also make use of the above *dummy* constants when describing disabled queues in their MT configurations.

See also the *demo projects* for actual examples of MT configuration.

3.3 Mock Turtle Instantiation

A VHDL component declaration for the MT top-level entity is included in the **mock_turtle_pkg**:

```
component mock_turtle_core is
  generic (
    g_CONFIG             : t_mt_config := c_DEFAULT_MT_CONFIG;
    g_SYSTEM_CLOCK_FREQ : integer      := 62500000;
    g_WITH_WHITE_RABBIT : boolean      := FALSE);
  port (
    clk_i            : in  std_logic;
    rst_n_i          : in  std_logic;
    sp_master_o      : out t_wishbone_master_out;
    sp_master_i      : in  t_wishbone_master_in := c_DUMMY_WB_MASTER_IN;
    dp_master_o      : out t_wishbone_master_out_array(0 to g_CONFIG.cpu_count-1);
    dp_master_i      : in  t_wishbone_master_in_array(0 to g_CONFIG.cpu_count-1) :=␣
↪(others => c_DUMMY_WB_MASTER_IN);
    host_slave_i     : in  t_wishbone_slave_in;
    host_slave_o     : out t_wishbone_slave_out;
    rmq_src_o        : out t_mt_stream_source_out_array2d;
    rmq_src_i        : in  t_mt_stream_source_in_array2d := (others => (others => c_
↪MT_DUMMY_SOURCE_IN));
    rmq_snk_o        : out t_mt_stream_sink_out_array2d;
    rmq_snk_i        : in  t_mt_stream_sink_in_array2d := (others => (others => c_
↪MT_DUMMY_SINK_IN));
    clk_ref_i        : in  std_logic := '0';
    tm_i             : in  t_mt_timing_if := c_DUMMY_MT_TIMING;
    gpio_o           : out std_logic_vector(31 downto 0);
    gpio_i           : in  std_logic_vector(31 downto 0) := (others => '0');
    hmq_in_irq_o     : out std_logic;
    hmq_out_irq_o    : out std_logic;
    notify_irq_o     : out std_logic;
    console_irq_o    : out std_logic);
end component mock_turtle_core;
```

All generics and all input ports (except **clk_i** and **rst_n_i**) have default values if left unconnected.

3.3.1 Generics

g_CONFIG VHDL record of type **t_mt_config**, as described in *Mock Turtle Configuration*.

g_SYSTEM_CLOCK_FREQ Frequency of system clock (provided on the *clk_i* input port). This is used internally for keeping track of time when *White Rabbit Support* is not enabled and it is also used by software to calculate delays.

g_WITH_WHITE_RABBIT Controls enabling of *White Rabbit Support*.

3.3.2 Ports

clk_i is the system clock.

rst_n_i is an active-low reset input, which is expected to be syncrhonous to the system clock.

sp_master_i, sp_master_o they form the *Shared Peripheral (SP)* Wishbone bus. Wishbone peripherals attached to these ports will be accessible by all MT soft-CPUs.

dp_master_i, dp_master_o they form the *Dedicated Peripheral (DP)* Wishbone buses. There is one DP Wishbone bus per soft CPU configured. Wishbone peripherals attached to these ports will be accessible only by their respective soft-CPU.

host_slave_i, host_slave_o they provide a Wishbone slave interface, to be attached to a controlling *host*, typically through some sort of bridge (eg. PCI to Wishbone). This interface provides access to the following Wishbone peripherals inside the MT:

- Control/Status registers
- Shared memory
- per-CPU host message queues
- Configuration ROM

Important: When mapping the MT address area to the host, users should keep in mind that the whole MT address space is 128KB (0x20000).

---

t_wishbone_master_out, **t_wishbone_master_in** and their respective arrays, as well as **t_wishbone_slave_out** and **t_wishbone_slave_in** are VHDL record types declared in the ***wishbone_pkg** of OHWR general-cores, along with their default/dummy constants:

```
type t_wishbone_master_out is record
  cyc : std_logic;
  stb : std_logic;
  adr : t_wishbone_address;
  sel : t_wishbone_byte_select;
  we  : std_logic;
  dat : t_wishbone_data;
end record t_wishbone_master_out;

subtype t_wishbone_slave_in is t_wishbone_master_out;

type t_wishbone_slave_out is record
  ack   : std_logic;
  err   : std_logic;
  rty   : std_logic;
  stall : std_logic;
  dat   : t_wishbone_data;
end record t_wishbone_slave_out;

subtype t_wishbone_master_in is t_wishbone_slave_out;

constant c_DUMMY_WB_ADDR : std_logic_vector(c_WISHBONE_ADDRESS_WIDTH-1 downto 0) :=
  (others => 'X');
constant c_DUMMY_WB_DATA : std_logic_vector(c_WISHBONE_DATA_WIDTH-1 downto 0) :=
  (others => 'X');
constant c_DUMMY_WB_SEL : std_logic_vector(c_WISHBONE_DATA_WIDTH/8-1 downto 0) :=
  (others => 'X');
constant c_DUMMY_WB_SLAVE_IN   : t_wishbone_slave_in :=
  ('0', '0', c_DUMMY_WB_ADDR, c_DUMMY_WB_SEL, 'X', c_DUMMY_WB_DATA);
constant c_DUMMY_WB_MASTER_OUT : t_wishbone_master_out := c_DUMMY_WB_SLAVE_IN;
constant c_DUMMY_WB_SLAVE_OUT  : t_wishbone_slave_out :=
```

(continues on next page)

(continued from previous page)

```
  ('1', '0', '0', '0', c_DUMMY_WB_DATA);
constant c_DUMMY_WB_MASTER_IN  : t_wishbone_master_in := c_DUMMY_WB_SLAVE_OUT;
```

rmq_src_o, rmq_src_i, rmq_snk_o, rmq_snk_i These ports provide the bidirectional interface from each of the configured soft CPUs to their repspective remote message queue(s).

t_mt_stream_source_out_array2d, **t_mt_stream_source_in_array2d**, **t_mt_stream_sink_out_array2d** and **t_mt_stream_sink_in_array2d** are two-dimensional arrays of VHDL records, defined in **mock_turtle_pkg** (for the first array dimension) and **mt_mqueue_pkg** (for the second array dimension and for the records):

```
subtype t_maxcpu_range is natural range 0 to 7;
subtype t_maxslot_range is natural range 0 to 7;

-- defined in mock_turtle_pkg
type t_mt_stream_sink_in_array2d is
  array(t_maxcpu_range) of t_mt_stream_sink_in_array(t_maxslot_range);
type t_mt_stream_sink_out_array2d is
  array(t_maxcpu_range) of t_mt_stream_sink_out_array(t_maxslot_range);

subtype t_mt_stream_source_in_array2d is t_mt_stream_sink_out_array2d;
subtype t_mt_stream_source_out_array2d is t_mt_stream_sink_in_array2d;

-- defined in mt_mqueue_pkg
type t_mt_stream_sink_in_array is array(integer range<>) of t_mt_stream_sink_in;
type t_mt_stream_sink_out_array is array(integer range<>) of t_mt_stream_sink_out;

subtype t_mt_stream_source_in_array is t_mt_stream_sink_out_array;
subtype t_mt_stream_source_out_array is t_mt_stream_sink_in_array;

type t_mt_stream_sink_in is record
  data  : std_logic_vector(31 downto 0);
  hdr   : std_logic;
  valid : std_logic;
  last  : std_logic;
  error : std_logic;
end record t_mt_stream_sink_in;

type t_mt_stream_sink_out is record
  ready     : std_logic;
  pkt_ready : std_logic;
end record t_mt_stream_sink_out;

subtype t_mt_stream_source_in is t_mt_stream_sink_out;
subtype t_mt_stream_source_out is t_mt_stream_sink_in;

constant c_MT_DUMMY_SOURCE_IN : t_mt_stream_sink_out :=
  ('0', '0');
constant c_MT_DUMMY_SINK_IN : t_mt_stream_sink_in :=
  (x"00000000", '0', '0', '0', '0');
```

Todo: Describe how the RMQ interface works and update after Tom's endpoint work if necessary.

clk_ref_i When *White Rabbit Support* is enabled (via the **g_WITH_WHITE_RABBIT** generic), the White Rabbit 125MHz reference clock should be connected here.

tm_i All other timing signals from the White Rabbit PTP Core go to the **tm_i** input.

t_mt_timing_if is a VHDL record type, defined in **mock_turtle_pkg**, together with its default/dummy constant as:

```
type t_mt_timing_if is record
  link_up    : std_logic;
  dac_value  : std_logic_vector(23 downto 0);
  dac_wr     : std_logic;
  time_valid : std_logic;
  tai        : std_logic_vector(39 downto 0);
  cycles     : std_logic_vector(27 downto 0);
  aux_locked : std_logic_vector(7 downto 0);
end record t_mt_timing_if;

constant c_DUMMY_MT_TIMING : t_mt_timing_if := (
  link_up    => '0',
  dac_value  => (others => '0'),
  dac_wr     => '0',
  time_valid => '0',
  tai        => (others => '0'),
  cycles     => (others => '0'),
  aux_locked => (others => '0'));
```

For an explanation of these fields, please refer to White Rabbit PTP core manual.

gpio_o, gpio_i These are the 32-bit inputs and outputs to the GPIO registers of each configured MT soft CPU. Inputs are delivered to all soft-CPUs while each bit of the output is a logic-OR of the respective GPIO output bit of each CPU.

hmq_in_irq_o Interrupt output to signal that one of the incoming host message queues is not empty.

hmq_out_irq_o Interrupt output to signal that one of the outgoing host message queues is not full.

notify_irq_o Interrupt output to signal that one of the soft-CPUs needs to notify the host.

console_irq_o Interrupt output to signal that one of the soft CPUs has pending data in its console output.

See also the *demo projects* for actual examples of MT instantiation.

3.4 White Rabbit Support

Support for White Rabbit (WR) is controlled via the *g_WITH_WHITE_RABBIT* generic (see *Generics*).

When WR is enabled, MT expects a reference clock on input port *clk_ref_i* and a *t_mt_timing_if* record on input port *tm_i* (see *Ports*).

Once WR is enabled and the WR link is up and running, each CPU core will have access to WR time via the *TAI cycles* and *TAI seconds* registers in *Local Registers (LR)*.

3.5 Simulation Testbenches

Several simulation testbenches are available under *hdl/testbench*. They all make use of the Mock Turtle SystemVerilog simulation environment, which can be found under *hdl/testbench/include*.

All available testbenches are built using Hdlmake and Modelsim/Questa.

Note: Building and running of the testbenches has been verified with Modelsim 10.2a and Questa 10.5c.

Important: Due to bugs present in Hdlmake release v3.0, it is necessary to use the *develop* branch of hdlmake, commit db4e1ab (or later).

Once the necessary tools are installed, building a particular testbench is simply a question of running:

```
$ cd <testbench_folder>
$ hdlmake
$ make
```

This will compile all the sources.

Running the testbench is simply done with:

```
$ cd <testbench_folder>
$ vsim -c -do run_ci.do
```

Alternatively, *vsim* can be launched interactively. In that case, users can launch the *run.do* file:

```
simulator_prompt> do run.do
```

This will run the simulation and log *all* signals. When the execution is done, users can inspect the signals, store them for future reference, display them as waveforms, etc.

3.5.1 mock_turtle_core

The mock_turtle_core uses the top-level module of the MT as the Device Under Test (DUT). It loads and executes a dedicated simulation verification program on the first CPU.

This program tests the following subsystems of the MT:

- UART messages
- Notification interrupts
- Host message queues
- Remote message queues

The expected output from the simulation is:

```
App ID: 0x115790de
Core count: 2
UART MSG from core 0: #1 console
UART MSG from core 0: #2 notify irq
UART MSG from core 0: #3 hmq
UART MSG from core 0: #4 rmq
UART MSG from core 0: Done
```

Note: The mock_turtle_core testbench expects an already compiled software binary under *tests/firmware/sim-verif*. Please compile the software prior to running the simulation.

CHAPTER

FOUR

THE SOFTWARE

This section explains the Mock Turtle software architecture as well as the necessary steps to develop software layers on top of the Mock Turtle ones.

The software integration discussion will assume that you have a general understanding of *The Mock Turtle Architecture*

The Mock Turtle software stack consists in two main development domains: the *Firmware Development* and the *Linux Development* (libraries or applications).

The Mock Turtle software stack is made of different layers which main objectives are:

- to manage the Mock Turtle cores from the host;
- to allow firmwares to access Mock Turtle resources;
- to provide a communication infrastructure between firmware and host
- to provide a communication infrastructure with remote nodes

The development of a firmware is necessary to make the system work. In the section *Firmware Development* you will learn how to write a firmware using Mock Turtle API.

On the other hand, the development of a software support layer on the host depends on your needs. If you need a custumized control/monitor infrastructure for firmwares, then it is recommended to develop your software support layer(s) on top of the Mock Turtle ones. Keep in mind that *The Mock Turtle Tools* can be used for basic control/monitor operations. This means that for basic requirements you can directly use the tools without developing any support layer.

We strongly recommend you to start the develpment of a new Mock Turtle project by using the *Mock Turtle Project Creator*.

Following a list of generic topics which are not specific to Linux development of firmware development.

4.1 Common Data Structures

4.1.1 Configuration ROM

The *configuration ROM* is, indeed, a ROM where at synthesis time we put information about the synthesis configuration. This configuration is the one used to tailor Mock Turtle to fit users needs. The configuration can be read, with different APIs, by both host system and firmwares.

struct trtl_config_rom
The synthesis configuration ROM descriptor shows useful configuration options during synthesis.

Public Members

uint32_t *trtl_config_rom*__signature__
we expect to see a known value

uint32_t *trtl_config_rom*.**version**
Mock Turtle version

uint32_t *trtl_config_rom*.**clock_freq**
clock frequency in Hz

uint32_t *trtl_config_rom*.**flags**
miscellaneous flags

uint32_t *trtl_config_rom*.**app_id**
Application ID

uint32_t *trtl_config_rom*.**n_cpu**
number of CPU

uint32_t *trtl_config_rom*.**smem_size**
shared memory size

uint32_t *trtl_config_rom*.**mem_size**[**TRTL_MAX_CPU**]
memory size for each CPU

uint32_t *trtl_config_rom*.**n_hmq**[**TRTL_MAX_CPU**]
number of HMQ for each CPU

uint32_t *trtl_config_rom*.**n_rmq**[**TRTL_MAX_CPU**]
number of RMQ for each CPU

struct *trtl_config_rom_mq* *trtl_config_rom*.**hmq**[**TRTL_MAX_CPU**][**TRTL_MAX_MQ_CHAN**]
HMQ config

struct *trtl_config_rom_mq* *trtl_config_rom*.**rmq**[**TRTL_MAX_CPU**][**TRTL_MAX_MQ_CHAN**]
RMQ config

struct trtl_config_rom_mq
The synthesis configuration for a single MQ. Note that there is always an input and output channel for each declaration.

Public Members

uint32_t *trtl_config_rom_mq*.**sizes**
it contains the MQ sizes. Use the MACROs to extract them

TRTL_CONFIG_ROM_MQ_SIZE_ENTRIES(_size)
It extracts the number of message queue entries from the sizes value in the configuration ROM

TRTL_CONFIG_ROM_MQ_SIZE_PAYLOAD(_size)
It extracts the maximum payload size (32bit words) from the sizes value in the configuration ROM

TRTL_CONFIG_ROM_MQ_SIZE_HEADER(_size)
It extracts the maximum header size (32bit words) from the sizes value in in the configuration ROM

4.1.2 Host Message Queue Protocol

When exchanging messages between two entities it is always handy to have a protocol. Any Mock Turtle message queue has an header part and a payload part. It is within the header part that users put the information to handle the choosen protocol.

In order to standardize the message exchange between host and firmwares a message header has been defined. This header is expected to be in the message queue header buffer.

Different Mock Turtle layers make a different use of this message header. The HDL code does not process the header, while the driver uses the `trtl_hmq_header.len` to optimize the amount of data copied.

This protocol is mostly used by libraries and firmwares, which are the two end-points of Host Message Queue communication channel.

struct trtl_hmq_header
HMQ header descriptor. It helps the various software layers to process messages.

Public Members

uint16_t *trtl_hmq_header*__rt_app_id__
firmware application unique identifier. Used to validate a message against the firmware that receives it

uint8_t *trtl_hmq_header*__flags__
flags

uint8_t *trtl_hmq_header*__msg_id__
It uniquely identify the message type. The first __TRTL_MSG_ID_MAX_USER are free use

uint16_t *trtl_hmq_header*__len__
message-length in 32bit words

uint16_t *trtl_hmq_header*__sync_id__
synchronous identifier

uint32_t *trtl_hmq_header*__seq__
sequence number (automatically set by the library?)

TRTL_HMQ_HEADER_FLAG_SYNC
Synchronous. When set, the sync_id is valid and the receiver must answer with a message with the same sync_id

TRTL_HMQ_HEADER_FLAG_ACK
Acknowledgment. When set, the sync_id is valid and it is the same as the sync message previously sent

TRTL_HMQ_HEADER_FLAG_RPC
Remote Procedure Call. when set the message ID will be used to execute a special function by the receiver

---

Todo: Should the framework/library handle sync_id and seq?

Remote Message Queue Protocol

Within this project we have defined a protocol for the communication with the host system. How to handle the communication with remote nodes is left to the user who can chose among dozen of existing protocols (for example UDP). Also for remote queues a message is made of header and payload; whatever is your choosen protocol, its header will lay in the header part and the payload the payload part. End-points can use the header part to configure them selves with user parameters.

4.2 Linux Development

It includes an explaination about the host development (libraries or applications) and the Mock Turtle API on a Linux host. Any future reference to host in this section will assume a Linux host because it is the only supported platform for the time being.

Mock Turtle offers 3 interfaces: a Pyhton module, a C library and a Linux kernel interface. Mock Turtle users are not supposed to use the driver interface directly.

4.2.1 The Linux Device Driver

The Mock Turtle device driver is a software component that exposes the Mock Turtle gateware core to the host. Any interaction with the Mock Turtle gateware core pass trought the device driver. This implies that if the driver does not support a Mock Turtle feature, neither the other layers (the library or the Python module) will do.

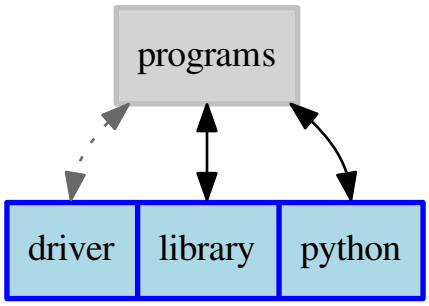

Fig. 1: Mock Turtle Linux Interfaces.

Requirements

The Mock Turtle device driver has been developed and tested on Linux 3.6. Other Linux versions might work as well but it is not guaranteed.

The FPGA address space must be visible on the host system. This requires a driver for the FPGA carrier that export the FPGA address space to the host.

Compile And Install

The Mock Turtle device driver compilation and installation requires only to execute `make`.:

```
$ cd /path/to/mockturtle/software/kernel
$ export LINUX=/path/to/linux/sources
$ make
$ make install
```

Load Driver

The Mock Turtle device driver module needs to be loaded in order to be used.:

```
$ cd /path/to/mockturtle/software/kernel
$ sudo insmod mock-turtle.ko
```

Following the list of module parameters that can be used to custumize the driver instance.

Name	Default Value	Description
hmq_buf_max_msg	16	Maximum number of messages stored by the driver for each direction

Load Gateware

Of course, a Mock Turtle instance must exist on your FPGA in order to be able to drive it. Loading the gateware bitstream depends on the FPGA carrier in use. Just keep in mind that you must load the bitstream **before** adding a Mock Turtle device instance in the Linux kernel. If you do not do so, you will get crashes because the device driver will try to access something that does not exist yet.

Load Device

The Mock Turtle device driver is based on the platform Linux subsystem[1] . This means that you need a mechanism to load a platform device that describes a Mock Turtle device. Typically, this mechanism involves the development of a Linux module or the DeviceTree.

This driver handles all platform_device instances which name is one of the following: “mock-turtle”, “mockturtle”.

The Mock Turtle device driver expects 5 `resources` from the platform device.

memory address The gateware core base address within the virtual addres space.

hmq IRQ input The Linux IRQ number to use for the hmq input.

hmq IRQ output The Linux IRQ number to use for the hmq output.

console IRQ The Linux IRQ number to use for the serial interface.

notification IRQ The Linux IRQ number to use for Mock Turtle cores notifications.

Since not all developer knows how to write such module, you can use the *platform-device-loader*[2]. This application is part of the CERN BE-CO-HT tools collection, you may need special permission to access the repository.

If you are not sure about how to write the Linux module to load your platform device, consider to have a look at the source code templates from the *platform-device-loader* tool.

Interfaces

Mock Turtle Device

The Mock Turtle driver exports a *char device* for each Mock Turtle. In */dev/mockturtle* you will have devices named *trtl-%04x* (trtl-<device-id>). This exports a set of `ioctl(2)` commands:

`TRTL_IOCTL_SMEM_IO`

You can find the cores *sysfs* attributes at:

```
/sys/class/mockturtle/trtl-%04x/
```

Name	Direction	Description
config-rom	RO	Binary data containing all the synthesis configuration
reset_mask	RW	Set or clear the soft-CPUs reset. It is a bit-mask where each bit correspond to a soft-CPU (e.g. bit 0 -> soft-CPU 0)

Mock Turtle Cores

The Mock Turtle driver exports a *char device* for each Mock Turtle core. All core instances will appear as child of a *Mock Turtle Device*; in */dev/mockturtle* you will have devices named *trtl-%04x-%02d‘‘* (trtl-<device-id>-<cpu-index>). The main purpose of this interface is to program (or rarely useful dump) firmwares into cores. These devices are bidirectional, so you can: `write(2)` to program a firmware, `read(2)` to dump a firmware; `lseek(2)` to move to different memory locations.

From the command line you can load a new program using `dd(1)` or similar tools.:

```
dd if=firmware.bin of=/dev/mockturtle/trtl-0001-00
```

The same command can be used also to dump the memory content.:

[1] https://www.kernel.org/doc/Documentation/driver-model/platform.txt
[2] https://gitlab.cern.ch/cohtdrivers/coht-tools/tree/master/drivers/platform-device-loader

```
dd if=/dev/mockturtle/trtl-0001-00 of=firmwaredump.bin
```

In both cases (loading and dumping) the driver automatically put in *reset* the core. This means that after your operation you need to *unreset* the core in order to make it running again.:

```
echo 0 > /sys/class/mockturtle/trtl-0001-00/reset
```

Mock Turtle uses the standard TTY layer from the Linux kernel. Each core has a dedicated serial interface which is used for communications from soft-CPU to host.

Linux TTY devices appear in the */dev* directory and they are named *ttytrtl-%04x-%d*.

Since it is a standard TTY interface you can use the tool you like to read it. For example:

```
minicom -D /dev/ttytrtl-0001-00
cat /dev/ttytrtl-0001-00
```

Note: The driver does not perform any data processing on the console. In other words whatever the firmware application writes is replicated on this interface.

You can find the cores *sysfs* attributes at:

```
/sys/class/mockturtle/trtl-%04x/trtl-%04x-%02d-%02d
```

Name	Direction	Description
reset	RW	It asserts (1) or de-asserts (0) the soft-CPU reset line
last_notification	RO	It shows the last notification ID received
notification_history	RO	It shows le last 128 notification IDs received

Host Message Queue

The Mock Turtle driver exports a *char device* for each Mock Turtle HMQ. All HMQ instances will appear as child of a *Mock Turtle Cores*; in */dev/mockturtle/* you will have devices named `trtl-%04x-%02d-%02d` (trtl-<device-id>-<cpu-index>-<hmq-index>). The main purposes of this interface is to exchange messages and configure filters. These devices are bidirectional, so you can: `write(2)` to send messaged; `read(2)` to receive messages; `poll(2)` or `select(2)` to wait for the interface to become accessible.

This *char device* provides a set of `ioctl(2)` commands:

TRTL_IOCTL_MSG_FILTER_ADD
: The IOCTL command to add a filter to an HMQ

TRTL_IOCTL_MSG_FILTER_CLEAN
: The IOCTL command to remove all filters from an HMQ

You can find the HMQ *sysfs* attributes at:

```
/sys/class/mockturtle/trtl-%04x/trtl-%04x-%02d-%02d
```

Name	Direction	Description
empty	RO	It shows the input and output empty status: 1 when the HMQ channel buffer is empty, 0 otherwise (hardware)
full	RO	It shows the input and output full status: 1 when the HMQ channel buffer is full, 0 otherwise (hardware)
occupied	RO	The number of entries in the queue for input and output
discard_all	WO	When written it flushes the HMQ from all pending messages
statistics/message_received	RO	Total number of messages received through the HMQ channel
statistics/message_sent	RO	Total number of messages sent through the HMQ channel

Debugging Interface

The driver exports on debugfs a file in YAML format which contains internal information about the driver: variable values, register values. This file is named as the Mock Turtle instance that represents "trtl-%04x".:

```
mount -t debugfs none /sys/kernel/debug
cat /sys/kernel/debug/trtl-0001/info
```

This is typically used by driver developers for debugging purposes.

Warning: Its content it is not stable and it may change at anytime. Do not consider this as a stable interface.

4.2.2 The Mock Turtle Linux Library

The Mock Turtle Library for host system development handles all the Mock Turtle features and it makes them available to any user. The Mock Turtle library mandate is to export all the Mock Turtle driver features to user-space programs in a more user-friendly way without paying much in terms of the flexibility that the driver offers.

The library layer covers all the driver features; for this reason, the user should use only the library or the Python module. The user can still access the Mock Turtle driver directly but it is strongly discouraged.

Installation

Requirements

The Mock Turtle library depends on:

- the standard C library and on;
- the Mock Turtle driver;

Compile And Install

The Mock Turtle library can be installed in your environment by running:

```
cd /path/to/mockturtle/software/lib
make install
```

This will install both the static library and the shared object library.

When using the shared object library, you can skip the installation and use the environment variable `LD_LIBRARY_PATH` to make the library visible to your programs/libraries:

```
$ export LD_LIBRARY_PATH=$LD_LIBRARY_PATH:/path/to/mockturtle/software/lib
```

In order to include this library in your project you must include in your source code::

```
#include <libmockturtle.h>
```

While in the compilation command you have to provide the following options (e.g. GCC)::

```
CC = /path/to/your/compiler
CFLAGS += -I/path/to/mockturtle/software/lib
CFLAGS += -I/path/to/mockturtle/software/include
CFLAGS += -L/path/to/mockturtle/software/lib
LDLIBS += -lmockturtle
$(CC) $(CFLAGS) $(LDLIBS)
```

The example above assumes that you are compiling from the Mock Turtle git repository or a copy of it; if on your environment libraries and header files are in different location, then fix the above code accordingly.

Initialization

The library initialization is done with function `trtl_init()`. You must initialize the library before calling any library function.

int **`trtl_init`**`()`
: It initializes the TRTL library. It must be called before doing anything else. If you are going to load/unload TRTL devices, then you have to un-load (*trtl_exit()*) e reload (*trtl_init()*) the library.

  Return 0 on success, otherwise -1 and errno is appropriately set

Once you are completely done with Mock Turtle you should properly close the library by calling `trtl_exit()`

void **`trtl_exit`**`()`
: It releases the resources allocated by *trtl_init()*. It must be called when you stop to use this library. Then, you cannot use functions from this library anymore.

Open And Close Devices

In order to be able to handle a Mock Turtle device you must open it with one of the following functions `trtl_open()`, `trtl_open_by_id()` or `trtl_open_by_lun()`. All these functions return a device token which is required by most Mock Turtle functions. When you do not want to use anymore the device, you should close it with `trtl_close()`.

`struct` *trtl_dev* ***`trtl_open`**`(`**`const`** char *_device_`)`
: It opens a TRTL device using a string descriptor. The descriptor correspond to the main char device name of the Mock-Turtle.

  Return the TRTL token, NULL on error and errno is appropriately set

  Parameters

  - `device`: name of the device to open

`struct` *trtl_dev* ***`trtl_open_by_id`**`(`uint32_t *device_id*`)`
: It opens a TRTL device using its device_id. The Mock-Turtle driver is based upon the platform bus infrastructure, so all trtl devices are identified with their platform id.

  Return the TRTL token, NULL on error and errno is appropriately set

  Parameters

  - `device_id`: device id of the device to use

struct *trtl_dev* ***trtl_open_by_lun** (unsigned int *lun*)

It opens a TRTL device using its Logical Unit Number. The Logical Unit Number is an instance number of a particular hardware. The LUN to use is the carrier one, and not the mezzanine one (if any). The driver is not aware of LUNs but only of device-id. So, if this function does not work it means that your installation lacks of symbolic links that convert LUNs to device-ids.

Return the TRTL token, NULL on error and errno is appropriately set

Parameters

- `lun`: Logical Unit Number of the device to use

void **trtl_close** (**struct** *trtl_dev* **trtl*)

It closes a TRTL device opened with one of the following functions: *trtl_open()*, wrcn_open_by_lun(), *trtl_open_by_id()*

Parameters

- `trtl`: device token

struct trtl_dev

It is obfuscated structure type. This becasue the user should not modify it but just use it as a token to access the library API.

Mock Turtle Cores Management

Library support for cores' management is limited to the firmware loading (dumping) and core enable/disable.

The typical use of these functions is to load an executable file into the soft-CPU. The following listing shows an example

```
void progr_cpu(struct trtl_dev *trtl, unsigned int cpu_idx, char *file_name)
{
    trtl_cpu_load_application_file(trtl, cpu_idx, file_name);
    trtl_cpu_enable(trtl, cpu_idx);
}
```

int **trtl_cpu_load_application_raw** (**struct** *trtl_dev* **trtl*, unsigned int *index*, void **code*, size_t *length*, unsigned int *offset*)

It loads a trtl CPU firmware from a given buffer The CPU must be in reset mode in order to be programmed. This is done automatically by the driver which will leave the CPU in reset mode. The user must clear the reset status in order to run the firmware.

Parameters

- `trtl`: device token
- `index`: CPU index
- `code`: buffer containing the CPU firmware binary code
- `length`: code length
- `offset`: memory offset where to start to write the code

Return the number of written byte, on error -1 and errno is set appropriately

int **trtl_cpu_dump_application_raw** (**struct** *trtl_dev* **trtl*, unsigned int *index*, void **code*, size_t *length*, unsigned int *offset*)

It dumps a TRTL CPU firmware into a given buffer For a reliable dump, the CPU must be in pause. This is done by the driver which then will set back the previous situation.

Parameters

- `trtl`: device token
- `index`: CPU index

- `code`: buffer containing the CPU firmware binary code
- `length`: code length
- `offset`: memory offset where to start to write the code

Return the number of written byte, on error -1 and errno is set appropriately

int **trtl_cpu_load_application_file**(**struct** *trtl_dev* *trtl, unsigned int *index*, char *path)
It loads a TRTL CPU firmware from a given file After extracting the data from the file, it internally uses *trtl_cpu_load_application_raw()*.

Parameters

- `trtl`: device token
- `index`: CPU index
- `path`: path to the firmware file

Return 0 on success, on error -1 and errno is set appropriately

int **trtl_cpu_dump_application_file**(**struct** *trtl_dev* *trtl, unsigned int *index*, char *path)
It dumps a TRTL CPU firmware into a given file It internally uses *trtl_cpu_dump_application_raw()*.

Parameters

- `trtl`: device token
- `index`: CPU index
- `path`: path to the firmware file

Return 0 on success, on error -1 and errno is set appropriately

int **trtl_cpu_enable**(**struct** *trtl_dev* *trtl, unsigned int *index*)
It enables a CPU; in other words, it clear the reset line of a CPU. This function is a wrapper of *trtl_cpu_reset_set()* that allow you to safely enable a single CPU.

Return 0 on success, -1 otherwise and errno is set appropriately

Parameters

- `trtl`: device token
- `index`: CPU index

int **trtl_cpu_disable**(**struct** *trtl_dev* *trtl, unsigned int *index*)
It disables a CPU; in other words, it sets the reset line of a CPU. This function is a wrapper of *trtl_cpu_reset_set()* that allow you to safely disable a single CPU.

Return 0 on success, -1 otherwise and errno is set appropriately

Parameters

- `trtl`: device token
- `index`: CPU index

int **trtl_cpu_is_enable**(**struct** *trtl_dev* *trtl, unsigned int *index*, unsigned int *enable)
It checks if the CPU is enabled (or not)

Return 0 on success, -1 otherwise and errno is set appropriately

Parameters

- `trtl`: device token
- `index`: CPU index
- `enable`: 1 if the CPU is enable

int **trtl_cpu_reset_set**(**struct** *trtl_dev* *trtl, uint32_t *mask*)
Assert or de-assert the reset line of the TRTL CPUs

Return 0 on success, -1 otherwise and errno is set appropriately

Parameters

- `trtl`: device to use
- `mask`: bit mask of the reset-lines

int **trtl_cpu_reset_get**(**struct** *trtl_dev* *trtl, uint32_t **mask*)
It returns the current status of the TRTL CPUs' reset line

Return 0 on success, -1 otherwise and errno is set appropriately

Parameters

- `trtl`: device token
- `mask`: bit mask of the reset-lines

Host Message Queue and Messages

This library has a set of functions to handle HMQs and to send/receveice messages to/from them.

Whenever you need to remove all the messages from the HMQ you can use the function `trtl_hmq_flush()`. The host system does not have access to the RMQ. If what you want to achive is a complete flush of all the message queues (host and remote) you should do it on firmware side so that the complete flush happens synchronously.

int **trtl_hmq_flush**(**struct** *trtl_dev* *trtl, unsigned int *idx_cpu*, unsigned int *idx_hmq*)
It flushes the content of an HMQ for both input and output channel

Return 0 on success, otherwise -1 and errno is set appropriately

Parameters

- `trtl`: device token
- `idx_cpu`: CPU index
- `idx_hmq`: HMQ index

The Mock Turtle driver has a basic message filtering mechanism. Each user can add a set of filter on any host message queue. Mock Turtle will deliver to the user only those messages which pass the filter. Rembember that this is user filter, this means that on the same host message queue, different users can have different filters.

int **trtl_hmq_filter_add**(**struct** *trtl_dev* *trtl, unsigned int *idx_cpu*, unsigned int *idx_hmq*, **struct** *trtl_msg_filter* **filter*)
It adds a new filter to the given hmq descriptor

Return 0 on success, -1 otherwise and errno is set appropriately

Parameters

- `trtl`: device token
- `idx_cpu`: CPU index
- `idx_hmq`: HMQ index
- `filter`: filter to add

int **trtl_hmq_filter_clean**(**struct** *trtl_dev* *trtl, unsigned int *idx_cpu*, unsigned int *idx_hmq*)
It removes all filters from the given hmq descriptor

Return 0 on success, -1 otherwise and errno is set appropriately

Parameters

- `trtl`: device token

- `idx_cpu`: CPU index
- `idx_hmq`: HMQ index

enum trtl_msg_filter_operation_type
List of available filter's operations

Values:

TRTL_MSG_FILTER_AND

TRTL_MSG_FILTER_OR

TRTL_MSG_FILTER_EQ

TRTL_MSG_FILTER_NEQ

__TRTL_MSG_FILTER_MAX

struct trtl_msg_filter
It describe a filter to apply to messages

Public Members

unsigned long *trtl_msg_filter***flags**
options for the filter

uint32_t *trtl_msg_filter***operation**
kind of operation to perform

uint32_t *trtl_msg_filter***word_offset**
offset of the word to check

uint32_t *trtl_msg_filter***mask**
mask to apply before the operation

uint32_t *trtl_msg_filter***value**
second operand of the operation

Then, there are the functions to exchange messages with firmwares running on Mock Turtle. This API offers a minimum set of function to allow users to send/receive synchronous/asynchronous messages. This library does not have any knowledge about the message content, it processes the header but the payload is transfered as is. Any processing on the payload is left to the user. This is the rule for most messages, anyway Mock Turtle offers a set of special messages which are completely handled by Mock Turtle.

struct polltrtl
This structure mimic the pollfd structure for the Mock Turtle needs

Public Members

struct *trtl_dev* **trtl**
device token

unsigned int *polltrtl***idx_cpu**
CPU index

unsigned int *polltrtl***idx_hmq**
HMQ index

short *polltrtl***events**
like in pollfd poll(2)

short *polltrtl***revents**
like in pollfd poll(2)

int **trtl_msg_poll**(**struct** *polltrtl* *trtlp, unsigned int *npolls*, int *timeout*)

It waits for one of a set of Mock Turtle HMQ to become ready to perform I/O

Return like poll(2)

Parameters

- `trtlp`: specific Mock Turtle poll descriptor
- `npolls`: like in poll(2)
- `timeout`: like in poll(2)

int **trtl_msg_sync**(**struct** *trtl_dev* *trtl, unsigned int *idx_cpu*, unsigned int *idx_hmq*, **struct** trtl_msg *msg_s, **struct** trtl_msg *msg_r, int *timeout*)

It sends and receives a synchronous message. It is up to the user to set the "sync_id" in the message that it would like to send. This function configure some filters, so it does some bit magic which have been tested on a little-endian host.

Return 0 on success, otherwise -1 and errno is set appropriately

Parameters

- `trtl`: device token
- `idx_cpu`: CPU index
- `idx_hmq`: HMQ index
- `msg_s`: message to send
- `msg_r`: message received
- `timeout`: like poll(2)

int **trtl_msg_async_send**(**struct** *trtl_dev* *trtl, unsigned int *idx_cpu*, unsigned int *idx_hmq*, **struct** trtl_msg *msg, unsigned int *n*)

It writes messages to a given HMQ

Return on success the number of valid messages, otherwise -1 and errno is set appropriately

Parameters

- `trtl`: device token
- `idx_cpu`: CPU index
- `idx_hmq`: HMQ index
- `msg`: messages to write
- `n`: maximum number of messages to write

int **trtl_msg_async_recv**(**struct** *trtl_dev* *trtl, unsigned int *idx_cpu*, unsigned int *idx_hmq*, **struct** trtl_msg *msg, unsigned int *n*)

It reads messages from a given HMQ

Return on success the number of valid messages, otherwise -1 and errno is set appropriately

Parameters

- `trtl`: device token
- `idx_cpu`: CPU index
- `idx_hmq`: HMQ index
- `msg`: pre-allocated memory where storing the messages
- `n`: maximum number of messages to read

Mock Turtle offers a set of special messages which can be used in combination with the firmware framework to ease the development. The idea behind these special messages is to offer an API for the most common operation that you will perform with Mock Turtle. Of course, you are always free to use the basic message exchange mechanism and build on top of them your high level API.

int **trtl_fw_ping**(**struct** *trtl_dev* **trtl*, unsigned int *idx_cpu*, unsigned int *idx_hmq*)

It checks if firmware core is running and answering to messages

Return 0 on success, -1 on error and errno is set appropriately

Parameters

- `trtl`: device token
- `idx_cpu`: CPU index
- `idx_hmq`: HMQ index

int **trtl_fw_version**(**struct** *trtl_dev* **trtl*, unsigned int *idx_cpu*, unsigned int *idx_hmq*, **struct** *trtl_fw_version* **version*)

Retrieve the current Real-Time Application version running. This is a synchronous message.

Return 0 on success, -1 on error and errno is set appropriately

Parameters

- `trtl`: device token
- `idx_cpu`: CPU index
- `idx_hmq`: HMQ index
- `version`: firmware version

int **trtl_fw_variable_set**(**struct** *trtl_dev* **trtl*, unsigned int *idx_cpu*, unsigned int *idx_hmq*, uint32_t **variables*, unsigned int *n_variables*)

It sends/receive a set of variables to/from the Real-Time application.

The 'variables' field data format is the following

```
0     1     2     3     4     5     ...
+-----+-----+-----+-----+-----+-----+
| IDX | VAL | IDX | VAL | IDX | VAL | ...
+-----+-----+-----+-----+-----+-----+
```

IDX is the variable index defined by the real-time application VAL is the associated value

By setting the flag 'sync' you will send a synchronous message, otherwise it is asyncrhonous. When synchronous the 'variables' field will be overwritten by the syncrhonous answer; the answer contains the read back values for the requested variable after the set operation. You can use this to verify. You can use synchrounous messages to verify that you variable are properly set. This function will change the header content, in particular it will change the following fields: msg_id, len

Return 0 on success, -1 on error and errno is appropriately set.

Parameters

- `trtl`: device token
- `idx_cpu`: CPU index
- `idx_hmq`: HMQ index
- `variables`: on input variable indexes and values. On output variable indexes and values.
- `n_variables`: number of variables to set. In other words, the number of indexes you have in the 'variables' fields

int **trtl_fw_variable_get**(**struct** *trtl_dev* **trtl*, unsigned int *idx_cpu*, unsigned int *idx_hmq*, uint32_t **variables*, unsigned int *n_variables*)

It receive a set of variables from the Real-Time application.

The 'variables' field data format is the following

```
0     1     2     3     4     5    ...
+-----+-----+-----+-----+-----+-----+
| IDX | VAL | IDX | VAL | IDX | VAL | ...
+-----+-----+-----+-----+-----+-----+
```

IDX is the variable index defined by the real-time application VAL is the associated value

This kind of message is always synchronous. The 'variables' field will be overwritten by the syncrhonous answer; the answer contains the read back values for the requested variables. This function will change the header content, in particular it will change the following fields: msg_id, flags, len

Return 0 on success, -1 on error and errno is appropriately set.

Parameters

- `trtl`: device token
- `idx_cpu`: CPU index
- `idx_hmq`: HMQ index
- `variables`: on input variable indexes. On output variable indexes and values.
- `n_variables`: number of variables to set. In other words, the number of indexes you have in the 'variables' fields

int **trtl_fw_buffer_set**(**struct** *trtl_dev* **trtl*, unsigned int *idx_cpu*, unsigned int *idx_hmq*, **struct** trtl_tlv **tlv*, unsigned int *n_tlv*)

It sends/receives a set of structures within TLV records. This function will change the header content, in particular it will change the following fields: msg_id, len

Parameters

- `trtl`: device token
- `idx_cpu`: CPU index
- `idx_hmq`: HMQ index
- `tlv`: the complete buffer
- `n_tlv`: number of tlv structures

int **trtl_fw_buffer_get**(**struct** *trtl_dev* **trtl*, unsigned int *idx_cpu*, unsigned int *idx_hmq*, **struct** trtl_tlv **tlv*, unsigned int *n_tlv*)

It receives a set of structures within TLV records. This function will change the header content, in particular it will change the following fields: msg_id, flags, len

Parameters

- `trtl`: device token
- `idx_cpu`: CPU index
- `idx_hmq`: HMQ index
- `tlv`: on input tlv with only the type, on output the complete buffer
- `n_tlv`: number of tlv structures

Shared Memory

The Mock Turtle shared memory is accessible also from the host.

In some cases accessing the shared memory from host is necessary, but this is not really encouraged as *normal usage* because it may affect the Mock Turtle determinism. This API is limited to the basic function to read and write: `trtl_smem_write()`, `trtl_smem_write()`.

int **trtl_smem_read**(**struct** *trtl_dev* *trtl, uint32_t *addr*, uint32_t *data, size_t *count*, **enum** *trtl_smem_modifier* *mod*)

It does a direct acces to the shared memory to read a set of cells

Return 0 on success, -1 otherwise and errno is set appropriately

Parameters

- `trtl`: device token
- `addr`: memory address where start the operations
- `data`: values read from in the shared memory. The function will replace this value with the read back value
- `count`: number of values in data
- `mod`: shared memory operation mode

int **trtl_smem_write**(**struct** *trtl_dev* *trtl, uint32_t *addr*, uint32_t *data, size_t *count*, **enum** *trtl_smem_modifier* *mod*)

It writes on the shared memory of the TRTL

Return 0 on success, -1 otherwise and errno is set appropriately

Parameters

- `trtl`: device to use
- `addr`: memory address
- `data`: values to write in the shared memory. The function will replace this value with the read back value
- `count`: number of values in data
- `mod`: shared memory operation mode

enum trtl_smem_modifier

Shared memory operation modifier. This is a list of operation modifier that you can use to access the shared memory.

Values:

TRTL_SMEM_TYPE_BASE = 0
no operation

TRTL_SMEM_TYPE_ADD
atomic addition

TRTL_SMEM_TYPE_SUB
atomic subtraction

TRTL_SMEM_TYPE_SET
atomic bit set

TRTL_SMEM_TYPE_CLR
atomic bit clear

TRTL_SMEM_TYPE_FLP
atomic bit flip

TRTL_SMEM_TYPE_TST_SET
atomic test and set

Utilities

This library offers a set of handlers.

char *__trtl_strerror__(int *err*)
It returns a string messages corresponding to a given error code. If it is not a libtrtl error code, it will run strerror(3)

Return a message error. No need to free the string.

Parameters

- `err`: error code. Typically 'errno' variable

uint32_t __trtl_count__()
It returns the number of available TRTLs. This is not calculated on demand. It depends on library initialization.

Return the number of TRTL available

char **__trtl_list__()
It allocates and returns the list of available TRTL devices. The user is in charge to free the allocated memory wit *trtl_list_free()*. The list contains *trtl_count()* + 1 elements. The last element is a NULL pointer.

Return a list of TRTL device's names. NULL on error

void __trtl_list_free__(char ***list*)
It release the list allocated memory

Parameters

- `list`: device list to release

char *__trtl_name_get__(__struct__ *trtl_dev* **trtl*)
It returns the device name

Return the string representing the name of the device

Parameters

- `trtl`: device token

void __trtl_print_header__(__struct__ trtl_msg **msg*)
It prints the message header in a human readable format

Parameters

- `msg`: message

void __trtl_print_payload__(__struct__ trtl_msg **msg*)
It prints the payload in a human readable format according to the message type

Parameters

- `msg`: message

void __trtl_print_message__(__struct__ trtl_msg **msg*)
It prints a message in a human readable format. This function assumes that the message contains a Mock Turtle message header. According to the message ID the format may change

Parameters

- `msg`: message

int __trtl_hmq_fd__(__struct__ *trtl_dev* **trtl*, unsigned int *idx_cpu*, unsigned int *idx_hmq*)
It returns the HMQ File Descriptor

Return the file descriptor

Parameters

- `trtl`: device token
- `idx_cpu`: CPU index
- `idx_hmq`: HMQ index

enum trtl_error_number
Error codes for Mock-Turtle applications

Values:

ETRTL_INVAL_PARSE = 83630
cannot parse data from sysfs

ETRTL_INVAL_SLOT
invalid slot

ETRTL_NO_IMPLEMENTATION
a prototype is not implemented

ETRTL_HMQ_CLOSE
The HMQ is closed

ETRTL_INVALID_MESSAGE
Invalid message

ETRTL_HMQ_READ
Error while reading messages

ETRTL_MSG_SYNC_FAILED_SEND
Send sync message failure

ETRTL_MSG_SYNC_FAILED_RECV
Receive sync message failure

ETRTL_MSG_SYNC_FAILED_RECV_TIMEOUT
Receive sync message failure: timeout

ETRTL_MSG_SYNC_FAILED_RECV_POLLERR
Receive sync message failure

ETRTL_MSG_SYNC_FAILED_INVAL
Receive sync message failure: invalid

__ETRTL_MAX

4.2.3 The Mock Turtle Python Support

The Mock Turtle Python Module (*PyMockTurtle*) is a Python module that wraps the Mock Turtle library described in *The Mock Turtle Linux Library*.

Most of the features that the Linux library provides are as well provided by the PyMockTurtle module.

Installation

Requirements

PyMockTurtle depends on:

- Python 3.x;
- Python ctype
- Mock Turtle Linux library `libmockturtle.so` installed;
- Mock Turtle Linux driver loaded;

Note: The module has been tested with Python 3.5. In principle it should work as well on any 3.x version. Compatibility with Python 2.7 has not been verified. Open an issue if you find Python version incompatibilities.

Install

You can use the Makefile to install PyMockTurtle module:

```
cd /path/to/mockturtle/software/lib/PyMockTurtle
make install
```

Alternatively, you can use the *distutil* script that takes care of the module installation in your Python environment:

```
cd /path/to/mockturtle/software/lib/PyMockTurtle
python setup.py install
```

On a successful installation you should be able to import PyMockTurtle:

```
import PyMockTurtle
```

The installation is not mandatory. What is really important is that both the shared object library and the Python module are visible to the Python interpreter. You can use the environment variables `PYTHONPATH` and `LD_LIBRARY_PATH` to make them visible:

```
export LD_LIBRARY_PATH=$LD_LIBRARY_PATH:/path/to/mock-turtle-sw/lib
export PYTHONPATH=$PYTHONPATH:/path/to/mock-turtle-sw/lib/PyMockTurtle

python3
>>> import PyMockTurtle
```

Distribution

If you want to create a package for the distribution of this module, you can use the `sdist` command:

```
cd /path/to/mock-turtle-sw/lib/PyMockTurtle
python setup.py sdist
```

This will create the dist directory in which you will find an archive corresponding to the version declared in the setup.py script.

The PyMockTurlte Basic Usage

The usage of this module is quite straight forward. The first thing you have to to do is to create an instance for *PyMockTurtle.TrtlDevice*. The instantiation process will autoconfigure the new object using the information from the configuration ROM. This means that all the cores (*PyMockTurtle.TrtlCpu*) and the respective host message queues (*PyMockTurtle.TrtlHmq*) will be instanciated automaticaly.:

```
import PyMockTurtle

trtl = PyMockTurtle.TrtlDevice(0x1)
```

At this point it should be enough to have a look at *The PyMockTurtle API* to start using the object.

The PyMockTurtle API

Here you can find the complete *PyMockTurtle* API. PyMockTurtle exports a set of objects used to handle Mock Turtle components. Then, it exports a set of *ctype* data structures used to exchange information with the Mock Turtle layers.

Note: Since this Pyhton module is nothing more than a wrapper on top of a C library, we suggest you to have a look at *The Mock Turtle Linux Library* for a better understanding of this API

PyMockTurtle Objects

class PyMockTurtle.**TrtlDevice**(*devid*)
It is a Python class that represent a Mock Turtle Device

Parameters **devid** – Mock Turtle device identifier

Variables

- **device_id** – device ID associated with the instance
- **tkn** – device token to be used with the libmockturtle library
- **libtrtl** – the libmockturtle library
- **rom** – configuration rom
- **cpu** – list of TrtlCpu instances
- **shm** – shared memory

class PyMockTurtle.**TrtlCpu**(*trtl_dev*, *idx_cpu*)
It is a Python class that represents a Mock Turtle device CPU

Parameters

- **trtl_dev** – the device instance to use
- **idx_cpu** – the core to access

Variables

- **trtl_dev** – parent device instance
- **idx_cpu** – the core index
- **hmq** – the list of TrtlHmq instances

disable()
It disables a CPU; in other words, it sets the reset line of a CPU.

Raises **OSError** – from C library errors

dump_application_file(*file_path*)
It dumps the running firmware to the given file

Parameters **file_path** – path to the firmware file

Raises **OSError** – from C library errors

enable()
It enables a CPU; in other words, it clear the reset line of a CPU.

Raises **OSError** – from C library errors

is_enable()
It checks if the CPU is enabled (or not)

Returns True when the CPU is enable, False otherwise

Raises **OSError** – from C library errors

load_application_file(*file_path*)
It loads a firmware from the given file

Parameters **file_path** – path to the firmware file

Raises **OSError** – from C library errors

ping(*idx_hmq=0*)
It pings the firmware running on the CPU

Parameters **idx_hmq** (*int*) – which HMQ to use for pinging (default: 0)

Returns True if the firmware is alive, False otherwise

Return type bool

Raises **OSError** – from C library errors

version(*idx_hmq=0*)
It pings the firmware running on the CPU. :param idx_hmq: which HMQ to use for pinging (default: 0) :type idx_hmq: int :return: True if the firmware is alive, False otherwise :raises OSError: from C library errors

class PyMockTurtle.**TrtlHmq**(*trtl_cpu*, *idx_hmq*)
Python wrapper for HMQ management

Parameters

- **trtl_cpu** – the cpu instance to use
- **idx_hmq** – the HMQ to access

flush()
It removes enqueued messages from the queue. It does it for both direction: input and output.

get_stats()
It gets statistics :return: a dictionary with the statistics :rtype: dict

recv_msg(*timeout=-1*)
It receives an asynchronous message.

Parameters **timeout** (*int*) – time to wait before returning

Returns an asynchronous message

Return type *TrtlMessage*

Raises **OSError** – from C library errors

send_msg(*msg*, *timeout=-1*)
It sends an asynchronous message.

Parameters

- **msg** (TrtlMessage) – the message to send
- **timeout** (*int*) – time to wait before returning

Raises **OSError** – from C library errors

sync_msg(*msg_s*, *timeout=1000*)
It sends a synchronous message.

Parameters

- **msg_s** (TrtlMessage) – the message to send
- **timeout** (*int*) – time to wait before returning in milli-seconds

Returns the synchronous answer

Return type *TrtlMessage*

Raises **OSError** – from C library errors

class PyMockTurtle.**TrtlSmem**(*trtl_dev*)
It allows to read and write the Mock Turtle shared memory

MOD_DIRECT = 0
Normal write

MOD_ADD = 1
Write modifier for atomic addition of the shared memory value with the given one

MOD_SUB = 2
Write modifier for atomic subtraction of the shared memory value with the given one

MOD_SET = 3
Write modifier for atomic OR of the shared memory value with the given one. In other words it sets the bit in the given value

MOD_CLEAR = 4
Write modifier for atomic AND NOT of the shared memory value with the given one. In other words it clears the bit in the given value

MOD_FLIP = 5
Write modifier for atomic XOR of the shared memory value with the given one. In other words it flips the bit in the given value

MOD_TEST_AND_SET = 6
Write modifier for atomic TEST and SET. FIXME

MOD_ADD = 1
Write modifier for atomic addition of the shared memory value with the given one

MOD_CLEAR = 4
Write modifier for atomic AND NOT of the shared memory value with the given one. In other words it clears the bit in the given value

MOD_DIRECT = 0
Normal write

MOD_FLIP = 5
Write modifier for atomic XOR of the shared memory value with the given one. In other words it flips the bit in the given value

MOD_SET = 3
Write modifier for atomic OR of the shared memory value with the given one. In other words it sets the bit in the given value

MOD_SUB = 2
Write modifier for atomic subtraction of the shared memory value with the given one

MOD_TEST_AND_SET = 6
Write modifier for atomic TEST and SET. FIXME

read(*address*, *count*)
It reads from the shared memory 'count' 32bit words starting from 'address'

Parameters

- **address** – memory address where start writing
- **count** – number of 32bit consectutive words to read

Returns the values

Return type list of int

Raises **OSError** – from C library errors

write(*address*, *values*, *modifier*)
It writes 'values' to the shared memory starting from 'address' using the access mode 'modifier'

Parameters

- **address** (*int*) – memory address where start writing

- **values** (*list of int*) – the values to write
- **modifier** (*int*) – write operation modifier. It changes the write behaviour

Raises **OSError** – from C library errors

PyMockTurtle Data Structures

class PyMockTurtle.**TrtlFirmwareVersion**
It is a descriptor for firmware versions

Variables

- **fw_id** – firmware unique identifier
- **fw_version** – firmware version
- **git_version** – firmware first 32bit git SHA

class PyMockTurtle.**TrtlMessage**
It is a container for Mock Turtle messages

Variables

- **header** – message header
- **payload** – message payload

class PyMockTurtle.**TrtlHmqHeader**
It describes the HMQ protocol header

Variables

- **rt_app_id** –
- **flags** –
- **msg_id** –
- **len** –
- **sync_id** –
- **seq** –

TRTL_HMQ_HEADER_FLAG_ACK = 2
Flag for synchronous acknowledgment

TRTL_HMQ_HEADER_FLAG_SYNC = 1
Flag for synchronous messages

class PyMockTurtle.**TrtlConfig**
The synthesis configuration ROM descriptor shows useful configuration options during synthesis.

TRTL_CONFIG_ROM_SIGNATURE = 1414681676
This signature must be present on all the configuration rom

class PyMockTurtle.**TrtlConfigMq**
It describe the configuration information for a single message queue

Variables

- **sizes** –
- **endpoint_id** –

4.3 Firmware Development

This section explains how to write firmwares using the Mock Turtle API.

The Mock Turtle offers 2 API for the firmware development: *The Mock Turtle Firmware Library* and *The Mock Turtle Firmware Framework*.

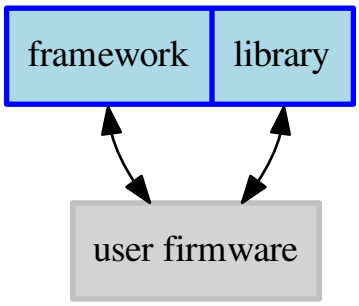

Fig. 2: Mock Turtle Firmware Interfaces.

It is strongly recommended to use the library because it offers a set of macros and functions that simplifies the access to Mock Turtle resources and to external gateware cores. This will help mainly in firmware development.

It is recommended to use the framework because it guides you in the development by keeping you focused only on your core logic without the need to deal with Mock Turtle architecture details.

The framework usage, rather than precluding the user to use library functions, is complementary to the library.

Of course, this framework provides more features and features cost space and computation time. If you need more space (and you can't allocate more memory) or you need much better performances: don't use this framework.

All the Mock Turtle firmware source code can be found in the directory `/path/to/mockturtle/software/firmware/`.

Mock Turtle has a generic `Makefile` that you should include in your `Makefile` in order to import all the Mock Turtle building rules.:

```
TRTL ?= /path/to/mcokturtle/
TRTL_SW = $(TRTL)/software

# Mandatory
OBJS = source1.o
OBJS += source2.o
OBJDIR += some/local/directory
OUTPUT = firmware-name

# Optional (prefer default when possible)
CFLAGS_OPT := -O1
CFLAGS_DBG := -ggdb
EXTRA_CFLGAS :=
MOCKTURTLE_LDSCRIPT := myfirmware.ld

# INCLUDE GENERIC Makefile
include $(TRTL_SW)/firmware/Makefile
```

Here the list of supported Makefile environment variables

OBJS (Mandatory) List of object files to generate from sources with the same name. The default is an empty variable, this mean that it will not compile any source file.

OUTPUT (Mandatory) Final binary name (the firmware).

MOCKTURTLE_LDSCRIPT (Optional) Local path to the linker script. The default is the standard Mock Turtle linker script.

Memory resources on Mock Turtle are very limited and the full framework may take more space than needed. For this reason Mock Turtle has *Kconfig* support which allows you to interactivly enable/disable both library and framework. You should create a local `Kconfig` file in your firmware directory; in this file you must include the generic one from Mock Turtle.:

```
mainmenu "Project Firmware Name"

comment "Project specific configuration"

# INCLUDE GENERIC Kconfig
source "Kconfig.mt"
```

Configuration options are not documented here. For more details use the help messages from Kconfig: run `make menuconfig` from your firmware directory.

Mock Turtle is using the *RISC-V* ISA, this means that your code must be compiled for this instruction-set. Mock Turtle uses the environment variable `CROSS_COMPILE_TARGET` to provide the path to the cross-compilation toolchain. By default, Mock Turtle expects the cross-compilation toolchain to be installed on your system and visible in `PATH`. If this is not the case you have to overwrite this variable.:

```
export CROSS_COMPILE_TARGET=/path/to/toolchain/bin/riscv32-elf-
```

At this point you can call `make(1)` to build your firmware.

Note: If you do not know how to get the cross-compilation toolchain or you need to build your own one, please have a look at the soft-cpu toolchain project on the OHWR.

4.3.1 The Mock Turtle Firmware Library

The Mock Turtle firmware library offers a set of macros and functions for the basic interaction with Mock Turtle resources. This API is available by including `mockturtle-rt.h` in your source file.:

```
#include <mockturtle-rt.h>
```

We strongly recommend users to use this library to develop any Mock Turtle firmware.

> **Warning:** Any firmware develop without this library will not receive any kind of support.

Read And Write Memory Locations

This firmware library offers a set of function to read/write memory locations. You can access local register with `lr_readl()` and `lr_writel()`. You can access device peripherals with `dp_readl()` and `dp_writel()`.:

```
#include <mockturtle-rt.h>

/*
 * List of device peripheral cores offsets from the point of view
 * of the soft-cpu
```

(continues on next page)

(continued from previous page)

```
 */
#define DP_CORE_1 0x1000

int main ()
{
  int val;

  /* Read a register from the device peripheral 1 */
  dp_writel(0xBADCOFFE, DP_CORE_1 +0x4);
  val = dp_readl(DP_CORE_1 + 0x4);

  /* Read registers from the local soft-core */
  lr_writel(0xDEADBEEF, 0x4);
  val = lr_readl(0x4);
}
```

static uint32_t **dp_readl** (uint32_t *reg*)
Read a 32bit word value from the Dedicated Peripheral address space

Return the value fromt the register

Parameters

- `reg`: register offset within the Dedicated Peripheral

static void **dp_writel** (uint32_t *value*, uint32_t *reg*)
Write a 32bit word value to the Dedicated Peripheral address space

Parameters

- `value`: value to write
- `reg`: register offset within the Dedicated Peripheral

static uint32_t **lr_readl** (uint32_t *reg*)
Read 32bit word value from the CPU Local Registers address space

Return the value fromt the register

Parameters

- `reg`: register offset within the Local Registers

static void **lr_writel** (uint32_t *value*, uint32_t *reg*)
Write 32bit word value to the CPU Local Registers address space

Parameters

- `value`: value to write
- `reg`: register offset within the Local Registers

Each Mock Turtle core has a group of GPIO lines which can be used to access external signals. You can handle the GPIO with the functions `gpio_set()`, `gpio_clear()`, `gpio_status()`.:

```
#include <mockturtle-rt.h>

int main ()
{
  int val;

  gpio_set(11); /* set BIT(11) to 1 */
  val = gpio_status(11);
  gpio_clear(11); /* set BIT(11) to 0 */
  val = gpio_status(11);
}
```

static void **gpio_set** (int *pin*)
Set a bit in the CPU GPIO Register

Parameters

- `pin`: GPIO pin to set

static void **gpio_clear** (int *pin*)
Clear a bit in the CPU GPIO Register

Parameters

- `pin`: GPIO pin to clear

static unsigned int **gpio_status** (int *pin*)
It gets the GPIO status

Return the GPIO status

Parameters

- `pin`: GPIO pin to query

All these functions are based on the generic `readl()` and `writel()`.:

```
#include <mockturtle-rt.h>

int main ()
{
  /* set GPIO 11 to 1 - using generic write */
  writel((1 << 11), TRTL_ADDR_LR(MT_CPU_LR_REG_GPIO_SET));
  /* set GPIO 11 to 1 */
  gpio_set(11);
}
```

Note: In order to keep your code clean and future proof, do not use generic functions when a specific one is available. The example above makes it evident.

static uint32_t **readl** (void **addr*)
Read a 32bit word value from the given address

Return the value fromt the register

Parameters

- `addr`: source address

static void **writel** (uint32_t *value*, void **addr*)
Write a 32bit word value to the given address

Parameters

- `value`: value to write
- `addr`: destination address

Message Queue

Mock Turtle cores' main communication mechanism is the message queues. The API is almost identical for both remote and host because most of these functions have the *message queue type* argument to distinguish them.

You can handle the message queue with the commands: *claim*, *send*, *discard*, *purge*. For each of these command there is a function that you can call to execute that command: `mq_claim()`, `mq_send()`, `mq_discard()`, `mq_purge()`. A part from performing active actions on the message queue, sometimes we are interested only in their status, expecialy we want to know if the message queue input channel is *not empty* (it means that there

is something to read) or the output channel is *not full* (it means that there space for writing). You can check the queue status with `mq_poll_in()` and `mq_poll_out()`.

The API usage is different for input and for output.

The typical procedure to send (output) messages over is the following.

1. *poll* the mq to see if there is at least an empty entry;
2. *claim* a mq in order to get exclusive access to it;
3. *map* the claimed mq slot in order to get the buffer where to write;
4. *write* your message;
5. *send* the data, which will also release the mq slot;

Here the output example.:

```
#define HMQ_NUM 0
struct trtl_fw_msg msg;
uint32_t status;

while ((mq_poll_out(TRTL_HMQ) & (1 << HMQ_NUM)) == 0)
    ; /* wait until queue not full */

mq_claim(TRTL_HMQ, HMQ_NUM);
mq_map_out_message(TRTL_HMQ, HMQ_NUM, &msg);

/* Play with the message queue entry */

mq_send(TRTL_HMQ, HMQ_NUM);
```

The typical procedure to receive (input) messages is the following.

1. *poll* the mq to see if there is available data on a slot;
2. *map* the mq slot in order to get the slot buffer;
3. *read* your data from the mapped buffer;
4. *discard* the slot, which will erase the data and point to the next one;

Here the input example.:

```
#define HMQ_NUM 0
struct trtl_fw_msg msg;
uint32_t status;

while ((mq_poll_in(TRTL_HMQ) & (1 << HMQ_NUM)) == 0)
    ; /* wait until queue not empty */

mq_map_in_message(TRTL_HMQ, HMQ_NUM, &msg);

/* Play with the message queue entry */

mq_discard(TRTL_HMQ, HMQ_NUM);
```

The library does not perform any validation on the data you write in the message. Any kind of overflow control is up to the user who can take the payload size from the configuration rom using `trtl_config_rom_get()`.

On the host you can read the messages using the tool *Mock Turtle Messages*

enum trtl_mq_type
List of Message Queue types

Values:

TRTL_HMQ = 0
Host Message Queue - Host-Firmware

TRTL_RMQ
Remote Message Queue - Network-Firmware

__TRTL_MAX_MQ_TYPE

static uint32_t **mq_poll_in**(**enum** *trtl_mq_type type*, uint32_t *mask*)
It gets the current MQ input status

Return message queues input status bitmask

Parameters

- `type`: MQ type to use
- `mask`: bitmask to set the bit of interest

static uint32_t **mq_poll_out**(**enum** *trtl_mq_type type*, uint32_t *mask*)
It gets the current MQ output status

Return message queues output status bitmask

Parameters

- `type`: MQ type to use
- `mask`: bitmask to set the bit of interest

static void **mq_claim**(**enum** *trtl_mq_type type*, int *slot*)

Parameters

- `type`: MQ type to use
- `slot`: slot number

static void **mq_send**(**enum** *trtl_mq_type type*, int *slot*)

Parameters

- `type`: MQ type to use
- `slot`: slot number

static void **mq_purge**(**enum** *trtl_mq_type type*, int *slot*)

Parameters

- `type`: MQ type to use
- `slot`: slot number

static void **mq_discard**(**enum** *trtl_mq_type type*, int *slot*)

Parameters

- `type`: MQ type to use
- `slot`: slot number

struct trtl_fw_msg
Messages descriptor for firmware. We map directly the HDL buffer

Public Members

struct *trtl_hmq_header* ***trtl_fw_msg*header**
points to HMQ header buffer

void ***trtl_fw_msg*payload**
points to HMQ payload buffer

static void **mq_map_out_message**(**enum** *trtl_mq_type type*, unsigned *idx_mq*, **struct** *trtl_fw_msg* ***msg*)

It maps a given MQ for outcoming messages

Parameters

- `type`: MQ type to use
- `idx_mq`: MQ index
- `msg`: where to map the message

static void **mq_map_in_message**(**enum** *trtl_mq_type type*, unsigned *idx_mq*, **struct** *trtl_fw_msg* ***msg*)

It maps a given MQ for incoming messages

Parameters

- `type`: MQ type to use
- `idx_mq`: MQ index
- `msg`: where to map the message

These functions are enough to send and receive messages with both HMQ and RMQ.

Following a list of lower level functions which actually are used to implement the ones above.

Note: In principle, you should never use the lower level API. These functions are used to provide services for the higher level API

---

static void ***trtl_mq_base_address**(**enum** *trtl_mq_type type*)

It gets the Message Queue base address

Return message queue base address

Parameters

- `type`: MQ type

static void ***mq_map_out_header**(**enum** *trtl_mq_type type*, int *slot*)

It gets the output slot header field pointer

Return pointer to the output header

Parameters

- `type`: MQ type to use
- `slot`: slot number

static void ***mq_map_out_buffer**(**enum** *trtl_mq_type type*, int *slot*)

It gets the output slot data field pointer

Return pointer to the input buffer

Parameters

- `type`: MQ type to use
- `slot`: slot number

static void ***mq_map_in_header**(**enum** *trtl_mq_type type*, int *slot*)

It gets the input slot header field pointer

Return pointer to the input header

Parameters

- `type`: MQ type to use
- `slot`: slot number

static void ***mq_map_in_buffer**(**enum** *trtl_mq_type type*, int *slot*)
It gets the input slot data field pointer

Return pointer to the input buffer

Parameters

- `type`: MQ type to use
- `slot`: slot number

Shared Memory

This is a collection of functions and macros which purposes are:

- to read/write the Mock Turtle shared memory
- to perform atomic operations on the shared memory

In order to declare a variable in shared memory, instead of the local soft-core RAM, you have to add `SMEM` before your variable declaration:

```
#include <mockturtle-rt.h>

SMEM int my_variable = 100;
```

Then you can use a shared memory variable as a normal variable:

```
#include <mockturtle-rt.h>

SMEM int my_variable = 100;

int main ()
{
  int b = 5;

  my_variable += 10; /* Not atomic operation */
  b = my_variable;
}
```

The shared memory provides a set of atomic operations, to avoid race conditions while different cores are writing. There is a dedicated API for such operations.

Here an example that uses all operations.:

```
#include <mockturtle-rt.h>

SMEM int my_variable = 100;

int main ()
{
  int b = 5, t;

  smem_atomic_add(&my_variable, 10);
  smem_atomic_sub(&my_variable, 10);
  smem_atomic_or(&my_variable, 0xF0);
  smem_atomic_and_not(&my_variable, 0xF0);
  smem_atomic_xor(&my_variable, 0xF0);
  t = smem_atomic_test_and_set(&my_variable);
  b = my_variable;
}
```

static void **smem_atomic_add**(**volatile** int *p, int *x*)
It performs an

```
(*p) = (*p) + x
```

Parameters

- p: address on the shared memory of the first operator and store location
- x: second operation argument

static void **smem_atomic_sub** (**volatile** int *p, int x)
It performs an

```
(*p) = (*p) - x
```

Parameters

- p: address on the shared memory of the first operator and store location
- x: second operation argument

static void **smem_atomic_or** (**volatile** int *p, int x)
It performs an

```
(*p) = (*p) | x
```

Parameters

- p: address on the shared memory of the first operator and store location
- x: second operation argument

static void **smem_atomic_and_not** (**volatile** int *p, int x)
It performs an

```
(*p) = (*p) & (~x)
```

Parameters

- p: address on the shared memory of the first operator and store location
- x: second operation argument

static void **smem_atomic_xor** (int *p, int x)
It performs an

```
(*p) = (*p) ^ x
```

Parameters

- p: address on the shared memory of the first operator and store location
- x: second operation argument

static int **smem_atomic_test_and_set** (int *p)
It performs an:

```
val = (*p);
if (val == 0) {
    (*p) = 1;
}
return val;
```

This is useful to implement mutex

Return the value before the set

Parameters

- `p`: address on the shared memory

Serial Interface

Over the serial interface you can print formatted string messages.

Note: Even if it is potentially possible to use the serial interface to exchange binary data, this is not supported. The only supported use of the *Serial Interface* is to send strings to the host system.

This API is based on `pp_printf()` and its different flavors: `pr_error()`, `pr_debug()`.

Here an example:

```
#include <mockturtle-rt.h>

int main ()
{
  pp_printf("Hello World\n");
  pr_debug("%s:%d something here\n", __func__, __LINE__);
  pr_error("%s:%d Error Message\n", __func__, __LINE__);
}
```

On the host side you can read the serial messages using any tool that can read a TTY interface (e.g. `cat`, `minicom`).

Since the standard `printf` function is heavy, Mock Turtle offers a light implementation named `pp_printf()`. This function relays on function `puts()` to send the strings over the serial interface.

static int **pp_printf**(**const** char *_fmt_, ...)

It prints a string on the serial interface. Internally, it uses the *puts()* function.

Return number of printed characters

Parameters

- `fmt`: string format
- `...`: argument according to the string format

static int **pr_debug**(**const** char *_fmt_, ...)

It prints a string on the serial interface only when the support for error messages is enable.

Kconfig ->CONFIG_MOCKTURTLE_LIBRARY_PRINT_DEBUG_ENABLE

Internally, it uses the *puts()* function.

Return number of printed characters

Parameters

- `fmt`: string format
- `...`: argument according to the string format

static int **pr_error**(**const** char *_fmt_, ...)

It prints a string on the serial interface only when the support for error messages is enable.

Kconfig -> CONFIG_MOCKTURTLE_LIBRARY_PRINT_ERROR_ENABLE

Internally, it uses the *puts()* function.

Return number of printed characters

Parameters

- `fmt`: string format
- `...`: argument according to the string format

void **pr_message**(**struct** *trtl_fw_msg* *msg)
It prints on the serial console the given message

Parameters

- `msg`: a mock turtle message

int **putchar**(int *c*)
It sends a character to the UART interface

Return 0 on success (for the time being it does not fail)

Parameters

- `c`: the character to sent

int **puts**(**const** char *p)
It sends a string over the serial interface The function does not add any special character. If you neeed the new-line or the carriage-return you have to add them to your strings.

Return number of sent characters

Parameters

- `p`: string to send

Host Notification

Mock Turtle has a mechanism that allows firmwares to send arbitrary interrupts to the host system. This mechanism is used by Mock Turtle software to deliver special notifications; but this mechanism can be used as well by the user to deliver custum notifications to their support layer.

The Mock Turtle notifications are enumerated by `trtl_cpu_notification`. The user must start their enumeration after the value *__TRTL_CPU_NOTIFY_MAX*.

enum trtl_cpu_notification
It lists all Mock Turtle notification's code.

Values:

TRTL_CPU_NOTIFY_APPLICATION = __TRTL_CPU_NOTIFY_APPLICATION_MAX
anonymous application notification (user)

TRTL_CPU_NOTIFY_INIT
the firmware is executing init phase

TRTL_CPU_NOTIFY_MAIN
the firmware is executing main phase

TRTL_CPU_NOTIFY_EXIT
the firmware is executing exit finishing

TRTL_CPU_NOTIFY_ERR
general error, firmware is not running anymore

__TRTL_CPU_NOTIFY_MAX

Mock Turtle will deliver a notification to the host when the firmware calls `trtl_notify()` or, suggested for user's notification, `trtl_notify_user()`.

static void **trtl_notify**(uint8_t *id*)
It generates a notification signal (IRQ) to ask the HOST CPU to take some action.

Parameters

- `id`: CPU notification identifier

static void **trtl_notify_user**(uint8_t *id*)
It generates a notification signal (IRQ) to ask the HOST CPU to take some action.

Parameters

- `id`: CPU notification identifier

Miscellaneous

At the end, this chapter is a collection of helpers

static void **delay**(int *n*)
Wait n cycles. This means that the wait can be different on different core with different clock.

Parameters

- `n`: number of cycles to wait

static struct *trtl_config_rom* ***trtl_config_rom_get**(void)
It returns a pointer to the config ROM

Return pointer to the configuration ROM

static uint32_t **trtl_get_core_id**(void)
It returns the core ID on which the firmware is running

Return the core ID

4.3.2 The Mock Turtle Firmware Framework

Mock Turtle firmware framework guides users development by keeping them focused only on the core logic without the need to deal with Mock Turtle architectural details.

This API is available by including `mockturtle-framework.h` in your source file.:

```
#include <mockturtle-framework.h>
```

We recommend this framework to develop Mock Turtle firmwares. You should consider alternatives if you see that it's consuming too much memory or the performances are not enough for your application.

Note: This framework internally uses the *The Mock Turtle Firmware Library*

Application

Firmwares developed with this framework need to be described by `trtl_fw_application`. Firmware applications develop with this framework does not have a `main()`. The `main()` is implemented within the framework itself. What you should do is to declare a new `trtl_fw_application` named **app** and implement the operations *init*, *main* and *exit*. These operations are all optional, it means that if you do not implement them nothing will be executed.

Here an example.:

```
#include <mockturtle-framework.h>

static init myinit()
{
    return 0;
}

static int mymain()
```

(continues on next page)

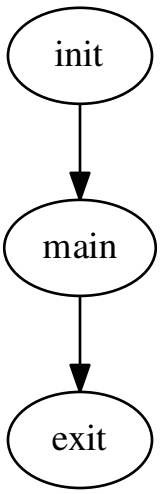

Fig. 3: Mock Turtle Firmware Framework Phases.

(continued from previous page)

```
{
    while (1) {
        /* main code here */
    }
    return 0;
}

static init myexit()
{
    return 0;
}

struct trtl_application app = {
    .name = "myfirmware",
    .fpga_id_compat = NULL,
    .fpga_id_compat_n = 0,
    .version = {
            .id = 0x12345678,
            .version = RT_VERSION(1, 0),
            .git = GIT_VERSION,
    },

    .init = myinit,
    .main = mymain,
    .exit = myexit,
};
```

The Mock Turtle is FPGA based, this means that the firmwares are typically loaded by the host. The procedure is error prone, so it may happen to load the wrong firmware with unpredictable consequences. To limit the damage, the `fpga_id_compat` can be used to declare a list of compatible gateware identifiers. The firmware framework will refuse to execute the firmware when it is not compatible with the gateware according to the given list. To disable this validation step, do not provide a compatibility list (like in the example above).

`struct trtl_fw_application`

Firmware Application Descriptor. It provides a set of useful information used by the framework to provide services to users.

Public Members

const char *trtl_fw_application***name**[16]
Firmware name

const uint32_t **trtl_fw_application***__**fpga_id_compat**
list of compatible FPGA application ID

const unsigned int *trtl_fw_application***fpga_id_compat_n**
number of entry in the fpga_id_compat list

struct *trtl_fw_version trtl_fw_application***version**
version running

struct trtl_fw_buffer **trtl_fw_application***__**buffers**
exported buffers

unsigned int *trtl_fw_application***n_buffers**
number or exported buffers

struct trtl_fw_variable **trtl_fw_application***__**variables**
exported variables

unsigned int *trtl_fw_application***n_variables**
number or exported variables

trtl_fw_action_t ***trtl_fw_application***actions**
list of custum actions

unsigned int *trtl_fw_application***n_actions**
number of custum actions

struct *trtl_config_rom* **trtl_fw_application***__**cfgrom**
sequence number reference configuration ROM

trtl_fw_phase_t **trtl_fw_application***__**init**
function where you prepare the application

trtl_fw_phase_t **trtl_fw_application***__**main**
function where you execute your application

trtl_fw_phase_t **trtl_fw_application***__**exit**
function where you clean-up firmware status

struct trtl_fw_version
It describes the version running on the embedded CPU

Public Members

uint16_t *trtl_fw_version***rt_id**
RT application identifier

uint32_t *trtl_fw_version***rt_version**
RT application version

uint32_t *trtl_fw_version***git_version**
git commit SHA1 of the compilation time

Actions

The *action* is s function that get executed when a special message arrives through a mq slot. In order words, an action is similar to an RPC. The firmware framework relies on the *Host Message Queue Protocol* to make this work.

The mechanism is quite simple. The Mock Turtle message header has a dedicated flag to mark a message as an RPC call *TRTL_HMQ_HEADER_FLAG_RPC*. When this flag is set, the firmware framework interprets the header's message-id as action-id. Thus, the correspondent function get executed.

This framework support up to __TRTL_MSG_ID_MAX actions. Part of it (__TRTL_MSG_ID_MAX_TRTL) is reserved for internal use, the rest (__TRTL_MSG_ID_MAX_USER) can be used to implement new actions. The reserved IDs are at the end of the number space and they are defined in `trtl_msg_id`.

The declaration of a new user action consists in 4 steps:

1. enumerate the action IDs, share it with host;
2. implement the function to execute;
3. add the function to the action list, and map it to an action-id. The framework uses the message id in `trtl_hmq_header` as action-id to execute the functions mapped here;
4. export the action list in the `trtl_fw_application` data structure;

The framework does not execute actions automatically. Once the actions are declared, the user must ask the framework to dispatch incoming actions. This is performed by `trtl_fw_mq_action_dispatch()`, which listens for RPC messages on a given mq. Here an example.:

```
#include <mockturtle-framework.h>

/* [POINT 1] */
enum id_actions {
    MY_ACTION_ID = 0,
    /* ... */
};

/* [POINT 2] */
static int my_action(struct trtl_msg *msg_i, struct trtl_msg *msg_o)
{
    /* ... */
}

/* [POINT 3] */
static trtl_fw_action_t *actions[] = {
    [MY_ACTION_ID] = my_action,
    /* ... */
};

static int mymain()
{
    int err;

    while (1) {
        /* ... */
        err = trtl_fw_mq_action_dispatch(TRTL_HMQ, 0);
        /* ... */
    }

    return 0;
}


/* POINT [4]*/
```

(continues on next page)

(continued from previous page)

```
struct trtl_application app = {
    /* ... */

    /* Export Actions */
    .actions = actions,
    .n_actions = ARRAY_SIZE(actions),

    .main = mymain,
};
```

typedef trtl_fw_action_t
Action prototype type

The header for the output message is prepared by the framework. The header's fields that the user should touch are

Return 0 on success. -1 on error

Parameters

- `msg_i`: input message
- `msg_o`: output message

On error the message will be sent anyway to the host. This is just in case of future development.

int **trtl_fw_mq_action_dispatch**(**enum** *trtl_mq_type type*, unsigned int *idx_mq*)
It dispatch messages coming from a given message queue. If the message is not acceptable, this function will discard it.

Return 0 on success. -1 on error

Parameters

- `type`: MQ type
- `idx_mq`: MQ index

Variables

The firmware framework offers the possibility to export local variables to the host system. Variables must be declared using the `trtl_fw_variable` and then exported in your `trtl_fw_application`.

The mean of variable in this context is extended to any memory location: local variable, Mock Turtle registers, device peripheral registers and so on.

The framework handles the variable exchange as a special action. Internally, the framework defines actions to write and to read variables. This implies that `trtl_fw_mq_action_dispatch()` must be used to dispatch incoming actions:

```
#include <mockturtle-framework.h>

#define REGISTER_TAI_SEC (CPU_LR_BASE + 0xC)
static int var1;
static int var2;

struct trtl_fw_variable variables[] = {
    [0] = {
        .addr = (void *)&var1,
        .mask = 0xFFFFFFFF,
        .offset = 0,
        .flags = 0,
    },
    [1] = {
```

(continues on next page)

(continued from previous page)

```
        .addr = (void *)&var2,
        .mask = 0xFFFFFFFF,
        .offset = 0,
        .flags = 0,
    },
    [2] = {
        .addr = (void *)REGISTER_TAI_SEC,
        .mask = 0xFFFFFFFF,
        .offset = 0,
        .flags = 0,
    },
};

static int mymain()
{
    /* ... */

    while (1) {
        trtl_fw_mq_action_dispatch(TRTL_HMQ, 0);
        /* ... */
    }

    return 0;
}

struct trtl_fw_application app = {
    /* ... */

    .variables = variables,
    .n_variable = ARRAY_SIZE(variables),

    .main = mymain,
};
```

The user can define any number of variables, the firmware framework does not impose any constraint.

From the host you can read/write the variable by using the *Mock Turtle Variable* tool.

```
$ mockturtle-variable -D 0xdead -i 0 read 0 1 2
[0] 0x00000100
[1] 0x00123fcb
[2] 0x003f42ca
```

In chapter *Host Message Queue and Messages* you can find the correspondent host API, which in few words consists in two function: `trtl_fw_variable_set()` and `trtl_fw_variable_get()`.

What we have seen till now it is handled automatically by this framework. The user can send, asynchronously, variables of choice using the function `trtl_fw_mq_send_buf()`.

int **trtl_fw_mq_send_uint32**(**enum** *trtl_mq_type type*, unsigned int *idx_mq*, uint8_t *msg_id*, unsigned int *n*, ...)

It builds and it sends a message over MQ. The vargs will be copied into the payload message.

Parameters

- `type`: MQ type
- `idx_mq`: MQ index within the declaration
- `msg_id`: message identifier
- `n`: number of vargs

Buffers

The firmware framework offers the possibility to export local buffers to the host system. The buffers must be declared using the `trtl_fw_buffer` and then exported in your `trtl_fw_application`.

The mean of buffer in this context is extended to any contiguous memory location.

The framework handles the buffer exchange as a special action. Internally, the framework defines actions to write and to read buffers. This implies that `trtl_fw_mq_action_dispatch()` must be used to dispatch incoming actions:

```
#include <mockturtle-framework.h>

static int buf1[32];
static int buf2[16];

struct trtl_fw_buffer buffers[] = {
    [0] = {
        .buf = buf1,
        .size = sizeof(buf1),
    },
    [1] = {
        .buf = buf2,
        .size = sizeof(buf2),
    },

};

static int mymain()
{
    /* ... */

    while (1) {
        trtl_fw_mq_action_dispatch(TRTL_HMQ, 0);
        /* ... */
    }

    return 0;
}

struct trtl_fw_application app = {
    /* ... */

    .buffers = buffers,
    .n_buffer = ARRAY_SIZE(buffers),

    .main = mymain,
};
```

The user can define any number of buffers, the firmware framework does not impose any constraint.

From the host you can read/write the buffer by using the *Mock Turtle Buffer* tool.

```
$ mockturtle-buffer -D 0xdead -i 0 read 0 32 1 16
Buffer 0 (32 bytes)

0000 : 0x00000001 0x34597332 0x12393903 0xf423a4c4
0004 : 0x00000005 0x32432ffe 0x432bbff3 0x2232342b

Buffer 1 (16 bytes)

0000 : 0x03fffffe 0x07fffffe 0x0ffffffe 0x1ffffffe
```

In chapter *Host Message Queue and Messages* you can find the correspondent host API, which in few words

consists in two function: `trtl_fw_buffer_set()` and `trtl_fw_buffer_get()`.

What we have seen till now it is handled automatically by this framework. The user can send, asynchronously, buffer of choice using the function `trtl_fw_mq_send_buf()`.

int **trtl_fw_mq_send_buf**(**enum** *trtl_mq_type type*, unsigned int *idx_mq*, uint8_t *msg_id*, unsigned int *n*, void **data*)

It builds and it sends a message over MQ. The buffer will be copied into the payload message. Beware that, internally, it uses trtl_fw_mq_send().

Parameters

- `type`: MQ type
- `idx_mq`: MQ index within the declaration
- `msg_id`: message identifier
- `n`: buffer size in bytes
- `data`: buffer to send

Utilities

This is a collection of miscellaneous function that can be of some helpers.

void **trtl_fw_time**(uint32_t **seconds*, uint32_t **cycles*)

It get the current time from the internal TRTL timer

Parameters

- `seconds`:
- `cycles`:

void **trtl_fw_message_error**(**struct** *trtl_fw_msg* **msg*, int *err*)

It makes the given message an error message

CHAPTER

FIVE

THE MOCK TURTLE TOOLS

This section describes the Mock Turtle tools. The description is limited to the main purpose of the tool, for more details use the tool's help message. All tools are available in the directory *software/tools*.

5.1 Mock Turtle List

The Mock Turtle List application (*lsmockturtle*) shows the list of available Mock Turtle devices. Optionally it shows the configuration ROM content of each device.

5.2 Mock Turtle Project Creator

The process of setting up a new Mock Turtle project can takes hours the first time if you do not know already what it is expected from the user. Fortunatelly, this process can be automated and there is not much knowledge in it.

The Mock Turtle Project Creator (*mockturtle-project-creator*) is a Python script that creates a basic Mock Turtle project. This project should be used to start the development of your project.

The generated Mock Turtle project is based on a template and it includes:

- a project library on top of *The Mock Turtle Linux Library* for the development of software support layer on Linux host;
- a basic firmware based on *The Mock Turtle Firmware Framework*;
- Makefiles to compile the project

Optionally, the mockturtle-project-creator can generate a git repository with some initial commits. This will give the possibility to rollback to the skeleton whenever you want.

5.3 Mock Turtle Firmware Loader

The Mock Turtle Loader application (*mockturtle-firmware-loader*) allows the user to load a firmware in a soft-cpu. It gives also the possibility to dump the RAM content from a soft-cpu.

5.4 Mock Turtle Messages

The Mock Turtle Messages application (*mockturtle-messages*) can be used to sniff the traffic over a hmq or to access the serial console of a soft-cpu.

5.5 Mock Turtle Shared Memory

The Mock Turtle Shared Memory application (*mockturtle-smem*) provides access to the Mock Turtle shm. The user can read/write any location of the shm using different access modes.

5.6 Mock Turtle CPU Restart

The Mock Turtle CPU Restart application (*mockturtle-cpu-restart*) is used to restart a soft-cpu. This will stop the firmware execution and start it again from the `main()` function.

5.7 Mock Turtle Ping

The purpose of the Mock Turtle Ping application (*mockturtle-ping*) is to be able to verify that a firmware is running. In addition, the mockturtle-ping application provides information about the firmware version running on a Mock Turtle soft-cpu. The tool works only with those firmwares developed using *The Mock Turtle Firmware Framework*.

5.8 Mock Turtle Variable

The Mock Turtle variable application(*mockturtle-variable*) allows the user to read/write the variables that a firmware exports. The tool works only with those firmwares developed using *The Mock Turtle Firmware Framework*.

5.9 Mock Turtle Buffer

The Mock Turtle buffer application(*mockturtle-buffer*) allows the user to read/write the buffers that a firmware exports. The tool works only with those firmwares developed using *The Mock Turtle Firmware Framework*.

5.10 Mock Turtle Debugger

The Mock Turtle Debugger agent (*mockturtle-debug.py*) is used to debug a CPU with the cross gdb (*riscv32-elf-gdb*).

The debugger agent is a standalone gdb server that implements the remote protocol. It is standalone because it doesn't need any linux device driver and relies of the Universal Access Library (UAL) to do hardware accesses (See https://gitlab.cern.ch/cohtdrivers/ual).

You need to start the debug agent on the host that drives the Mock Turtle.

The debug agent will create a TCP socket and listen on it on port 3000 for a connection from gdb. It displays this message before listening:

```
Waiting for connection on port 3000
```

At that point you can start the cross debugger on the host where you have compiled your program:

```
riscv32-elf-gdb ./myprog.elf
```

Then you have to connect to the debug agent (replace *address* by the name of the target machine which runs the debug agent):

```
(gdb) target remote address:3000
```

It is possible to load the program directly, and you should load before restarting a program to be sure about the state of it:

```
(gdb) load
```

The execution is started with the *continue* command:

```
(gdb) c
```

The usual gdb commands are available: you can set a breakpoint, display the backtrace, inspect registers, display a variable, do a single step, execute until the next line... Refer to the gdb manual or any gdb tutorial.

CHAPTER

SIX

THE DEMOS

This is a collection of demo applications which main purpose is to introduce the users to the Mock Turtle development. In the following demos you will find some example code to run and test the applications.

You will notice the usage of environment variable; these variables, of course, depend of your environment. Here a list of used variable

TRTL This is the path to the root directory of the Mock Turtle project.

CROSS_COMPILE_TARGET This is the path to the cross-compiler for the soft-CPU used by Mock Turtle.

DEVID This is the device-id that uniquely identify a Mock Turtle instance. This is an integer number in hexadecimal representation (e.g. 0x0201)

CPU_INDEX This is used to select a Mock Turtle soft-CPU starting from 0.

TTYTRTL This is the path to the TTY device in /dev (e.g. /dev/ttyTRTL0)

DEMO This is the path to the demo application directory that you can find in the `software/demos` main directory.

In principle you can compile all the demos by running *make* in the main directory. Then you can load the firmware using *Mock Turtle Firmware Loader* and restart the CPU with *Mock Turtle CPU Restart*:

```
# Compile
make -C $DEMO
# Program
mockturtle-loader -D $DEVID -i $CPU_INDEX -f $DEMO/firmware/fw01/hello_world.bin
# Restart and start execution
mockturtle-cpu-restart -D $DEVID -i $CPU_INDEX
```

6.1 The *Hello World* Demo

The *Hello World* demo is a firmware program that prints over the serial interface the string `"Hello World"` and exit.

This program makes use of the *The Mock Turtle Firmware Library*.

```
minicom -D $TTYTRTL
# On a different shell instance
mockturtle-cpu-restart -D $DEVID -i $CPU_INDEX
```

```
#include <mockturtle-rt.h>

int main()
{
        pp_printf("Hello World!\r\n");
        return 0;
}
```

There is also the *Hello World* demo based on the *The Mock Turtle Firmware Framework*. This demo will print on the serial interface general informations about the firmware application.

```
minicom -D $TTYTRTL
# On a different shell instance
mockturtle-cpu-restart -D $DEVID -i $CPU_INDEX
```

```
/*
 * Copyright (C)
 * Author:
 * License:
 */

#include <mockturtle-framework.h>

static int frm_init()
{
        pp_printf("Hello world!\r\n");

        return 0;
}

/**
 * Well, the main :)
 */
static int frm_main()
{
        while (1) {
                /*
                 * Handle all messages incoming from slot 0
                 * as actions
                 */
                trtl_fw_mq_action_dispatch(TRTL_HMQ, 0);
        }

        return 0;
}


struct trtl_fw_application app = {
        .name = "hellofrm",
        .version = {
                .rt_id = CONFIG_RT_APPLICATION_ID,
                .rt_version = RT_VERSION(0, 1),
                .git_version = GIT_VERSION,
        },

        .init = frm_init,
        .main = frm_main,
};
```

6.2 The *Data Generator* Demo

The *Data Generator* demo is a firmware program that pretends to be a little acquisition system. It makes use of the *Host Message Queue* to receive configuration options and to send messages to the host system.

This program makes use of the *The Mock Turtle Firmware Framework* and shows the use of *variables* and *buffers*. The application exports to the host a set of local variables which are used to configure the application; it also exports buffers for data array or structures.

```
mockturtle-variable -D $DEVID
mockturtle-buffer -D $DEVID
```

The application periodically generates data. The generation period can be adjusted using the variable DG_PERIOD_UPDATE. The application generates sequential values which can be adjusted using gain and offset; these 2 parameters are part of a data structure exported with the buffer DG_CONF. Finally, it is possible to read the data from the buffer DG_DATA.

The communication with the host happens through the message queues as described in the following table.

Direction	Index	Description
Input	0	Receive configuration from the host
Output	0	Send configuration to the host
Output	1	Send data to the host

```
/*
 * Copyright (C) CERN 2016
 * Author: Federico Vaga <federico.vaga@cern.ch>
 * License: GPLv3
 */

#include <mockturtle-framework.h>

static uint32_t period_c;
static unsigned int period = 1000000;

#define DG_BUF_SIZE (32)
static uint32_t dg_data[DG_BUF_SIZE];


struct dg_conf {
        uint32_t gain;
        uint32_t offset;
};
static struct dg_conf dg_conf;

static uint32_t dg_last_sample_idx;
static uint32_t dg_last_sample = 1;

enum dg_variable {
        DG_PERIOD_UPDATE = 0,
};

static struct trtl_fw_variable variables[] = {
        [DG_PERIOD_UPDATE] = {
                .addr = (void *)&period,
                .mask = 0xFFFFFFFF,
                .offset = 0,
                .flags = 0,
        },
};

enum dg_structures {
        DG_DATA = 0,
        DG_CONF,
};

struct trtl_fw_buffer buffers[] = {
        [DG_DATA] = {
                .buf = dg_data,
```

(continues on next page)

(continued from previous page)

```
                .len = sizeof(dg_data),
        },
        [DG_CONF] = {
                .buf = &dg_conf,
                .len = sizeof(struct dg_conf)
        },
};


static void generate_sample(void)
{
        if (dg_last_sample == 0)
                dg_last_sample = 1;

        dg_last_sample++;
        if (dg_last_sample * dg_conf.gain < dg_last_sample)
                dg_last_sample = 1;
        dg_last_sample = (dg_last_sample * dg_conf.gain) + dg_conf.offset;


        pr_debug("%s:%d [%"PRIu32"d/%d] = %"PRIu32"u\n\r", __func__, __LINE__,
                 dg_last_sample_idx + 1, DG_BUF_SIZE, dg_last_sample);

        dg_data[dg_last_sample_idx] = dg_last_sample;

        dg_last_sample_idx++;
        dg_last_sample_idx &= (DG_BUF_SIZE - 1);
}

static void dg_update(void)
{
        if ((--period_c) == 0) {
                period_c = period;
                generate_sample();
        }
}


/**
 * Firmware initialization
 */
static int dg_init(void)
{
        period_c = period;
        dg_last_sample_idx = 0;
        dg_last_sample = 1;
        dg_conf.offset = 0;
        dg_conf.gain = 1;

        return 0;
}


/**
 * Well, the main :)
 */
static int dg_main()
{
        while (1) {
                /* Handle all messages incoming from HMQ 0 as actions */
                trtl_fw_mq_action_dispatch(TRTL_HMQ, 0);
```

(continues on next page)

(continued from previous page)

```
                dg_update();
        }

        return 0;
}


struct trtl_fw_application app = {
        .name = "data-gen",
        .version = {
                .rt_id = CONFIG_RT_APPLICATION_ID,
                .rt_version = RT_VERSION(0, 1),
                .git_version = GIT_VERSION
        },

        .variables = variables,
        .n_variables = ARRAY_SIZE(variables),

        .buffers = buffers,
        .n_buffers = ARRAY_SIZE(buffers),

        .init = dg_init,
        .main = dg_main,
};
```

6.3 The *Alarm Clock* Demo

The *Alarm Clock* demo is a firmware program that makes use of the *Host Message Queue* to receive configuration options and to send messages to the host system.

This program makes use of the *The Mock Turtle Firmware Framework* and shows the use of *variables*. The application exports to the host a set of local variables which are used to configure the application.

```
mockturtle-variable -D $DEVID
```

The application counts the number of internal iterations; this is represented by the AC_TIME variable. According to the local variable AC_PERIOD_UPDATE it periodically sends messages to the host system to notify the current time. Using the variables AC_ALARM_EN and AC_ALARM_ITER, It is possible to enable and configure an alarm which will produce a message.

The communication with the host happens through the message queues as described in the following table.

Direction	Index	Description
Input	0	Receive configuration from the host
Output	0	Send configuration to the host
Output	1	Send notifications

```
/*
 * Copyright (C) CERN 2016
 * Author: Federico Vaga <federico.vaga@cern.ch>
 * License: GPLv3
 */

#include <mockturtle-framework.h>

static unsigned int iteration;
```

(continues on next page)

(continued from previous page)

```
static uint32_t period_c;
static unsigned int period = 1000000;
static unsigned int alarm_enable;
static uint32_t alarm_iter;

enum ac_variable {
        AC_TIME = 0,
        AC_PERIOD_UPDATE,
        AC_ALARM_EN,
        AC_ALARM_ITER,
};

static struct trtl_fw_variable variables[] = {
        [AC_TIME] = {
                .addr = (void *)&iteration,
                .mask = 0xFFFFFFFF,
                .offset = 0,
                .flags = 0,
        },
        [AC_PERIOD_UPDATE] = {
                .addr = (void *)&period,
                .mask = 0xFFFFFFFF,
                .offset = 0,
                .flags = 0,
        },
        [AC_ALARM_EN] = {
                .addr = (void *)&alarm_enable,
                .mask = 0xFFFFFFFF,
                .offset = 0,
                .flags = 0,
        },
        [AC_ALARM_ITER] = {
                .addr = (void *)&alarm_iter,
                .mask = 0xFFFFFFFF,
                .offset = 0,
                .flags = 0,
        },
};



static void ac_update(void)
{
        uint32_t sec, cyc;

        trtl_fw_time(&sec, &cyc);

        if ((--period_c) == 0) {
                period_c = period;
                trtl_fw_mq_send_uint32(TRTL_HMQ, 0, 0x12, 1,
                                       iteration);
                pr_debug("Iteration %d\n\r", iteration);
        }

        if (alarm_enable) {
                if (alarm_iter < iteration) {
                        trtl_fw_mq_send_uint32(TRTL_HMQ, 0, 0x34, 2,
                                               iteration, alarm_iter);
                        alarm_enable = 0;
                        alarm_iter = 0;
                }
```

(continues on next page)

(continued from previous page)

```
        }

        iteration++;
}


/**
 * Firmware initialization
 */
static int ac_init(void)
{
        period_c = period;

        return 0;
}


/**
 * Well, the main :)
 */
static int ac_main(void)
{
        while (1) {
                /* Handle all messages incoming from HMQ 0 as actions */
                trtl_fw_mq_action_dispatch(TRTL_HMQ, 0);

                ac_update();
        }

        return 0;
}


struct trtl_fw_application app = {
        .name = "alarm-clk",
        .version = {
                .rt_id = CONFIG_RT_APPLICATION_ID,
                .rt_version = RT_VERSION(0, 1),
                .git_version = GIT_VERSION
        },

        .variables = variables,
        .n_variables = ARRAY_SIZE(variables),

        .init = ac_init,
        .main = ac_main,
};
```

6.4 The *FMC SVEC* Demo

The *FMC SVEC* demo is a complete demo that uses hardware features from the FMC SVEC carrier. This demo offers an example of all layers, so it is a good starting point to understand how to create a complete Mock Turtle application. Apart from the board itself, no other hardware is necessary to run the demo.

The main aim of this demo is to handle the SVEC LEDs and LEMOs. The LEDs can be turned *on* (*red*, *green*, *orange*) and *off*. The LEMOs can be set to *input* or *output*; when output they can be set to *high* or *low* voltage; when input it is possible to read their status (*high* or *low*).

6.4.1 HDL Code

The top-level VHDL entity of the demo can be found under *hdl/top/svec_mt_demo/svec_mt_demo.vhd*, while an Hdlmake project file (able to produce an FPGA bitstream of the demo) is available under *hdl/syn/svec_mt_demo/Manifest.py*.

The SVEC demo defines the following *Mock Turtle Configuration*:

```
constant c_MT_CONFIG : t_mt_config := (
  app_id     => x"d330d330",
  cpu_count  => 2,
  cpu_config => (others =>
                 (memsize => 8192,
                  hmq_config => (2, (0      => (7, 3, 2, x"0000_0000"),
                                     1      => (5, 4, 3, x"0000_0000"),
                                     others => (c_DUMMY_MT_MQUEUE_SLOT))),
                  rmq_config => (1, (0      => (7, 2, 2, x"0000_0000"),
                                     others => (c_DUMMY_MT_MQUEUE_SLOT))))),
  shared_mem_size => 2048);
```

The above configuration instantiates two soft CPUs, each with two host message queues (of different sizes) and one remote message queue.

The SVEC demo *Mock Turtle Instantiation* is done using the following VHDL code:

```
U_Mock_Turtle : mock_turtle_core
  generic map (
    g_CONFIG            => c_MT_CONFIG,
    g_WITH_WHITE_RABBIT => FALSE)
  port map (
    clk_i           => clk_sys,
    rst_n_i         => rst_n_sys,
    dp_master_o     => dp_wb_out,
    dp_master_i     => dp_wb_in,
    host_slave_i    => cnx_master_out(c_SLAVE_MT),
    host_slave_o    => cnx_master_in(c_SLAVE_MT),
    rmq_src_o       => rmq_ds_o,
    rmq_src_i       => rmq_ds_i,
    rmq_snk_o       => rmq_us_o,
    rmq_snk_i       => rmq_us_i,
    hmq_in_irq_o    => mt_hmq_in_irq,
    hmq_out_irq_o   => mt_hmq_out_irq,
    notify_irq_o    => mt_notify_irq,
    console_irq_o   => mt_console_irq);
```

All unconnected inputs will get their default values.

The Wishbone host interface is attached to a Wishbone crossbar, and from there to the VME64x host interface. All interrupt lines are driven into a Vectored Interrupt Controller (VIC). The remote message queue interfaces are simply forming a loopback (for testing).

For each one of the two configured soft CPUs, their respective DP interface is attached to a 24-bit Wishbone GPIO peripheral. The outputs from the two GPIO peripherals are logically OR'ed, while their are copies of the same signals. The mapping of these 24 signals is the following:

- GPIO0 to GPIO3: 4 LEMO I/O connectors on the front panel of the SVEC
- GPIO4 to GPIO7: not used
- GPIO8 to GPIO23: control the 8 bi-color LEDs on the front panel of the SVEC

All mentioned peripherals (WB crossbar, VIC, WB GPIO) are available as part of OHWR general-cores. This demo also uses the VME64x core to provide the host interface.

Simulation

Note: read *Simulation Testbenches* chapter in order to run a simulation.

The spec_mt_demo testbench uses the top-level module as the Device Under Test (DUT). It loads and executes a simple "Hello World" in the first CPU.

The aim of this testbench is to simply verify that the SVEC demo design is working.

The expected output from the simulation is:

```
App ID: 0xd330d330
Core count: 2
UART MSG from core 0: Hello World!
UART MSG from core 0:
```

Note: The svec_mt_demo testbench expects an already compiled software binary under *demos/hello_world/firmware/fw-01*. Please compile the software prior to running the simulation.

6.4.2 Software

This demo has two firmwares. One is named *autosvec*, the other *manualsvec*.

The *autosvec* firmware runs autonomously without any communication with the host system or a remote node and for this reason it is the simplest one It does not use *The Mock Turtle Firmware Framework* but only *The Mock Turtle Firmware Library*. This firmware does the following things:

- it turns *on* and *off* all the LEDs one after the other;
- it reproduce on LEMO connector 2 whatever state is on LEMO connector 1
- it generates square signals on LEMO connectors 3
- it generates square signals on LEMO connectors 4
- it periodically prints messages on the console with the GPIO status (LEDs and LEMOs)

```
/**
 * Copyright (C) 2015 CERN (www.cern.ch)
 * Author: Tomasz Wlostowski <tomasz.wlostowski@cern.ch>
 * Author: Federico Vaga <federico.vaga@cern.ch>
 *
 * SPDX-License-Identifier: LGPL-3.0-or-later
 */

#include <inttypes.h>
#include <string.h>
#include "mockturtle-rt.h"
#include <fw-svec-common.h>

#define GPIO_CODR 0x0
#define GPIO_SODR 0x4
#define GPIO_DDR 0x8
#define GPIO_PSR 0xc
#define PIN_LEMO_L1 1
#define PIN_LEMO_L2 0
#define PIN_LEMO_L3 3
#define PIN_LEMO_L4 2
```

(continues on next page)

(continued from previous page)

```
void gpio_set_dir(int pin, int out)
{
        uint32_t ddr = dp_readl(GPIO_DDR);
        if(out)
                ddr |= (1<<pin);
        else
                ddr &= ~(1<<pin);
        dp_writel(ddr, GPIO_DDR);
}

void gpio_set_state(int pin, int state)
{
        if(state)
                dp_writel(1<<pin, GPIO_SODR);
        else
                dp_writel(1<<pin, GPIO_CODR);
}

int gpio_get_state(int pin)
{
        return dp_readl(GPIO_PSR) & (1 << pin) ? 1 : 0;
}

void autosvec()
{
        int state, on = 1;
        uint32_t i, j = 0;

        /* Print something on the debug interface */
        pp_printf("Running autosvec\n\r");

        gpio_set_dir(PIN_LEMO_L1, 0); // Lemo L1 = input
        gpio_set_dir(PIN_LEMO_L2, 1); // Lemo L2 = output
        gpio_set_dir(PIN_LEMO_L3, 1); // Lemo L3 = output
        gpio_set_dir(PIN_LEMO_L4, 1); // Lemo L4 = output

        /* Clear all GPIOs (LEDs and LEMOs) */
        dp_writel(~0, GPIO_CODR);

        for (i = 0;; i++) {
                /* Lemo 2 follows Lemo 1 */
                state = gpio_get_state(PIN_LEMO_L1);
                gpio_set_state(PIN_LEMO_L2, state);

                /* Turn on/off leds one by one */
                if ((i & autosvec_led_period) == 0) {
                        dp_writel(1 << (j + 8), on ? GPIO_SODR : GPIO_CODR);
                        j = (j >= 16 ? 0 : j + 1);
                        on = j ? on : !on;
                }

                /* Square signal on Lemo 3 */
                if ((i & autosvec_lemo3_period) == 0) {
                        state = gpio_get_state(PIN_LEMO_L3);
                        gpio_set_state(PIN_LEMO_L3, !state);
                }

                /* Square signal on Lemo 4 */
                if ((i & autosvec_lemo4_period) == 0) {
                        state = gpio_get_state(PIN_LEMO_L4);
                        gpio_set_state(PIN_LEMO_L4, !state);
```

(continues on next page)

(continued from previous page)

```
                }

                /* GPIO status on debug interface */
                if ((i & autosvec_print_period) == 0) {
                        /* This output is not reliable for debugging purpose,
                           it's just to show that you can do it */
                        pp_printf("GPIO direction 0x%"PRIx32"\n\rGPIO 0x%"PRIx32
↪"\n\r",
                                  dp_readl(GPIO_DDR),
                                  dp_readl(GPIO_PSR));
                }

                /* Check if someone else (HOST or other RT application) want
                   to stop this execution */
                if (!autosvec_run) {
                        pp_printf("Stopping autosvec\n\r");
                        break;
                }
        }

        /* Clear all GPIOs (LEDs and LEMOs) */
        dp_writel(~0, GPIO_CODR);
        /* Set all GPIO as output */
        dp_writel(~0, GPIO_DDR);
}


int main()
{
        /* By default allow this program to run */
        smem_atomic_or(&autosvec_run, 1);
#ifndef SIMULATION
        smem_atomic_or(&autosvec_led_period, 0x1FFFF);
        smem_atomic_or(&autosvec_lemo3_period, 0x1FFFF);
        smem_atomic_or(&autosvec_lemo4_period, 0x3FFFF);
        smem_atomic_or(&autosvec_print_period, 0xFFFFF);
#else
        smem_atomic_or(&autosvec_led_period, 0x1);
        smem_atomic_or(&autosvec_lemo3_period, 0x1);
        smem_atomic_or(&autosvec_lemo4_period, 0x3);
        smem_atomic_or(&autosvec_print_period, 0x7);
#endif

        while (1) {
                /* Wait that someone (HOST or other RT application) allows
                   this program to run */
                if (!autosvec_run)
                        continue;

                autosvec();
        }
}
```

The *manualsvec* firmware offers a manual control of all LEDs and LEMOs. It does use *The Mock Turtle Firmware Framework*. This firmware does the following things:

- it exportes as *Variables* the device peripheral registers to configure LEDs and LEMOs
- to exports a local *Buffers* where the user can read and write (it is not used)
- it exports a local *variable* that can be used to stop/start an *autosvec* firmware running on a different core.

Note: Rembember that the SVEC connects LEMO 3 and LEMO 4 to the same GPIO port. This means that they must have the same direction (both input, or both output)

```
/**
 * Copyright (C) 2015 CERN (www.cern.ch)
 * Author: Federico Vaga <federico.vaga@cern.ch>
 *
 * SPDX-License-Identifier: LGPL-3.0-or-later
 */

#include <string.h>
#include <mockturtle-rt.h>
#include <fw-svec-common.h>
#include <mockturtle-framework.h>

#define GPIO_CODR        0x0 /* Clear Data Register */
#define GPIO_SODR        0x4 /* Set Data Register */
#define GPIO_DDR        0x8 /* Direction Data Register */
#define GPIO_PSR        0xC /* Status Register */

static struct svec_structure svec_struct;

struct trtl_fw_buffer svec_buffers[] = {
        [SVEC_BUF_TEST] = {
                .buf = &svec_struct,
                .len = sizeof(struct svec_structure),
        }
};

struct trtl_fw_variable svec_variables[] = {
        [SVEC_VAR_LEMO_STA] = {
                .addr = TRTL_ADDR_DP(GPIO_PSR),
                .mask = PIN_LEMO_MASK,
                .offset = 0,
        },
        [SVEC_VAR_LEMO_DIR] = {
                .addr = TRTL_ADDR_DP(GPIO_DDR),
                .mask = PIN_LEMO_MASK,
                .offset = 0,
        },
        [SVEC_VAR_LEMO_SET] = {
                .addr = TRTL_ADDR_DP(GPIO_SODR),
                .mask = PIN_LEMO_MASK,
                .offset = 0,
        },
        [SVEC_VAR_LEMO_CLR] = {
                .addr = TRTL_ADDR_DP(GPIO_CODR),
                .mask = PIN_LEMO_MASK,
                .offset = 0,
        },
        [SVEC_VAR_LED_STA] = {
                .addr = TRTL_ADDR_DP(GPIO_PSR),
                .mask = PIN_LED_MASK,
                .offset = PIN_LED_OFFSET,
        },
        [SVEC_VAR_LED_SET] = {
                .addr = TRTL_ADDR_DP(GPIO_SODR),
                .mask = PIN_LED_MASK,
                .offset = PIN_LED_OFFSET,
        },
```

(continues on next page)

(continued from previous page)

```
        [SVEC_VAR_LED_CLR] = {
                .addr = TRTL_ADDR_DP(GPIO_CODR),
                .mask = PIN_LED_MASK,
                .offset = PIN_LED_OFFSET,
        },
        [SVEC_VAR_AUTO] = {
                .addr = &autosvec_run,
                .mask = 0xFFFFFFFF,
                .offset = 0,
        },
};


/**
 * It sends messages over the debug interface
 */
static int svec_debug_interface(void)
{
        pp_printf("Hello world.\n\r");
        pp_printf("We are messages over the serial interface.\n\r");
        pp_printf("Print here your messages.\n\r");

        return 0;
}


/**
 * Well, the main :)
 */
static int svec_main()
{
        while (1) {
                /* Handle all messages incoming from HMQ 0 as actions */
                trtl_fw_mq_action_dispatch(TRTL_HMQ, 0);
        }

        return 0;
}


struct trtl_fw_application app = {
        .name = "manualsvec",
        .version = {
                .rt_id = RT_APPLICATION_ID,
                .rt_version = RT_VERSION(1, 0),
                .git_version = GIT_VERSION
        },

        .buffers = svec_buffers,
        .n_buffers = ARRAY_SIZE(svec_buffers),

        .variables = svec_variables,
        .n_variables = ARRAY_SIZE(svec_variables),

        .init = svec_debug_interface,
        .main = svec_main,
};
```

This firmware has also a support layer on the host side. This is not really necessary because you can always uses the generic Mock Turtle tools to *read/write variables* and to *read/write buffers*; but for the sake of make this demos as complete as possible we added an host support layer which is made of a C library and a C program. A part from the standard operations to open and close a device, the library exports an API to handle the LEDs and LEMOs

status and functions to set/get a dummy data structure. This library is mainly a wrapper around the Mock Turtle one.

```
/*
 * Copyright (C) 2014 CERN (www.cern.ch)
 * Author: Federico Vaga <federico.vaga@cern.ch>
 *
 * SPDX-License-Identifier: LGPL-3.0-or-later
 */

/*
 * This is just a SVEC, the code is not optimized
 */
#include <stdlib.h>
#include <unistd.h>
#include <errno.h>
#include <libmockturtle.h>
#include <libsvec-internal.h>

const char *svec_errors[] = {
        "Received an invalid answer from white-rabbit-node-code CPU",
        "Real-Time application does not acknowledge",
};


/**
 * It returns a string messages corresponding to a given error code. If
 * it is not a libwrtd error code, it will run trtl_strerror()
 * @param[in] err error code
 * @return a message error
 */
const char *svec_strerror(unsigned int err)
{
        if (err < ESVEC_INVALID_ANSWER_ACK || err >= __ESVEC_MAX_ERROR_NUMBER)
                return trtl_strerror(err);

        return svec_errors[err - ESVEC_INVALID_ANSWER_ACK];
}


/**
 * It initializes the SVEC library. It must be called before doing
 * anything else.
 * This library is based on the libmockturtle, so internally, this function also
 * run svec_init() in order to initialize the WRNC library.
 * @return 0 on success, otherwise -1 and errno is appropriately set
 */
int svec_init()
{
        int err;

        err = trtl_init();
        if (err)
                return err;

        return 0;
}


/**
 * It releases the resources allocated by svec_init(). It must be called when
 * you stop to use this library. Then, you cannot use functions from this
```

(continues on next page)

(continued from previous page)

```
 * library.
 */
void svec_exit()
{
        trtl_exit();
}


/**
 * Open a WRTD node device using ID ID
 * @param[in] device_id ID device identificator
 * @return It returns an anonymous svec_node structure on success.
 *         On error, NULL is returned, and errno is set appropriately.
 */
struct svec_node *svec_open_by_id(uint32_t device_id)
{
        struct svec_desc *svec;

        svec = malloc(sizeof(struct svec_desc));
        if (!svec)
                return NULL;

        svec->trtl = trtl_open_by_id(device_id);
        if (!svec->trtl)
                goto out;

        svec->dev_id = device_id;
        return (struct svec_node *)svec;

out:
        free(svec);
        return NULL;
}


/**
 * It closes a SVEC device opened with one of the following function:
 * svec_open_by_id()
 * @param[in] dev device token
 */
void svec_close(struct svec_node *dev)
{
        struct svec_desc *svec = (struct svec_desc *)dev;

        trtl_close(svec->trtl);
        free(svec);
        dev = NULL;
}


/**
 * It returns the WRNC token in order to allows users to run
 * functions from the WRNC library
 * @param[in] dev device token
 * @return the WRNC token
 */
struct trtl_dev *svec_get_trtl_dev(struct svec_node *dev)
{
        struct svec_desc *svec = (struct svec_desc *)dev;

        return (struct trtl_dev *)svec->trtl;
```

(continues on next page)

(continued from previous page)

```
}

int svec_lemo_dir_set(struct svec_node *dev, uint32_t value)
{
        struct svec_desc *svec = (struct svec_desc *)dev;
        uint32_t fields[] = {SVEC_VAR_LEMO_DIR, value};

        return trtl_fw_variable_set(svec->trtl,
                                    SVEC_CPU_MANUAL, SVEC_CPU_MANUAL_HMQ,
                                    fields, 1);
}

int svec_lemo_set(struct svec_node *dev, uint32_t value)
{
        struct svec_desc *svec = (struct svec_desc *)dev;
        uint32_t fields[] = {SVEC_VAR_LEMO_SET, value,
                             SVEC_VAR_LEMO_CLR, ~value};

        return trtl_fw_variable_set(svec->trtl,
                                    SVEC_CPU_MANUAL, SVEC_CPU_MANUAL_HMQ,
                                    fields, 2);
}


/**
 * Convert the given LEDs value with color codification
 */
static uint32_t svec_apply_color(uint32_t value, enum svec_color color)
{
        uint32_t val = 0;
        int i;

        for (i = 0; i < PIN_LED_COUNT; ++i) {
                if (!((value >> i) & 0x1))
                        continue;
                switch (color) {
                case SVEC_GREEN:
                case SVEC_RED:
                        val |= (0x1 << (color + (i * 2)));
                        break;
                case SVEC_ORANGE:
                        val |= (0x3 << ((i * 2)));
                        break;
                }
        }

        return val;
}


/**
 * Set LED's register
 */
int svec_led_set(struct svec_node *dev, uint32_t value, enum svec_color color)
{
        struct svec_desc *svec = (struct svec_desc *)dev;
        uint32_t real_value = svec_apply_color(value, color);
        uint32_t fields[] = {SVEC_VAR_LED_SET, real_value,
                             SVEC_VAR_LED_CLR, ~real_value};

        return trtl_fw_variable_set(svec->trtl,
```

(continues on next page)

(continued from previous page)

```
                                    SVEC_CPU_MANUAL, SVEC_CPU_MANUAL_HMQ,
                                    fields, 2);
}


/**
 * It gets the status of the SVEC program
 */
int svec_status_get(struct svec_node *dev, struct svec_status *status)
{
        struct svec_desc *svec = (struct svec_desc *)dev;
        uint32_t fields[] = {SVEC_VAR_LEMO_STA, 0,
                             SVEC_VAR_LED_STA, 0,
                             SVEC_VAR_LEMO_DIR, 0};
        int err;

        err = trtl_fw_variable_get(svec->trtl,
                                   SVEC_CPU_MANUAL, SVEC_CPU_MANUAL_HMQ,
                                   fields, 3);
        if (err)
                return err;

        status->lemo = fields[1];
        status->led = fields[3];
        status->lemo_dir = fields[5];

        return 0;
}

int svec_run_autosvec(struct svec_node *dev, uint32_t run)
{
        struct svec_desc *svec = (struct svec_desc *)dev;
        uint32_t fields[] = {SVEC_VAR_AUTO, run};

        return trtl_fw_variable_set(svec->trtl,
                                    SVEC_CPU_MANUAL, SVEC_CPU_MANUAL_HMQ,
                                    fields, 1);
}

int svec_version(struct svec_node *dev, struct trtl_fw_version *version)
{
        struct svec_desc *svec = (struct svec_desc *)dev;

        return trtl_fw_version(svec->trtl,
                               SVEC_CPU_MANUAL, SVEC_CPU_MANUAL_HMQ,
                               version);
}

int svec_test_struct_get(struct svec_node *dev, struct svec_structure *test)
{
        struct svec_desc *svec = (struct svec_desc *)dev;
        struct trtl_tlv tlv = {
                .type = SVEC_BUF_TEST,
                .size = sizeof(struct svec_structure),
                .buf = test,
        };

        return trtl_fw_buffer_get(svec->trtl,
                                  SVEC_CPU_MANUAL, SVEC_CPU_MANUAL_HMQ,
                                  &tlv, 1);
}
```

(continues on next page)

(continued from previous page)

```
int svec_test_struct_set(struct svec_node *dev, struct svec_structure *test)
{
        struct svec_desc *svec = (struct svec_desc *)dev;
        struct trtl_tlv tlv = {
                .type = SVEC_BUF_TEST,
                .size = sizeof(struct svec_structure),
                .buf = test,
        };

        return trtl_fw_buffer_set(svec->trtl,
                                  SVEC_CPU_MANUAL, SVEC_CPU_MANUAL_HMQ,
                                  &tlv, 1);
}
```

At the end, the host program. This program is a command line tool that uses the svec library described above to handle the SVEC board. Again, it gives users the possibility to play with LEDs and LEMOs status.

```
/*
 * Copyright (C) 2014 CERN (www.cern.ch)
 * Author: Federico Vaga <federico.vaga@cern.ch>
 *
 * SPDX-License-Identifier: GPL-3.0-or-later
 */

#include <stdint.h>
#include <stdlib.h>
#include <stdio.h>
#include <errno.h>
#include <string.h>
#include <getopt.h>
#include <libsvec.h>
#include <inttypes.h>


static void help()
{
        fprintf(stderr,
                "svec -D 0x<hex-number> -L 0x<hex-number> -l 0x<hex-number> -c
↪<char> -a <char> -s\n");
        fprintf(stderr, "-D device id\n");
        fprintf(stderr, "-l value to write into the LED register\n");
        fprintf(stderr, "-L value to write into the LEMO register\n");
        fprintf(stderr, "-d value to write into the LEMO direction register\n");
        fprintf(stderr, "-s it reports the content of LED and LEMO registers\n");
        fprintf(stderr, "-c set led color (g: green, r: red, o: orange)\n");
        fprintf(stderr, "-a set autosvec status (r: run, s: stop)\n");
        fprintf(stderr, "-v show version\n");
        fprintf(stderr, "-t send random value to the structure and read them back\n
↪");
        fprintf(stderr, "\n");
        exit(1);
}

static void svec_print_status(struct svec_status *status)
{
        fprintf(stdout, "Status:\n");
        fprintf(stdout, "\tled\t0x%x\n", status->led);
        fprintf(stdout, "\tlemo\t0x%x\n", status->lemo);
        fprintf(stdout, "\t\tdirection\t0x%x\n", status->lemo_dir);
        fprintf(stdout, "\tautosvec\t%s\n", status->autosvec ? "run" : "stop");
```

(continues on next page)

(continued from previous page)

```
}

static void svec_print_version(struct trtl_fw_version *version)
{
        fprintf(stdout, "Version:\n");
        fprintf(stdout, "\tRT: 0x%x\n", version->rt_id);
        fprintf(stdout, "\tRT Version: 0x%x\n", version->rt_version);
        fprintf(stdout, "\tGit Version: 0x%x\n", version->git_version);
}

static void svec_print_structure(struct svec_structure *test)
{
        int i;

        fprintf(stdout, "\tfield1: 0x%x\n", test->field1);
        fprintf(stdout, "\tfield2: 0x%x\n", test->field2);
        for (i = 0; i < SVEC_BUF_MAX_ARRAY; i++)
                fprintf(stdout, "\tarray[%d]: 0x%x\n", i, test->array[i]);
}

int main(int argc, char *argv[])
{
        struct svec_node *svec;
        struct svec_status status;
        uint32_t dev_id = 0;
        int led = -1, lemo = -1, lemo_dir = -1, i;
        char c, c_color = 0, autosvec = 0;
        int err = 0, show_status = 0, show_version = 0, structure = 0;
        enum svec_color color = SVEC_RED;
        struct trtl_fw_version version;
        struct svec_structure test, test_rb;

        while ((c = getopt (argc, argv, "hD:l:L:d:c:sa:vt")) != -1) {
                switch (c) {
                case 'h':
                case '?':
                        help();
                        break;
                case 'D':
                        sscanf(optarg, "0x%x", &dev_id);
                        break;
                case 'l':
                        sscanf(optarg, "0x%x", &led);
                        break;
                case 'L':
                        sscanf(optarg, "0x%x", &lemo);
                        break;
                case 'd':
                        sscanf(optarg, "0x%x", &lemo_dir);
                        break;
                case 'c':
                        sscanf(optarg, "%c", &c_color);
                        switch (c_color) {
                        case 'g':
                                color = SVEC_GREEN;
                                break;
                        case 'r':
                                color = SVEC_RED;
                                break;
                        case 'o':
                                color = SVEC_ORANGE;
```

(continues on next page)

(continued from previous page)

```
                                break;
                        }
                        break;
                case 's':
                        show_status = 1;
                        break;
                case 'a':
                        sscanf(optarg, "%c", &autosvec);
                        break;
                case 'v':
                        show_version = 1;
                        break;
                case 't':
                        structure = 1;
                        break;
                }
        }

        if (dev_id == 0) {
                help();
                exit(1);
        }

        atexit(svec_exit);
        err = svec_init();
        if (err) {
                fprintf(stderr, "Cannot init svec library: %s\n",
                        svec_strerror(errno));
                exit(1);
        }

        svec = svec_open_by_id(dev_id);
        if (!svec) {
                fprintf(stderr, "Cannot open svec: %s\n", svec_strerror(errno));
                exit(1);
        }


        /* Set autosvec status */
        if (autosvec != 0)
                svec_run_autosvec(svec, autosvec == 'r' ? 1 : 0);

        if (lemo_dir >= 0) {
                /* Set LEMO direction */
                err = svec_lemo_dir_set(svec, lemo_dir);
                if (err)
                        fprintf(stderr, "Cannot set LEMO direction: %s\n",
                                svec_strerror(errno));
        }

        if (led >= 0) {
                /* Set LED register */
                err = svec_led_set(svec, led, color);
                if (err)
                        fprintf(stderr, "Cannot set LED: %s\n", svec_
↪strerror(errno));
        }

        if (lemo >= 0) {
                /* Set LEMO register */
                err = svec_lemo_set(svec, lemo);
```

(continues on next page)

(continued from previous page)

```
                if (err)
                        fprintf(stderr, "Cannot set LEMO: %s\n", svec_
↪strerror(errno));
        }

        if (show_status) {
                /* Get the current status */
                err = svec_status_get(svec, &status);
                if (err)
                        fprintf(stderr, "Cannot get status: %s\n", svec_
↪strerror(errno));
                else
                        svec_print_status(&status);
        }

        if (show_version) {
                err = svec_version(svec, &version);
                if (err)
                        fprintf(stderr, "Cannot get version: %s\n",
                                svec_strerror(errno));
                else
                        svec_print_version(&version);
        }

        if (structure) {
                /* Generate random numbers (cannot use getrandom(2) because
                   of old system) */
                uint32_t seq = 0;
                test.field1 = seq++;
                test.field2 = seq++;
                for (i = 0; i < SVEC_BUF_MAX_ARRAY; i++)
                        test.array[i] = seq++;

                fprintf(stdout, "Generated structure:\n");
                svec_print_structure(&test);

                err = svec_test_struct_set(svec, &test);
                if (err) {
                        fprintf(stderr, "Cannot set structure: %s\n",
                                svec_strerror(errno));
                } else {
                        err = svec_test_struct_get(svec, &test_rb);
                        if (err) {
                                fprintf(stderr, "Cannot get structure: %s\n",
                                        svec_strerror(errno));
                        } else {
                                if (memcmp(&test, &test_rb, sizeof(struct svec_
↪structure))) {
                                        fprintf(stderr, "Got wrong structure: %s\n
↪",
                                                svec_strerror(errno));
                                        svec_print_structure(&test_rb);
                                } else {
                                        fprintf(stderr,
                                                "Structure correctly read back\n");
                                }
                        }
                }
        }

        svec_close(svec);
```

(continues on next page)

(continued from previous page)

```
        exit(0);
}
```

6.5 The *FMC SPEC* Demo

The *FMC SPEC* demo is a complete demo that uses hardware features from the FMC SPEC carrier. This demo offers an example of all layers, so it is a good starting point to understand how to create a complete Mock Turtle application. Apart from the board itself, no other hardware is necessary to run the demo.

The main aim of this demo is to handle the SPEC LEDs and buttons. The LEDs can be turned *on* and *off*. Buttuns' status can be read.

6.5.1 HDL Code

The top-level VHDL entity of the demo can be found under *hdl/top/spec_mt_demo/spec_mt_demo.vhd*, while an Hdlmake project file (able to produce an FPGA bitstream of the demo) is available under *hdl/syn/spec_mt_demo/Manifest.py*.

The SPEC demo defines the following *Mock Turtle Configuration*:

```
constant c_MT_CONFIG : t_mt_config := (
  app_id     => x"d331d331",
  cpu_count  => 2,
  cpu_config => (others =>
                 (memsize => 8192,
                  hmq_config => (2, (0      => (7, 3, 2, x"0000_0000"),
                                     1      => (5, 4, 3, x"0000_0000"),
                                     others => (c_DUMMY_MT_MQUEUE_SLOT))),
                  rmq_config => (1, (0      => (7, 2, 2, x"0000_0000"),
                                     others => (c_DUMMY_MT_MQUEUE_SLOT))))),
  shared_mem_size => 2048);
```

The above configuration instantiates two soft CPUs, each with two host message queues (of different sizes) and one remote message queue.

the SPEC demo *Mock Turtle Instantiation* is done using the following VHDL code:

```
U_Mock_Turtle : mock_turtle_core
  generic map (
    g_CONFIG            => c_MT_CONFIG,
    g_WITH_WHITE_RABBIT => FALSE)
  port map (
    clk_i           => clk_sys,
    rst_n_i         => rst_n_sys,
    dp_master_o     => dp_wb_out,
    dp_master_i     => dp_wb_in,
    host_slave_i    => cnx_master_out(c_SLAVE_MT),
    host_slave_o    => cnx_master_in(c_SLAVE_MT),
    rmq_src_o       => rmq_ds_o,
    rmq_src_i       => rmq_ds_i,
    rmq_snk_o       => rmq_us_o,
    rmq_snk_i       => rmq_us_i,
    hmq_in_irq_o    => mt_hmq_in_irq,
    hmq_out_irq_o   => mt_hmq_out_irq,
    notify_irq_o    => mt_notify_irq,
    console_irq_o   => mt_console_irq);
```

All unconnected inputs will get their default values.

The Wishbone host interface is attached to a Wishbone crossbar, and from there to the PCIe host interface. All interrupt lines are driven into a Vectored Interrupt Controller (VIC). The remote message queue interfaces are simply forming a loopback (for testing).

For each one of the two configured soft CPUs, their respective DP interface is attached to an 8-bit Wishbone GPIO peripheral. The outputs from the two GPIO peripherals are logically OR'ed, while their inputs are copies of the same signals. The mapping of these 8 signals is the following:

- GPIO0 to GPIO1: Two push buttons on the SPEC board
- GPIO2 to GPIO5: Four LEDs on the SPEC board
- GPIO6 to GPIO7: Two LEDs on the front panel of the SPEC

All mentioned peripherals (WB crossbar, VIC, WB GPIO) are available as part of OHWR general-cores. The SPEC demo also uses the Gennum GN4124 core to provide the host interface.

Simulation

Note: read *Simulation Testbenches* chapter in order to run a simulation.

The spec_mt_demo testbench uses the top-level module as the Device Under Test (DUT). It loads and executes a simple "Hello World" in the first CPU.

The aim of this testbench is to simply verify that the SPEC Demo design is working.

The expected output from the simulation is:

```
App ID: 0xd331d331
Core count: 2
UART MSG from core 0: Hello World!
UART MSG from core 0:
```

Note: The spec_mt_demo testbench expects an already compiled software binary under *demos/hello_world/firmware/fw-01*. Please compile the software prior to running the simulation.

6.5.2 Software

Todo: to be done

CHAPTER

SEVEN

REGISTER TABLES

7.1 Control/Status Registers (CSR)

This is a list of all the control and status registers, which are accessible from the Host. There is one CSR for the whole MT.

7.1.1 Mock Turtle CPU Control/Status registers block

Memory map summary

SW Offset	Type	Name	HW prefix	C prefix
0x0	REG	Core Reset Register	mt_cpu_csr_reset	RESET
0x4	REG	Core Notification Interrupt Register	mt_cpu_csr_int	INT
0x8	REG	Core 0-3 Notification Value Register	mt_cpu_csr_int_val_lo	INT_VAL_LO
0xc	REG	Core 4-7 Notification Value Register	mt_cpu_csr_int_val_hi	INT_VAL_HI
0x18	REG	SMEM Operation Select	mt_cpu_csr_smem_op	SMEM_OP
0x1c	REG	HMQ Select Register	mt_cpu_csr_hmq_sel	HMQ_SEL
0x40	REG	HMQ IN Status Core 0-3 Register	mt_cpu_csr_hmqi_status_lo	HMQI_STATUS_LO
0x48	REG	HMQ IN Status Core 4-7 Register	mt_cpu_csr_hmqi_status_hi	HMQI_STATUS_HI
0x60	REG	HMQ OUT Status Core 0-3 Register	mt_cpu_csr_hmqo_status_lo	HMQO_STATUS_LO
0x68	REG	HMQ OUT Status Core 4-7 Register	mt_cpu_csr_hmqo_status_hi	HMQO_STATUS_HI
0x80	REG	HMQ IN Interrupt Enable Core 0-3 Register	mt_cpu_csr_hmqi_inten_lo	HMQI_INTEN_LO
0x88	REG	HMQ IN Interrupt Enable Core 4-7 Register	mt_cpu_csr_hmqi_inten_hi	HMQI_INTEN_HI
0x90	REG	HMQ OUT Interrupt Enable Core 0-3 Register	mt_cpu_csr_hmqo_inten_lo	HMQO_INTEN_LO
0x98	REG	HMQ OUT Interrupt Enable Core 4-7 Register	mt_cpu_csr_hmqo_inten_hi	HMQO_INTEN_HI
0xc0	REG	Core Select Register	mt_cpu_csr_core_sel	CORE_SEL
0xc4	REG	Core Upload Address Register	mt_cpu_csr_uaddr	UADDR
0xc8	REG	Core Upload Data Register	mt_cpu_csr_udata	UDATA
0x100	REG	Core Serial Console Message Register	mt_cpu_csr_uart_msg	UART_MSG
0x104	REG	Core Serial Console Message Poll Register	mt_cpu_csr_uart_poll	UART_POLL
0x108	REG	Core Serial Console Message Interrupt Mask Register	mt_cpu_csr_uart_imsk	UART_IMSK
0x180	REG	Debug Interface Status Register	mt_cpu_csr_dbg_status	DBG_STATUS
0x184	REG	Debug Interface Force Register	mt_cpu_csr_dbg_force	DBG_FORCE
0x188	REG	Debug Interface Instruction Ready Register	mt_cpu_csr_dbg_insn_ready	DBG_INSN_READY
0x18c	REG	Debug Interface Core[0] Instruction Register	mt_cpu_csr_dbg_core0_insn	DBG_CORE0_INSN
0x190	REG	Debug Interface Core[1] Instruction Register	mt_cpu_csr_dbg_core1_insn	DBG_CORE1_INSN
0x194	REG	Debug Interface Core[0] Mailbox Data Register	mt_cpu_csr_dbg_core0_mbx	DBG_CORE0_MBX
0x198	REG	Debug Interface Core[1] Mailbox Data Register	mt_cpu_csr_dbg_core1_mbx	DBG_CORE1_MBX

Register description

Core Reset Register

HW prefix: mt_cpu_csr_reset
HW address: 0x0
SW prefix: RESET
SW offset: 0x0

31	30	29	28	27	26	25	24
-	-	-	-	-	-	-	-
23	22	21	20	19	18	17	16
-	-	-	-	-	-	-	-
15	14	13	12	11	10	9	8
-	-	-	-	-	-	-	-
7	6	5	4	3	2	1	0
RESET[7:0]							

- **RESET** [*read/write*]: Core reset lines

Core Notification Interrupt Register

HW prefix: mt_cpu_csr_int
HW address: 0x1
SW prefix: INT
SW offset: 0x4

31	30	29	28	27	26	25	24
-	-	-	-	-	-	-	-
23	22	21	20	19	18	17	16
-	-	-	-	-	-	-	-
15	14	13	12	11	10	9	8
-	-	-	-	-	-	-	-
7	6	5	4	3	2	1	0
INT[7:0]							

- **INT** [*read/write*]: Notification interrupt lines. Cleared on read.
  Each notification interrupt line has an associated 8-bit notification value, available in the INT_VAL_LO and INT_VAL_HI registers below.

Core 0-3 Notification Value Register

HW prefix: mt_cpu_csr_int_val_lo
HW address: 0x2
SW prefix: INT_VAL_LO
SW offset: 0x8

31	30	29	28	27	26	25	24
INT_VAL_LO[31:24]							
23	22	21	20	19	18	17	16
INT_VAL_LO[23:16]							
15	14	13	12	11	10	9	8
INT_VAL_LO[15:8]							
7	6	5	4	3	2	1	0
INT_VAL_LO[7:0]							

- **INT_VAL_LO** [*read-only*]: 8-bit notification values from cores 0-3.

Core 4-7 Notification Value Register

HW prefix: mt_cpu_csr_int_val_hi
HW address: 0x3
SW prefix: INT_VAL_HI
SW offset: 0xc

31	30	29	28	27	26	25	24
INT_VAL_HI[31:24]							
23	22	21	20	19	18	17	16
INT_VAL_HI[23:16]							
15	14	13	12	11	10	9	8
INT_VAL_HI[15:8]							
7	6	5	4	3	2	1	0
INT_VAL_HI[7:0]							

- **INT_VAL_HI** [*read-only*]: 8-bit otification values from cores 4-7.

SMEM Operation Select

HW prefix: mt_cpu_csr_smem_op
HW address: 0x6
SW prefix: SMEM_OP
SW offset: 0x18

31	30	29	28	27	26	25	24
-	-	-	-	-	-	-	-
23	22	21	20	19	18	17	16
-	-	-	-	-	-	-	-
15	14	13	12	11	10	9	8
-	-	-	-	-	-	-	-
7	6	5	4	3	2	1	0
-	-	-	-	-	SMEM_OP[2:0]		

- **SMEM_OP** [*read/write*]: Operation code
  Selects the operation mode for Shared Memory writes from host.
  When reading, the operation mode is ignored (and always treated as a direct access).
  Accepted values:
  0x0: direct

0x1: add
0x2: subtract
0x3: bit set
0x4: bit clear
0x5: bit flip
0x6: test and set

HMQ Select Register

HW prefix: mt_cpu_csr_hmq_sel
HW address: 0x7
SW prefix: HMQ_SEL
SW offset: 0x1c

Select the active HMQ for accessing the GCR registers of that queue.

31	30	29	28	27	26	25	24
-	-	-	-	-	-	-	-
23	22	21	20	19	18	17	16
-	-	-	-	-	-	-	-
15	14	13	12	11	10	9	8
-	-	-	-	-	CORE[2:0]		
7	6	5	4	3	2	1	0
-	-	-	-	-	QUEUE[2:0]		

- **QUEUE** [*read/write*]: Queue select
- **CORE** [*read/write*]: Core select

HMQ IN Status Core 0-3 Register

HW prefix: mt_cpu_csr_hmqi_status_lo
HW address: 0x10
SW prefix: HMQI_STATUS_LO
SW offset: 0x40

31	30	29	28	27	26	25	24
HMQI_STATUS_LO[31:24]							
23	22	21	20	19	18	17	16
HMQI_STATUS_LO[23:16]							
15	14	13	12	11	10	9	8
HMQI_STATUS_LO[15:8]							
7	6	5	4	3	2	1	0
HMQI_STATUS_LO[7:0]							

- **HMQI_STATUS_LO** [*read-only*]: HMQ IN status core 0-3.
  Returns 1 when the respective queue is not empty.

HMQ IN Status Core 4-7 Register

HW prefix: mt_cpu_csr_hmqi_status_hi
HW address: 0x12
SW prefix: HMQI_STATUS_HI
SW offset: 0x48

31	30	29	28	27	26	25	24
HMQI_STATUS_HI[31:24]							
23	22	21	20	19	18	17	16
HMQI_STATUS_HI[23:16]							
15	14	13	12	11	10	9	8
HMQI_STATUS_HI[15:8]							
7	6	5	4	3	2	1	0
HMQI_STATUS_HI[7:0]							

- **HMQI_STATUS_HI** [*read-only*]: HMQ IN status core 4-7.
  Returns 1 when the respective queue is not empty.

HMQ OUT Status Core 0-3 Register

HW prefix: mt_cpu_csr_hmqo_status_lo
HW address: 0x18
SW prefix: HMQO_STATUS_LO
SW offset: 0x60

31	30	29	28	27	26	25	24
HMQO_STATUS_LO[31:24]							
23	22	21	20	19	18	17	16
HMQO_STATUS_LO[23:16]							
15	14	13	12	11	10	9	8
HMQO_STATUS_LO[15:8]							
7	6	5	4	3	2	1	0
HMQO_STATUS_LO[7:0]							

- **HMQO_STATUS_LO** [*read-only*]: HMQ OUT status core 0-3.
  Returns 1 when the respective queue is not full.

HMQ OUT Status Core 4-7 Register

HW prefix: mt_cpu_csr_hmqo_status_hi
HW address: 0x1a
SW prefix: HMQO_STATUS_HI
SW offset: 0x68

31	30	29	28	27	26	25	24
HMQO_STATUS_HI[31:24]							
23	22	21	20	19	18	17	16
HMQO_STATUS_HI[23:16]							
15	14	13	12	11	10	9	8
HMQO_STATUS_HI[15:8]							
7	6	5	4	3	2	1	0
HMQO_STATUS_HI[7:0]							

- **HMQO_STATUS_HI** [*read-only*]: HMQ OUT status core 4-7.
  Returns 1 when the respective queue is not full.

HMQ IN Interrupt Enable Core 0-3 Register

HW prefix: mt_cpu_csr_hmqi_inten_lo
HW address: 0x20
SW prefix: HMQI_INTEN_LO
SW offset: 0x80

31	30	29	28	27	26	25	24
HMQI_INTEN_LO[31:24]							
23	22	21	20	19	18	17	16
HMQI_INTEN_LO[23:16]							
15	14	13	12	11	10	9	8
HMQI_INTEN_LO[15:8]							
7	6	5	4	3	2	1	0
HMQI_INTEN_LO[7:0]							

- **HMQI_INTEN_LO** [*read/write*]: HMQ IN interrupt enable for core 0-3.
  Set to 1 to enable interrupts from respective queue.

HMQ IN Interrupt Enable Core 4-7 Register

HW prefix: mt_cpu_csr_hmqi_inten_hi
HW address: 0x22
SW prefix: HMQI_INTEN_HI
SW offset: 0x88

31	30	29	28	27	26	25	24
HMQI_INTEN_HI[31:24]							
23	22	21	20	19	18	17	16
HMQI_INTEN_HI[23:16]							
15	14	13	12	11	10	9	8
HMQI_INTEN_HI[15:8]							
7	6	5	4	3	2	1	0
HMQI_INTEN_HI[7:0]							

- **HMQI_INTEN_HI** [*read/write*]: HMQ IN interrupt enable for core 4-7.
  Set to 1 to enable interrupts from respective queue.

HMQ OUT Interrupt Enable Core 0-3 Register

HW prefix: mt_cpu_csr_hmqo_inten_lo
HW address: 0x24
SW prefix: HMQO_INTEN_LO
SW offset: 0x90

31	30	29	28	27	26	25	24
HMQO_INTEN_LO[31:24]							
23	22	21	20	19	18	17	16
HMQO_INTEN_LO[23:16]							
15	14	13	12	11	10	9	8
HMQO_INTEN_LO[15:8]							
7	6	5	4	3	2	1	0
HMQO_INTEN_LO[7:0]							

- **HMQO_INTEN_LO** [*read/write*]: HMQ OUT interrupt enable for core 0-3.
  Set to 1 to enable interrupts from respective queue.

HMQ OUT Interrupt Enable Core 4-7 Register

HW prefix: mt_cpu_csr_hmqo_inten_hi
HW address: 0x26
SW prefix: HMQO_INTEN_HI
SW offset: 0x98

31	30	29	28	27	26	25	24
HMQO_INTEN_HI[31:24]							
23	22	21	20	19	18	17	16
HMQO_INTEN_HI[23:16]							
15	14	13	12	11	10	9	8
HMQO_INTEN_HI[15:8]							
7	6	5	4	3	2	1	0
HMQO_INTEN_HI[7:0]							

- **HMQO_INTEN_HI** [*read/write*]: HMQ OUT interrupt enable for core core 4-7.
  Set to 1 to enable interrupts from respective queue.

Core Select Register

HW prefix: mt_cpu_csr_core_sel
HW address: 0x30
SW prefix: CORE_SEL
SW offset: 0xc0

31	30	29	28	27	26	25	24
-	-	-	-	-	-	-	-
23	22	21	20	19	18	17	16
-	-	-	-	-	-	-	-
15	14	13	12	11	10	9	8
-	-	-	-	-	-	-	-
7	6	5	4	3	2	1	0
-	-	-	-	CORE_SEL[3:0]			

- **CORE_SEL** [*read/write*]: Select core to access.
  Applies to UADDR, UDATA and all serial console registers.

Core Upload Address Register

HW prefix: mt_cpu_csr_uaddr
HW address: 0x31
SW prefix: UADDR
SW offset: 0xc4

31	30	29	28	27	26	25	24
-	-	-	-	-	-	-	-
23	22	21	20	19	18	17	16
-	-	-	-	ADDR[19:16]			
15	14	13	12	11	10	9	8
ADDR[15:8]							
7	6	5	4	3	2	1	0
ADDR[7:0]							

- **ADDR** [*read/write*]: Address to access in selected core's local memory.

Core Upload Data Register

HW prefix: mt_cpu_csr_udata
HW address: 0x32
SW prefix: UDATA
SW offset: 0xc8

31	30	29	28	27	26	25	24
UDATA[31:24]							
23	22	21	20	19	18	17	16
UDATA[23:16]							
15	14	13	12	11	10	9	8
UDATA[15:8]							
7	6	5	4	3	2	1	0
UDATA[7:0]							

- **UDATA** [*read/write*]: Read/Write data from/to selected core's local memory.
  The address to read/write from/to is specified in the UADDR register.

Core Serial Console Message Register

HW prefix: mt_cpu_csr_uart_msg
HW address: 0x40
SW prefix: UART_MSG
SW offset: 0x100

31	30	29	28	27	26	25	24
-	-	-	-	-	-	-	-
23	22	21	20	19	18	17	16
-	-	-	-	-	-	-	-
15	14	13	12	11	10	9	8
-	-	-	-	-	-	-	-
7	6	5	4	3	2	1	0
DATA[7:0]							

- **DATA** [*read-only*]: Serial console message byte for the selected core

Core Serial Console Message Poll Register

HW prefix: mt_cpu_csr_uart_poll
HW address: 0x41
SW prefix: UART_POLL
SW offset: 0x104

31	30	29	28	27	26	25	24
-	-	-	-	-	-	-	-
23	22	21	20	19	18	17	16
-	-	-	-	-	-	-	-
15	14	13	12	11	10	9	8
-	-	-	-	-	-	-	-
7	6	5	4	3	2	1	0
READY[7:0]							

- **READY** [*read-only*]: Serial console message data available

Core Serial Console Message Interrupt Mask Register

HW prefix: mt_cpu_csr_uart_imsk
HW address: 0x42
SW prefix: UART_IMSK
SW offset: 0x108

31	30	29	28	27	26	25	24
-	-	-	-	-	-	-	-
23	22	21	20	19	18	17	16
-	-	-	-	-	-	-	-
15	14	13	12	11	10	9	8
-	-	-	-	-	-	-	-
7	6	5	4	3	2	1	0
ENABLE[7:0]							

- **ENABLE** [*read/write*]: Per-Core Serial console message Interrupt Enable
  1: IRQ enabled

Debug Interface Status Register

HW prefix: mt_cpu_csr_dbg_status
HW address: 0x60
SW prefix: DBG_STATUS
SW offset: 0x180

31	30	29	28	27	26	25	24
-	-	-	-	-	-	-	-
23	22	21	20	19	18	17	16
-	-	-	-	-	-	-	-
15	14	13	12	11	10	9	8
-	-	-	-	-	-	-	-
7	6	5	4	3	2	1	0
DBG_STATUS[7:0]							

- **DBG_STATUS** [*read-only*]: Per Core debug mode bit

Debug Interface Force Register

HW prefix: mt_cpu_csr_dbg_force
HW address: 0x61
SW prefix: DBG_FORCE
SW offset: 0x184

31	30	29	28	27	26	25	24
-	-	-	-	-	-	-	-
23	22	21	20	19	18	17	16
-	-	-	-	-	-	-	-
15	14	13	12	11	10	9	8
-	-	-	-	-	-	-	-
7	6	5	4	3	2	1	0
DBG_FORCE[7:0]							

- **DBG_FORCE** [*read/write*]: Core debug force

Debug Interface Instruction Ready Register

HW prefix: mt_cpu_csr_dbg_insn_ready
HW address: 0x62
SW prefix: DBG_INSN_READY
SW offset: 0x188

31	30	29	28	27	26	25	24
-	-	-	-	-	-	-	-
23	22	21	20	19	18	17	16
-	-	-	-	-	-	-	-
15	14	13	12	11	10	9	8
-	-	-	-	-	-	-	-
7	6	5	4	3	2	1	0
DBG_INSN_READY[7:0]							

- **DBG_INSN_READY** [*read-only*]: Core instruction ready

Debug Interface Core[0] Instruction Register

HW prefix: mt_cpu_csr_dbg_core0_insn
HW address: 0x63
SW prefix: DBG_CORE0_INSN
SW offset: 0x18c

31	30	29	28	27	26	25	24
DBG_CORE0_INSN[31:24]							
23	22	21	20	19	18	17	16
DBG_CORE0_INSN[23:16]							
15	14	13	12	11	10	9	8
DBG_CORE0_INSN[15:8]							
7	6	5	4	3	2	1	0
DBG_CORE0_INSN[7:0]							

- **DBG_CORE0_INSN** [*read/write*]: Instruction to be executed

Debug Interface Core[1] Instruction Register

HW prefix: mt_cpu_csr_dbg_core1_insn
HW address: 0x64
SW prefix: DBG_CORE1_INSN
SW offset: 0x190

31	30	29	28	27	26	25	24
DBG_CORE1_INSN[31:24]							
23	22	21	20	19	18	17	16
DBG_CORE1_INSN[23:16]							
15	14	13	12	11	10	9	8
DBG_CORE1_INSN[15:8]							
7	6	5	4	3	2	1	0
DBG_CORE1_INSN[7:0]							

- **DBG_CORE1_INSN** [*read/write*]: Instruction to be executed

Debug Interface Core[0] Mailbox Data Register

HW prefix: mt_cpu_csr_dbg_core0_mbx
HW address: 0x65
SW prefix: DBG_CORE0_MBX
SW offset: 0x194

31	30	29	28	27	26	25	24
DBG_CORE0_MBX[31:24]							
23	22	21	20	19	18	17	16
DBG_CORE0_MBX[23:16]							
15	14	13	12	11	10	9	8
DBG_CORE0_MBX[15:8]							
7	6	5	4	3	2	1	0
DBG_CORE0_MBX[7:0]							

- **DBG_CORE0_MBX** [*read/write*]: Mailbox data

Debug Interface Core[1] Mailbox Data Register

HW prefix: mt_cpu_csr_dbg_core1_mbx
HW address: 0x66
SW prefix: DBG_CORE1_MBX
SW offset: 0x198

31	30	29	28	27	26	25	24
DBG_CORE1_MBX[31:24]							
23	22	21	20	19	18	17	16
DBG_CORE1_MBX[23:16]							
15	14	13	12	11	10	9	8
DBG_CORE1_MBX[15:8]							
7	6	5	4	3	2	1	0
DBG_CORE1_MBX[7:0]							

- **DBG_CORE1_MBX** [*read/write*]: Mailbox data

7.2 Local Registers (LR)

This is a list of all the local registers, which are accessible from the soft-CPUs. Each soft CPU has its own copy of the LR.

7.2.1 Mock Turtle CPU Per-Core Local Registers

Memory map summary

SW Offset	Type	Name	HW prefix	C prefix
0x0	REG	Status Register	mt_cpu_lr_stat	STAT
0x4	REG	Notification Interrupt Register	mt_cpu_lr_ntf_int	NTF_INT
0x8	REG	Serial Console Output	mt_cpu_lr_uart_chr	UART_CHR
0x40	REG	HMQ Status Register	mt_cpu_lr_hmq_stat	HMQ_STAT
0x44	REG	RMQ Status Register	mt_cpu_lr_rmq_stat	RMQ_STAT
0x80	REG	White Rabbit Status Register	mt_cpu_lr_wr_stat	WR_STAT
0x84	REG	TAI Cycles	mt_cpu_lr_tai_cycles	TAI_CYCLES
0x88	REG	TAI Seconds	mt_cpu_lr_tai_sec	TAI_SEC
0x8c	REG	Delay Counter Register	mt_cpu_lr_delay_cnt	DELAY_CNT
0xc0	REG	GPIO Input	mt_cpu_lr_gpio_in	GPIO_IN
0xc4	REG	GPIO Set	mt_cpu_lr_gpio_set	GPIO_SET
0xc8	REG	GPIO Clear	mt_cpu_lr_gpio_clear	GPIO_CLEAR

Register description

Status Register

HW prefix: mt_cpu_lr_stat
HW address: 0x0
SW prefix: STAT
SW offset: 0x0

31	30	29	28	27	26	25	24
-	-	-	-	-	-	-	-
23	22	21	20	19	18	17	16
-	-	-	-	-	-	-	-
15	14	13	12	11	10	9	8
-	-	-	-	-	-	-	-
7	6	5	4	3	2	1	0
-	-	-	-	CORE_ID[3:0]			

- **CORE_ID** [*read-only*]: ID (number) of the CPU core owning this register.

Notification Interrupt Register

HW prefix: mt_cpu_lr_ntf_int
HW address: 0x1
SW prefix: NTF_INT
SW offset: 0x4

31	30	29	28	27	26	25	24
-	-	-	-	-	-	-	-
23	22	21	20	19	18	17	16
-	-	-	-	-	-	-	-
15	14	13	12	11	10	9	8
-	-	-	-	-	-	-	-
7	6	5	4	3	2	1	0
NTF_INT[7:0]							

- **NTF_INT** [*read/write*]: Write a value to send a notification interrupt to the host.

Serial Console Output

HW prefix: mt_cpu_lr_uart_chr
HW address: 0x2
SW prefix: UART_CHR
SW offset: 0x8

31	30	29	28	27	26	25	24
-	-	-	-	-	-	-	-
23	22	21	20	19	18	17	16
-	-	-	-	-	-	-	-
15	14	13	12	11	10	9	8
-	-	-	-	-	-	-	-
7	6	5	4	3	2	1	0
UART_CHR[7:0]							

- **UART_CHR** [*write-only*]: Write port for serial console.

HMQ Status Register

HW prefix: mt_cpu_lr_hmq_stat
HW address: 0x10
SW prefix: HMQ_STAT
SW offset: 0x40

31	30	29	28	27	26	25	24
-	-	-	-	-	-	-	-
23	22	21	20	19	18	17	16
OUT[7:0]							
15	14	13	12	11	10	9	8
-	-	-	-	-	-	-	-
7	6	5	4	3	2	1	0
IN[7:0]							

- **IN** [*read-only*]: HMQ IN Slot Status
  Returns 1 if not empty (a message is available)
- **OUT** [*read-only*]: HMQ OUT Slot Status
  Returns 1 if not full (a message can be sent)

RMQ Status Register

HW prefix: mt_cpu_lr_rmq_stat
HW address: 0x11
SW prefix: RMQ_STAT
SW offset: 0x44

31	30	29	28	27	26	25	24
-	-	-	-	-	-	-	-
23	22	21	20	19	18	17	16
OUT[7:0]							
15	14	13	12	11	10	9	8
-	-	-	-	-	-	-	-
7	6	5	4	3	2	1	0
IN[7:0]							

- **IN** [*read-only*]: RMQ IN Slot Status
  Returns 1 if not empty (a message is available)
- **OUT** [*read-only*]: RMQ OUT Slot Status
  Returns 1 if not full (a message can be sent)

White Rabbit Status Register

HW prefix: mt_cpu_lr_wr_stat
HW address: 0x20
SW prefix: WR_STAT
SW offset: 0x80

31	30	29	28	27	26	25	24
-	-	-	-	-	-	-	-
23	22	21	20	19	18	17	16
-	-	-	-	-	-	-	-
15	14	13	12	11	10	9	8
AUX_CLOCK_OK[7:0]							
7	6	5	4	3	2	1	0
-	-	-	-	-	-	TIME_OK	LINK_OK

- **LINK_OK** [*read-only*]: WR Link Up
- **TIME_OK** [*read-only*]: WR Time OK
- **AUX_CLOCK_OK** [*read-only*]: WR Aux Clock OK

TAI Cycles

HW prefix: mt_cpu_lr_tai_cycles
HW address: 0x21
SW prefix: TAI_CYCLES
SW offset: 0x84

31	30	29	28	27	26	25	24
-	-	-	-	TAI_CYCLES[27:24]			
23	22	21	20	19	18	17	16
TAI_CYCLES[23:16]							
15	14	13	12	11	10	9	8
TAI_CYCLES[15:8]							
7	6	5	4	3	2	1	0
TAI_CYCLES[7:0]							

- **TAI_CYCLES** [*read-only*]: When White Rabbit is enabled, this returns the TAI clock ticks. Without WR, it just counts ticks of the system clock

TAI Seconds

HW prefix: mt_cpu_lr_tai_sec
HW address: 0x22
SW prefix: TAI_SEC
SW offset: 0x88

31	30	29	28	27	26	25	24
TAI_SEC[31:24]							
23	22	21	20	19	18	17	16
TAI_SEC[23:16]							
15	14	13	12	11	10	9	8
TAI_SEC[15:8]							
7	6	5	4	3	2	1	0
TAI_SEC[7:0]							

- **TAI_SEC** [*read-only*]: When White Rabbit is enabled, this returns the TAI seconds. Without WR, it just counts seconds based on ticks of the system clock

Delay Counter Register

HW prefix: mt_cpu_lr_delay_cnt
HW address: 0x23
SW prefix: DELAY_CNT
SW offset: 0x8c

31	30	29	28	27	26	25	24
DELAY_CNT[31:24]							
23	22	21	20	19	18	17	16
DELAY_CNT[23:16]							
15	14	13	12	11	10	9	8
DELAY_CNT[15:8]							
7	6	5	4	3	2	1	0
DELAY_CNT[7:0]							

- **DELAY_CNT** [*read/write*]: Counts down at every system clock cycle and stops at 0.
  Useful for generating delays.

GPIO Input

HW prefix: mt_cpu_lr_gpio_in
HW address: 0x30
SW prefix: GPIO_IN
SW offset: 0xc0

31	30	29	28	27	26	25	24
GPIO_IN[31:24]							
23	22	21	20	19	18	17	16
GPIO_IN[23:16]							
15	14	13	12	11	10	9	8
GPIO_IN[15:8]							
7	6	5	4	3	2	1	0
GPIO_IN[7:0]							

- **GPIO_IN** [*read-only*]: GPIO In

GPIO Set

HW prefix: mt_cpu_lr_gpio_set
HW address: 0x31
SW prefix: GPIO_SET
SW offset: 0xc4

31	30	29	28	27	26	25	24
GPIO_SET[31:24]							
23	22	21	20	19	18	17	16
GPIO_SET[23:16]							
15	14	13	12	11	10	9	8
GPIO_SET[15:8]							
7	6	5	4	3	2	1	0
GPIO_SET[7:0]							

- **GPIO_SET** [*write-only*]: GPIO Set

GPIO Clear

HW prefix: mt_cpu_lr_gpio_clear
HW address: 0x32
SW prefix: GPIO_CLEAR
SW offset: 0xc8

31	30	29	28	27	26	25	24
GPIO_CLEAR[31:24]							
23	22	21	20	19	18	17	16
GPIO_CLEAR[23:16]							
15	14	13	12	11	10	9	8
GPIO_CLEAR[15:8]							
7	6	5	4	3	2	1	0
GPIO_CLEAR[7:0]							

- **GPIO_CLEAR** [*write-only*]: GPIO Clear

CHAPTER

EIGHT

GLOSSARY

Control System It is a system that manage, commands, regulates the behaviour of a set of devices.

Digital Signal Processing The use of digital processing to perform a wide variety of signal processing operations.

CPU-Core It is a soft-CPU in Mock Turtle

Embedded System It is an autonomus system made of software, hardware (and gateware), implementing dedicated functions

End-Point It is a gateware core connected to a Mock Turtle RMQ that provides connection to an external network.

Firmware It is an embedded software system running on the Mock Turtle cpu-core.

Gateware It is a bitstream which configures an FPGA, or the HDL sources from which it was generated.

Gateware Core It is an HDL component part of a more complex gateware design.

Hardware It is a physical component.

Host It is the system that hosts the hardware in use.

Host Application It is an user space program running on the host system.

HMQ

Host Message Queue It is a message queue that connects Mock Turtle to the host system.

MQ

Message Queue It is a communication system based on queues with FIFO policy. Messages are put on the queue and they are sent to the programmed destination. Each message queue has two direction: input and output. Mock Turtle supports two message queues: host and remote.

RTC

Real-Time Computing It is an hardware and software system subject to time constraints.

RMQ

Remote Message Queue It is a message queue that connects the Mock Turtle to a network.

SHM

Shared Memory It is a memory shared among soft-CPUs and the host system.

Soft CPU

soft-cpu It is an HDL implementation of a CPU running on an FPGA.

MQ Entry It is a single element in the MQ.

User Space It is a software running on the host and it is not in kernel mode. This includes libraries and programs.

CHAPTER

NINE

INDICES AND TABLES

- genindex
- modindex
- search
- *Control/Status Registers (CSR)*
- *Local Registers (LR)*

INDEX

Symbols

C

D

E

F

G

H

I

L

M

P

R

S

T

U

V

W