

ReconOS User's Manual

The license notice (i.e. BSD-like) goes here.

Copyright © 2006-2010 Enno Luebbers <luebbers@reconos.de>

Table of Contents

1	Introduction	1
2	Installation	2
2.1	Prerequisites	2
2.1.1	FPGA synthesis, implementation, and simulation tools	2
2.1.2	Scripting languages and parsers	2
3	Quick Start	3
3.1	Prerequisites	3
3.2	Create Project	3
3.3	Hardware Design	4
3.3.1	Add Static Hardware Thread	4
3.3.2	Generate Static Hardware Design	4
3.3.2.1	Option 1: Using XPS tool	4
3.3.2.2	Option 2: Using Makefile	4
3.4	Software Design	5
3.4.1	Code the Software Application	5
3.4.2	Create an eCos Configuration	5
3.4.3	Compile SW application	5
3.5	Configure the FPGA and Upload the Application	5
3.5.1	Start Modem	5
3.5.2	Configure the FPGA	5
3.5.3	Upload the Application	5
4	Designing Hardware Threads	7
4.1	Hardware Thread Files	7
4.2	Hardware Thread Interface	7
4.3	Hardware Thread FSM	7
4.4	Hardware Thread Tools	7
4.5	Hardware Thread API	7
4.5.1	Data Types	8
4.5.2	Constants	8
4.5.3	System Call Reference	8
4.5.3.1	reconos_sem_post()	11
4.5.3.2	reconos_sem_wait()	11
4.5.3.3	reconos_write()	12
4.5.3.4	reconos_read() / reconos_read_s()	12
4.5.3.5	reconos_write_burst()	14
4.5.3.6	reconos_read_burst()	15
4.5.3.7	reconos_get_init_data() / reconos_get_init_data_s()	16
4.5.3.8	reconos_mutex_lock()	18
4.5.3.9	reconos_mutex_trylock()	18

4.5.3.10	reconos_mutex_unlock()	19
4.5.3.11	reconos_mutex_release()	19
4.5.3.12	reconos_cond_wait()	20
4.5.3.13	reconos_cond_signal()	21
4.5.3.14	reconos_cond_broadcast()	21
4.5.3.15	reconos_mbox_get() / reconos_mbox_get_s()	22
4.5.3.16	reconos_mbox_tryget() / reconos_mbox_tryget_s()	24
4.5.3.17	reconos_mbox_put()	26
4.5.3.18	reconos_mbox_tryput()	27
4.5.3.19	reconos_begin()	28
4.5.3.20	reconos_ready()	28
4.5.3.21	reconos_reset()	28
4.5.3.22	reconos_thread_exit()	29
5	Hardware Architecture	30
6	Software Architecture	31
7	Partial Reconfiguration	32
8	Tools	33
9	Target Platforms	34
10	Sample Applications	35
	Topics to be placed	36
	Execution Model	36
	Multi Cycle Commands	36
	reconos_hwthread_create	36
	Message Boxes	36
	OS Interface	36
	Environment Variables	36

1 Introduction

The vision behind this project is to raise design productivity and flexibility for dynamically reconfigurable (FPGA-based) systems to a level currently achieved only for processor-based systems. To this end, we are developing ReconOS, an operating system designed to alleviate the difficulty of programming reconfigurable hardware by offering common service abstractions for both hardware and software tasks. ReconOS provides application programmers with an RTOS-like programming model that abstracts away from the device details and an execution model that performs hardware multitasking on current FPGA technology.

The importance of the ReconOS project stems from the fact that hardware reconfigurability will become an essential feature of future architectures and applications, especially networked embedded systems. While the underlying hardware technology already evolves, there is a lack of appropriate design tools. ReconOS intends to address this problem and to build the necessary foundation for these future applications.

Furthermore, an extension to the basic ReconOS system aims to take the usability of dynamically reconfigurable hardware to the next level while maintaining efficiency. We think this can be reached by looking into the areas of higher-level front ends and system monitoring, both of which have not been sufficiently investigated yet in the context of reconfigurable hardware, as well as into the novel idea of adaptive run-time systems.

2 Installation

These instructions will guide you step-by-step through the process of setting up a development box including all the tools you need.

2.1 Prerequisites

The ReconOS tool chain depends on quite a few other tools, both commercial and open-source. While we try to rely on as much free software as possible, some of the tools (most notably the FPGA synthesis and implementation tools) are not, and will probably never be, available for free. In some cases, we will provide links to free evaluation versions or alternatives to our preferred choices.

2.1.1 FPGA synthesis, implementation, and simulation tools

ReconOS requires that the following commercial FPGA tools be installed.

- Xilinx ISE version 9.2.04i ([Xilinx Website](#)). Note that version 9.2 is no longer available for download but continues to be the only supported version for the early-access partial reconfiguration tool flow. There is also a free, limited version of ISE, the WebPack.
- Xilinx EDK version 9.2.02 ([Xilinx Website](#)). The EDK version should match the ISE version for the least hassle.
- ModelSim SE version 6.1f or higher ([Modeltech Website](#)). This tool is not necessary, but greatly recommended for hardware thread and system simulation as well as automated regression testing. Please note that several [[Environment Variables](#)], [page 36](#) related to the commercial tools need to be set for the ReconOS tool chain. Also, setting up the drivers for the Xilinx Platform USB cable, which is used for programming most of the target platforms, is often tedious (instructions are [here](#) and [here \(German\)](#)).

2.1.2 Scripting languages and parsers

ReconOS heavily relies on Python for the automated parts of its tool chain. It requires the following software:

- Python version 2.5.1 ([Download](#)). Newer versions may work, but have not been tested.
- pyparsing 1.5.1 ([Download](#)). Newer versions may work, but have not been tested.
- Cheetah 2.0.1 ([Download](#)). Newer versions may work, but have not been tested.
- Cross-compilers, binutils and libraries

TODO: crostool (for Linux), ecos 2.0 (for eCos)

3 Quick Start

DRAFT

This guide will lead you through the steps necessary for building a static design using ReconOS. We will be using a simple example with one slot containing a static hardware thread. As application we implement a sorter. In a first step, the application generates random data. In a next step, the data is divided into chunks of 8 Kbyte. These chunks will be sorted by either the hardware thread or a software thread using the bubble sort algorithm. The chunks will then be merged, such that in the end the entire data is sorted. To ensure correctness, the application checks, if the data is sorted correctly.

3.1 Prerequisites

This howto will assume that you already have checked out the ReconOS repository and build all required tools and libraries. Refer to the documentation pages that cover these topics: [Chapter 2 \[Installation\]](#), [page 2](#). Furthermore, this tutorial assumes that you have a Xilinx XUPV2P board with a compatible memory module installed.

3.2 Create Project

First we create the structure of the project. This is easily done by using the command

```
reconos_mkprj.py <project_name>
```

This creates the typical ReconOS Projekt environment with the sw and hw directory. Now we have to change the project settings. From this point, we assume that `$WORK` describes the path to your project folder.

```
file: <project_name>.rprj
```

The line beginning with `STATIC-THREADS` has to be edited, because we add a hardware thread that can sort 8 KBytes of data. Name the thread `sort8k` by adding this to the line.

Note: This tutorial creates the multicore environment on the XUP-Board. If you want to create it for the virtex4 , you have to change the reference design in the project file.

Now edit the layout file:

```
hw/<project_name>.lyt
```

Delete everything except the `TARGET` definition. The final file should look like this (for the XUP-Board):

```
target
  device XC2VP30
  family xc2vp
end
```

In a last step, set the environment variable `$HW_THREADS`

```
source $WORK/settings.sh
```

After this, the project structure is ready to be worked with.

3.3 Hardware Design

3.3.1 Add Static Hardware Thread

In this tutorial, we will use an existing simple example thread that sorts 8 kbytes of data. The thread waits for a message from an incoming message queue containing the address of the data chunk and sends a message to an outgoing queue when sorting is done. The thread is composed in two VHDL files that can be found under ‘\$RECONOS/demos/sort_demo/src/bubble_sorter.vhd’ and ‘\$RECONOS/demos/sort_demo/src/sort8k.vhd’. Copy both files to the ‘\$PROJECT_NAME/hw’ directory:

```
cp $RECONOS/demos/sort_demo/src/*.vhd $WORK/hw/*.vhd
```

Have a look at the VHDL code - most threads will be of a similar structure. `sort8k.vhd` contains the synchronous state machine that is connected to the operating system interface (OSIF), i.e. waiting for messages, while ‘`bubble_sorter.vhd`’ contains the user logic for sorting the data.

```
cd $WORK/hw/hw_threads
reconos_addwthread.py sort8k sort8k ../bubble_sorter.vhd ../sort8k.vhd
```

The arguments to the `reconos_addwthread.py` script are the hardware thread’s entity name, the user logic entity’s name (often the same as the one before), and the source files sorted after dependency such that the top file comes last. The script now creates an EDK pcore that contains the interface structures necessary to connect our hardware thread to the already instantiated OSIF.

Note that you can instantiate the same hardware thread multiple times.

3.3.2 Generate Static Hardware Design

3.3.2.1 Option 1: Using XPS tool

To generate the hardware design you first copy a reference design and insert static threads. This is done by the following command:

```
cd $WORK/hw
make static-threads
```

Now we create the software libraries and the final bitstream. This can be done using the Xilinx Platform Studio (XPS).

You have to open the project which can be found in ‘\$WORK/hw/edk-static/system.xmp’.

Compile the software drivers and library functions into a BSP, using the **Software->Generate libraries and BSPs** menu item. This will generate the Xilinx headers and particularly ‘`libxil.a`’ which we will need when compiling the eCos library. You need to regenerate this BSP whenever you change the hardware architecture (e.g. add OSIFs/slots, peripherals, change the memory map, etc.).

Finally, generate the bitstream, using the **Hardware->Generate Bitstream** menu item.

3.3.2.2 Option 2: Using Makefile

Alternatively, you can also use the makefile to do these steps

```
cd $WORK/hw
make bits-static
```

3.4 Software Design

3.4.1 Code the Software Application

Now copy the software part of the demo application into your project.

```
cp -r $RECONOS/demos/sort_demo/src/sw $WORK/sw
```

3.4.2 Create an eCos Configuration

ReconOS extends the embedded operating system eCos that is composed of packages. The eCos configuration file `sort.ecc` defines the eCos configuration. (You can modify it using the `configtool`.)

```
cd $WORK/sw
make mrproper setup
```

3.4.3 Compile SW application

Compile the software part of the application and link them into an executable.

```
make clean ecos
```

3.5 Configure the FPGA and Upload the Application

3.5.1 Start Modem

In a new shell, start the `minicom` modem, such that the print-functions, which are called by the software part of application and forwarded through the serial port to your computer, is shown to you.

```
minicom
```

When you have uploaded an executable, the print output will be shown here.

3.5.2 Configure the FPGA

To configure the FPGA with your hardware design, you have to download the bitstream to the board.

```
cd $WORK/sw
dow ../hw/edk-static/implementation/system.bit
```

3.5.3 Upload the Application

We have four different executables, which you can test.

- The first one executes the entire application in a function that runs on the CPU.

```
dow sort_ecos_st_sw.elf
```

- The second setting instantiates several software threads for the sorting part. Each thread can sort 8 kbytes of data. The application divides the entire data into such chunks and sends the starting addresses to a message queues. Each thread waits for such a message, then sorts the corresponding cunk and sends a message to an outgoing message box when the sorting is done. Then it waits for the next chunk message, and so on. Here we use only software threads.

```
dow sort_ecos_mt_sw.elf
```

- The third setting instantiates a hardware thread instead of multiple software threads.

```
dow sort_ecos_st_hw.elf
```

- The last setting combines the second with the third setting, such that multiple software threads and a single hardware threads run concurrently. The CPU and the hardware thread can run in parallel. Note, that the threads are independent of each other.

```
dow sort_ecos_mt_hw.elf
```

4 Designing Hardware Threads

In order to connect hardware components (either as VHDL modules or as black-box netlists) as threads to a ReconOS system, they need to conform to a set of requirements. In particular, hardware threads need to expose a strict [Section 4.2 \[Hardware Thread Interface\]](#), [page 7](#), which is used to connect them to their [\[OS Interface\]](#), [page 36](#), and to conform to a certain signalling protocol on this interface, which is most conveniently done by structuring all OS interactions in a dedicated state machine description.

The former requirement is easily satisfied by reusing a common VHDL port template for all hardware threads (which may be extensible with specially annotated user ports). The latter requires the use of a predefined set of VHDL functions and procedures to ensure proper synchronization with the OSIF and to invoke the individual operating system calls from within the hardware component. This set of VHDL callables, when used in a structured fashion within the hardware thread's [Section 4.3 \[Hardware Thread FSM\]](#), [page 7](#), mirrors the OS kernel's API as closely as possible and allows transparent interaction with the operating system's services and other threads.

4.1 Hardware Thread Files

4.2 Hardware Thread Interface

4.3 Hardware Thread FSM

4.4 Hardware Thread Tools

4.5 Hardware Thread API

The ReconOS hardware thread API, together with the [\[OS Interface\]](#), [page 36](#), provides a hardware thread written in VHDL with the capabilities to call kernel functions (or "system calls"). To implement blocking system calls, these VHDL procedures have to be called in a finite state machine that is written in a specific way. You can read more about this state machine in [Section 4.3 \[Hardware Thread FSM\]](#), [page 7](#).

This section describes the procedure prototypes without going into implementation details. It is meant to serve as a reference when writing hardware threads. To understand how the internals of the hardware thread / OS communications work, see [\[Execution Model\]](#), [page 36](#).

The interface a hardware thread has to have is detailed in [Section 4.2 \[Hardware Thread Interface\]](#), [page 7](#).

The datatypes and procedures for communication between a user task and the ReconOS OS interface are defined in `/hw/pccores/reconos_${RECONOS_VERSION}/hdl/vhdl/reconos_pkg.vhd`. This package must be included in every ReconOS hardware thread, for example:

```
library reconos_v2_01_a;
use reconos_v2_01_a.reconos_pkg.all;
for version 2.01.a.
```

4.5.1 Data Types

ReconOS defines two records for hardware thread / OS communication, one for each direction:

```
-- OS to task communication
type osif_os2task_t is record
  -- [...]
end record;

-- task to OS communication
type osif_task2os_t is record
  -- [...]
end record;
```

The actual contents of these records are not important to the thread developer—they are manipulated through the ReconOS hardware API calls. The `i_osif` and `o_osif` ports in every `HardwareThread` entity have to be passed to the API calls.

4.5.2 Constants

There are constants for the binary encoding of all ReconOS commands passed between hardware threads and the OS interface. They are listed in the system call overview table for reference and debugging purposes. When writing hardware threads, you do not need to specify these—they are implicitly encoded in the hardware API procedures.

Other constants include the bit width of the different osif fields, which can be interesting for thread designers, e.g. for determining the width of values read from memory. Which is fixed at 32 bits, but anyway...

```
-- width of OSIF commands, data and status registers
constant C_OSIF_CMD_WIDTH      : natural := 8;
constant C_OSIF_DATA_WIDTH    : natural := 32;
constant C_OSIF_STATUS_WIDTH  : natural := C_OSIF_DATA_WIDTH;
constant C_OSIF_NUM_STATUS_REGS : natural := 2; -- access latency and busy load

-- number of bits in communication records
constant C_OSIF_OS2TASK_REC_WIDTH : natural := C_OSIF_CMD_WIDTH + C_OSIF_DATA_WIDTH
constant C_OSIF_TASK2OS_REC_WIDTH : natural := C_OSIF_CMD_WIDTH + C_OSIF_DATA_WIDTH

-- maximum steps a multicycle command can take
constant C_MAX_MULTICYCLE_STEPS : natural := 4;

-- common constants
constant C_RECONOS_FAILURE : std_logic_vector(0 to C_OSIF_DATA_WIDTH-1) := X"00000000"
```

4.5.3 System Call Reference

ReconOS hardware system calls can be split into two classes: blocking and non-blocking calls. The meaning of 'blocking' here is a little different than in the context of software system calls. A 'blocking' call will halt the execution of the synchronization state machine (and thus all OS interaction) until the corresponding software call (initiated by the delegate

thread) returns. This way the hardware thread will also block while the delegate thread is blocking, but it is also necessary, for example, to transfer the return value of a call back to the hardware thread. Other system calls (e.g. `semaphore_post()`) that are 'software-nonblocking' and do not return anything will be 'hardware-nonblocking', thus allowing the state machine to continue as soon as the delegate thread has relayed the call to the eCos kernel.

In essence, every 'software-blocking' call is also 'hardware-blocking', but additionally all 'software-nonblocking' calls **with a return value** also have to be 'hardware-blocking'.

We also have to distinguish between "single-cycle" and "multi-cycle" commands. Multi-cycle commands are used if the communication between the thread and the OSIF cannot be completed in a single cycle. This is the case, for example, if the OSIF needs more than 32 bits of data, or if the OSIF will return a value to the thread upon completion. These commands have to be used in a certain way, which is described in [\[Multi Cycle Commands\]](#), page 36.

System calls that return a value usually exist in two variants, e.g. `reconos_read()` and `reconos_read_s()`. The first flavour returns the return value in a variable, so that it can be evaluated in the same state of the synchronization FSM. The `_s()` variety is a convenience wrapper which returns the return value in a signal, so that it can be directly connected to a separate process. This introduces one cycle of latency. The table below lists only the 'non-`_s()`' variant.

Some of the commands (such as memory accesses) are handled directly in hardware, while others (like calls to kernel synchronization primitives) have to be handled in software. See the ExecutionModel for details.

Most programming model objects (like mutexes or semaphores) are referenced by a handle. This handle is usually encoded as a 32 bit value, which is transferred to software and translated into the actual address of the object. See ResourceHandling for details.

NOTE: There can be only one OS command per state in the synchronization state machine!

The following table shows all available system calls, their associated constants and binary encodings (for debugging/simulation purposes), where applicable:

command	bin	symbolic name	SW	HW	MC	SW- proc
Section 4.5.3.1 [reconos_sem_post] , page 11	[re- 0x00	OSIF_CMD_SEM_POST			X	
Section 4.5.3.2 [reconos_sem_wait] , page 11	[re- 0x81	OSIF_CMD_SEM_WAIT	X	X		X
Section 4.5.3.3 [reconos_write] , page 12	0x49	OSIF_CMD_WRITE			X	
Section 4.5.3.4 [reconos_read] , page 12	0x48	OSIF_CMD_READ			X	

Section 4.5.3.20
[reconos_ready], page 28

Section 4.5.3.21
[reconos_reset], page 28

Section 4.5.3.22 [re- 0xF0 OSIF_CMD_THREAD_EXIT X X
conos_thread_exit],
page 29

Procedure Descriptions

The following is a short description of the individual function calls:

4.5.3.1 reconos_sem_post()

```
procedure reconos_sem_post (signal osif_task2os : out osif_task2os_t;
                            osif_os2task      : in  osif_os2task_t;
                            handle           : in  std_logic_vector(0 to C_OSIF_D
```

Posts (increments) the semaphore identified by `handle`.

Parameter	Description
<code>osif_task2os</code>	record of communication signals to the OS interface
<code>osif_os2task</code>	record of communication signals from the OS interface
<code>handle</code>	the thread-local identifier for the semaphore to post

Example (inside the synchronization FSM process):

```
...
case state is
...
when STATE_POST =>
    reconos_sem_post(o_osif, i_osif, C_MY_SEMAPHORE);
    state <= STATE_IDLE;
...

```

4.5.3.2 reconos_sem_wait()

```
procedure reconos_sem_wait (signal osif_task2os : out osif_task2os_t;
                            osif_os2task      : in  osif_os2task_t;
                            handle           : in  std_logic_vector(0 to C_OSIF_D
```

Waits for (then decrements) the semaphore identified by `handle`. Blocks until semaphore becomes available.

Parameter	Description
<code>osif_task2os</code>	record of communication signals to the OS interface
<code>osif_os2task</code>	record of communication signals from the OS interface
<code>handle</code>	the thread-local identifier for the semaphore to wait on

Example (inside the synchronization FSM process):

```
...
case state is
```

```

...
when STATE_WAIT =>
    reconos_sem_wait(o_osif, i_osif, C_MY_SEMAPHORE);
    state <= STATE_READ;
...

```

4.5.3.3 reconos_write()

```

procedure reconos_write (variable completed : out boolean;
                        signal osif_task2os : out osif_task2os_t;
                        signal osif_os2task : in  osif_os2task_t;
                        address            : in  std_logic_vector(0 to C_OSIF_DATA_W
                        data               : in  std_logic_vector(0 to C_OSIF_DATA_W

```

Writes a single word (32 Bits) to system memory.

Parameter	Description
completed	variable to indicate completion of this MultiCycle command
osif_task2os	record of communication signals to the OS interface
osif_os2task	record of communication signals from the OS interface
address	address to write to (can be memory or memory-mapped peripheral)
data	data word to write

Example (inside the synchronization FSM process):

```

...
variable done : boolean;
...
begin
...
    case state is
...
        when STATE_WRITE =>
            reconos_write(done, o_osif, i_osif, my_address, my_data);
            if done then
                state <= STATE_DO_SOMETHING;
            end if;
...

```

4.5.3.4 reconos_read() / reconos_read_s()

```

procedure reconos_read (variable completed : out boolean;
                        signal osif_task2os : out osif_task2os_t;
                        signal osif_os2task : in  osif_os2task_t;
                        address            : in  std_logic_vector(0 to C_OSIF_DATA_W
                        variable data       : out std_logic_vector(0 to C_OSIF_DATA_W

...

procedure reconos_read_s (variable completed : out boolean;
                        signal osif_task2os : out osif_task2os_t;
                        signal osif_os2task : in  osif_os2task_t;

```

```

address          : in  std_logic_vector(0 to C_OSIF_DATA_WIDTH-1)
signal data      : out std_logic_vector(0 to C_OSIF_DATA_WIDTH-1)

```

Reads a single word (32 Bits) from system memory. `reconos_read()` reads into a variable for immediate evaluation, while `reconos_read_s()` reads into a signal for evaluation in a different process.

Parameter	Description
<code>completed</code>	variable to indicate completion of this MultiCycle command
<code>osif_task2os</code>	record of communication signals to the OS interface
<code>osif_os2task</code>	record of communication signals from the OS interface
<code>address</code>	address to read from (can be memory or memory-mapped peripheral)
<code>data</code>	data word (signal/variable) to read into

Example 1 (inside the synchronization FSM process):

```

...
variable done : boolean;
variable my_data : std_logic_vector(0 to C_OSIF_DATA_WIDTH-1);
...
begin
...
  case state is
...
    when STATE_READ =>
      reconos_read(done, o_osif, i_osif, my_address, my_data);
      if done then
        if my_data = SOME_VALUE then
          state <= STATE_DO_SOMETHING;
        else
          state <= STATE_DO_SOMETHING_ELSE;
        end if;
      end if;
    end if;
  end if;
...

```

Example 2 (inside the synchronization FSM process):

```

signal my_data : std_logic_vector(0 to C_OSIF_DATA_WIDTH-1);
...
fsm_proc : process(...)
...
variable done : boolean;
...
begin
...
  case state is
...
    when STATE_READ =>
      reconos_read_s(done, o_osif, i_osif, my_address, my_data);
    end if;
  end if;
end process;

```

```

        if done then
            state <= STATE_DO_SOMETHING;
        end if;
    ...
end process;
...
some_other_proc : process(...)
...
begin
    ...
    if my_data = SOME_VALUE then
        ...
    end if;
    ...
end process

```

4.5.3.5 reconos_write_burst()

```

procedure reconos_write_burst (variable completed : out boolean;
                               signal osif_task2os : out osif_task2os_t;
                               signal osif_os2task : in  osif_os2task_t;
                               my_address          : in  std_logic_vector(0 to C_OSIF_...);
                               target_address       : in  std_logic_vector(0 to C_OSIF_...));

```

Writes / copies a burst of 32 words (32x32 Bits) from the local burst RAM to system memory.

Parameter	Description
completed	variable to indicate completion of this MultiCycle command
osif_task2os	record of communication signals to the OS interface
osif_os2task	record of communication signals from the OS interface
my_address	address in the local RAM to get data from (byte address)
target_address	address in the system memory space to write data to (byte address)

Example 1 (inside the synchronization FSM process):

```

...
variable done : boolean;
...
begin
...
    case state is
...
        when STATE_WRITE_BURST =>
            reconos_write_burst(done, o_osif, i_osif, local_address, global_address);
            if done then
                state <= STATE_DO_SOMETHING;
            end if;
...

```

Example 2 (inside the synchronization FSM process, type conversions omitted for simplicity):

```

...
variable done : boolean;
variable count : natural;
...
begin
...
  case state is
...
    when STATE_WRITE_BURST =>
      reconos_write_burst(done, o_osif, i_osif, local_address + count, global_address);
      if done then
        count := count + 128;
        if count > MAX_COUNT then
          state <= STATE_DO_SOMETHING;
        end if;
      end if;
    end if;
  end if;
...

```

4.5.3.6 reconos_read_burst()

```

procedure reconos_read_burst (variable completed : out boolean;
                              signal osif_task2os : out osif_task2os_t;
                              signal osif_os2task : in  osif_os2task_t;
                              my_address          : in  std_logic_vector(0 to C_OSIF_ADDR_WIDTH-1);
                              target_address      : in  std_logic_vector(0 to C_OSIF_ADDR_WIDTH-1));

```

Reads / copies a burst of 32 words (32x32 Bits) from the system memory space to local burst RAM.

Parameter	Description
completed	variable to indicate completion of this MultiCycle command
osif_task2os	record of communication signals to the OS interface
osif_os2task	record of communication signals from the OS interface
my_address	address in the local RAM to write data to (byte address)
target_address	address in the system memory space to get data from (byte address)

Example 1 (inside the synchronization FSM process):

```

...
variable done : boolean;
...
begin
...
  case state is
...
    when STATE_READ_BURST =>
      reconos_read_burst(done, o_osif, i_osif, local_address, global_address);

```

```

        if done then
            state <= STATE_DO_SOMETHING;
        end if;
    ...

```

Example 2 (inside the synchronization FSM process, type conversions omitted for simplicity):

```

...
variable done : boolean;
variable count : natural;
...
begin
...
    case state is
...
        when STATE_READ_BURST =>
            reconos_read_burst(done, o_osif, i_osif, local_address + count, global_address)
            if done then
                count := count + 128;
                if count > MAX_COUNT then
                    state <= STATE_DO_SOMETHING;
                end if;
            end if;
        end if;
    ...

```

4.5.3.7 reconos_get_init_data() / reconos_get_init_data_s()

```

procedure reconos_get_init_data (variable completed      : out boolean;
                                signal  osif_task2os    : out osif_task2os_t;
                                signal  osif_os2task    : in  osif_os2task_t;
                                variable data           : out std_logic_vector(0 to C_0))

procedure reconos_get_init_data_s (variable completed : out boolean;
                                   signal osif_task2os : out osif_task2os_t;
                                   signal osif_os2task  : in  osif_os2task_t;
                                   signal data         : out std_logic_vector(0 to C_0))

```

Reads the thread initialization data (32 bit) from the OS interface. This data is passed to the OSIF on hardware thread initialization (see [\[reconos_hwthread_create\]](#), page 36). `reconos_get_init_data()` returns a variable for immediate evaluation, while `reconos_get_init_data_s()` returns a signal for evaluation in a separate process.

Parameter	Description
completed	variable to indicate completion of this MultiCycle command
osif_task2os	record of communication signals to the OS interface
osif_os2task	record of communication signals from the OS interface
data	variable / signal to store initialization data in

Example 1 (inside the synchronization FSM process):

```

...
variable done : boolean;
variable my_data : std_logic_vector(0 to C_OSIF_DATA_WIDTH-1);
...
begin
...
    case state is
...
        when STATE_INIT =>
            reconos_get_init_data(done, o_osif, i_osif, my_data);
            if done then
                if my_data = SOME_VALUE then
                    state <= STATE_DO_SOMETHING;
                else
                    state <= STATE_DO_SOMETHING_ELSE;
                end if;
            end if;
        ...

```

Example 2 (inside the synchronization FSM process):

```

signal my_data : std_logic_vector(0 to C_OSIF_DATA_WIDTH-1);
...
fsm_proc : process(...)
...
variable done : boolean;
...
begin
...
    case state is
...
        when STATE_INIT =>
            reconos_get_init_data_s(done, o_osif, i_osif, my_data);
            if done then
                state <= STATE_DO_SOMETHING;
            end if;
        ...
    end process;
...
some_other_proc : process(...)
...
begin
    ...
    if my_data = SOME_VALUE then
        ...
    end if;
    ...
end process

```

4.5.3.8 reconos_mutex_lock()

```

procedure reconos_mutex_lock(variable completed      : out boolean;
                             variable success       : out boolean;
                             signal  osif_task2os   : out osif_task2os_t;
                             signal  osif_os2task   : in  osif_os2task_t;
                             handle          : in  std_logic_vector(0 to C_OSIF_

```

Locks a mutex. `reconos_mutex_lock()` will block until mutex becomes available, lock it, and return. If unsuccessful (e.g. on error or thread termination) `success` will be false.

Parameter	Description
<code>completed</code>	variable to indicate completion of this MultiCycle command
<code>success</code>	variable to indicate success of the mutex lock operation
<code>osif_task2os</code>	record of communication signals to the OS interface
<code>osif_os2task</code>	record of communication signals from the OS interface
<code>handle</code>	handle of the mutex to lock

Example (inside the synchronization FSM process):

```

...
variable done : boolean;
variable okay : boolean;
...
begin
...
    case state is
...
        when STATE_LOCK =>
            reconos_mutex_lock(done, okay, o_osif, i_osif, C_MY_MUTEX);
            if done and okay then -- note: this will loop for
                state <= STATE_DO_SOMETHING;
            end if;
...

```

4.5.3.9 reconos_mutex_trylock()

```

procedure reconos_mutex_trylock(variable completed      : out boolean;
                                variable success       : out boolean;
                                signal  osif_task2os   : out osif_task2os_t;
                                signal  osif_os2task   : in  osif_os2task_t;
                                handle          : in  std_logic_vector(0 to C_OSIF_

```

Locks a mutex, if available. `reconos_mutex_trylock()` will lock the mutex indicated by `handle`, if and only if it is available, and return true in `success`. Otherwise (e.g. mutex is already locked, an error occurred or the thread terminates) `success` will be false.

Parameter	Description
<code>completed</code>	variable to indicate completion of this MultiCycle command
<code>success</code>	variable to indicate success of the mutex trylock operation
<code>osif_task2os</code>	record of communication signals to the OS interface

osif_os2task record of communication signals from the OS interface
 handle handle of the mutex to lock

Example (inside the synchronization FSM process):

```

...
variable done : boolean;
variable okay : boolean;
...
begin
...
  case state is
...
    when STATE_LOCK =>
      reconos_mutex_trylock(done, okay, o_osif, i_osif, C_MY_MUTEX);
      if done then
        if okay then
          state <= STATE_DO_SOMETHING;
        else
          state <= STATE_DO_SOMETHING_ELSE;
        end if;
      end if;
    end if;
  end if;
...

```

4.5.3.10 reconos_mutex_unlock()

```

procedure reconos_mutex_unlock(signal osif_task2os : out osif_task2os_t;
                               signal osif_os2task : in  osif_os2task_t;
                               handle               : in  std_logic_vector(0 to C_OSIF_

```

Unlocks a mutex.

Parameter	Description
osif_task2os	record of communication signals to the OS interface
osif_os2task	record of communication signals from the OS interface
handle	handle of the mutex to unlock

Example (inside the synchronization FSM process):

```

...
begin
...
  case state is
...
    when STATE_UNLOCK =>
      reconos_mutex_unlock(o_osif, i_osif, C_MY_MUTEX);
      state <= STATE_DO_SOMETHING;
    end if;
  end if;
...

```

4.5.3.11 reconos_mutex_release()

```

procedure reconos_mutex_release(signal osif_task2os : out osif_task2os_t;

```

```

        signal osif_os2task : in  osif_os2task_t;
        handle           : in  std_logic_vector(0 to C_OSIF_D

```

Releases a mutex (i.e. wakes all threads waiting on it, which will get a return value of false).

Parameter	Description
osif_task2os	record of communication signals to the OS interface
osif_os2task	record of communication signals from the OS interface
handle	handle of the mutex to release

Example (inside the synchronization FSM process):

```

...
begin
...
  case state is
...
    when STATE_RELEASE =>
      reconos_mutex_release(o_osif, i_osif, C_MY_MUTEX);
      state <= STATE_DO_SOMETHING;
...

```

4.5.3.12 reconos_cond_wait()

```

procedure reconos_cond_wait(variable completed   : out boolean;
                           variable success     : out boolean;
                           signal  osif_task2os : out osif_task2os_t;
                           signal  osif_os2task : in  osif_os2task_t;
                           handle           : in  std_logic_vector(0 to C_OSIF_D

```

Waits on a condition variable. `reconos_cond_wait()` will block until a change of the condition variable referenced by `handle` is signalled. If unsuccessful (e.g. on error or thread termination) `success` will be false.

Parameter	Description
completed	variable to indicate completion of this MultiCycle command
success	variable to indicate success of the cond_wait operation
osif_task2os	record of communication signals to the OS interface
osif_os2task	record of communication signals from the OS interface
handle	handle of the condvar to wait on

TODO

Example (inside the synchronization FSM process):

```

...
variable done : boolean;
variable okay : boolean;
...
begin
...
  case state is

```

```

...
when STATE_WAIT =>
  reconos_cond_wait(done, okay, o_osif, i_osif, C_MY_CONDVAR);
  if done and okay then
    state <= STATE_DO_SOMETHING;
  end if;
...

```

4.5.3.13 reconos_cond_signal()

```

procedure reconos_cond_signal(signal osif_task2os : out osif_task2os_t;
                             signal osif_os2task : in  osif_os2task_t;
                             handle           : in  std_logic_vector(0 to C_OSIF_D

```

Signals a change of a condition variable, i.e. wakes up the **next** thread waiting on that condition variable.

Parameter	Description
osif_task2os	record of communication signals to the OS interface
osif_os2task	record of communication signals from the OS interface
handle	handle of the condvar to signal

Example (inside the synchronization FSM process):

```

...
begin
...
  case state is
...
  when STATE_SIGNAL =>
    reconos_cond_signal(o_osif, i_osif, C_MY_MUTEX);
    state <= STATE_DO_SOMETHING;
...

```

4.5.3.14 reconos_cond_broadcast()

```

procedure reconos_cond_broadcast(signal osif_task2os : out osif_task2os_t;
                                signal osif_os2task : in  osif_os2task_t;
                                handle           : in  std_logic_vector(0 to C_OSIF_D

```

Broadcasts a change of a condition variable, i.e. wakes **all** threads waiting on that condition variable.

Parameter	Description
osif_task2os	record of communication signals to the OS interface
osif_os2task	record of communication signals from the OS interface
handle	handle of the condvar to broadcast

Example (inside the synchronization FSM process):

```

...
begin
...

```

```

    case state is
    ...
    when STATE_SIGNAL =>
        reconos_cond_broadcast(o_osif, i_osif, C_MY_MUTEX);
        state <= STATE_DO_SOMETHING;
    ...

```

4.5.3.15 reconos_mbox_get() / reconos_mbox_get_s()

```

procedure reconos_mbox_get(variable completed      : out boolean;
                           variable success       : out boolean;
                           signal  osif_task2os  : out osif_task2os_t;
                           signal  osif_os2task  : in  osif_os2task_t;
                           handle         : in  std_logic_vector(0 to C_OSIF_
                           variable data        : out std_logic_vector(0 to C_OSIF_

procedure reconos_mbox_get_s(variable completed      : out boolean;
                             variable success       : out boolean;
                             signal  osif_task2os  : out osif_task2os_t;
                             signal  osif_os2task  : in  osif_os2task_t;
                             handle         : in  std_logic_vector(0 to C_OSIF_
                             signal  data         : out std_logic_vector(0 to C_OSIF_

```

Retrieves a 32bit value from the specified message box. The call may block if the message box is empty.

This call can also be handled in hardware, if the message box is mapped to a hardware FIFO connected to the executing thread's OSIF (see [Message Boxes], page 36). If the call is handled in software, it will block until the return value arrives.

`reconos_mbox_get()` returns the value from the message box in a variable for immediate evaluation, while `reconos_mbox_get_s()` returns a signal for evaluation in a separate process.

Parameter	Description
completed	variable to indicate completion of this MultiCycle command
success	signals whether the call succeeded. A failure can for example mean that the blocking call was interrupted by a signal
osif_task2os	record of communication signals to the OS interface
osif_os2task	record of communication signals from the OS interface
handle	the handle (resource number) of the message box
data	variable / signal to store initialization data in

Example 1 (inside the synchronization FSM process):

```

...
constant C_MBOX_HANDLE : std_logic_vector(0 to C_OSIF_DATA_WIDTH-1) := X"00000001"
...
variable success : boolean;
variable done : boolean;

```

```

variable my_data : std_logic_vector(0 to C_OSIF_DATA_WIDTH-1);
...
begin
...
  case state is
...
    when STATE_GET =>
      reconos_mbox_get(done, success, o_osif, i_osif, C_MBOX_HANDLE, my_data);
      if done then
        if success then
          if my_data = SOME_VALUE then
            state <= STATE_DO_SOMETHING;
          else
            state <= STATE_DO_SOMETHING_ELSE;
          end if;
        else -- no success
          state <= HANDLE_ERROR;
        end if;
      end if;
    end if;
...

```

Example 2 (inside the synchronization FSM process):

```

...
constant C_MBOX_HANDLE : std_logic_vector(0 to C_OSIF_DATA_WIDTH-1) := X"00000001";
...
variable done : boolean;
variable my_data : std_logic_vector(0 to C_OSIF_DATA_WIDTH-1);
...
begin
...
  case state is
...
    when STATE_GET =>
      reconos_mbox_get(done, success, o_osif, i_osif, C_MBOX_HANDLE, my_data);
      if done then
        if success then
          state <= STATE_DO_SOMETHING;
        else -- no success
          state <= HANDLE_ERROR;
        end if;
      end if;
    end if;
...
some_other_proc : process(...)
...
begin
...
  if my_data = SOME_VALUE then

```

```

    ...
    end if;
    ...
end process

```

4.5.3.16 reconos_mbox_tryget() / reconos_mbox_tryget_s()

```

procedure reconos_mbox_tryget(variable completed    : out boolean;
                             variable success      : out boolean;
                             signal  osif_task2os : out osif_task2os_t;
                             signal  osif_os2task : in  osif_os2task_t;
                             handle       : in  std_logic_vector(0 to C_OSIF_HANDLE_WIDTH-1);
                             variable data      : out std_logic_vector(0 to C_OSIF_DATA_WIDTH-1));

procedure reconos_mbox_tryget_s(variable completed    : out boolean;
                                variable success      : out boolean;
                                signal  osif_task2os : out osif_task2os_t;
                                signal  osif_os2task : in  osif_os2task_t;
                                handle       : in  std_logic_vector(0 to C_OSIF_HANDLE_WIDTH-1);
                                signal  data        : out std_logic_vector(0 to C_OSIF_DATA_WIDTH-1));

```

Retrieves a 32bit value from the specified message box. The call will **not** block if the message box is empty, but fail (`success = false`).

This call can also be handled in hardware, if the message box is mapped to a hardware FIFO connected to the executing thread's OSIF (see [\[Message Boxes\]](#), page 36). If the call is handled in software, it will block until the return value arrives.

`reconos_mbox_tryget()` returns the value from the message box in a variable for immediate evaluation, while `reconos_mbox_tryget_s()` returns a signal for evaluation in a separate process.

Parameter	Description
<code>completed</code>	variable to indicate completion of this MultiCycle command
<code>success</code>	signals whether the call succeeded.
<code>osif_task2os</code>	record of communication signals to the OS interface
<code>osif_os2task</code>	record of communication signals from the OS interface
<code>handle</code>	the handle (resource number) of the message box
<code>data</code>	variable / signal to store initialization data in

Example 1 (inside the synchronization FSM process):

```

...
constant C_MBOX_HANDLE : std_logic_vector(0 to C_OSIF_DATA_WIDTH-1) := X"00000001";
...
variable success : boolean;
variable done : boolean;
variable my_data : std_logic_vector(0 to C_OSIF_DATA_WIDTH-1);
...

```

```

begin
...
  case state is
...
    when STATE_GET =>
      reconos_mbox_tryget(done, success, o_osif, i_osif, C_MBOX_HANDLE, my_data);
      if done then
        if success then
          if my_data = SOME_VALUE then
            state <= STATE_DO_SOMETHING;
          else
            state <= STATE_DO_SOMETHING_ELSE;
          end if;
        else -- no success
          state <= HANDLE_ERROR;
        end if;
      end if;
...

```

Example 2 (inside the synchronization FSM process):

```

...
constant C_MBOX_HANDLE : std_logic_vector(0 to C_OSIF_DATA_WIDTH-1) := X"00000001";
...
variable done : boolean;
variable my_data : std_logic_vector(0 to C_OSIF_DATA_WIDTH-1);
...
begin
...
  case state is
...
    when STATE_GET =>
      reconos_mbox_tryget(done, success, o_osif, i_osif, C_MBOX_HANDLE, my_data);
      if done then
        if success then
          state <= STATE_DO_SOMETHING;
        else -- no success
          state <= HANDLE_ERROR;
        end if;
      end if;
...
some_other_proc : process(...)
...
begin
...
  if my_data = SOME_VALUE then
    ...
  end if;

```

```

...
end process

```

4.5.3.17 reconos_mbox_put()

```

procedure reconos_mbox_put(variable completed      : out boolean;
                           variable success       : out boolean;
                           signal  osif_task2os   : out osif_task2os_t;
                           signal  osif_os2task   : in  osif_os2task_t;
                           handle    : in  std_logic_vector(0 to C_OSIF_...
                           data      : in  std_logic_vector(0 to C_OSIF_...

```

Sends a 32bit value to the specified message box. The call may block if the message box is full.

This call can also be handled in hardware, if the message box is mapped to a hardware FIFO connected to the executing thread's OSIF (see [\[Message Boxes\]](#), page 36). If the call is handled in software, it will block until the return value arrives.

Parameter	Description
completed	variable to indicate completion of this MultiCycle command
success	signals whether the call succeeded. A failure can for example mean that the blocking call was interrupted by a signal
osif_task2os	record of communication signals to the OS interface
osif_os2task	record of communication signals from the OS interface
handle	the handle (resource number) of the message box
data	variable / signal to store initialization data in

Example (inside the synchronization FSM process):

```

...
constant C_MBOX_HANDLE : std_logic_vector(0 to C_OSIF_DATA_WIDTH-1) := X"00000001"
...
variable success : boolean;
variable done : boolean;
signal/variable my_data : std_logic_vector(0 to C_OSIF_DATA_WIDTH-1);
...
begin
...
  case state is
...
    when STATE_PUT =>
      reconos_mbox_put(done, success, o_osif, i_osif, C_MBOX_HANDLE, my_data);
      if done then
        if success then
          state <= STATE_DO_SOMETHING;
        else -- no success
          state <= HANDLE_ERROR;

```

```

        end if;
    end if;
    ...

```

4.5.3.18 reconos_mbox_tryput()

```

procedure reconos_mbox_tryput(variable completed    : out boolean;
                             variable success      : out boolean;
                             signal  osif_task2os  : out osif_task2os_t;
                             signal  osif_os2task  : in  osif_os2task_t;
                             handle      : in  std_logic_vector(0 to C_OS
                             data        : in  std_logic_vector(0 to C_OS

```

Sends a 32bit value to the specified message box. The call will **not** block if the message box is full, but fail (`success = false`).

This call can also be handled in hardware, if the message box is mapped to a hardware FIFO connected to the executing thread's OSIF (see [Message Boxes], page 36). If the call is handled in software, it will block until the return value arrives.

Parameter	Description
<code>completed</code>	variable to indicate completion of this MultiCycle command
<code>success</code>	signals whether the call succeeded.
<code>osif_task2os</code>	record of communication signals to the OS interface
<code>osif_os2task</code>	record of communication signals from the OS interface
<code>handle</code>	the handle (resource number) of the message box
<code>data</code>	variable / signal to store initialization data in

Example (inside the synchronization FSM process):

```

...
constant C_MBOX_HANDLE : std_logic_vector(0 to C_OSIF_DATA_WIDTH-1) := X"00000001";
...
variable success : boolean;
variable done : boolean;
variable/signal my_data : std_logic_vector(0 to C_OSIF_DATA_WIDTH-1);
...
begin
...
    case state is
...
        when STATE_PUT =>
            reconos_mbox_tryput(done, success, o_osif, i_osif, C_MBOX_HANDLE, my_data);
            if done then
                if success then
                    state <= STATE_DO_SOMETHING;
                else -- no success
                    state <= HANDLE_ERROR;
                end if;
            end if;

```

```

        end if;
    ...

```

4.5.3.19 reconos_begin()

```

procedure reconos_begin (signal osif_task2os : out osif_task2os_t;
                        osif_os2task      : osif_os2task_t);

```

Sets all OSIF signals to valid starting values. Used at the beginning of a synchronization state machine (see [Section 4.3 \[Hardware Thread FSM\]](#), page 7).

Parameter	Description
osif_task2os	record of communication signals to the OS interface
osif_os2task	record of communication signals from the OS interface

Example:

```

...
fsm_proc : process(clk, reset)
...
begin
    if reset = '1' then
        ...
        reconos_reset(o_osif, i_osif);
        ...
    elsif rising_edge(clk) then
        ...
        reconos_begin(o_osif, i_osif);
        ...
        if reconos_ready(i_osif) then
            case state is
                ...
                when STATE_SIGNAL =>
                    ...
            end if;
        end if;
    end if;
...

```

4.5.3.20 reconos_ready()

```

function reconos_ready (osif_os2task : osif_os2task_t) return boolean;

```

Returns true if the hardware thread is neither busy nor blocking. see [Section 4.3 \[Hardware Thread FSM\]](#), page 7 and [\[Execution Model\]](#), page 36 for details.

Parameter	Description
osif_os2task	record of communication signals from the OS interface

Example: see [Section 4.5.3.19 \[reconos_begin\]](#), page 28

4.5.3.21 reconos_reset()

```

procedure reconos_reset (signal osif_task2os : out osif_task2os_t;

```

```
osif_os2task      : osif_os2task_t);
```

Resets all OSIF signals to their initialization values. Used in the asynchronous reset block of a [[ReconOSHardwareTaskFSM][synchronization state machine]].

Parameter	Description
osif_task2os	record of communication signals to the OS interface
osif_os2task	record of communication signals from the OS interface

Example: see [Section 4.5.3.19 \[reconos_begin\]](#), page 28

4.5.3.22 reconos_thread_exit()

```
procedure reconos_thread_exit (signal osif_task2os : out osif_task2os_t;
                               osif_os2task      : in  osif_os2task_t;
                               retval            : in  std_logic_vector(0 to C_OSIF_D
```

Causes the hardware thread to terminate itself. Depending on whether it was created using `reconos_hwthread_create()` or `rthread_create()` (eCos or POSIX), it uses `cyg_thread_exit()` or `pthread_exit()`, respectively. The latter can pass a return value to possible joiner threads.

Parameter	Description
osif_task2os	record of communication signals to the OS interface
osif_os2task	record of communication signals from the OS interface
retval	the return value to be passed to possible joiners (POSIX) or printed on diag (eCos)

Example (inside the synchronization FSM process):

```
...
case state is
...
  when STATE_EXIT =>
    reconos_thread_exit(o_osif, i_osif, X"00000000");
...

```

5 Hardware Architecture

6 Software Architecture

7 Partial Reconfiguration

8 Tools

9 Target Platforms

10 Sample Applications

Topics to be placed

This is a list of topics which have not been placed properly within the manual's outline.

Execution Model

TODO

Multi Cycle Commands

TODO

`reconos_hwthread_create`

Message Boxes

OS Interface

Environment Variables