

Department of Electrical and Computer Engineering  
Aarhus University

# Embedded Real Time Systems

## Assignment 1

**Students:**

Deyana Atanasova  
Micha Heiß

September 19, 2021

# 1 Exercise 3.1

## 1.1 Implementation

The exercise is implemented with a systemc module. The module has an event, a counter variable, a method, and a thread. The counter value is a 4bit wide unsigned integer, which can represent values between 0 and 15. The method 'notifyMe' is sensitive to the event 'e' and will therefore be executed once for every time the event 'e' is triggered. It will first increment the counter and then print the current simulation time and the counter value. The thread 'notifier' calls the notify method of the event 'e' every 2ms in a loop. It will be started when the simulation starts.

```
2  #include <systemc>
3  #include <sysc/datatypes/int/sc_uint.h>
4  using namespace sc_core;
5
6  SC_MODULE (ModuleSingle) {
7      sc_dt::sc_uint<4> counter;      // counter value
8      sc_event e;                    // create event
9      SC_CTOR(ModuleSingle){
10         SC_METHOD(notifyMe);        // register method
11         sensitive << e;              // set sensitivity to event
12         SC_THREAD(notifier);        // register thread
13     }
14     void notifyMe(void){
15         counter++;                  // increment counter
16         std::cout << sc_time_stamp()
17             << " counter: " << counter << std::endl;
18     }
19     void notifier(void){
20         while (true){
21             wait(2, SC_MS);          // delay 2ms
22             e.notify();              // notify method with event
23         }
24     }
25 };
26 // main function
27 int sc_main(int, char*[]) {
28     ModuleSingle mS("test");        // initialize module
29     sc_start(200, SC_MS);           // start systemc sim kernel
30     return 0;
31 }
```

Figure 1: Implementation of Module Single, exercise 3.1

To start the simulation an instance of the module has to be created first and then the simulation can be started. In this exercise the simulation is restricted

to 200ms and will stop execution of all threads and methods after that time.

## 1.2 Results

In the simulation output it can be observed, that the counter value overflows after counting up to 15. This is to be expected, because the counter cannot hold a bigger value than that. The output in figure 2 is cut after 50ms but the actual simulation ran until 200ms. The observations after 50ms are the same, as the output continues likewise. The last counter output is 4 after 198ms of simulation.

```
SystemC 2.3.1-Accellera --- Sep 12 2021 09:49:44
Copyright (c) 1996-2014 by all Contributors,
ALL RIGHTS RESERVED
0 s counter: 1
2 ms counter: 2
4 ms counter: 3
6 ms counter: 4
8 ms counter: 5
10 ms counter: 6
12 ms counter: 7
14 ms counter: 8
16 ms counter: 9
18 ms counter: 10
20 ms counter: 11
22 ms counter: 12
24 ms counter: 13
26 ms counter: 14
28 ms counter: 15
30 ms counter: 0
32 ms counter: 1
34 ms counter: 2
36 ms counter: 3
38 ms counter: 4
40 ms counter: 5
42 ms counter: 6
44 ms counter: 7
46 ms counter: 8
48 ms counter: 9
50 ms counter: 10
```

Figure 2: Simulation results of Module Single, exercise 3.1

## 2 Exercise 3.2

### 2.1 Implementation

For this exercise the module "module\_double" was implemented. It contains 2 threads - "thread\_a" and "thread\_b", and 1 method - "method\_a". Apart from that it contains 4 events - "event\_a", "event\_b", "ack\_a", and "ack\_b". Thread A notifies event A every 3 ms, and thread B notifies event B every 2ms. Each thread then waits for an acknowledgement event or a timeout after it notifies its event. Method A is used to wait on event A or B alternating between them, starting with event B because it's the first event triggered in the simulation (at 2 ms). Dynamic sensitivity is used to set the next event to trigger the method depending on which event triggered the method. The implementation was separated into 3 files - module\_double.h, module\_double.cpp, and main.cpp. Code snippets can be observed in Figure 3 and Figure 2.1.

```
SC_MODULE(module_double) {  
    void thread_a(void);  
    void thread_b(void);  
    void method_a(void);  
    sc_event event_a, event_b, ack_a, ack_b;  
  
    SC_CTOR(module_double) {  
        SC_THREAD(thread_a);  
        SC_THREAD(thread_b);  
        SC_METHOD(method_a);  
    }  
};
```

Figure 3: module\_double.h

### 2.2 Results

In Figure 2.2 it can be observed from the timestamps that events A and B are triggered every 3ms and 2ms respectively. It can also be observed that method A alternates between waiting for event A and B. In case an event occurs and method A is not listening for it, it does not send an acknowledgement back and a timeout occurs, after which the event is triggered again. This behaviour can be observed when event B goes unnoticed at 6ms, so the next time it is triggered

```

void module_double::thread_a(void) {
    sc_time time_a(3, SC_MS);
    while (true) {
        wait(time_a); //notify event A in 3ms
        event_a.notify();
        cout << sc_time_stamp() << ": Event A triggered, waiting for acknowledgment A" << endl;
        wait(time_a, ack_a); //wait for timeout or acknowledgement
    }
}

void module_double::thread_b(void) {
    sc_time time_b(2, SC_MS);
    while (true)
    {
        wait(time_b); //notify event B in 2ms
        event_b.notify();
        cout << sc_time_stamp() << ": Event B triggered, waiting for acknowledgment B" << endl;
        wait(time_b, ack_b); //wait for timeout or acknowledgement
    }
}

void module_double::method_a(void) {
    next_trigger(event_b);
    if (event_a.triggered()) {
        ack_a.notify();
        cout << sc_time_stamp() << ": Sending acknowledgement for A" << endl;
        next_trigger(event_b);
        cout << sc_time_stamp() << ": Waiting for event B" << endl;
    }
    else if (event_b.triggered()) {
        ack_b.notify();
        cout << sc_time_stamp() << ": Sending acknowledgement for B" << endl;
        next_trigger(event_a);
        cout << sc_time_stamp() << ": Waiting for event A" << endl;
    }
}

```

Figure 4: module\_double.cpp

is at 10ms (it takes 2ms to timeout and 2ms for the event to be triggered again).

## 3 Exercise 3.3

### 3.1 Implementation

In this exercise, two modules were created, that communicate with each other over TCP packets. The packets are sent over a fifo queue. The two modules are connected in the 'tcp\_top' module.

The producer module has a thread, that sends out packets containing a sequence number with a random time delay between 2 and 10 milliseconds (implementation see figure 6).

The consumer module has a thread, that waits for the 'data\_written\_event' of the fifo queue and reads the packet, if the event is received. It will then print the sequence number and current simulation time (implementation see figure 7).

```
2 ms: Event B triggered, waiting for acknowledgment B
2 ms: Sending acknowledgement for B
2 ms: Waiting for event A
3 ms: Event A triggered, waiting for acknowledgment A
3 ms: Sending acknowledgement for A
3 ms: Waiting for event B
4 ms: Event B triggered, waiting for acknowledgment B
4 ms: Sending acknowledgement for B
4 ms: Waiting for event A
6 ms: Event B triggered, waiting for acknowledgment B
6 ms: Event A triggered, waiting for acknowledgment A
6 ms: Sending acknowledgement for A
6 ms: Waiting for event B
9 ms: Event A triggered, waiting for acknowledgment A
10 ms: Event B triggered, waiting for acknowledgment B
10 ms: Sending acknowledgement for B
10 ms: Waiting for event A
12 ms: Event B triggered, waiting for acknowledgment B
15 ms: Event A triggered, waiting for acknowledgment A
15 ms: Sending acknowledgement for A
```

Figure 5: Simulation result

As an extension, a fork module was implemented for this exercise. Its task is to support multiple output queues, sending every packet that it receives on the input fifo queue to each of the registered output queues. This allows for multiple consumers listening to the same producer (see figure 8).

### 3.2 Results

In the output (figure 9) one can observe, that the time interval between the packets sent from the producer is random and the corresponding reception simulation time printed from the consumers matches those intervals. Namely the printed simulation time increases by the random time interval for each packet sent over the fifo queue. This works equally well with two consumer attached to the producer via the fork module.

```

void tcp_producer::producer_thread(void){
    srand(time(NULL));
    while(true){
        // get random number between 2-10ms
        double delay = ( rand() % 8 ) + 2;
        // wait for random time
        wait(delay, sc_core::SC_MS);          // delay 2ms
        std::cout << "random delay: " << delay << std::endl;
        // update sequence number
        packet.SequenceNumber++;
        // send tcp packet to consumer over fifo queue
        out->write(&packet);
    }
}

```

Figure 6: Implementation of producer thread, exercise 3.3

```

void tcp_consumer::consumer_thread(void){
    while(true){
        TCPHeader* packet;
        // wait for message to arrive
        if(in.num_available() == 0)
            wait(in.data_written_event());
        packet = in.read();

        // print simulation time and sequence number
        std::cout << sc_core::sc_time_stamp() << ": "
            << this->id << ", received "
            << packet->SequenceNumber << std::endl;
    }
}

```

Figure 7: Implementation of consumer thread, exercise 3.3

```

template <class T>
class tcp_fork : public sc_core::sc_module
{
public:
    sc_core::sc_fifo_in<T> in;
    sc_core::sc_port<sc_core::sc_fifo_out_if<T>,0> out;
    SC_HAS_PROCESS(tcp_fork);
    tcp_fork(sc_core::sc_module_name name) : sc_module(name)
    {
        SC_THREAD(do_fork);
    }
private:
    void do_fork()
    {
        T sample;
        while (1)
        {
            sample = in.read();
            for (int i = 0; i < out.size(); i++)
            {
                out[i]->write(sample);
            }
        }
    }
};

```

Figure 8: Implementation of fork module, similar to example code, exercise 3.3



```
random delay: 7
7 ms: c1, received 1
7 ms: c2, received 1
random delay: 3
10 ms: c1, received 2
10 ms: c2, received 2
random delay: 8
18 ms: c1, received 3
18 ms: c2, received 3
random delay: 2
20 ms: c1, received 4
20 ms: c2, received 4
random delay: 8
28 ms: c1, received 5
28 ms: c2, received 5
random delay: 4
32 ms: c1, received 6
32 ms: c2, received 6
random delay: 9
41 ms: c1, received 7
41 ms: c2, received 7
```

Figure 9: Test output of simulation for exercise3.3

## 4 Exercise 3.4

### 4.1 Implementation

Several modules and a configuration file were implemented to solve this exercise. The data, error, and channel size, as well as the clock period(CLK\_PERIOD) and the ready latency(READY\_LATENCY) are configured in the `config.h` file, which can be seen in Figure 10.

```
#ifndef CONFIG_H
# define CONFIG_H

#define CHANNEL_BITS      4    // number of channel wires
#define ERROR_BITS        2    // number of error wires
#define DATA_BITS        16   // number of data wires
#define MAX_CHANNEL        2    // maximum number of channels actually used
#define CLK_PERIOD        20   // clock period in ns
#define READY_LATENCY     1    // ready latency

#define TRACE_FILE_NAME    "trace.vcd"    // name of file the trace gets saved to

#endif
```

Figure 10: config.h

The source module awaits a `ready` signal from the sink. Upon receiving a `ready=true` signal, the source waits for `READY_LATENCY` clock cycles, and then transmits data until the sink transmits a `ready=false` signal. The code for transmitting the data to the sink can be seen in Figure 11.

```
void source_module::transmit_data()
{
    while (true)
    {
        while (!data_ready)
        {
            wait();
        }

        for (int i = 0; i < READY_LATENCY; i++) {
            wait();
        }

        for (int i = 0; data_ready; i++)
        {
            data_valid->write(true);
            out_error->write(0);
            for (int j = 0; j < out_channel.size(); j++)
            {
                out_channel[j]->write(0);
                out_data->write(i % (1 << DATA_BITS));
                wait();
            }

            data_valid->write(false);
        }
    }
}
```

Figure 11: Transmitting data to sink

The sink module has two processes - one to receive data from the sink and one to transmit the `ready` signal.

The `transmit_data_ready` process, shown in Figure 12 alternates between sending a `ready=true` and `ready=false` signals with an interval of `CLK_PERIOD*3`.

```
void sink_module::transmit_data_ready() {
    while (true)
    {
        data_ready->write(true);
        wait(CLK_PERIOD*3, SC_NS);
        data_ready->write(false);
        wait(CLK_PERIOD * 3, SC_NS);
    }
}
```

Figure 12: Transmitting `ready` signal

The `receive_data` process, which can be seen in Figure 13, waits for a `valid` signal from the source, signifying valid data is being sent, and then reads the and writes the data bits and error bits received from the source into a `.txt` file.

```
void sink_module::receive_data()
{
    ofstream received_data;
    received_data.open("received_data.txt");

    while (true)
    {
        while (!data_valid)
            wait();

        while (data_valid)
        {
            received_data << in_data->read() << " received at simulation time " << sc_time_stamp() << endl;
            in_channel->read();

            int errorno = in_error->read();
            if (errorno != 0)
            {
                received_data << "Error occurred with code ERRNO " << in_error << " at simulation time " << sc_time_stamp() << endl;
            }

            wait();
        }
    }
    received_data.close();
}
```

Figure 13: Receiving data from source

The monitor module contains a few basic processes for displaying the simulation results in the console and creating a VCD trace file. The code for creating the VCD trace file can be seen in Figure 14.

## 4.2 Results

The simulation runs for `CLK_PERIOD*200` ns. A part of the simulation result, shown in Figure 15, is visualized using GTK viewer.

```

void monitor_module::start_of_simulation() {
    // Create tracefile
    trace_file = sc_create_vcd_trace_file("simulation_result");
    if (!trace_file) cout << "Could not create trace file." << endl;

    // Set resolution of trace file to be in 1 ns
    trace_file->set_time_unit(1, SC_NS);

    sc_trace(trace_file, clk, "clk");
    sc_trace(trace_file, data_valid, "valid");
    sc_trace(trace_file, data_ready, "ready");
    sc_trace(trace_file, in_data, "data");
    sc_trace(trace_file, in_error, "error");
    for (int i = 0; i < in_channel.size(); i++)
        sc_trace(trace_file, *in_channel[i], "channel(" + std::to_string(i) + ")");
}

void monitor_module::end_of_simulation() {
    sc_close_vcd_trace_file(trace_file);
    cout << "Created simulation_result.vcd" << endl;
}

```

Figure 14: Creating VCD trace file with the simulation results

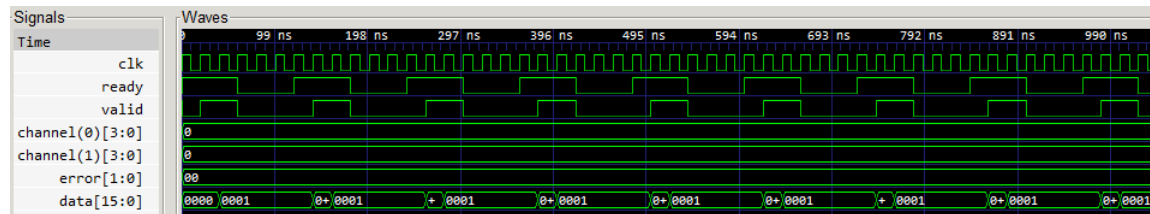
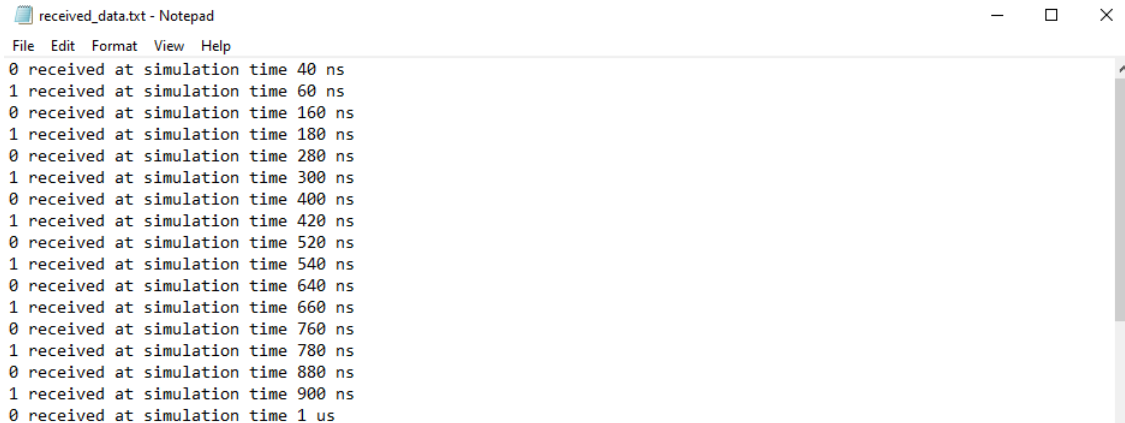


Figure 15: Simulation result in GTK wave viewer

The simulation also produces a .txt file with the received data from the source, which is shown in Figure 16.



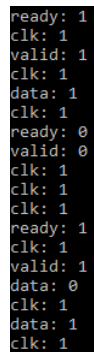
```

received_data.txt - Notepad
File Edit Format View Help
0 received at simulation time 40 ns
1 received at simulation time 60 ns
0 received at simulation time 160 ns
1 received at simulation time 180 ns
0 received at simulation time 280 ns
1 received at simulation time 300 ns
0 received at simulation time 400 ns
1 received at simulation time 420 ns
0 received at simulation time 520 ns
1 received at simulation time 540 ns
0 received at simulation time 640 ns
1 received at simulation time 660 ns
0 received at simulation time 760 ns
1 received at simulation time 780 ns
0 received at simulation time 880 ns
1 received at simulation time 900 ns
0 received at simulation time 1 us

```

Figure 16: Data from source

In case the sink transmits a `ready=false` signal, the source does not send data to it until it receives a `ready=true` signal. This behaviour can be observed from the simulation results displayed in the console and shown in Figure 17. We can see that data is being received only when `ready=true`.



```

ready: 1
clk: 1
valid: 1
clk: 1
data: 1
clk: 1
ready: 0
valid: 0
clk: 1
clk: 1
clk: 1
ready: 1
clk: 1
valid: 1
data: 0
clk: 1
data: 1
clk: 1

```

Figure 17: Data sink signals `ready=false`

## 5 Exercise 3.5

### 5.1 Implementation

The implementation of this exercise extends the functionality of exercise 3.4 by an input adapter, that converts from a fifo queue (TLM) to the Avalon

Streaming Interface (BCAM) and a master module that sends test data to the adapter, instead of the previously used source. The 'InAdapter' template was mostly taken from the given examples, changing only minor things. It's implementation can be seen in figure 18. The master module simply counts

```
void write(const T &value)
{
    if(reset == SC_LOGIC_0){
        while(ready == SC_LOGIC_0)
            wait(clock.posedge_event());
        data.write(value);
        channel.write(0);
        error.write(0);
        // changed to neg edge event...
        wait(clock.negedge_event());
        valid.write(SC_LOGIC_1);
        // ... to make sure, valid signal is high only at exactly one rising edge...
        // ... otherwise each data point might be received twice
        wait(clock.negedge_event());
        valid.write(SC_LOGIC_0);
    }
    else wait(clock.posedge_event());
}
```

Figure 18: Implementation of InAdapter, exercise 3.5

down from 10, sending each number as an integer into the fifo queue (figure 19). It waits two microseconds between sending integers.

```
while(1){
    for(int i = 10; i>0; i--){
        std::cout << "Master sending " << i << std::endl;
        this->out.write(i);
        wait(2, SC_US);
    }
}
```

Figure 19: Implementation of master module, exercise 3.5

## 5.2 Results

In the wave output of the simulation (figure 20) one can observe that the sink signals 'ready', expecting to receive data from the master (via the adapter). The 'InAdapter' module will then, if data from the master is available in the fifo queue, send that data by putting it on the data lines and signaling valid during the next rising flank of the clock. The pauses that the master creates by waiting in between transmissions cause time intervals where the sink signals

ready but doesn't receive any data (signalled by the valid signal not being high).

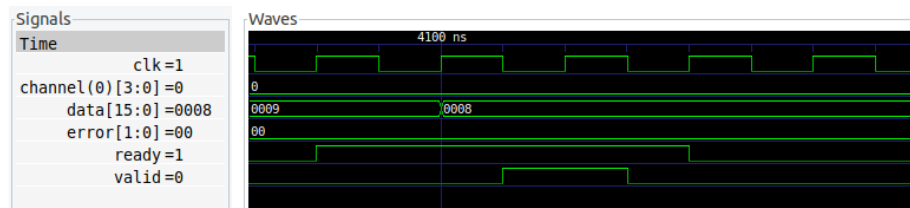


Figure 20: Wave output of exercise 3.5