Embedded Real Time Systems - Assignment 1

Henrik Bagger Jensen, David Jensen and Christian M. Lillelund

September 12, 2018

1 Introduction

This report details the five exercises completed in assignment 1 for the course "Embedded Real Time Systems". All exercises revolve around the modeling language SystemC and uses it to build models at the system level of abstraction using modules, methods, threads etcetera. Each exercise contains a short description of the goal, how we did it with SystemC and code snippets for demonstration purposes.

All code listings are presented in the following manner:

```
1  #include <stdio.h>
2  #define N 10
3  /* Block
4  * comment */
5
6  int main()
7  {
8          int i;
9
10         // Line comment.
11         puts("Hello world!");
12
13         for (i = 0; i < N; i++) {
14                 puts("LaTeX is also great for programmers!");
15         }
16
17         return 0;
18 }
```

Listing 1: Example listing.

2 Exercise 3.1 - ModuleSingle

We implemented the application with a main.cpp and a ModuleSingle.h file, containing the headers and the implementation code. The main.cpp starts the simulation by instantiating the Single class and calling *sc_start()* running for $200MS$. This is shown in listing 2

```
1 int sc_main(int argc, char* argv[]) {
2         Single ModuleSingle("my_instance 1");
3         sc_start(200,SC_MS);
4         return 0;
5 }
```

Listing 2: Application file for ModuleSingle

The ModuleSingle.h defines the Single module functionality that performs the exercise task. It contains a single thread *my_thread_process* and a single method *my_method*. The method is static sensitive to the event *my_event*. Inside the method is simply a counter that increments a *sc_uint¡4¿* variable when invoked and outputs the value using cout. The thread executes in a while-loop that notifies the method every $2MS$ by calling the event *my_event*. The counter variable is just 4 bits, so it resets when it wraps around (16 or higher). The code for ModuleSingle.h is shown in listing 3.

```
1  SC_MODULE(Single) {
2          sc_event my_event;
3          sc_uint<4> counter;
4          void my_thread_process(void) {
5                  while(true) {
6                          my_event.notify();
7                          wait(2,SC_MS);
8                  }
9          }
10         void my_method(void) {
11                 counter++;
12                 std::cout << "counter: " << counter << "
13                 - Timestamp: " << sc_time_stamp() << std::endl;
14         }
15
16         SC_CTOR (Single) {
17                 counter = 0;
18                 SC_THREAD(my_thread_process)
19                         sensitive << my_event;
20                 SC_METHOD(my_method)
21                         sensitive << my_event;
22         } };
```

Listing 3: Implementation of ModuleSingle

3 Exercise 3.2 - ModuleDouble

For ModuleDouble, the application now includes two threads, one method and four events. We use dynamic sensitivity in this exercise to subscribe and wait for an event to trigger the method. Figure 1 shows the interaction between the threads A and B and method A.

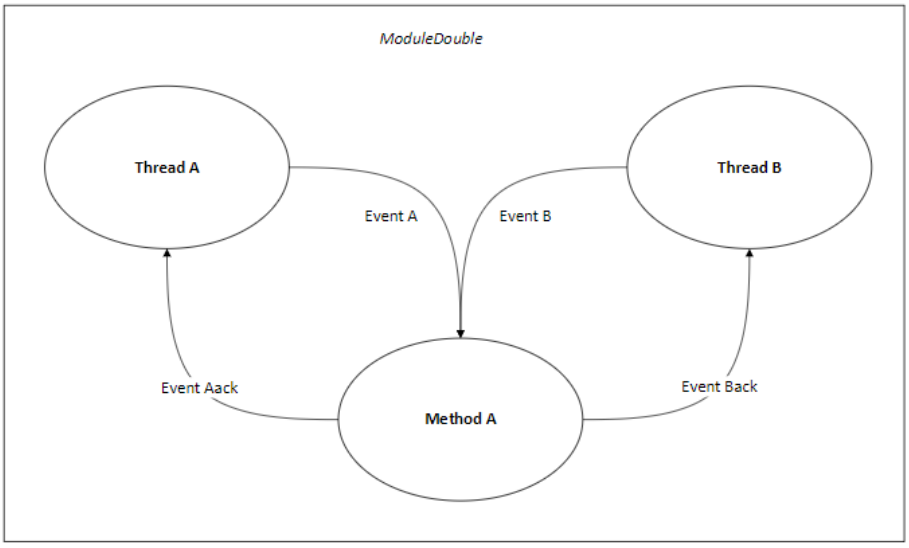

Figure 1: Architecture of ModuleDouble with threads, methods and events.

The application file contains the main.cpp which instantiates ModuleDouble and starts the simulation as shown in listing 4.

```
1 int sc_main(int argc, char* argv[]) {
2         moduleDouble my_module("moduleDouble");
3         sc_start(2000,SC_MS);
4 return 0;
5 }
```

Listing 4: Application file for ModuleDouble.

The functionality is implemented in the ModuleDouble.h file. Firstly, we define the four events that are used to trigger the threads and methods as shown in the architecture. We then introduce a boolean event_tracker variable to enable the method A to alternate between waiting on event A and B. The two threads A and B notifies event A and B respectively and waits for an acknowledgment from the method by calling the *wait()* method. This suspends the thread and waits for the sensitive list event to occur (event Aack and Back respectively). The method alternates between the two events, prints the current time and which event it has received before notifying the acknowledge event and waits for the

opposite thread than the one being acknowledged to invoke it again using the *next_trigger()* method. The code for ModuleDouble.h is shown in listing 5. Note that the *dont_initialize()* method in the constructor prohibits the threads from being executed before the event occur.

```
1  SC_MODULE(moduleDouble){
2
3  sc_event event_a, event_b, event_a_ack, event_b_ack;
4  bool event_tracker;
5
6  void my_threadA(void) {
7          while(true) {
8                  wait(3,SC_MS);
9                  event_a.notify();
10                 wait(3,SC_MS,event_a_ack);
11         }
12 }
13 void my_threadB(void) {
14         while(true) {
15                 wait(2,SC_MS);
16                 event_b.notify();
17                 wait(2,SC_MS,event_b_ack);
18         }
19 }
20 void my_methodA(void){
21         if(event_tracker) {
22                 std::cout << "Timestamp: " << sc_time_stamp() <<
                       "
23                 - event: A" <<  std::endl;
24                 event_a_ack.notify();
25                 event_tracker = false;
26                 // When the method is called next time it must
                       be by event_b
27                 next_trigger(event_b);
28         } else {
29                 std::cout << "Timestamp: " << sc_time_stamp() <<
                       "
30                 - event: B" <<  std::endl;
31                 event_b_ack.notify();
32                 event_tracker = true;
33                 // When the method is called next time it must
                       be by event_a
34                 next_trigger(event_a);
35         }
36 }
37 SC_CTOR (moduleDouble) {
38         SC_THREAD(my_threadA)
39         sensitive << event_a_ack;
40         SC_THREAD(my_threadB)
41         sensitive << event_b_ack;
42         SC_METHOD(my_methodA)
43         sensitive << event_a << event_b;
44         dont_initialize();
45 } };
```

Listing 5: Implementation of ModuleDouble.

4 Exercise 3.3 - Consumer and producer with TCP

This exercise models a producer and consumer thread by two independent modules. They exchange messages (in this case TCP packets) via a SystemC *sc_fifo* queue, with the producer transmitting a new packet every 2-10MS. The consumer receives the packet on its queue and prints out the sequence number and time stamp. The definition of the *TCPHeader* is found in the exercise. We extended the model to include multiple consumers receiving packets on different ports.
Listing 6 shows the application file for our producer/consumer setup. We create one producer, two consumers and then two *sc_fifo* channels for each consumer. The outbound channel of the producer is set to both channels.

```
1  int sc_main(int argc, char* argv[]) {
2
3          producer my_producer("producer");
4          consumer my_consumer1("consumer1");
5          consumer my_consumer2("consumer2");
6
7          sc_fifo<TCPHeader*> C1("C1");
8          sc_fifo<TCPHeader*> C2("C2");
9
10         my_producer.out(C1);
11         my_producer.out(C2);
12         my_consumer1.in(C1);
13         my_consumer2.in(C2);
14
15         sc_start(2000,SC_MS);
16         return 0;
17 }
```

Listing 6: Application file for producer/consumer.

The producer internally uses a *sc_port* class to bind the two channel interfaces. The zero parameter allows binding a unlimited number of interfaces. We can access each binded channel using the index operator as shown in the *producer_thread* and write to it. The producer waits a random time interval, creates a new TCP packet, set a sequence number and writes that to each binded channel. See listing 7.

```
1  SC_MODULE(producer){
2          sc_port<sc_fifo_out_if<TCPHeader*>,0> out;
3          int min = 2;
4          int max = 10;
5          int counter = 0;
6
7          void producer_thread(void) {
8                  while(true) {
9                          int random = rand() % min+max;
10                         wait(random,SC_MS);
11                         TCPHeader* newpacket = new TCPHeader();
12                         counter++;
13                         newpacket->SequenceNumber = counter;
14
15                         for(int i = 0; i < out.size(); i++) {
16                                 out[i]->write(newpacket);
17                         }
18                 }
19         }
20 SC_CTOR (producer) {
21         SC_THREAD(producer_thread)
22 }
23 };
```

Listing 7: Implementation of TCP producer.

The consumer receives the TCP packets in a thread and prints out the time stamp and the sequence number. Note the *object* variable is a pointer to a *TCPHeader* type, hence the arrow notation to dereference it. See listing 8.

```
1  SC_MODULE(consumer){
2          sc_fifo_in<TCPHeader*> in;
3
4          void consumer_thread(void) {
5          while(true) {
6                  TCPHeader* object = in->read();
7                  std::cout << "Timestamp: " << sc_time_stamp() <<
8                  " - Seq: " << object->SequenceNumber <<
9                  " " << name() << std::endl;
10                 }
11         }
12 SC_CTOR (consumer) {
13         SC_THREAD(consumer_thread)
14 }
15 };
```

Listing 8: Implementation of TCP consumer.

5 Exercise 3.4 - Master and slave with ST

In this exercise we create a cycle accurate communication model that uses the Avalon Streaming Bus interface (ST) to send data from a master (data source) to a slave (data sink) by a common clock frequency. The sink notifies the source whenever it is ready to receive new data and stores the received data in a text file. The data simulation is shown in a GTKWaveViewer file. The channel configuration is defined in the exercise. Figure 2 shows the architecture of the source and sink.

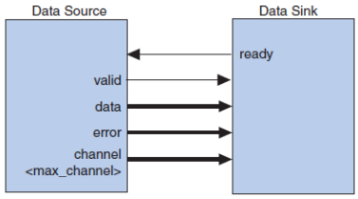

Figure 2: Architecture of the data source and sink.

Listing 10 shows the application file that takes care of the declaration and wiring of channels, the clock and starting the application.

```
1  int sc_main(int argc, char* argv[]) {
2
3          Source Master("Master"); DataSink Slave("Slave");
4
5          sc_clock clock("clock", sc_time(CLK_PERIODE, SC_NS) );
               // 50 MHz
6
7          sc_signal<bool> ready_channel("C1"), valid_channel("C2")
               ;
8
9          sc_signal<sc_uint<DATA_BITS> >  data_channel("C3");
10         sc_signal<sc_uint<ERROR_BITS> > error_channel("C4");
11         sc_signal<sc_uint<CHANNEL_BITS> > channel_channel("C5");
12
13         Master.valid(valid_channel);
14         Master.data(data_channel);
15         Master.ready(ready_channel);
16         Master.error(error_channel);
17         Master.channel(channel_channel);
18         Master.clk(clock);
19         Slave.valid(valid_channel);
20         Slave.data(data_channel);
21         Slave.ready(ready_channel);
22         Slave.error(error_channel);
23         Slave.channel(channel_channel);
24         Slave.clk(clock);
25         sc_start(2000,SC_NS); return 0; }
```

Listing 9: Application file for mater/slave.

Listing 10 shows the implementation of the source. The outgoing channels are modeled with the *sc_out* class and ingoing with *sc_in*. The constructor declares a *source_method()* sensitive to the clock configured in the application file. The method detects the ready signal from the sink and then writes a arbitrary number to the data channel, a random error code and a channel bit. Finally it sets the valid channel to true, letting the sink know data is ready.

```
1  SC_MODULE(Source){
2
3          sc_in<bool> ready;
4          sc_in_clk clk;
5
6          sc_out<bool> valid;
7          sc_out< sc_uint<DATA_BITS> > data;
8          sc_out< sc_uint<ERROR_BITS> > error;
9          sc_out< sc_uint<CHANNEL_BITS> > channel;
10
11         void source_method(void) {
12                 if(ready.read() == true){
13                 data.write(rand() % 65536);
14                 error.write(rand() % 4);
15                 channel.write(rand() % MAX_CHANNEL);
16                 valid.write(true);
17         } else {
18                 valid.write(false);
19         }
20 }
21
22 SC_CTOR (Source) {
23         SC_METHOD(source_method)
24         sensitive << clk;
25         dont_initialize();
26         }
27 };
```

Listing 10: Implementation of the data source.

Listing 11 details the sink. Data is received when valid is signaled, stored and written to an external text file. The global clock invokes the *sink_method()* method. The text file and the waveform file are opened in the constructor and closed in the destructor.

```
SC_MODULE(DataSink){

        ofstream myfile;

        sc_out<bool> ready;
        sc_in_clk clk;

        sc_in<bool> valid;
        sc_in< sc_uint<DATA_BITS> > data;
        sc_in< sc_uint<ERROR_BITS> > error;
        sc_in< sc_uint<CHANNEL_BITS> > channel;

        sc_trace_file * tf;

        void sink_method(void) {
                if(valid.read() == true) {
                        int data = this->data.read();
                        int error = this->error.read();
                        int channel = this->channel.read();

                        ready.write(false);

                        .. // print data to console and file
                } else {
                        ready.write(true);
                }
}

SC_CTOR (DataSink) {
        SC_METHOD(sink_method)
        sensitive << clk;
        dont_initialize();

        ...     // code for generating VCD file
        myfile.open ("data.txt");
}

~DataSink() {
        sc_close_vcd_trace_file(tf);
        myfile.close();
        }
};
```

Listing 11: Implementation of the data sink.

Listing 12 shows a small snippet from the text file with data entries and figure 3 displays the signal states when the first data transfer occurs from the source at time $10NS$. It starts when the clock goes low. The data value is 4567, and the valid signal is set accordingly.

```
Data: 17767
Data: 18547
Data: 37962
Data: 31949
Data: 16882
Data: 57670
```

Listing 12: Output at the sink.

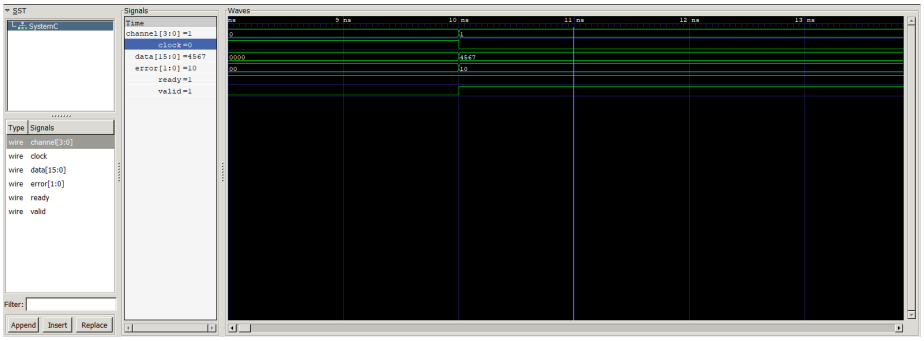

Figure 3: The signal state monitored in GTKWaveViewer.

6 Exercise 3.5 - Master and slave with adapter

In this exercise we move from the TCL modeling layer to BCAM by implementing a adapter between the master and slave (source and sink) from the previous exercise. The source simply puts data on a FIFO queue, then the adapter is responsible for handling and converting the data to the bus cycle accurate interface on the receiving sink. Figure 4 exhibits the setup.

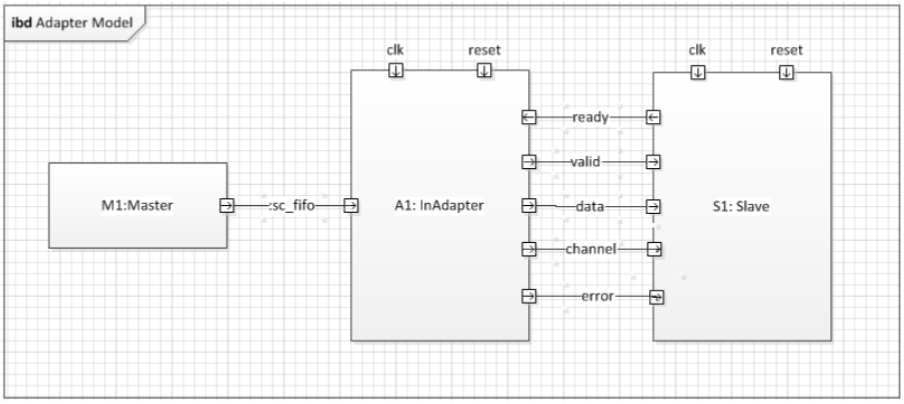

Figure 4: BCAM model with an adapter between the master and slave.

The application file creates instances of the master, slave and the input adapter. We set an appropriate clock, create signals that mimic the ones used in 3.4 and then wire everything together. We introduce a reset and a clock for the adapter to use. It waits for a positive rising edge to sample and write data to the sink. Listing 13 shows the application file.

```
1  int sc_main(int argc, char* argv[]) {
2
3          Source Master("Master");
4          DataSink Slave("Slave");
5          InAdapter<int> inAdapt("instAdapter");
6
7          sc_clock clock("clock", sc_time(CLK_PERIODE, SC_NS) );
               // 50 MHz
8
9          sc_signal<sc_logic> reset;
10         sc_signal<sc_logic> ready_channel("C1");
11         sc_signal<sc_logic> valid_channel("C2") ;
12
13         sc_signal<sc_uint<DATA_BITS> >  data_channel("C3");
14         sc_signal<sc_uint<ERROR_BITS> > error_channel("C4");
15         sc_signal<sc_uint<CHANNEL_BITS> > channel_channel("C5");
16
17         reset = SC_LOGIC_0; // Reset release
18
19         Master.out(inAdapt);
20
21         inAdapt.valid(valid_channel);
22         inAdapt.data(data_channel);
23         inAdapt.ready(ready_channel);
24         inAdapt.error(error_channel);
25         inAdapt.channel(channel_channel);
26         inAdapt.clock(clock);
27         inAdapt.reset(reset);
28
29
30         Slave.valid(valid_channel);
31         Slave.data(data_channel);
32         Slave.ready(ready_channel);
33         Slave.error(error_channel);
34         Slave.channel(channel_channel);
35         Slave.clk(clock);
36
37         sc_start(2000,SC_NS);
38         return 0;
39 }
```

Listing 13: Application file for master/slave with an adapter.

The source has now become very simple, as it just pushes arbitrary data onto a FIFO queue in a thread for the adapter to receive. Listing 14 shows the source.

```
1  SC_MODULE(Source){
2
3          sc_fifo_out<int> out;
4
5          void source_thread(void) {
6                  while(true){
7                          out.write(rand() % 65536);
8                          wait(40,SC_NS);
9                  }}
10         SC_CTOR (Source) {
11                 SC_THREAD(source_thread)
12         }
13 };
```

Listing 14: Source module that generates data.

The adapter is implemented as a template C++ class that inherits the *sc_fifo_out_if* of some type T and *sc_module* class making it a *SC_MODULE*. The point of interest is the inherited *write()* method that accepts any type T scalar and convert it to a BCAM model by putting data on specific channels to the sink when it receives the ready signal. Also the reset signal must not have been set. Listing 15 shows the adapter.

```
1
2         template <class T>
3         class InAdapter: public sc_fifo_out_if <T>, public
              sc_module
4         {
5                 // Clock and reset
6                 sc_in_clk clock; // Clock
7                 sc_in<sc_logic> reset; // Reset
8
9                 // Handshake ports for ST bus
10                sc_in<sc_logic> ready; // Ready signal
11                sc_out<sc_logic> valid; // Valid signal
12
13                // Channel, error and data ports ST bus
14                sc_out<sc_uint<CHANNEL_BITS> > channel;
15                sc_out<sc_uint<ERROR_BITS> > error;
16                sc_out<sc_uint<DATA_BITS> > data;
17
18                void write (const T & value)
19                {
20                        if (reset == SC_LOGIC_0)
21                        {
22                        // Output sample data on negative edge
                              of clock
23                        while (ready == SC_LOGIC_0)     {
24                                wait(clock.posedge_event());
25                        }
26                        wait(clock.posedge_event());
27                        data.write(value);
28                        channel.write(0); // Channel number
29                        error.write(0); // Error
30                        valid.write(SC_LOGIC_1); // Signal valid
                               new data
31                        wait(clock.posedge_event());
32                        valid.write(SC_LOGIC_0);
33                }
34                else wait(clock.posedge_event());
35        }
36        InAdapter (sc_module_name name_)
37        : sc_module (name_)
38        { }
39
40        ... // Other inherited methods implemented here.
41        };
```

Listing 15: Adapter that receives FIFO data and converts it.

The sink is similar to the one in 3.4. It is shown in listing 16. We used the SystemC boolean data type *SC_LOGIC* in this exercise rather than the default C++ one.

```
SC_MODULE(DataSink){

        ofstream myfile;

        sc_out<sc_logic> ready;
        sc_in_clk clk;

        sc_in<sc_logic> valid;
        sc_in< sc_uint<DATA_BITS> > data;
        sc_in< sc_uint<ERROR_BITS> > error;
        sc_in< sc_uint<CHANNEL_BITS> > channel;

        sc_trace_file * tf;

        void sink_method(void) {
                if(valid.read() == SC_LOGIC_1){
                        int data = this->data.read();
                        int error = this->error.read();
                        int channel = this->channel.read();

                        ready.write(SC_LOGIC_0);

                        ... // Output data to cout and text file
                            .
                } else {
                        ready.write(SC_LOGIC_1);
                }
        }

        SC_CTOR (DataSink) {
                SC_METHOD(sink_method)
                sensitive << clk;
                dont_initialize();

                tf = sc_create_vcd_trace_file("Waveform");
                ... // Set signal traces
                myfile.open ("data.txt");
        }

        ~DataSink(){
                sc_close_vcd_trace_file(tf);
                myfile.close();
        }
};
```

Listing 16: Sink that receives data.

Listing 17 shows the output at the sink and figure 5 shows the GTKWaveViewer diagram of the signal states.

```
Data:  17767
Data:  17767
Data:  9158
Data:  9158
Data:  39017
Data:  39017
Data:  18547
```

Listing 17: Output at the sink.

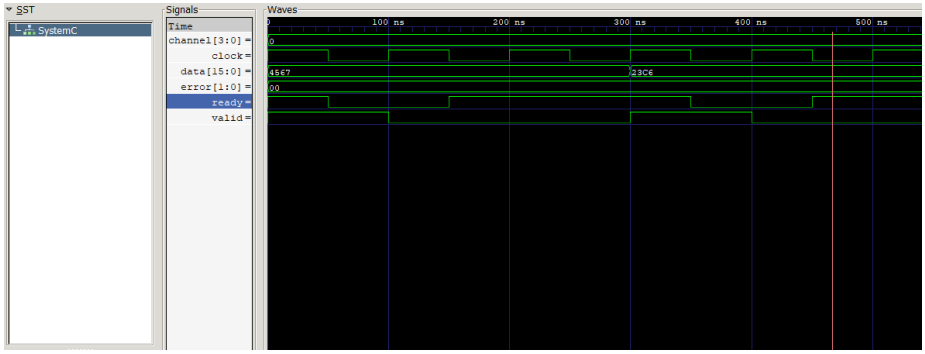

Figure 5: The signal state monitored in GTKWaveViewer.