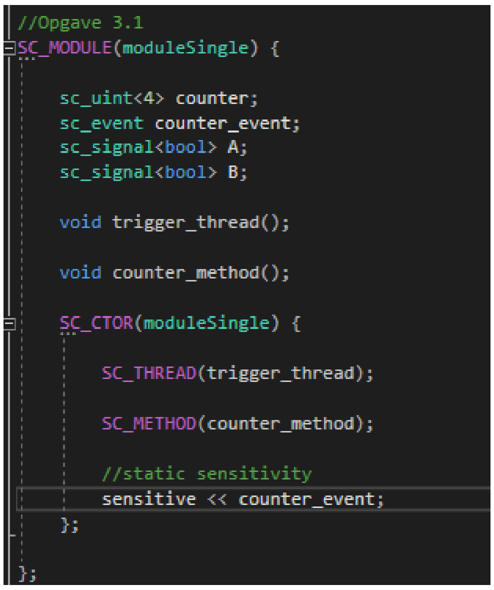
Participants:
Jacob Munkholm, 20140479
Jens Jakob Mikkelsen, 201506215
Yang Hanshou, 201902791

Embedded Real Time Systems – Assignment 1
System level modeling using SystemC

Modules, threads, methods and events

3.1:

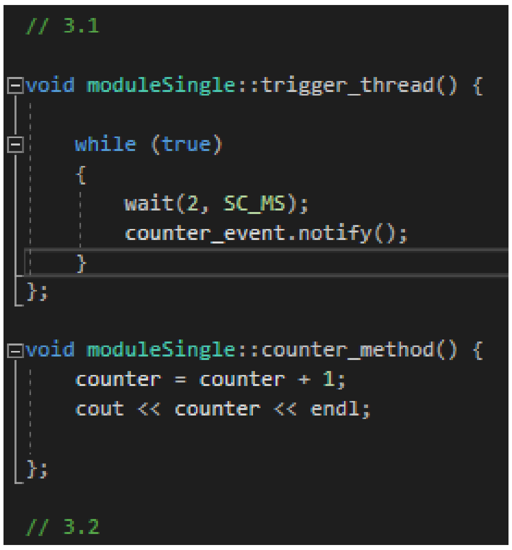
The module ’moduleSingle’ is created:



```
//Opgave 3.1
SC_MODULE(moduleSingle) {

    sc_uint<4> counter;
    sc_event counter_event;
    sc_signal<bool> A;
    sc_signal<bool> B;

    void trigger_thread();

    void counter_method();

    SC_CTOR(moduleSingle) {

        SC_THREAD(trigger_thread);

        SC_METHOD(counter_method);

        //static sensitivity
        sensitive << counter_event;
    };

};
```

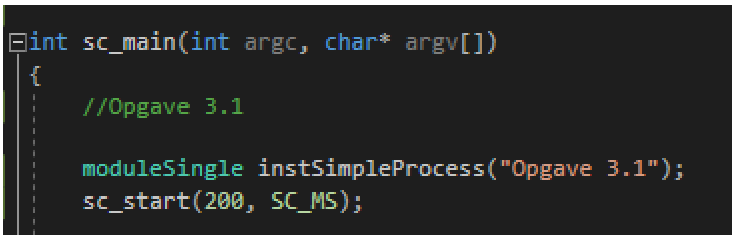
The thread to continuously notify the method, and the method to count, is initialized in the constructor. Static sensitivity is used, so counter_method() is called everytime the event is raised.

’Trigger_thread() og ’counter_method’ looks like this



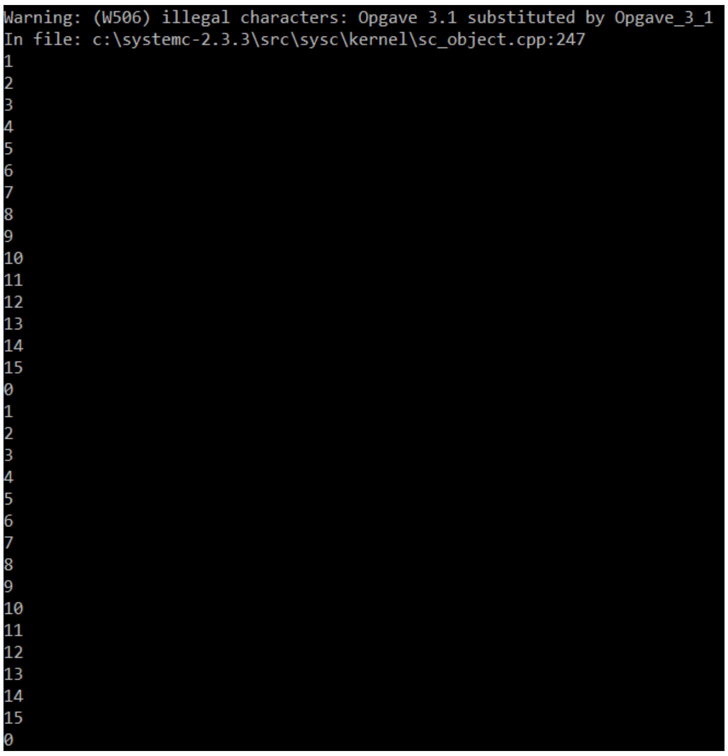
```
// 3.1

void moduleSingle::trigger_thread() {

    while (true)
    {
        wait(2, SC_MS);
        counter_event.notify();
    }
};

void moduleSingle::counter_method() {
    counter = counter + 1;
    cout << counter << endl;

};

// 3.2
```

’Trigger thread’ is an unending loop, calling counter_event.notify() every other ms. In the main function, the simulation is limited to 200 ms:



```
int sc_main(int argc, char* argv[])
{
    //Opgave 3.1

    moduleSingle instSimpleProcess("Opgave 3.1");
    sc_start(200, SC_MS);
```

The following is the result:

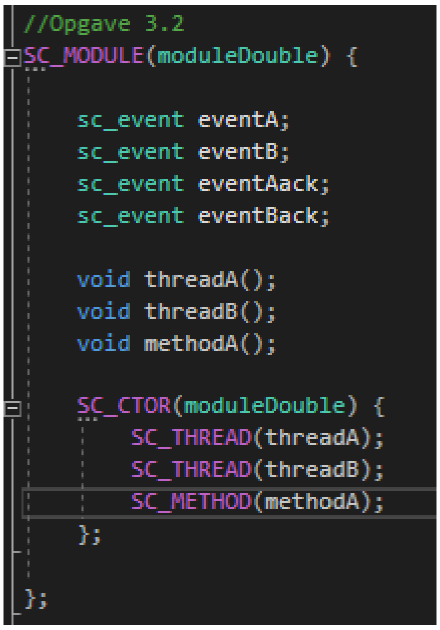


```
Warning: (W506) illegal characters: Opgave 3.1 substituted by Opgave_3_1
In file: c:\systemc-2.3.3\src\sysc\kernel\sc_object.cpp:247
1
2
3
4
5
6
7
8
9
10
11
12
13
14
15
0
1
2
3
4
5
6
7
8
9
10
11
12
13
14
15
0
```

When sc_uint<4> counter overflows, the program will count from zero again.

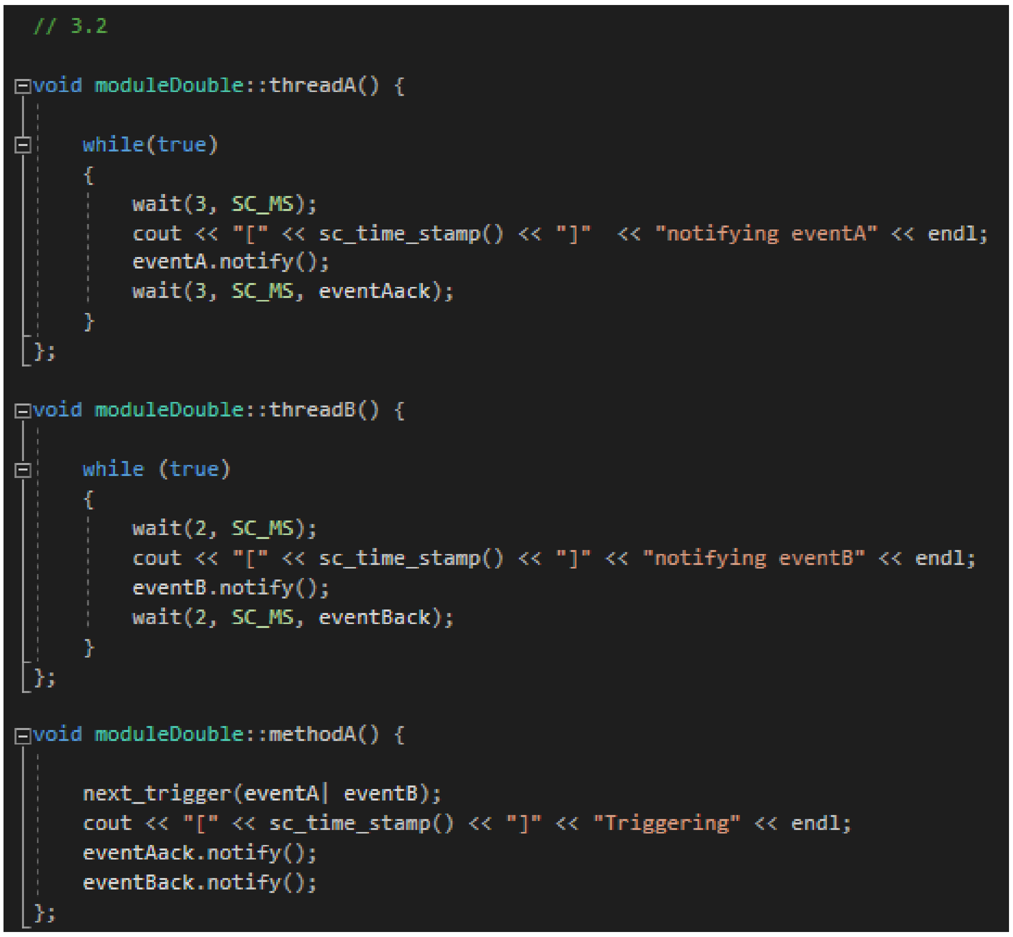
3.2.

The different threads and events and the method is created:



```
//Opgave 3.2
SC_MODULE(moduleDouble) {

    sc_event eventA;
    sc_event eventB;
    sc_event eventAack;
    sc_event eventBack;

    void threadA();
    void threadB();
    void methodA();

    SC_CTOR(moduleDouble) {
        SC_THREAD(threadA);
        SC_THREAD(threadB);
        SC_METHOD(methodA);
    };

};
```

The threads and the method looks like this:

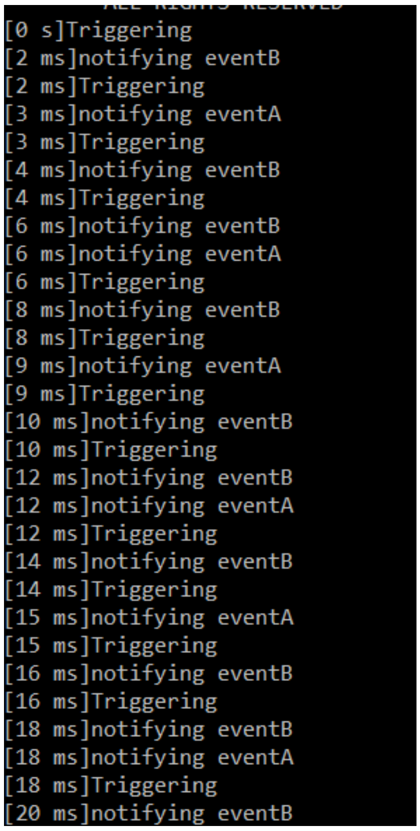


```
 // 3.2

void moduleDouble::threadA() {

    while(true)
    {
        wait(3, SC_MS);
        cout << "[" << sc_time_stamp() << "]"  << "notifying eventA" << endl;
        eventA.notify();
        wait(3, SC_MS, eventAack);
    }
};

void moduleDouble::threadB() {

    while (true)
    {
        wait(2, SC_MS);
        cout << "[" << sc_time_stamp() << "]" << "notifying eventB" << endl;
        eventB.notify();
        wait(2, SC_MS, eventBack);
    }
};

void moduleDouble::methodA() {

    next_trigger(eventA| eventB);
    cout << "[" << sc_time_stamp() << "]" << "Triggering" << endl;
    eventAack.notify();
    eventBack.notify();
};
```

In methodA(), dynamic sensitivity in the form of next_trigger(eventA | eventB) is used, så the method will be called by if either of the events are called. If the program should work so that methodA knows exactly which event is called, signals would need to be used.

Both event threadA and threadB wait for their event to be called. If no event is called, the thread will timeout and restart. This it done in the wait function by specifying eventAack and eventBack as parameters.

The result looks like this:



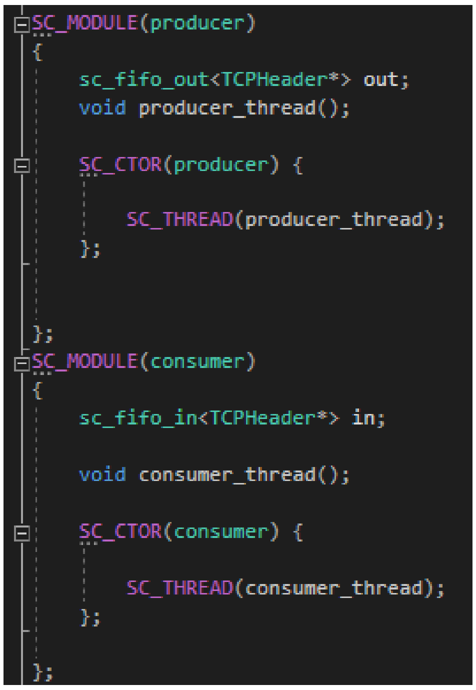
```
[0 s]Triggering
[2 ms]notifying eventB
[2 ms]Triggering
[3 ms]notifying eventA
[3 ms]Triggering
[4 ms]notifying eventB
[4 ms]Triggering
[6 ms]notifying eventB
[6 ms]notifying eventA
[6 ms]Triggering
[8 ms]notifying eventB
[8 ms]Triggering
[9 ms]notifying eventA
[9 ms]Triggering
[10 ms]notifying eventB
[10 ms]Triggering
[12 ms]notifying eventB
[12 ms]notifying eventA
[12 ms]Triggering
[14 ms]notifying eventB
[14 ms]Triggering
[15 ms]notifying eventA
[15 ms]Triggering
[16 ms]notifying eventB
[16 ms]Triggering
[18 ms]notifying eventB
[18 ms]notifying eventA
[18 ms]Triggering
[20 ms]notifying eventB
```

At 6, 12 and 18 ms the events overlap, but eventB is seen to be quickest everytime (because it is called first).

Channels, signals, hierarchy, communication

3.3

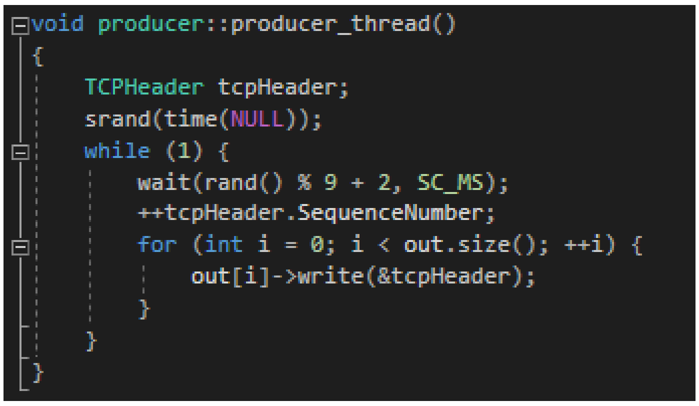
A producer and consumer-thread is created:



```
SC_MODULE(producer)
{
    sc_fifo_out<TCPHeader*> out;
    void producer_thread();

    SC_CTOR(producer) {

        SC_THREAD(producer_thread);
    };


};
SC_MODULE(consumer)
{
    sc_fifo_in<TCPHeader*> in;

    void consumer_thread();

    SC_CTOR(consumer) {

        SC_THREAD(consumer_thread);
    };

};
```

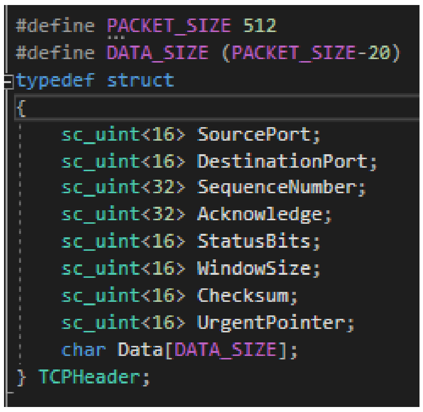
The main object to be used is the sc_fifo_in/out, through which data is transferred.

The producer thread is written like this:



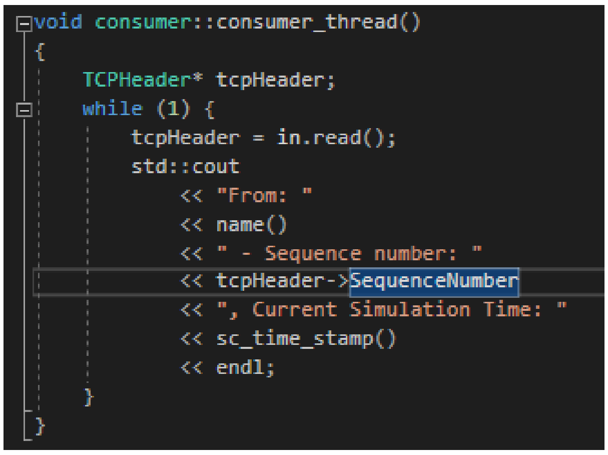
```
void producer::producer_thread()
{
    TCPHeader tcpHeader;
    srand(time(NULL));
    while (1) {
        wait(rand() % 9 + 2, SC_MS);
        ++tcpHeader.SequenceNumber;
        for (int i = 0; i < out.size(); ++i) {
            out[i]->write(&tcpHeader);
        }
    }
}
```

The thread will wait a random amount of time between 2 and 10 ms, increment the sequence-number (the whole TCP segment structure is not used) and write the sequence to the queue. The TCP-struct is defined, but only the sequenceNumber is used to demonstrate the concept:



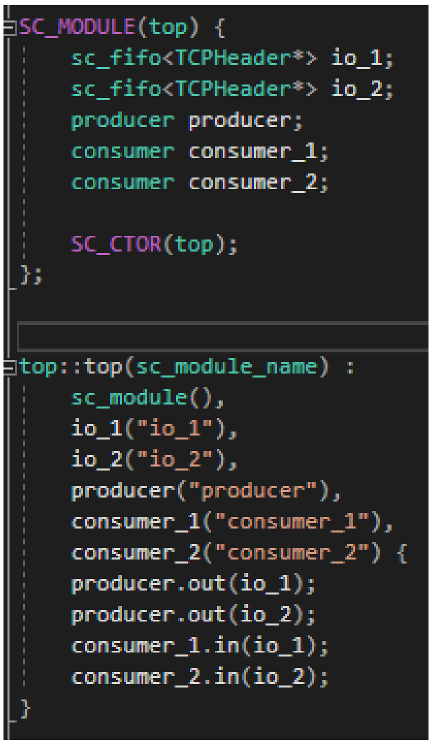
```
#define PACKET_SIZE 512
#define DATA_SIZE (PACKET_SIZE-20)
typedef struct
{
    sc_uint<16> SourcePort;
    sc_uint<16> DestinationPort;
    sc_uint<32> SequenceNumber;
    sc_uint<32> Acknowledge;
    sc_uint<16> StatusBits;
    sc_uint<16> WindowSize;
    sc_uint<16> Checksum;
    sc_uint<16> UrgentPointer;
    char Data[DATA_SIZE];
} TCPHeader;
```

The consumer-thread read the queues continuously and then writes out the sequenceNumber:



```
void consumer::consumer_thread()
{
    TCPHeader* tcpHeader;
    while (1) {
        tcpHeader = in.read();
        std::cout
            << "From: "
            << name()
            << " - Sequence number: "
            << tcpHeader->SequenceNumber
            << ", Current Simulation Time: "
            << sc_time_stamp()
            << endl;
    }
}
```

In the top-module, the producer, consumer and fifos are created. More consumers and producers could be added like this, if need be.



```
SC_MODULE(top) {
    sc_fifo<TCPHeader*> io_1;
    sc_fifo<TCPHeader*> io_2;
    producer producer;
    consumer consumer_1;
    consumer consumer_2;

    SC_CTOR(top);
};

top::top(sc_module_name) :
    sc_module(),
    io_1("io_1"),
    io_2("io_2"),
    producer("producer"),
    consumer_1("consumer_1"),
    consumer_2("consumer_2") {
    producer.out(io_1);
    producer.out(io_2);
    consumer_1.in(io_1);
    consumer_2.in(io_2);
}
```

3.4