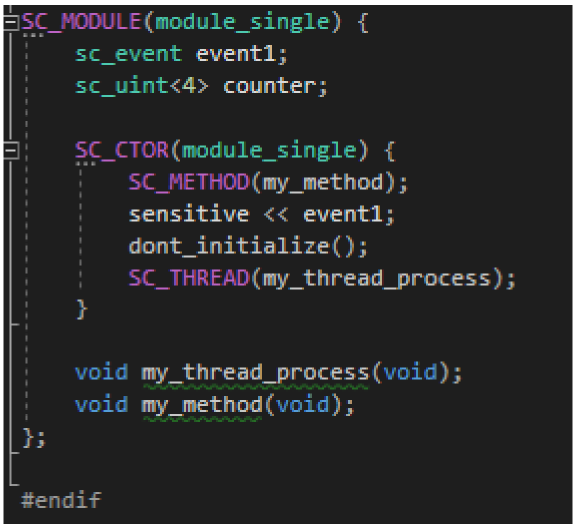
Participants:
Jacob Munkholm, 20140479
Jens Jakob Mikkelsen, 201506215
Yang Hanshou, 201902791

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

Modules, threads, methods and events

3.1:

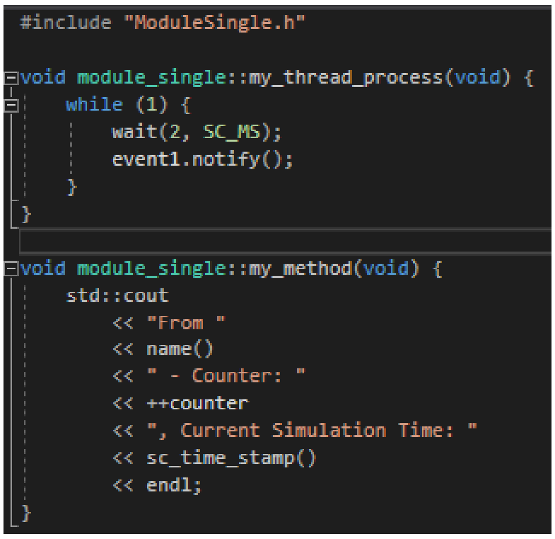
The module 'moduleSingle' is created:



```
SC_MODULE(module_single) {
    sc_event event1;
    sc_uint<4> counter;

    SC_CTOR(module_single) {
        SC_METHOD(my_method);
        sensitive << event1;
        dont_initialize();
        SC_THREAD(my_thread_process);
    }

    void my_thread_process(void);
    void my_method(void);
};

#endif
```

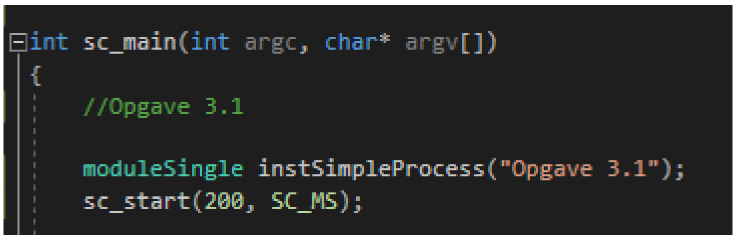
The thread to continuously notify the method, and the method to count, is initialized in the constructor. Static sensitivity is used, my_method is called everytime the event is raised.

The functions looks like this:



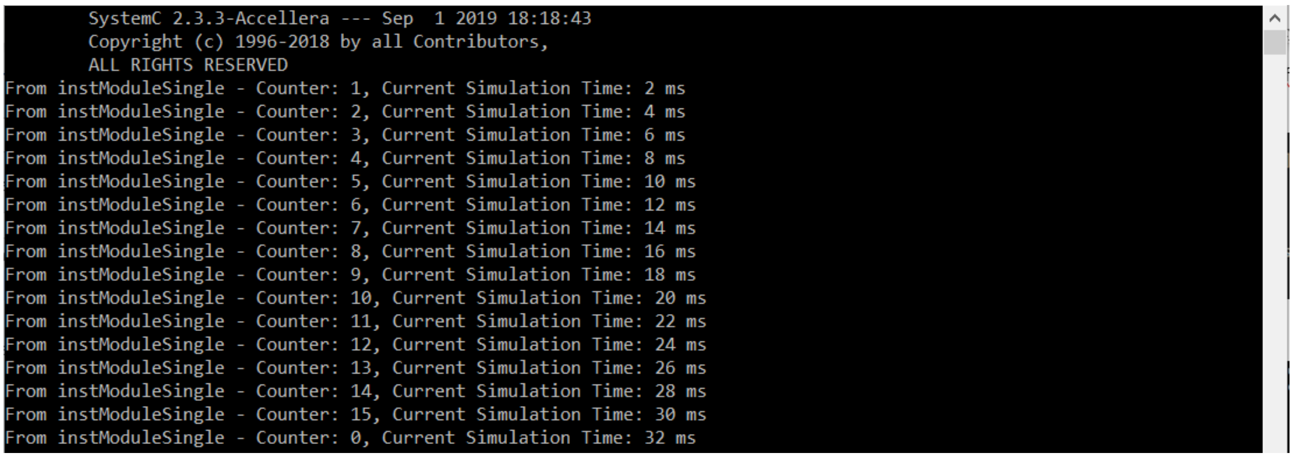
```
#include "ModuleSingle.h"

void module_single::my_thread_process(void) {
    while (1) {
        wait(2, SC_MS);
        event1.notify();
    }
}

void module_single::my_method(void) {
    std::cout
        << "From "
        << name()
        << " - Counter: "
        << ++counter
        << ", Current Simulation Time: "
        << sc_time_stamp()
        << endl;
}
```

’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:

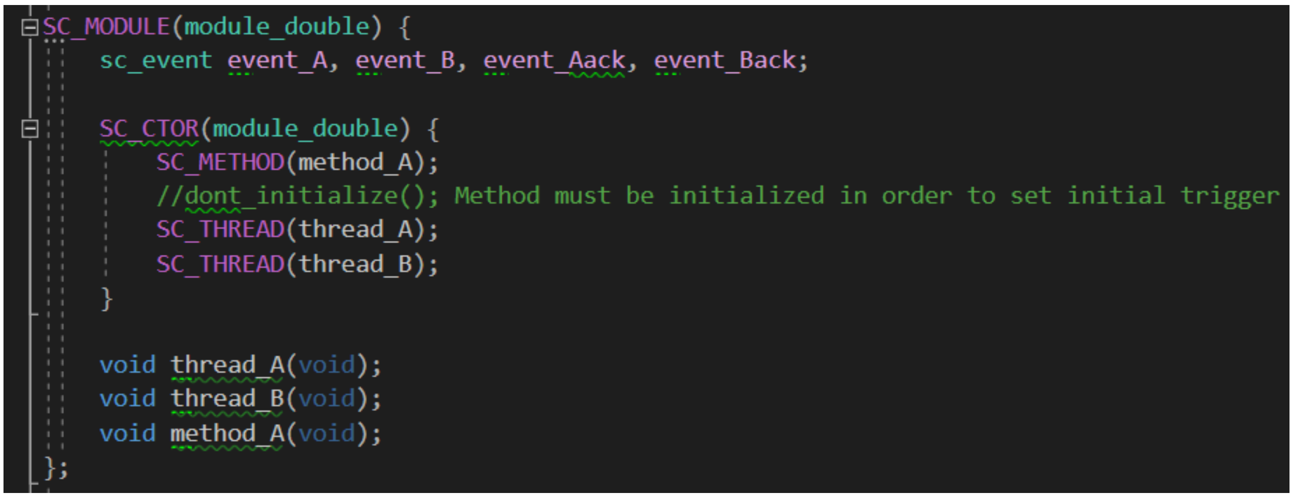


```
        SystemC 2.3.3-Accellera --- Sep  1 2019 18:18:43
        Copyright (c) 1996-2018 by all Contributors,
        ALL RIGHTS RESERVED
From instModuleSingle - Counter: 1, Current Simulation Time: 2 ms
From instModuleSingle - Counter: 2, Current Simulation Time: 4 ms
From instModuleSingle - Counter: 3, Current Simulation Time: 6 ms
From instModuleSingle - Counter: 4, Current Simulation Time: 8 ms
From instModuleSingle - Counter: 5, Current Simulation Time: 10 ms
From instModuleSingle - Counter: 6, Current Simulation Time: 12 ms
From instModuleSingle - Counter: 7, Current Simulation Time: 14 ms
From instModuleSingle - Counter: 8, Current Simulation Time: 16 ms
From instModuleSingle - Counter: 9, Current Simulation Time: 18 ms
From instModuleSingle - Counter: 10, Current Simulation Time: 20 ms
From instModuleSingle - Counter: 11, Current Simulation Time: 22 ms
From instModuleSingle - Counter: 12, Current Simulation Time: 24 ms
From instModuleSingle - Counter: 13, Current Simulation Time: 26 ms
From instModuleSingle - Counter: 14, Current Simulation Time: 28 ms
From instModuleSingle - Counter: 15, Current Simulation Time: 30 ms
From instModuleSingle - Counter: 0, Current Simulation Time: 32 ms
```

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

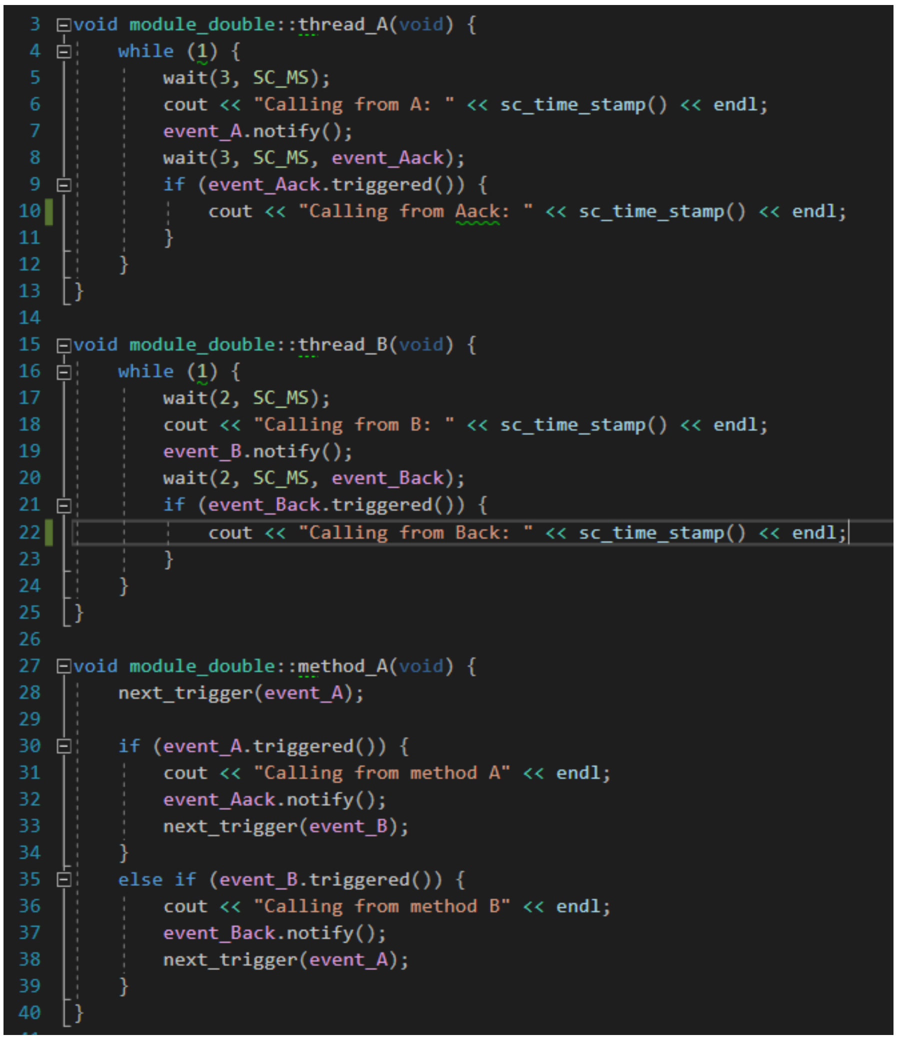
3.2.

The different threads and events and the method are created. It’s worth noting, that since the trigger function of method A is dynamically sensitive, it needs to be initialized in order to call next_trigger() the first time.



```cpp
SC_MODULE(module_double) {
    sc_event event_A, event_B, event_Aack, event_Back;

    SC_CTOR(module_double) {
        SC_METHOD(method_A);
        //dont_initialize(); Method must be initialized in order to set initial trigger
        SC_THREAD(thread_A);
        SC_THREAD(thread_B);
    }

    void thread_A(void);
    void thread_B(void);
    void method_A(void);
};
```

The threads and the method look like this:



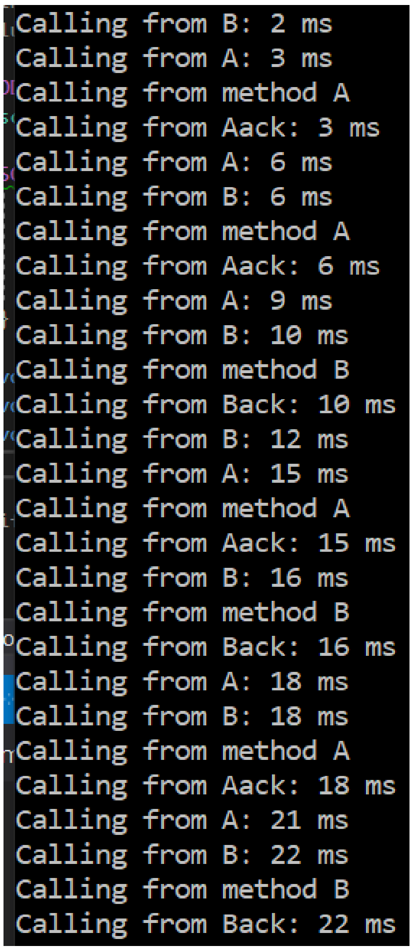
```cpp
void module_double::thread_A(void) {
    while (1) {
        wait(3, SC_MS);
        cout << "Calling from A: " << sc_time_stamp() << endl;
        event_A.notify();
        wait(3, SC_MS, event_Aack);
        if (event_Aack.triggered()) {
            cout << "Calling from Aack: " << sc_time_stamp() << endl;
        }
    }
}

void module_double::thread_B(void) {
    while (1) {
        wait(2, SC_MS);
        cout << "Calling from B: " << sc_time_stamp() << endl;
        event_B.notify();
        wait(2, SC_MS, event_Back);
        if (event_Back.triggered()) {
            cout << "Calling from Back: " << sc_time_stamp() << endl;
        }
    }
}

void module_double::method_A(void) {
    next_trigger(event_A);

    if (event_A.triggered()) {
        cout << "Calling from method A" << endl;
        event_Aack.notify();
        next_trigger(event_B);
    }
    else if (event_B.triggered()) {
        cout << "Calling from method B" << endl;
        event_Back.notify();
        next_trigger(event_A);
    }
}
```

In method_A(), the method is initialized to be triggered by event_A, when the program is running. After that, when the method has been triggered it shifts between setting event_B and event_A as the next trigger. When either of the these has triggered the method, their corresponding acknowledgement event (event_Aack and event_Back) is notified, and the other event is set as the next trigger.

There is one problem with the current solution. The method determines which event triggered the method, by calling triggered() on the event. If both events are triggered at the same time, the method will assume that the triggering event was event_A, as this is the first to be checked. If the program should work so that methodA knows exactly which event is called, signals would need to be used.

Both thread_A and thread_B first notifies their respective event, whereafter they wait for the respective acknowledgement event. Thread_A notifies its event every third second and waits on the acknowledgement for three seconds. The same goes for thread_B but the timespan is to two seconds. This it done in the wait function by specifying event_Aack and event_Back as parameters. If the acknowledgement event is not notified, the thread will timeout and restart. This is checked by calling triggered() on the event.

The result looks like this:



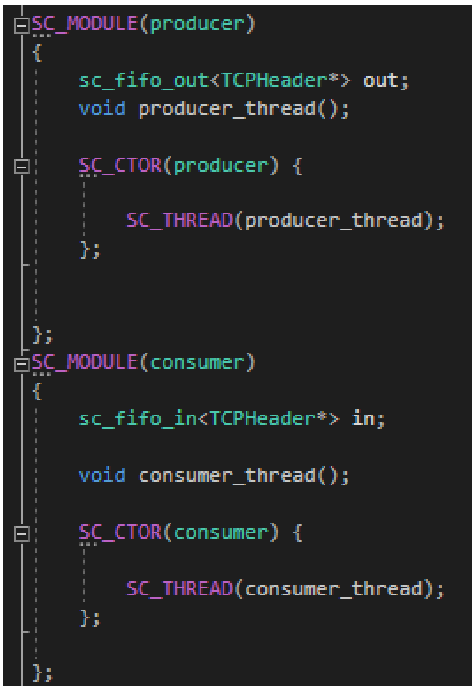
```
Calling from B: 2 ms
Calling from A: 3 ms
Calling from method A
Calling from Aack: 3 ms
Calling from A: 6 ms
Calling from B: 6 ms
Calling from method A
Calling from Aack: 6 ms
Calling from A: 9 ms
Calling from B: 10 ms
Calling from method B
Calling from Back: 10 ms
Calling from B: 12 ms
Calling from A: 15 ms
Calling from method A
Calling from Aack: 15 ms
Calling from B: 16 ms
Calling from method B
Calling from Back: 16 ms
Calling from A: 18 ms
Calling from B: 18 ms
Calling from method A
Calling from Aack: 18 ms
Calling from A: 21 ms
Calling from B: 22 ms
Calling from method B
Calling from Back: 22 ms
```

It can be seen, that even though thread_B is the first to trigger an event, event_A is the first to trigger the method. The aforementioned problem of determining the triggering event, can be seen at 6 ms, where event_B triggered the method, but the method assumed it was event_A, so that only event_Aack was notified. After that, the method steadily shifts between notifying event_Aack and event_Back.

Channels, signals, hierarchy, communication

3.3

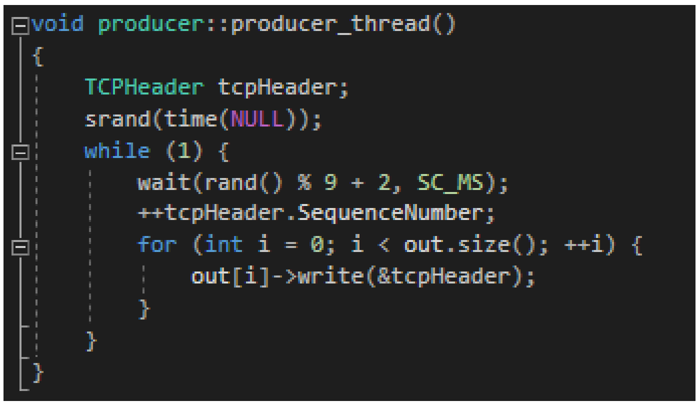
A producer and consumer-thread is created:



```
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);
    };

};
```

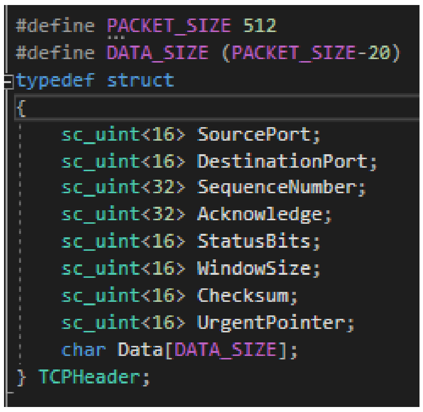
The main object to be used is the sc_fifo_in/out, through which data is transferred.

The producer thread is written like this:



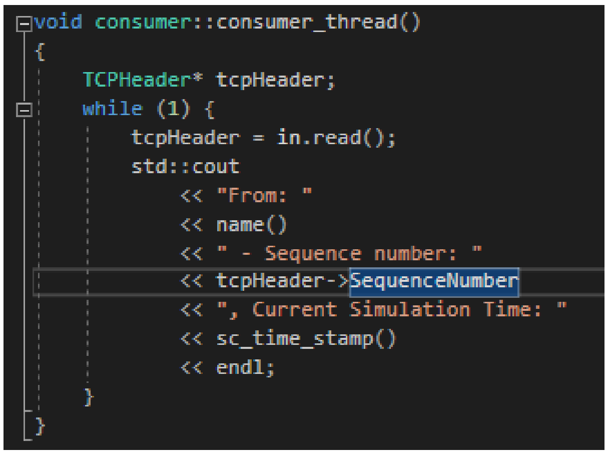
```
void producer::producer_thread()
{
    TCPHeader tcpHeader;
    srand(time(NULL));
    while (1) {
        wait(rand() % 9 + 2, SC_MS);
        ++tcpHeader.SequenceNumber;
        for (int i = 0; i < out.size(); ++i) {
            out[i]->write(&tcpHeader);
        }
    }
}
```

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:



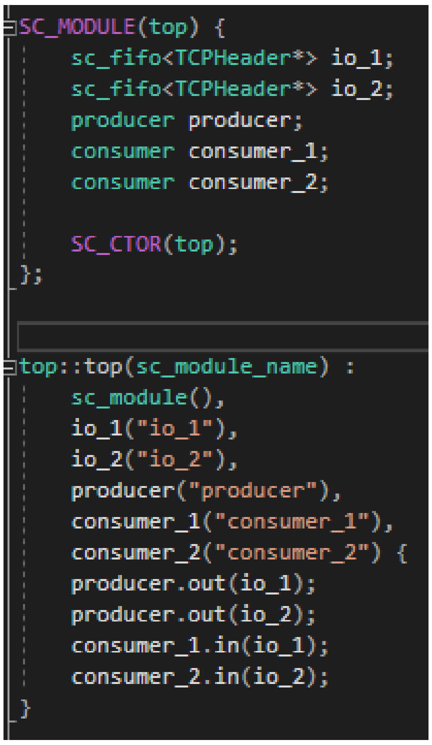
```
#define PACKET_SIZE 512
#define DATA_SIZE (PACKET_SIZE-20)
typedef struct
{
    sc_uint<16> SourcePort;
    sc_uint<16> DestinationPort;
    sc_uint<32> SequenceNumber;
    sc_uint<32> Acknowledge;
    sc_uint<16> StatusBits;
    sc_uint<16> WindowSize;
    sc_uint<16> Checksum;
    sc_uint<16> UrgentPointer;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

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



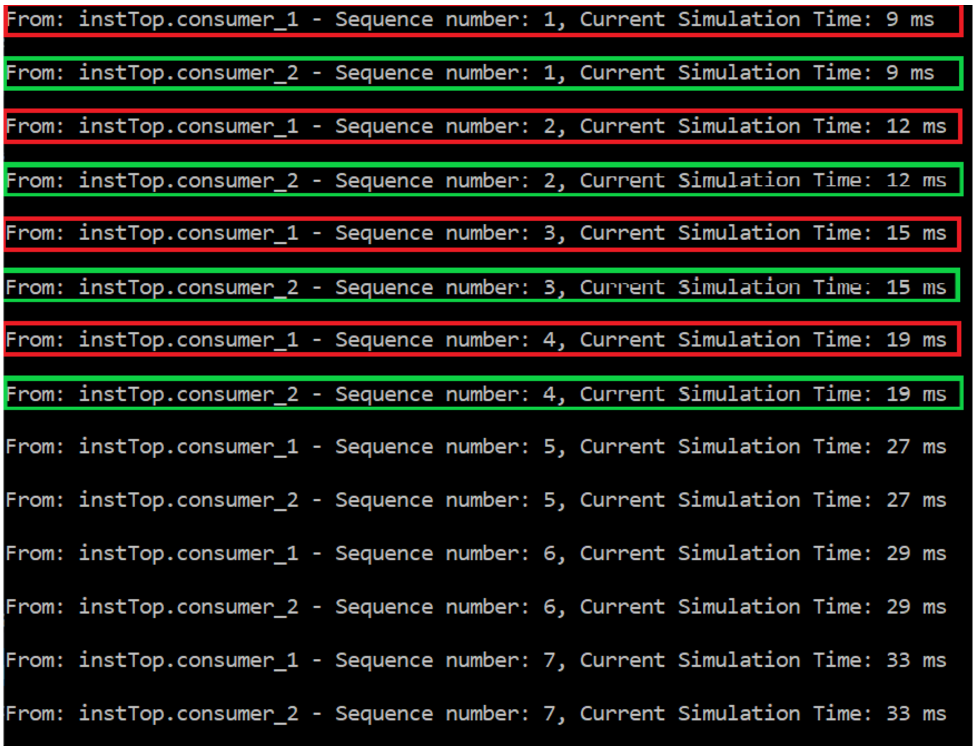
```
void consumer::consumer_thread()
{
    TCPHeader* tcpHeader;
    while (1) {
        tcpHeader = in.read();
        std::cout
            << "From: "
            << name()
            << " - Sequence number: "
            << tcpHeader->SequenceNumber
            << ", Current Simulation Time: "
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

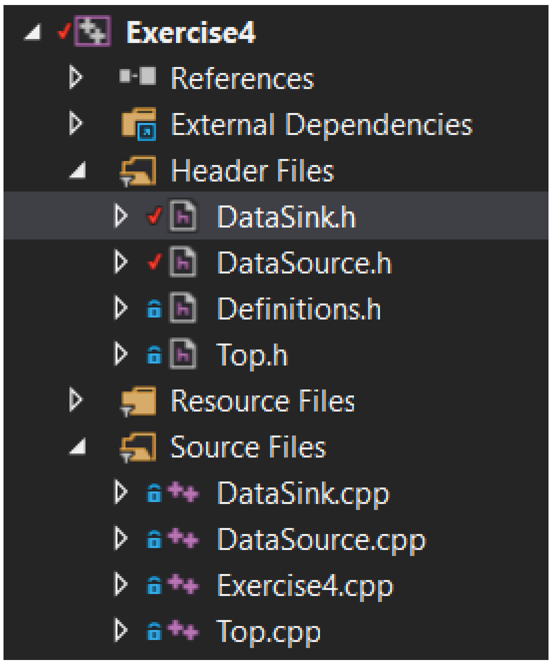
In the console output, it can be seen that each consumer receives the same package (marked by the sequence number) at the same time. Consumer 1 is marked in red, consumer 2 is marked in green.



```
From: instTop.consumer_1 - Sequence number: 1, Current Simulation Time: 9 ms
From: instTop.consumer_2 - Sequence number: 1, Current Simulation Time: 9 ms
From: instTop.consumer_1 - Sequence number: 2, Current Simulation Time: 12 ms
From: instTop.consumer_2 - Sequence number: 2, Current Simulation Time: 12 ms
From: instTop.consumer_1 - Sequence number: 3, Current Simulation Time: 15 ms
From: instTop.consumer_2 - Sequence number: 3, Current Simulation Time: 15 ms
From: instTop.consumer_1 - Sequence number: 4, Current Simulation Time: 19 ms
From: instTop.consumer_2 - Sequence number: 4, Current Simulation Time: 19 ms
From: instTop.consumer_1 - Sequence number: 5, Current Simulation Time: 27 ms
From: instTop.consumer_2 - Sequence number: 5, Current Simulation Time: 27 ms
From: instTop.consumer_1 - Sequence number: 6, Current Simulation Time: 29 ms
From: instTop.consumer_2 - Sequence number: 6, Current Simulation Time: 29 ms
From: instTop.consumer_1 - Sequence number: 7, Current Simulation Time: 33 ms
From: instTop.consumer_2 - Sequence number: 7, Current Simulation Time: 33 ms
```

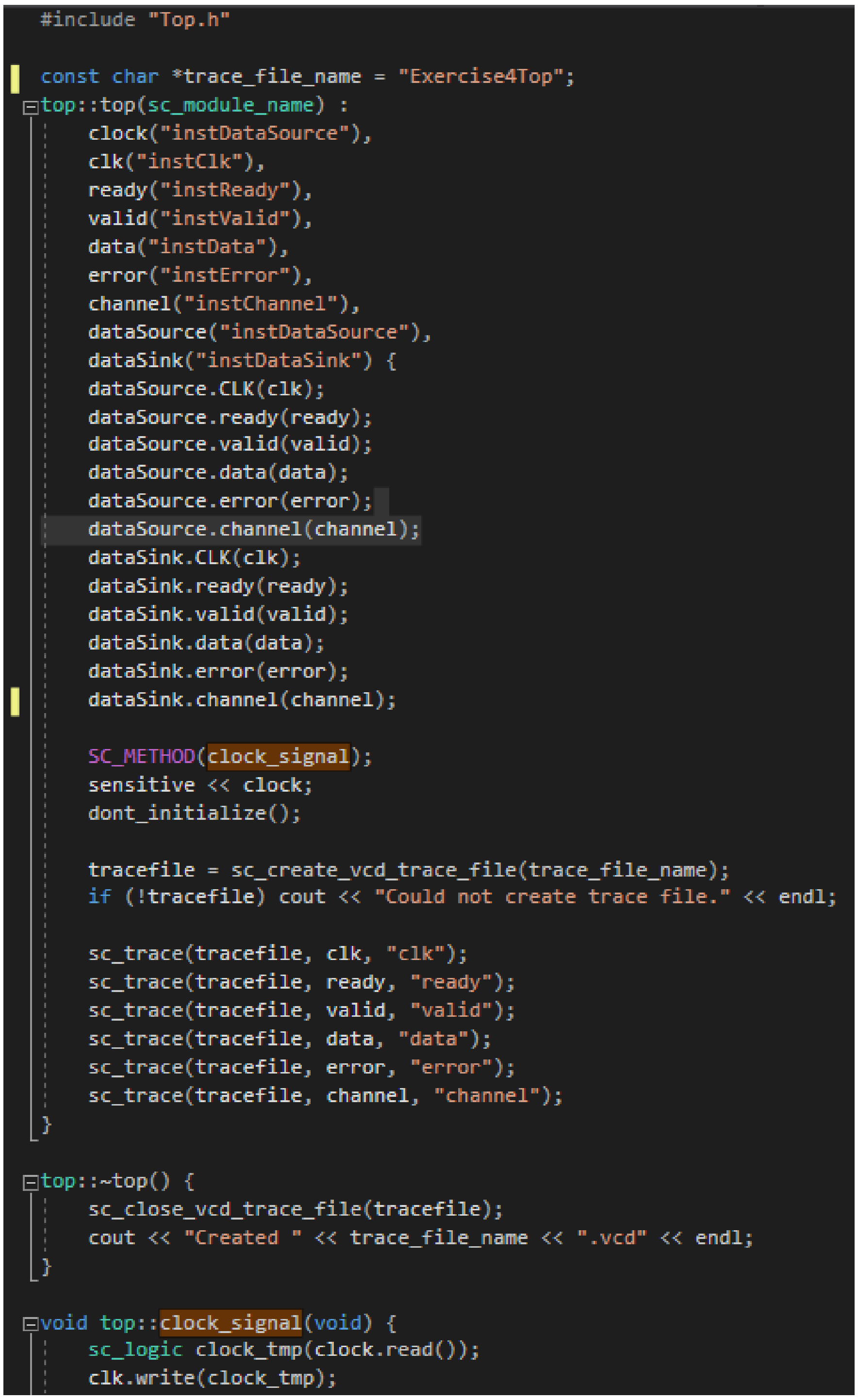
3.4

The program consists of the following files:



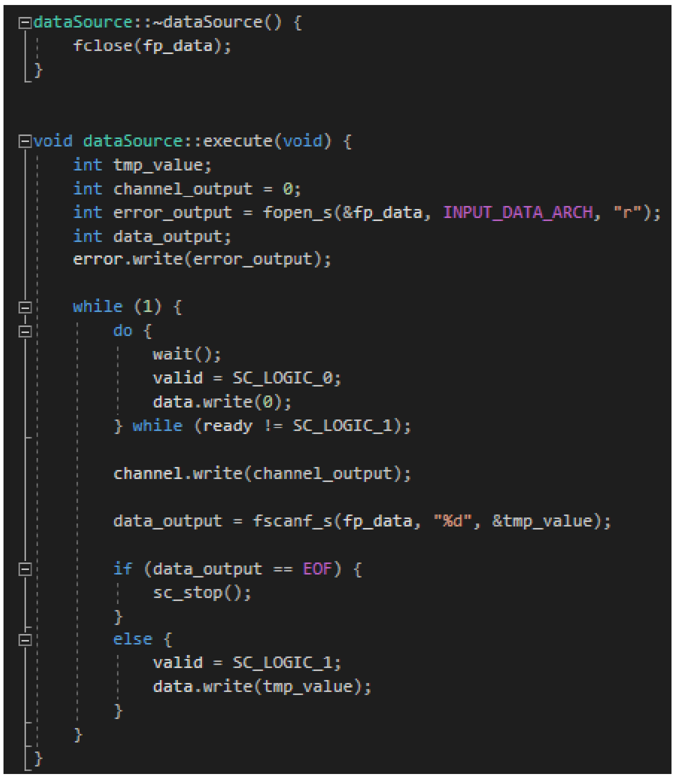
```
◢ Exercise4
   ▷ References
   ▷ External Dependencies
   ◢ Header Files
      ▷ DataSink.h
      ▷ DataSource.h
      ▷ Definitions.h
      ▷ Top.h
   ▷ Resource Files
   ◢ Source Files
      ▷ DataSink.cpp
      ▷ DataSource.cpp
      ▷ Exercise4.cpp
      ▷ Top.cpp
```

As usually, the top module initializes all the signals mm. It also creates the clock and creates the traces on the signals. SC_METHOD(clock_signal) uses static sensitivity to be called every time the clock raised.



```cpp
#include "Top.h"

const char *trace_file_name = "Exercise4Top";
top::top(sc_module_name) :
    clock("instDataSource"),
    clk("instClk"),
    ready("instReady"),
    valid("instValid"),
    data("instData"),
    error("instError"),
    channel("instChannel"),
    dataSource("instDataSource"),
    dataSink("instDataSink") {
    dataSource.CLK(clk);
    dataSource.ready(ready);
    dataSource.valid(valid);
    dataSource.data(data);
    dataSource.error(error);
    dataSource.channel(channel);
    dataSink.CLK(clk);
    dataSink.ready(ready);
    dataSink.valid(valid);
    dataSink.data(data);
    dataSink.error(error);
    dataSink.channel(channel);

    SC_METHOD(clock_signal);
    sensitive << clock;
    dont_initialize();

    tracefile = sc_create_vcd_trace_file(trace_file_name);
    if (!tracefile) cout << "Could not create trace file." << endl;

    sc_trace(tracefile, clk, "clk");
    sc_trace(tracefile, ready, "ready");
    sc_trace(tracefile, valid, "valid");
    sc_trace(tracefile, data, "data");
    sc_trace(tracefile, error, "error");
    sc_trace(tracefile, channel, "channel");
}

top::~top() {
    sc_close_vcd_trace_file(tracefile);
    cout << "Created " << trace_file_name << ".vcd" << endl;
}

void top::clock_signal(void) {
    sc_logic clock_tmp(clock.read());
    clk.write(clock_tmp);
```

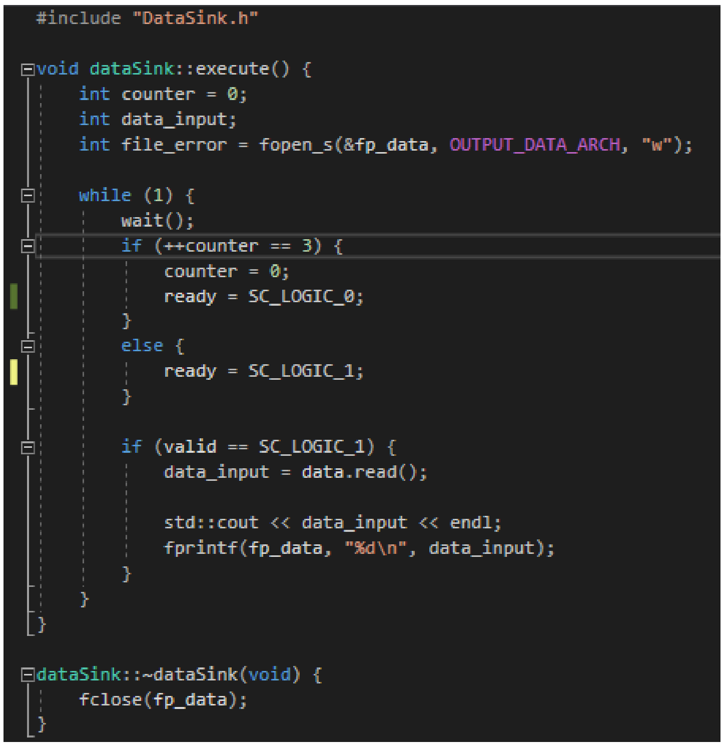
The dataSource looks like this:



```
dataSource::~dataSource() {
    fclose(fp_data);
}


void dataSource::execute(void) {
    int tmp_value;
    int channel_output = 0;
    int error_output = fopen_s(&fp_data, INPUT_DATA_ARCH, "r");
    int data_output;
    error.write(error_output);

    while (1) {
        do {
            wait();
            valid = SC_LOGIC_0;
            data.write(0);
        } while (ready != SC_LOGIC_1);

        channel.write(channel_output);

        data_output = fscanf_s(fp_data, "%d", &tmp_value);

        if (data_output == EOF) {
            sc_stop();
        }
        else {
            valid = SC_LOGIC_1;
            data.write(tmp_value);
        }
    }
}
```

The loop runs until the end of the file fp_data points to has been read. The loop follows the protocol, so it only sends data when the valid signal is high, and 1 with a 1 clock cycle delay from when the ready signal is high. If not, it only sends zero’s to simulate not receiving data.

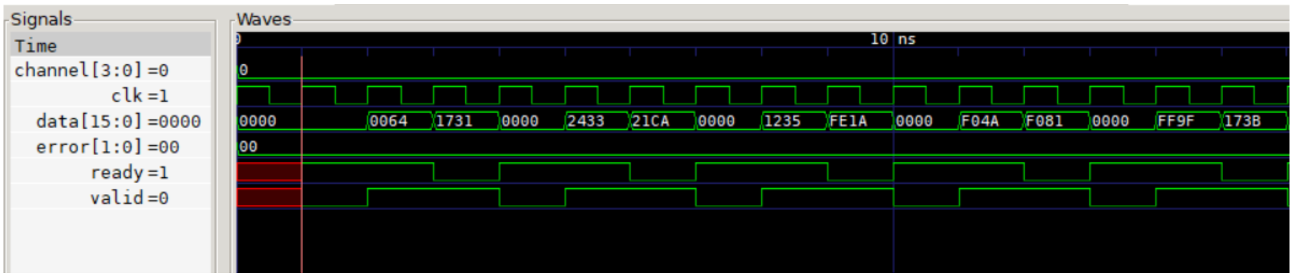
The sink then looks like this:



```
#include "DataSink.h"

void dataSink::execute() {
    int counter = 0;
    int data_input;
    int file_error = fopen_s(&fp_data, OUTPUT_DATA_ARCH, "w");

    while (1) {
        wait();
        if (++counter == 3) {
            counter = 0;
            ready = SC_LOGIC_0;
        }
        else {
            ready = SC_LOGIC_1;
        }

        if (valid == SC_LOGIC_1) {
            data_input = data.read();

            std::cout << data_input << endl;
            fprintf(fp_data, "%d\n", data_input);
        }
    }
}

dataSink::~dataSink(void) {
    fclose(fp_data);
}
```

It waits one clock cycle, and pulls the ready signal high for the source to react on. Then it itself expects the valid signal, after which it will read the data from the queue. On the third cycle, it sets the ready-signal low, which is the cycle in which it will receive zero's from the source.

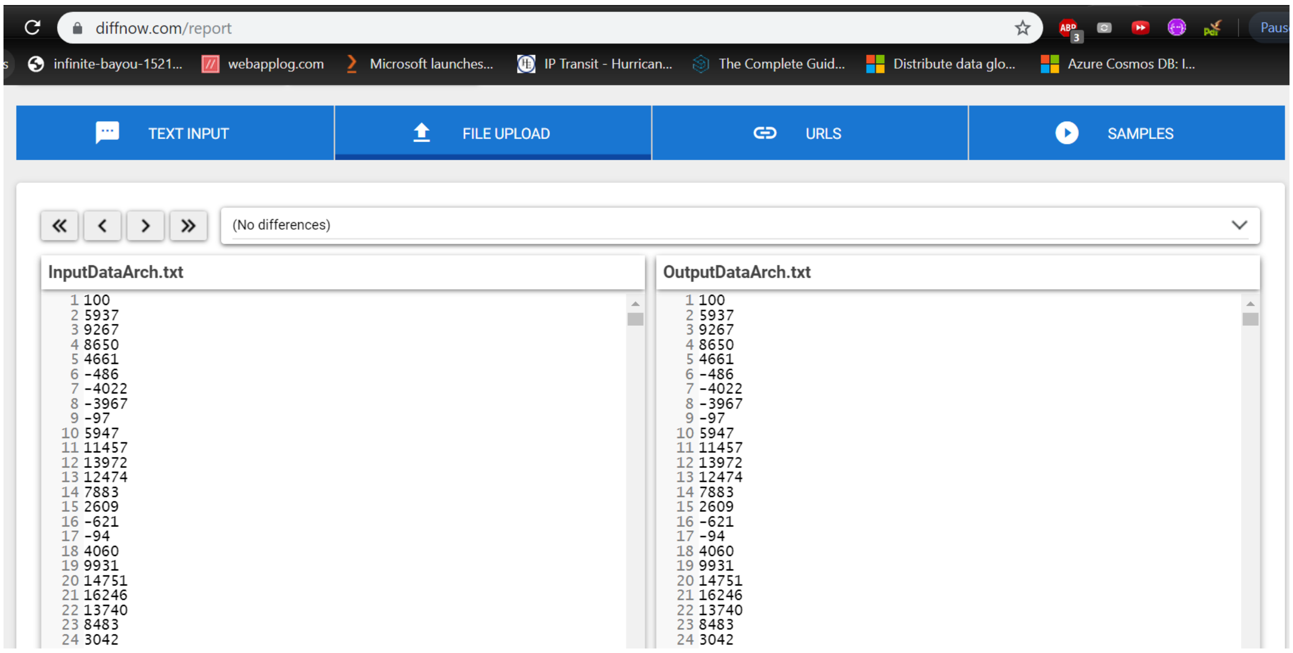
The tracefile in the end, looks like this:



It is seen that then one data byte is send out every clock-cycle when the valid signal is high, delayed by one clock cycle from the ready signal.

The error byte and data signals are made so they can be chosen manually, but does not affect the code in the source or sink for now.

The data is collected from a text file "InputDataArch.txt" in DataSource and saved in another text file "OutputDataArch.txt" in DataSink. The content has been determined to be identical using an online comparison tool "diffnow.com" to compare the two text files and look for differences:



(No differences)

InputDataArch.txt

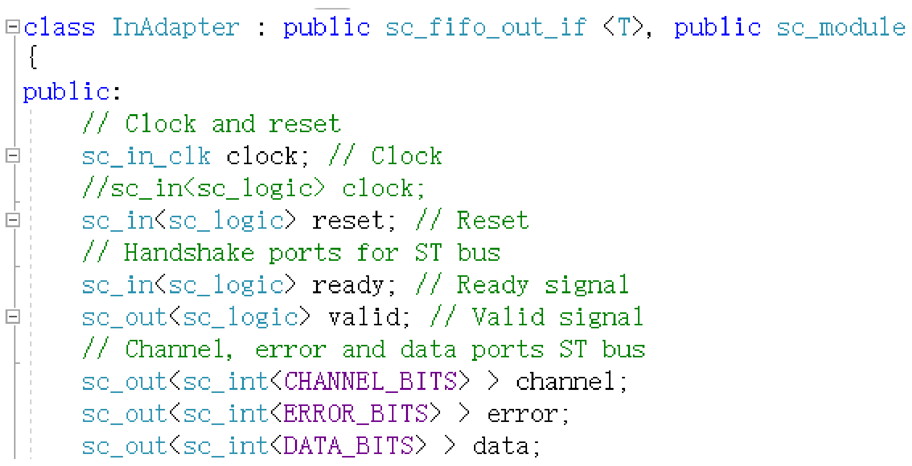
```
100
5937
9267
8650
4661
-486
-4022
-3967
-97
5947
11457
13972
12474
7883
2609
-621
-94
4060
9931
14751
16246
13740
8483
3042
```

OutputDataArch.txt

```
100
5937
9267
8650
4661
-486
-4022
-3967
-97
5947
11457
13972
12474
7883
2609
-621
-94
4060
9931
14751
16246
13740
8483
3042
```

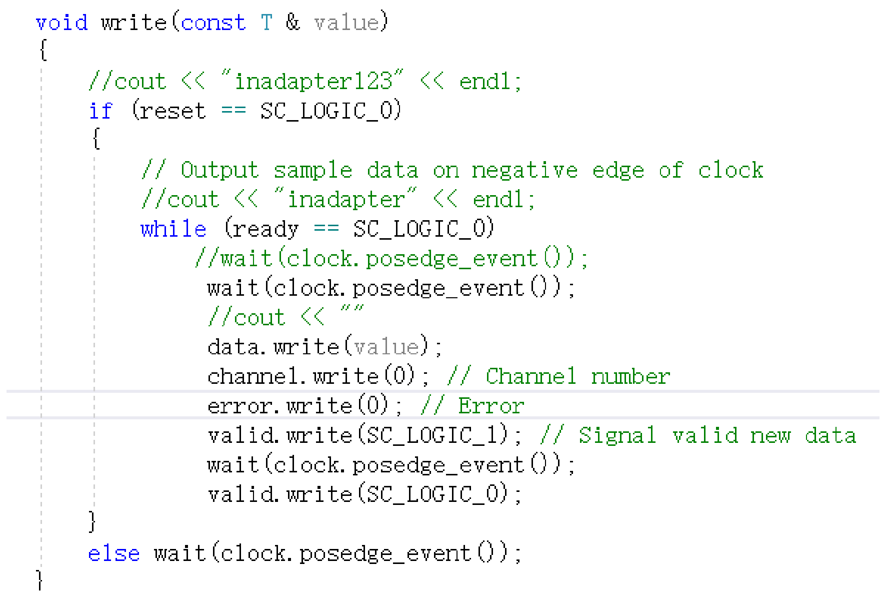
As the files are identical, all data has been transferred from the source to the sink successfully.

3.5



```
class InAdapter : public sc_fifo_out_if <T>, public sc_module
{
public:
    // Clock and reset
    sc_in_clk clock; // Clock
    //sc_in<sc_logic> clock;
    sc_in<sc_logic> reset; // Reset
    // Handshake ports for ST bus
    sc_in<sc_logic> ready; // Ready signal
    sc_out<sc_logic> valid; // Valid signal
    // Channel, error and data ports ST bus
    sc_out<sc_int<CHANNEL_BITS> > channel;
    sc_out<sc_int<ERROR_BITS> > error;
    sc_out<sc_int<DATA_BITS> > data;
```

Definition of the InAdapter. The job of the InAdapter: Receiving data from Master, receiving ready signal from Slave, transfer the data to the Slave, output the signal error, channel to the Salve, and also receive the command reset and clock.

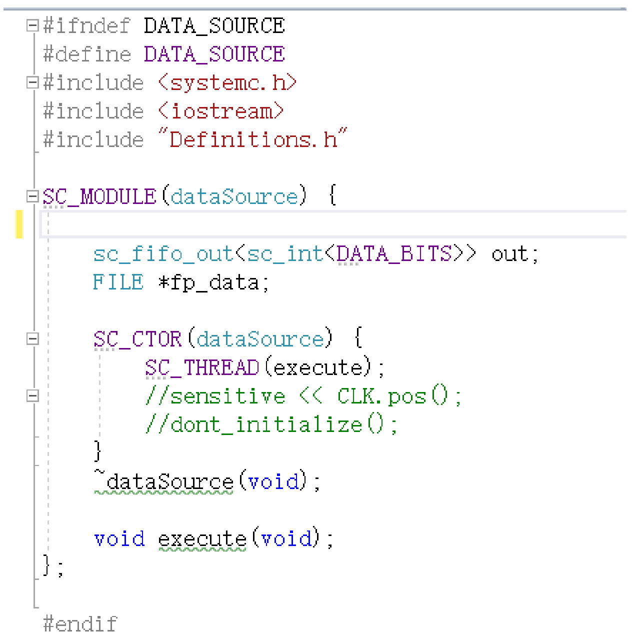


```
void write(const T & value)
{
    //cout << "inadapter123" << endl;
    if (reset == SC_LOGIC_0)
    {
        // Output sample data on negative edge of clock
        //cout << "inadapter" << endl;
        while (ready == SC_LOGIC_0)
            //wait(clock.posedge_event());
             wait(clock.posedge_event());
             //cout << ""
             data.write(value);
             channel.write(0); // Channel number
             error.write(0); // Error
             valid.write(SC_LOGIC_1); // Signal valid new data
             wait(clock.posedge_event());
             valid.write(SC_LOGIC_0);
    }
    else wait(clock.posedge_event());
}
```

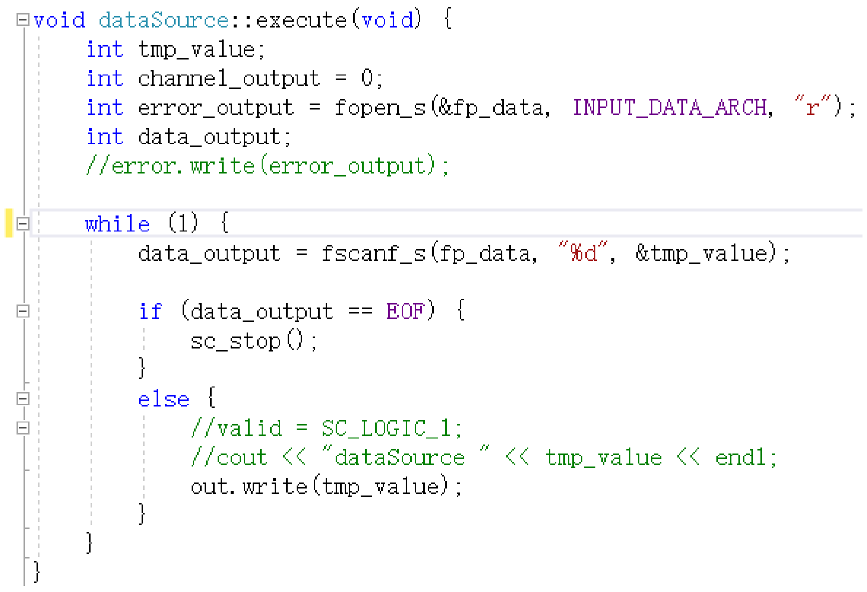
The write function of the InAdapter:

If it receives the command “reset”, then it would be idle and wait for next clock;

If it does not receive the “reset”, it waits for the “slave” to be ready, and when the slave is ready, sends the “data”, “channel” and “valid” to the Slave.

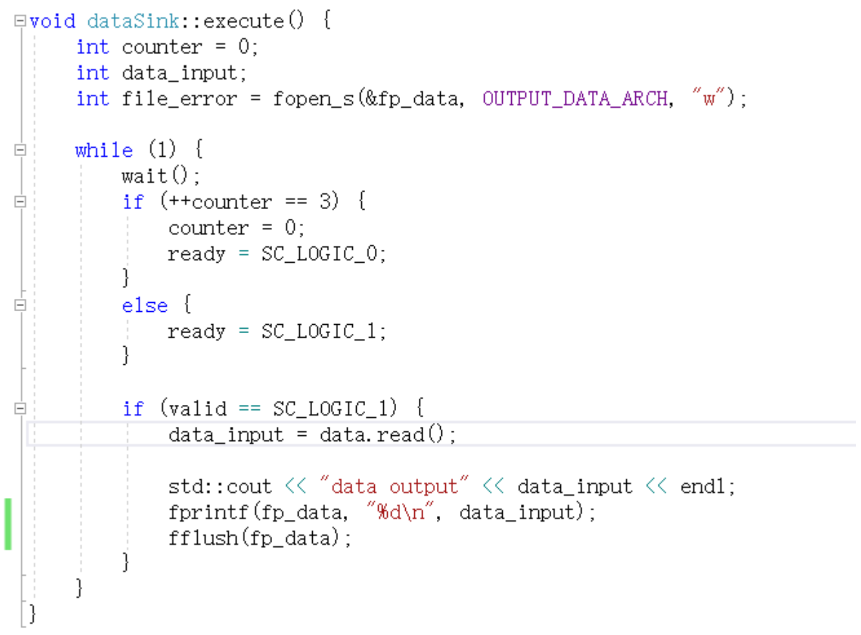


```
#ifndef DATA_SOURCE
#define DATA_SOURCE
#include <systemc.h>
#include <iostream>
#include "Definitions.h"

SC_MODULE(dataSource) {

    sc_fifo_out<sc_int<DATA_BITS>> out;
    FILE *fp_data;

    SC_CTOR(dataSource) {
        SC_THREAD(execute);
        //sensitive << CLK.pos();
        //dont_initialize();
    }
    ~dataSource(void);

    void execute(void);
};

#endif
```



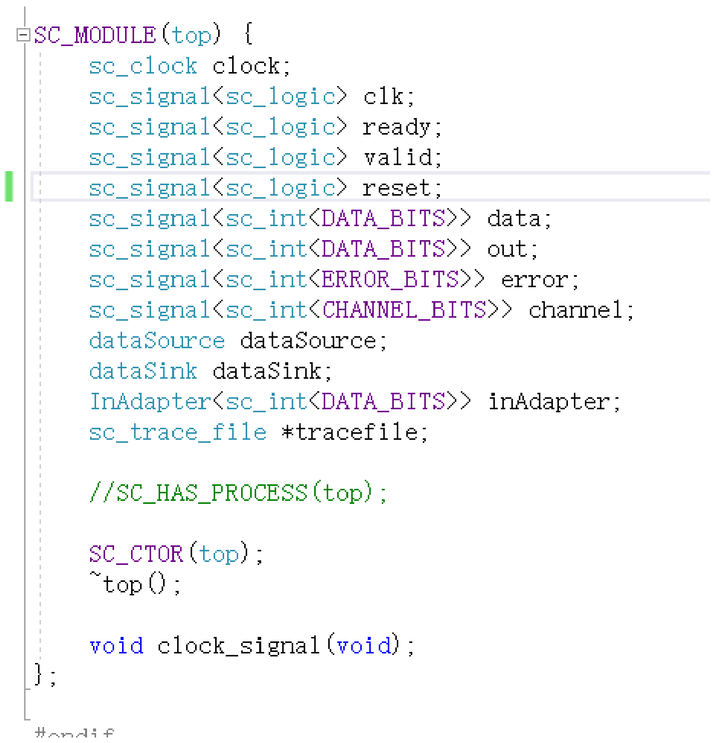
```
void dataSource::execute(void) {
    int tmp_value;
    int channel_output = 0;
    int error_output = fopen_s(&fp_data, INPUT_DATA_ARCH, "r");
    int data_output;
    //error.write(error_output);

    while (1) {
        data_output = fscanf_s(fp_data, "%d", &tmp_value);

        if (data_output == EOF) {
            sc_stop();
        }
        else {
            //valid = SC_LOGIC_1;
            //cout << "dataSource " << tmp_value << endl;
            out.write(tmp_value);
        }
    }
}
```

The definition and implement of “master”: it only does the job of reading data from input file and sending data to the InAdapter.

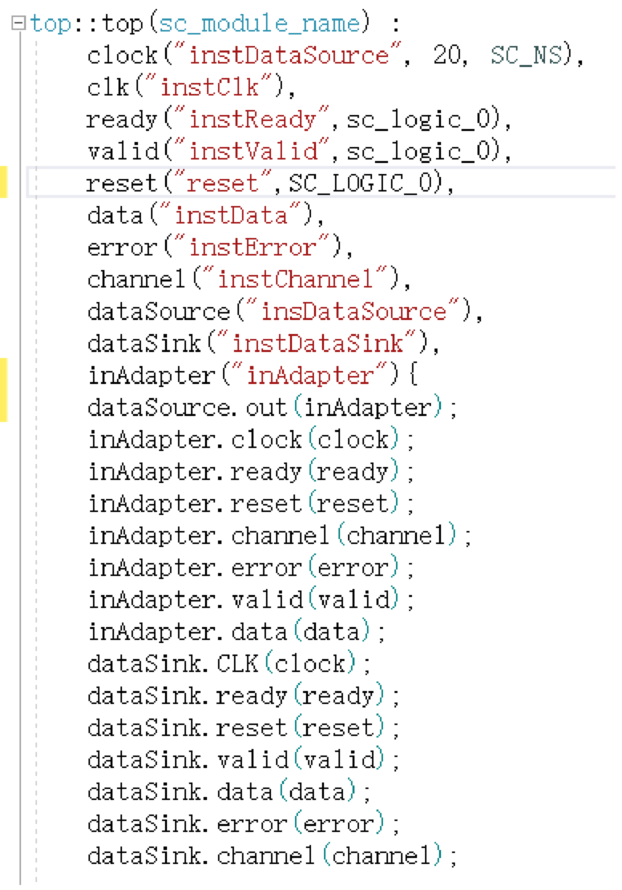


```
void dataSink::execute() {
    int counter = 0;
    int data_input;
    int file_error = fopen_s(&fp_data, OUTPUT_DATA_ARCH, "w");

    while (1) {
        wait();
        if (++counter == 3) {
            counter = 0;
            ready = SC_LOGIC_0;
        }
        else {
            ready = SC_LOGIC_1;
        }

        if (valid == SC_LOGIC_1) {
            data_input = data.read();

            std::cout << "data output" << data_input << endl;
            fprintf(fp_data, "%d\n", data_input);
            fflush(fp_data);
        }
    }
}
```

The “Slave”, which is “dataSink” is the same as the on in ex 3.4, but with a new line fflush(fp_data); just clean up the space cache for the new data waiting to be written to the output file.



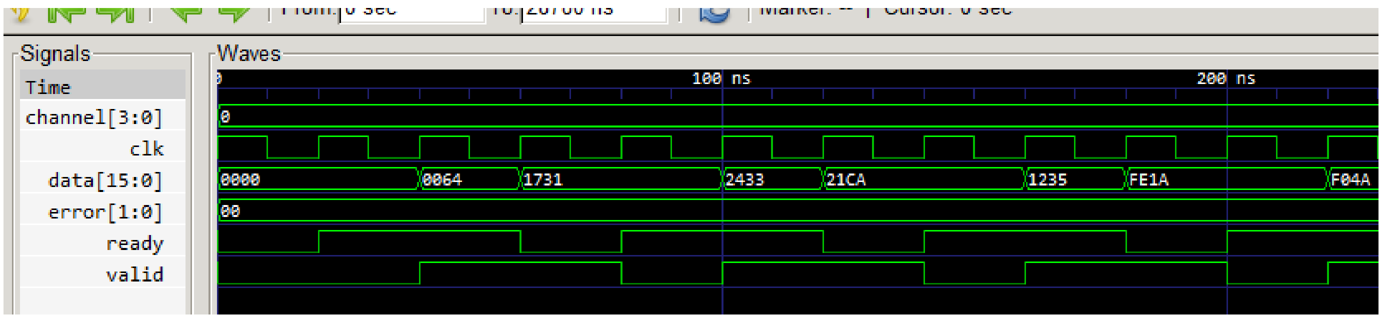
```
SC_MODULE(top) {
    sc_clock clock;
    sc_signal<sc_logic> clk;
    sc_signal<sc_logic> ready;
    sc_signal<sc_logic> valid;
    sc_signal<sc_logic> reset;
    sc_signal<sc_int<DATA_BITS>> data;
    sc_signal<sc_int<DATA_BITS>> out;
    sc_signal<sc_int<ERROR_BITS>> error;
    sc_signal<sc_int<CHANNEL_BITS>> channel;
    dataSource dataSource;
    dataSink dataSink;
    InAdapter<sc_int<DATA_BITS>> inAdapter;
    sc_trace_file *tracefile;

    //SC_HAS_PROCESS(top);

    SC_CTOR(top);
    ~top();

    void clock_signal(void);
};

#endif
```



```
top::top(sc_module_name) :
    clock("instDataSource", 20, SC_NS),
    clk("instClk"),
    ready("instReady",sc_logic_0),
    valid("instValid",sc_logic_0),
    reset("reset",SC_LOGIC_0),
    data("instData"),
    error("instError"),
    channel("instChannel"),
    dataSource("insDataSource"),
    dataSink("instDataSink"),
    inAdapter("inAdapter"){
    dataSource.out(inAdapter);
    inAdapter.clock(clock);
    inAdapter.ready(ready);
    inAdapter.reset(reset);
    inAdapter.channel(channel);
    inAdapter.error(error);
    inAdapter.valid(valid);
    inAdapter.data(data);
    dataSink.CLK(clock);
    dataSink.ready(ready);
    dataSink.reset(reset);
    dataSink.valid(valid);
    dataSink.data(data);
    dataSink.error(error);
    dataSink.channel(channel);
```

The definition and implement of the Top: Since InAdapter inherits from the class sc_fifo_out_if (as stated in the definition) and the "master" has a sc_fifo_out, they can be connected to each other directly (dataSource.out(inAdapter));

And then, InAdapter does the same job as the "master" did in the ex3.4: connecting with the "slave".



Simulation result:

As the code stated in InAdapter, when the InAdapter receives the "ready" signal, it would sent signal "valid" with value 1 after another clock. As stated in the simulation, the "valid" is one clock lagged behind the "ready".

The InAdapter has to wait for the "Slave" to be ready to transfer data to it, so in the simulation picture, some data`s transfer period is long while the others` is short, because it have to wait the "slave" to be ready.

The “channel” and the “error”, as stated in the code in InAdapter, always has the same value.