

SystemC Tutorial

1. Introduction to SystemC

SystemC is a system based on C++ that is widely used to simulate architectures. It is a library of C++ classes, global functions, data types and a simulation kernel that can be used for creating simulators for hardware. By using C/C++ development tools and the SystemC library executable specifications of a model can be created in order to simulate, validate, and optimize the system being designed. An executable specification is essentially a C++ program that exhibits the same behavior as the system when executed.

The SystemC library can be downloaded from:

www.systemc.org

http://staff.science.uva.nl/~bousias/GHI_2006/systemc

Documentation and User guides can also be downloaded from the above sites.

2. Using SystemC

A SystemC system consists of a set of one or more modules. Modules provide the ability to describe structure. Modules typically contain processes, ports, internal data, channels and possibly instances of other modules. All processes are conceptually concurrent and can be used to model functionality of the module. Ports are objects through which the module communicates with other modules. In C++ terms modules, ports and channels are classes from which objects are created. Processes are member functions of a module that when registered as a process to the simulation kernel, are executed every time an event is triggered.

Events are the basic synchronization objects. They are used to synchronize between processes. Processes are triggered or caused to run based on their sensitivity to events. That means that every time a module member function is registered as a process, the events on which it is “sensitive” are also defined.

Access to all SystemC classes and functions is provided in a single header file named “systemc.h”. This file includes other files, but the end user only needs to use this. In order to use the SystemC classes this file must be included and the application must be linked with the SystemC library.

SystemC comes in source code, thus in order to use it must be compiled.

Task 1

Download the SystemC library from one of the above addresses and then extract the file.

For Linux platforms do the following:

- 1) Execute the “configure” script.**
- 2) Execute the command “gmake”**
- 3) Execute the command “gmake install”**

After doing the above a file named “libsystemc.a” will be created in the directory <SystemC Installation Directory>/lib-linux. This file must be linked with the application.

For Windows platforms with Visual C++ 6.0 do the following:

- 1. Open the workspace file in the folder**
<SystemC Installation folder> \msvc60\systemc.
- 2. Select Build -> Build systemc.lib**

If using Visual C++ 7.1 (.NET), then:

- 1. Open the solution file in the folder**
<SystemC Installation folder> \msvc71\systemc.
- 2. Select Build -> Build Solution**

A file named “systemc.lib” will be created which is the file that must be linked with the application.

It is important that the application that uses the library is compiled with Run Time Type Information (RTTI).

If using Visual C++ 7.1 (.NET) the “/vmg” switch must also be set for the application that uses the library. This can be done as follows:

From the C/C++ tab, select the Command Line Properties and add /vmg to the “Additional Options:” box.

Further Information about building, installing and using the library can be found in the file “INSTALL”.

3. Execution Semantics

SystemC is an event based simulator. Events occur at a given simulation time. Time starts at 0 and moves forward only. Time increments are based on the default time unit and the time resolution.

Every C/C++ program has a **main()** function. When the SystemC library is used the **main()** function is not used. Instead the function **sc_main()** is the entry point of the application. This is because the **main()** function is defined into the library so that when the program starts initialization of the simulation kernel and the structures this requires to be performed, before execution of the application code. The **main()** defined into the library calls the **sc_main()** function after it has finished with the initialization process.

The **sc_main()** function is defined as follows:

int sc_main(int argc, char* argv[]) which is the same as a **main()** function in common C++ programs. The values of the arguments are forwarded from the **main()** defined in the library.

In the **sc_main()** function the structural elements of the system are created and connected throughout the system hierarchy. This is facilitated by the C++ class object construction behavior. When a module comes into existence all sub-modules this contains are also created.. After all modules have been created and connections have been established the simulation can start by calling the function **sc_start()** where the simulation kernel takes control and executes the processes of each module depending on the events occurring at every simulated cycle. In the first cycle all the processes are executed at least once unless otherwise stated with a call to the function **dont_initialize()** when the process is registered.

4. Example

The following code creates a module which simulates a Random Access Memory. How the code works is explained later.

```
#define MEM_SIZE 512

SC_MODULE(Memory) {
public:
    enum Function {
        FUNC_NONE,
        FUNC_READ,
        FUNC_WRITE
    };

    enum RETSignal {
        RSIG_NONE,
        RSIG_READ_FIN,
        RSIG_WRITE_FIN,
        RSIG_ERROR
    };

    sc_in<bool>      Port_CLK;
    sc_in<Function> Port_Func;

    sc_in<int>      Port_Addr;
    sc_inout<int>   Port_Data;
    sc_out<RETSignal> Port_DoneSig;

    SC_CTOR(Memory) {
        SC_METHOD(execute);
        sensitive_neg(Port_CLK);

        m_clkCnt = 0;
        m_curAddr = 0;
        m_curData = 0;
        m_curFunc = Memory::FUNC_NONE;

        m_data = new int[MEM_SIZE];

        m_writesCnt = 0;
        m_readsCnt = 0;
        m_errorsCnt = 0;
        m_errorCode = 0;
    }

    ~Memory() {
        delete [] m_data;
    }

private:
    int      m_clkCnt;
    int      m_curAddr;
    int      m_curData;
    Function m_curFunc;
};
```

```

int* m_data;
int m_errorCode;

int m_writesCnt;
int m_readsCnt;
int m_errorsCnt;

RETSigal read() {
    if(m_errorCode == 0) {
        Port_Data.write(m_data[m_curAddr]);

        m_readsCnt++;
        return RSIG_READ_FIN;
    } else {
        m_errorsCnt++;
        return RSIG_ERROR;
    }
}

RETSigal write() {
    if(m_errorCode == 0) {
        m_data[m_curAddr] = m_curData;

        m_writesCnt++;
        return RSIG_WRITE_FIN;
    } else {
        m_errorsCnt++;
        return RSIG_ERROR;
    }
}

void execute() {
    if(m_curFunc != Memory::FUNC_NONE) {
        m_clkCnt++;

        if(m_clkCnt == 100) {
            RETSigal retSig = Memory::RSIG_ERROR;

            switch(m_curFunc) {
                case Memory::FUNC_READ : { retSig = read(); break; }
                case Memory::FUNC_WRITE : { retSig = write(); break; }
            }

            Port_DoneSig.write(retSig);
            m_clkCnt = 0;
            m_curFunc = Memory::FUNC_NONE;
        }

        return;
    }

    if(Port_Func.read() == Memory::FUNC_NONE) {
        return;
    }

    m_curFunc = Port_Func.read();
}

```

```
m_curAddr = Port_Addr.read();
m_curData = Port_Data.read();

    Port_DoneSig.write(Memory::RSIG_NONE);
}
};
```

As can be seen from the above code, a definition of a module in SystemC is the same as declaration of a class in C++ and this is because it is exactly that. The `SC_MODULE` statement is just a macro that provides a simple form of module definition.

The macro is defined as follows :

```
#define SC_MODULE(user_module_name) \
    struct user_module_name : public sc_module
```

As it can be seen from the definition of the macro every module is a sub-class of the `sc_module` class.

SystemC uses many macros like `SC_MODULE` in order to simplify definitions and to be similar to SystemC terminology.

The first elements of the module's declaration are two enumeration data types that the module uses. The four lines that follow are the declarations of the **ports** the module has in order to communicate with other modules. In order to understand the concept of ports, first we must look briefly at interfaces and channels.

In SystemC interfaces define a set of member functions a channel that implements the interface must have. They only provide signatures of the functions and not implementation. In C++ terms they are classes with all their functions being pure virtual functions.

Channels define how the functions of an interface are implemented. They are classes directly derived from the interface(s) they implement and provide implementations for the functions of the interface(s).

SystemC provides a number of ready defined interfaces and channels which implement them. If those do not provide the required functionality others can be defined. The most common used interface is the **sc_signal_inout_if** and the channel class that implements it is the **sc_signal**.

Finally ports are objects through which a module can be connected into one or more channels. Port objects are directly or indirectly derived from the template class **sc_port**. This class is a template so that it can be customized according to the interface the port can connect. That means that when a port object is defined using this class the interface (and consequently the channel) on which it can connect must also be defined. The following is an example port declaration that can connect to the **sc_signal_inout_if** interface mentioned earlier.

```
sc_port<sc_signal_inout<int>,1> port_name
```

In the above statement we can also see that the **sc_signal_inout_if** is also a template class. SystemC makes extreme use of template classes, because that mechanism gives the flexibility the class needs to be customized for a specific data-type.

In our example though, we have not used such declarations to define the ports of the module. This is because we have used what is called **specialized ports**. Specialized ports are classes derived from the **sc_port** class, which are customized with a particular (set of) interface(s). These classes also provide additional support for use with a channel or for easier use. The ports used in the example are: **sc_in**, **sc_out** and **sc_inout**, which are ports specific for the **sc_signal_inout_if** mentioned earlier. Again here we see that those classes are also template classes. This gives the ability to define a port that can handle data of any C++ or SystemC data-type.

4.1 The module's constructor

The code which starts with the statement **SC_CTOR** is the code of the constructor of the module. The **SC_CTOR** statement is also a macro used for the constructor of the class.

At the beginning of this document it had been mentioned that a module also has processes, which are member functions registered as processes to the simulation kernel and executed by it when an event is triggered. The first line of the constructor's code does exactly that. It registers to the simulation kernel that the Memory module has a process which when executed the code of the **execute** member function must run. The **SC_METHOD** is also a macro that makes the code more readable.

Processes have a list of events on which they are sensitive. The second line of the constructor's code **sensitive(Port_CLK)** creates that list for the process registered by the previous **SC_METHOD** statement. It specifies that this process will execute when an event is triggered by the **Port_CLK** port. Ports trigger events owned by the channel they are connected to, when the value of the port itself changes or the value of another port also connected to the same channel changes. In simple terms when the **Port_CLK** port changes the **execute** member function will run. The rest of the constructor code allocates storage for the memory and initializes the counters with which the module keeps track of the requests sent to it.

The next lines of code is the code of the member functions of the module. In this code it can be seen that data values are written and read from the ports. The functions that are used for this operation actually belong to the interface/channel the port can connect to. By using operator overloading the **sc_port** class gives access to those functions in order for the code to be more meaningful.

Task 2.a

Enter the code of the example into a new C++ source code file. Do not forget to include the `systemc.h` file at the beginning of the file.

Create an `sc_main()` function in the same file with the following code:

```
int sc_main(int argc, char* argv[])
{
    Memory mem("main_memory");
    sc_start();

    return 0;
}
```

Compile the and run the program.

As you can see the program when run does nothing. This is because there are no events triggered. When there are no more events the simulation kernel stops the simulation. That is the `sc_start()` function returns and the program ends (Actually in this example errors about port binding will appear because the module's ports are not connected). The absence of events is because even though we create an instance of the Memory module we have not connected anything to its ports and no change to the ports' values is taking place to trigger any events. In order to test our module we need a second one that will be connected to it and make changes to its ports. Such a module can be created with the following code.

```
SC_MODULE(CPU) {
public:
    sc_in<bool> Port_CLK;
    sc_in<Memory::RETSignal> Port_MemDone;
    sc_out<Memory::Function> Port_MemFunc;
    sc_out<int> Port_MemAddr;
    sc_inout<int> Port_MemData;

    SC_CTOR(CPU) {
        SC_METHOD(execCycle);
        sensitive_pos(Port_CLK);
        dont_initialize();

        SC_METHOD(memDone);
        sensitive(Port_MemDone);
        dont_initialize();

        m_waitMem = false;
    }
private:
    bool m_waitMem;

    Memory::Function getrndfunc() {
        int rndnum=(rand() % 10);
```

```

        if(rndnum < 5)
            return Memory::FUNC_READ;
        else
            return Memory::FUNC_WRITE;
    }

    int getRndAddress() {
        return (rand() % MEM_SIZE);
    }

    int getRndData() {
        return rand();
    }

    void execCycle() {
        if(m_waitMem) {
            return;
        }

        int addr          = getRndAddress();
        Memory::Function f = getrndfunc();

        Port_MemFunc.write(f);
        Port_MemAddr.write(addr);

        if(f == Memory::FUNC_WRITE)
            Port_MemData.write(getRndData());

        m_waitMem = true;
    }

    void memDone() {
        if(Port_MemDone.read() == Memory::RSIG_NONE) {
            return;
        }

        m_waitMem = false;

        Port_MemFunc.write(Memory::FUNC_NONE);
    }
};

```

This module simulates a CPU that has the appropriate ports to connect to our Memory module and make read/write requests for random addresses. First thing to note in the above code is that the CPU module uses a **sc_in** port for every **sc_out** port of the Memory module, and a **sc_out** for every **sc_in**. The constructor of the module is similar to the one of the Memory module. There are though two major differences. The first is that the CPU module registers two processes with the kernel. This shows that in SystemC a module can have more than one processes. The second difference is the “`dont_initialize();`” statements after the declaration of each process. When the simulation starts the first thing the kernel does is to go through the list of all the processes from every module and execute the function associated with that process. This is done for all the processes despite whether any events on which the process is sensitive have been triggered (in fact at this point there will always be no events triggered because the

simulation has just started). By executing all the processes that way events are starting to be triggered (because the processes write to the ports) thus the simulation can continue. The exception to this rule is when the “dont_initialize()” function is called after a process has been registered with the kernel (with the SC_METHOD macro). If that is the case the kernel will not execute that process during the initialization stage.

Task 2.b

Add the above code of the CPU module to your previous program and then modify the sc_main() function to have the following code.

```
int sc_main(int argc, char* argv[])
{
    cout << "\n\nCreating Modules.....";

    /* Instantiate Modules */
    Memory mem("main_memory");
    CPU     cpu("cpu");

    /* Signals */
    sc_signal<Memory::Function> sigMemFunc;
    sc_signal<Memory::RETSignal> sigMemDone;
    sc_signal<int>               sigMemAddr;
    sc_signal<int>               sigMemData;

    /* The clock that will drive the CPU and Memory*/
    sc_clock clk;

    cout << "DONE\nConnecting Modules' Ports...";

    /* Connecting module ports with signals */
    mem.Port_Func(sigMemFunc);
    mem.Port_Addr(sigMemAddr);
    mem.Port_Data(sigMemData);
    mem.Port_DoneSig(sigMemDone);

    cpu.Port_MemFunc(sigMemFunc);
    cpu.Port_MemAddr(sigMemAddr);
    cpu.Port_MemData(sigMemData);
    cpu.Port_MemDone(sigMemDone);

    mem.Port_CLK(clk);
    cpu.Port_CLK(clk);

    cout << "DONE\n" << endl
         << "\n\nRunning (press CTRL+C to exit)... ";

    /* Start Simulation */
    sc_start();

    return 0;
}
```

Compile and run the program.

What the code in the **sc_main** function does is to create instances of the modules, four channels of type **sc_signal** and with those to connect the two modules through their ports.

It also creates an instance of the **sc_clock** class and connects that to the Port_CLK ports of the CPU and Memory modules. The **sc_clock** class is a predefined primitive channel derived from the class **sc_signal** and intended to model the behavior of a digital clock signal. In effect an instance of **sc_clock** triggers an event in regular intervals (there are constructor overloads that can be used to set certain properties to the clock) so when a module's port is connected to the clock the process sensitive to the port is executed.

Finally the **sc_start()** statement tells the simulation kernel to start the simulation.

Until now we have seen how a basic simulator can be built using SystemC and the basic aspects of the framework. A more complete implementation of the example in Task 2 can be downloaded from:

http://staff.science.uva.nl/~bousias/GHI_2006/tutorial/task_2_src

More detailed information on the topics described here and many others can be found in the document: *SystemC 2.0.1 Language Reference Manual*, which can be downloaded from:

www.systemc.org.