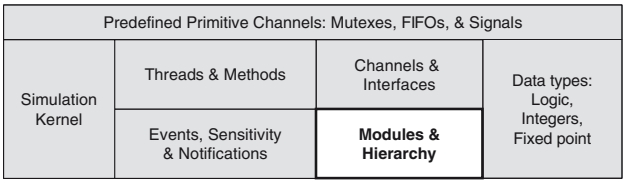
Chapter 4
Modules



This chapter lays the foundation for SystemC models. Here, we explore how to write a minimal SystemC program in preparation for an exploration of time and concurrency in later chapters. With respect to hierarchy, this chapter only touches the very top level. A later chapter on structure will discuss hierarchy in more detail.

4.1 A Starting Point: sc_main

All programs need a starting point. In C/C++, the starting point is called **`main`**`()`. In SystemC, the starting point is called **`sc_main`**`()`, and this is where you will start your code. SystemC as in Verilog and VHDL, might superficially appear to start every process simultaneously. In reality, all of these simulation languages have initialization requirements that are handled here.

The top level of a C/C++ program is a function named **`main`**`()`. Its declaration is generally:

```
int main(int argc, char* argv[]) {
    BODY_OF_PROGRAM
  return EXIT_CODE; // Zero indicates success
}
```

Fig. 4.1 Syntax of C++ **`main`**`()`

In Fig. 4.1, `argc` represents the number of command-line arguments including the program name itself. `argv[]` is an array of C-style character strings representing the command line that invoked the program. Thus, `argv[0]` is the program name itself.

SystemC usurps this procedure and provides **`sc_main()`** as replacement. The SystemC library provides its own definition of **`main()`**, which in turn calls **`sc_main()`** and passes along the command-line arguments. The syntax for **`sc_main()`** is shown in Fig. 4.2.

D.C. Black et al., *SystemC: From the Ground Up*, Second Edition,
DOI 10.1007/978-0-387-69958-5_4, © Springer Science+Business Media, LLC 2010

```
int sc_main(int argc, char* argv[]) {
  ELABORATION
  sc_start(); // <-- Simulation begins & ends
              //     in this function!
  [POST-PROCESSING]
  return EXIT_CODE; // Zero indicates success
}
```

Fig. 4.2 Syntax of `sc_main()`

By convention, SystemC programmers simply name the file containing **`sc_main()`**, as main.cpp to indicate to the C/C++ programmer that this is the place where everything begins[1]. The actual **`main()`** routine is located in the SystemC library itself and is not exposed to the user.

SystemC provides access to argc and argv throughout the model hierarchy by providing the functions as shown in Fig. 4.3.

```
int         sc_argc(); //access to argc
const char* sc_argv(); //access to argv
```

Fig. 4.3 Syntax of `sc_argc()` and `sc_argv()`

Within **`sc_main()`**, code executes in three distinct major stages. We will now examine these stages, which are elaboration, simulation, and post-processing.

During elaboration, connectivity for the model is established. This includes hierarchal modules, leaf modules, channels, simulation processes, and data structures. Elaboration invokes code to register simulation processes and performs the connections between design modules.

At the end of elaboration, **`sc_start()`** invokes the simulation stage. During simulation, code representing the behavior of the model executes. Within the simulation stage, the scheduler is responsible for process execution. A following chapter on concurrency will explore this stage in greater detail.

Finally, after returning from **`sc_start()`**, the post-processing stage begins. Post-processing is mostly optional. During post-processing, code may read data created during simulation and format reports or otherwise handle the results of simulation.

Post-processing finishes with the return of an exit status from **`sc_main()`**. A nonzero return status indicates failure that can be a computed result of post-processing. A zero return should indicate success (i.e., confirmation that the model correctly passed all tests). Many developers neglect this aspect and simply return zero by default. We recommend that you explicitly confirm that the model passed all tests.

[1] This naming convention is not without some controversy in some programming circles; however, most groups have accepted it and deal with the name mismatch.

Callbacks may be used to intercept some points in the preceding stages. For instance, **end_of_elaboration()**, **start_of_simulation()**, and **end_of_simulation()**. These are discussed later in the book.

We now turn our attention to the components used to create a system model.

4.2 The Basic Unit of Design: SC_MODULE

Complex systems consist of many independently functioning components. These components may represent hardware, software, or any physical entity. Components may be large or small, and often contain hierarchies of smaller components. The smallest components represent behaviors and state. In SystemC, we use a concept known as the **SC_MODULE** to represent components.

DEFINITION: A SystemC module is the smallest container of functionality with state, behavior, and structure for hierarchical connectivity.

A SystemC module is simply a C++ class definition. For convenience, the macro **SC_MODULE** is used to declare the class in Fig. 4.4.

```
#include <systemc>
SC_MODULE(module_name) {
  MODULE_BODY
};
```

Fig. 4.4 Syntax of **SC_MODULE**

where **SC_MODULE** is a simple cpp[2] macro as shown in Fig. 4.5. Don't forget the trailing semicolon, which is a fairly common error.

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

Fig. 4.5 **SC_MODULE** macro definition

We prefer the following method for coding an **SC_MODULE** as illustrated in Fig. 4.6.

```
#include <systemc>
class module_name : public sc_module {
public:
  MODULE_BODY
};
```

Fig. 4.6 Syntax without the **SC_MODULE** macro

[2] cpp is the C/C++ pre-processor that handles # directives such as # **define**.

Within this derived module class, a variety of elements make up the *MODULE BODY:*

- Ports
- Member channel instances
- Member data instances
- Member module instances (sub-designs)
- Constructor
- Destructor
- Simulation process member functions (processes)
- Other methods (i.e., member functions)

Of these, only the constructor is required. However, to contain any useful behavior in your design, you must include either a process or a sub-design. We will first look at the constructor, followed by a simple process. This sequence lets us finish with a basic example of a minimal design and a few alternatives.

4.3 The SC_MODULE Class Constructor: SC_CTOR

Since **SC_MODULE** is a C++ class, it requires a constructor. The **SC_MODULE** constructor performs several tasks specific to SystemC. These tasks include:

- Initializing/allocating sub-designs
  - Chapter 10 Structure
- Connecting sub-designs
  - Chapter 10 Structure
  - Chapter 11 Connectivity
- Registering processes with the SystemC kernel
  - Chapter 6 Concurrency
- Providing static sensitivity
  - Chapter 6 Concurrency
- Miscellaneous user-defined setup

To simplify coding, SystemC provides the macro, **SC_CTOR**(). The syntax using this macro follows in Fig. 4.7.

In a simple design, you may only require process registration and setup, whereas in complicated designs, you may need to include multiple designs as well as multiple processes. Let us now examine processes, and see how they fit into the design concept.

```
SC_MODULE(module_name) {
  SC_CTOR(module_name)
  : Initialization // C++ initialization list
  {
    Subdesign_Allocation
    Subdesign_Connectivity
    Process_Registration
    Miscellaneous_Setup
  }
};
```

Fig. 4.7 Syntax of **SC_CTOR**

4.4 The Basic Unit of Execution: Simulation Process

The SystemC simulation process is the basic unit of execution. All simulation processes are registered with the SystemC simulation kernel and are called by the kernel, and *only* from the SystemC simulation kernel. We discuss the SystemC simulation kernel in excruciating detail in a following chapter on concurrency. From the time the simulator begins until simulation ends, all executing code is initiated from one or more processes. Simulation processes appear to execute concurrently.

DEFINITION: A SystemC simulation process is a method (member function) of an **SC_MODULE** that is invoked by the scheduler in the SystemC simulation kernel.

The prototype of a basic simulation process for SystemC is:

```
void PROCESS_NAME(void);
```

Fig. 4.8 Syntax of SystemC process

There are several kinds of simulation processes, and we will discuss all of them eventually. For the purpose of simplification, we will look only at the most basic simulation process type in this chapter.

The most straightforward type of process to understand is the SystemC thread, **SC_THREAD**. Conceptually, a SystemC thread is similar to a single software thread.

A SystemC simulation is a simple C/C++ program. There is only one thread running for the entire program. The SystemC simulation kernel, on the other hand, allows the illusion that many SystemC simulation threads are executing in parallel, as we shall learn in the chapters on concurrency.

A simple **SC_THREAD** begins execution when the scheduler calls it. An **SC_THREAD** may also suspend itself, but we will discuss that topic in the next two chapters.

4.5 Registering the Basic Process: SC_THREAD

Once you have defined a process method in the module, you must identify and register it with the simulation kernel. This step allows the thread to be invoked by the simulation kernel's scheduler. The registration occurs within the module class constructor, SC_CTOR, as previously indicated in Fig. 4.7.

To register a SystemC thread, use the cpp macro **SC_THREAD** inside the constructor as shown in Fig. 4.9.

```
SC_THREAD(process_name);//Must be INSIDE constructor
```

Fig. 4.9 Syntax of **SC_THREAD**

The *process_name* is the name of the corresponding method of the class and also needs to be declared as a method within the **SC_MODULE** class. Figure 4.10 is a complete example of an **SC_THREAD** defined within a module:

```
//FILE: basic_process_ex.h
SC_MODULE(basic_process_ex) {
  SC_CTOR(basic_process_ex) {
    SC_THREAD(my_thread_process);
  }
  void my_thread_process(void);
};
```

Fig. 4.10 Example of basic **SC_THREAD**

Traditionally, the code above is placed in a header file that has the same name as the module and has a `.h` filename extension. Thus, the preceding example could appear inside a file named `basic_process_ex.h`.

Notice that `my_thread_process` is not implemented in the module definition, but only declared. In the manner of C++, it is legal to implement the member function within the class, but implementations are traditionally placed in a separate file, the `.cpp` file.

It is also possible to place the implementation of the constructor in the `.cpp` file, as we shall see in the next section. In the following example, we show an implementation for the `my_thread_process` in the implementation file `basic_process_ex.cpp`.

```
//FILE: basic_process_ex.cpp
void basic_process_ex::my_thread_process(void) {
  cout << "my_thread_process executed within "
       << name() //returns sc_module instance name
       << endl;
}
```

Fig. 4.11 Example of basic **SC_THREAD** implementation

Testbench code typically uses **SC_THREAD** processes to accomplish a series of tasks and to eventually stop the simulation. On the other hand, high-level abstraction hardware models commonly include infinite loops to model hardware logic. It is a requirement that such loops explicitly hand over control to other parts of the simulation. This topic will be discussed in a later chapter on concurrency.

4.6 Completing the Simple Design: main.cpp

Now we complete the design with an example of the top-level file for `basic_process_ex`. The top-level file for a SystemC model is placed in the traditional file, `main.cpp` as illustrated in Fig. 4.12.

Notice the string name constructor argument "my_instance" in the preceding example. The reason for this apparent duplication is to store the name of the instance internally for use when debugging. The **SC_MODULE** class member function **name**() may be used to obtain the name of the current instance to print information or debug messages.

```
//FILE: main.cpp
int sc_main(int argc, char* argv[]) { // args unused
  basic_process_ex my_instance("my_instance");
  sc_start();
  return 0; // unconditional success (not
            // recommended)
}
```

Fig. 4.12 Example of simple **sc_main**()

4.7 Alternative Constructors: SC_HAS_PROCESS

Before leaving this chapter on modules, we need to discuss an alternative approach to creating constructors. The alternative approach uses a cpp macro named **SC_HAS_PROCESS**. An explanation of the name will become clear in the chapter on concurrency.

You can use this macro in two situations. First, use **SC_HAS_PROCESS** when you require constructors with arguments beyond just the SystemC module instance name string passed into **SC_CTOR** (e.g., to provide configurable modules). Second, use **SC_HAS_PROCESS** when you want to place the constructor in the implementation (i.e., `.cpp`) file.

You can use constructor arguments to specify sizes of included memories, address ranges for decoders, FIFO depths, clock divisors, FFT depth, and other configuration information. For instance in Fig. 4.13, a memory design might allow selection of different sizes of memories with an argument:

```
My_memory instance("instance", 1024);
```

Fig. 4.13 Example of **SC_HAS_PROCESS** instantiation

To use this alternative approach, invoke **SC_HAS_PROCESS**, just prior to the definition of your conventional constructor. One caveat applies. You must construct or initialize the module base class, **sc_module**, with an instance name string. This requirement is why **SC_CTOR** has an argument.

There are alternate forms using **SC_HAS_PROCESS**. We will first describe in Fig. 4.14 a prevalent style with all of the constructor code defined in the header file. We will then present our preferred approach, which cleanly separates declaration from definition.

```
//FILE: module_name.h
SC_MODULE(module_name) {
  SC_HAS_PROCESS(module_name);
  module_name(sc_module_name
              instname[, other_args…])
  : sc_module(instname)
  [, other_initializers]
  {
    CONSTRUCTOR_BODY
  }
};
```

Fig. 4.14 Syntax of **SC_HAS_PROCESS** in the header

The syntax for using **SC_HAS_PROCESS** in a separate implementation (i.e., separate compilation situation) is similar as shown in Fig. 4.15 and Fig. 4.16. **SC_HAS_PROCESS** can also reside in additional locations but the authors prefer to keep it in the module class definition and close to the constructor or constructor declaration.

In the preceding examples, the *other_args* are optional.

```
//FILE: module_name.h
SC_MODULE(module_name) {
  module_name(sc_module_name
              instname[,other_args…]);
};
```

Fig. 4.15 Syntax of **SC_HAS_PROCESS** separated

```
//FILE: module_name.cpp
SC_HAS_PROCESS(module_name);
module_name::module_name(
  sc_module_name instname[, other_args…])
: sc_module(instname)
[, other_initializers]
{
  CONSTRUCTOR_BODY
}
```

Fig. 4.16 Syntax of **SC_HAS_PROCESS** in the implementation file

4.8 Two Styles Using SystemC Macros

We finish this chapter with two styles for coding SystemC designs. First, we provide the more traditional style, which depends heavily on headers. Second, our recommended style places more elements into the implementation. Creating a C++ templated module usually precludes this style due to C++ compiler restrictions.

You may use either one of these styles for your project, though separating your implementation code in a different file will have certain advantages. We'll visit these topics again in more detail when we discuss the details of hierarchy and structure.

4.8.1 The Traditional Coding Style

The traditional style illustrated in Fig. 4.17 and Fig. 4.18 places all the instance creation and constructor definitions in the header (`.h`) files. Only the implementation of processes and helper functions are coded in the compiled (`.cpp`) file.

```
#ifndef NAME_H
#define NAME_H
#include "submodule.h"
…
SC_MODULE(NAME) {
    Port declarations
    Channel/submodule instances
    SC_CTOR(NAME)
    : Initializations
     {
        Connectivity
        Process registrations
    }
    Process declarations
    Helper declarations
};
#endif
```

Fig. 4.17 Traditional style ***NAME*.h**

```
#include <systemc>
#include "NAME.h"
NAME::Process {implementations }
NAME::Helper {implementations }
```

Fig. 4.18 Traditional style ***NAME*.cpp**

Let's remind ourselves of the basic components in each file. First, the **#ifndef/#define/#endif** preprocessor directives prevent compile problems when the header file is included in multiple files. Using *NAME*_H definition is a standard name for the conditional directive. This definition is followed by file inclusions of any submodule header files by way of **#include**.

Next, the **SC_MODULE**{...}; defines the class definition. Within the class definition, ports are usually the first constructs declared because they represent the interface to the module. Local channels and submodule instances come next.

Next, we place the class constructor, and optionally the destructor. In most cases, using the **SC_CTOR**() {...}; macro proves sufficient in declaring the constructor. Note that **SC_CTOR**() {...}; implies **SC_HAS_PROCESS** and can only be used when no additional constructor arguments are needed. The body of the constructor usually includes initializations, connectivity of submodules, and registration of processes. The constructs mentioned will be discussed in greater detail in the following chapters.

The header finishes out with the declarations of processes, helper functions and possibly other private data. Note that C++ and SystemC do not dictate the ordering of the elements within the class declaration.

The body of a traditional style for the implementation simply includes the SystemC header file, and the corresponding module header described above. The rest of this file simply contains external function member implementations of the processes and functions. Note that it is possible to have no implementation file if there are no processes or helper functions in the module.

4.8.2 Recommended Alternate Style

Here is another style that has some advantages over the preceding style and is illustrated in Fig. 4.19 and Fig. 4.20.

First, the header contains the same **#define** and **SC_MODULE** components as the traditional style. The differences reside in how the channel and submodule definitions are implemented and how the constructor is placed into the implementation body. Notice in the first case that the channel and submodules are implemented using pointers instead of direct instantiation. This method allows for dynamic design configuration that is not possible with direct instantiation. In the second case, placing the constructor code in the implementation file hides the details from potential users.

```
#ifndef NAME_H
#define NAME_H
Submodule forward class declarations
SC_MODULE(NAME) {
    Port declarations
    Channel/Submodule* definitions
    // Constructor declaration:
    SC_CTOR(NAME);
    Process declarations
    Helper declarations
};
#endif
```

Fig. 4.19 Recommended style ***NAME*.h**

```
#include <systemc>
#include "NAME.h"
SC_HAS_PROCESS(NAME);
 NAME::NAME(sc_module_name nm)
: sc_module(nm)
, Initializations
{
    Channel allocations
    Submodule allocations
    Connectivity
    Process registrations
}
NAME::Process {implementations }
NAME::Helper {implementations }
```

Fig. 4.20 Recommended style **NAME.cpp**

4.9 Exercises

For the following exercises, use the samples provided at www.scftgu.com

Exercise 4.1: Compile and run the `basic_process_ex` example from the web site. Add an output statement before **`sc_start()`** indicating the end of elaboration and beginning of simulation.

Exercise 4.2: Rewrite `basic_process_ex` using **`SC_HAS_PROCESS`**. Compile and run the code.

Exercise 4.3: Create two concurrent threads by adding a second **`SC_THREAD`** to `basic_process_ex`. Be sure the output message is unique. Compile and run.

Exercise 4.4: Create two design instances (and hence two concurrent threads) by adding a second instantiation of `basic_process_ex`. Compile and run.

Exercise 4.5: Write a module from scratch using what you know. The output should count down from 3 to 1 and display the corresponding words “Ready”, “Set”, “Go” with each count. Compile and run.

Try writing the code without using **SC_MODULE**. What negatives can you think of for not using **SC_MODULE**? [HINT: Think about EDA vendor-supplied tools that augment SystemC.]