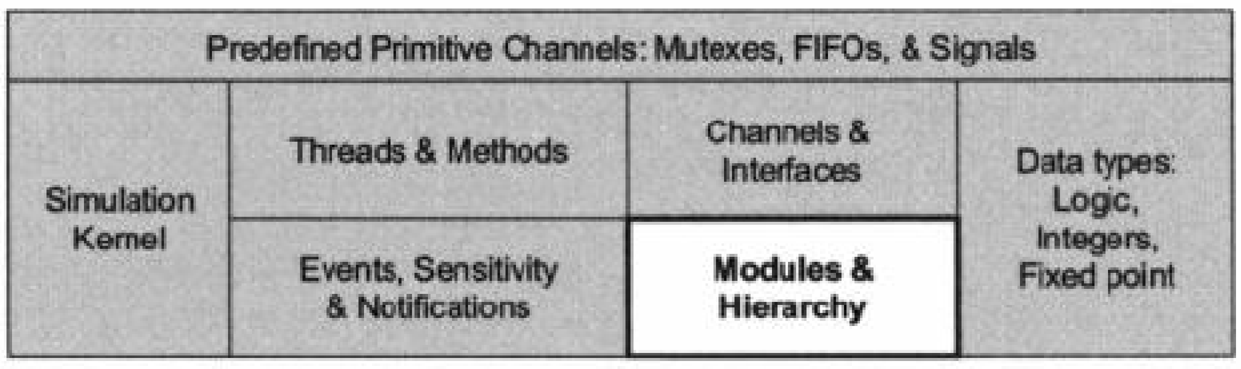
Predefined Primitive Channels: Mutexes, FIFOs, & Signals
Simulation Kernel
Threads & Methods
Channels & Interfaces
Data types: Logic, Integers, Fixed point
Events, Sensitivity & Notifications
Modules & Hierarchy

Chapter 5

MODULES
SC_MODULE

This chapter lays the foundation for SystemC models. In this chapter, we explore how to put together a minimal SystemC program in preparation for an exploration of time and concurrency in the later chapters. With respect to hierarchy, this chapter only touches the very top level. Chapter 10, Structure will discuss hierarchy in more detail.

5.1 A Starting Point: sc_main

All programs need a starting point. In C/C++, the starting point is called **main**. In Verilog and VHDL, it might superficially appear that every process starts at once. In reality, some time passes between initializing the code and beginning the simulation. SystemC having its roots in C/C++ exposes the starting point, and that starting point is known as **sc_main**.

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
}
```

Figure 5-1. Syntax of C++ main()

In the figure above, argc represents the number of command-line arguments including the program name itself. The second argument, 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 a replacement known as **sc_main**(). The SystemC library provides its own definition of **main**(), which in turn calls `sc_main()` and passes along the command-line arguments. Thus, the form of `sc_main()` follows:

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

Figure 5-2. 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[14]. The actual `main()` routine is located in the SystemC library itself.

Within `sc_main()`, code executes in three distinct major phases. Let us examine these phases, which are elaboration, simulation, and post-processing.

During elaboration, structures needed to describe the interconnections of the system are connected. Elaboration establishes hierarchy and initializes the data structures. Elaboration consists of creating instances of clocks, design modules, and channels that interconnect designs. Additionally, elaboration invokes code to register processes and perform the connections between design modules. Within each design, additional layers of design hierarchy are possible.

At the end of elaboration, `sc_start()` invokes the simulation phase. During simulation, code representing the behavior of the model executes. Chapter 7, Concurrency will explore this phase in detail.

Finally, after returning from `sc_start()`, the post-processing phase 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 non-zero return status indicates failure, which may be a computed result of post-processing. A zero return should indicate success (i.e., confirmation that the model correctly passed all tests). Many coders

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

neglect this aspect and simply return zero by default. We recommend that you explicitly confirm the model passed all tests.

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

5.2 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. Components 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. Normally, a macro **SC_MODULE** is used to declare the class:

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

Figure 5-3. Syntax of SC_MODULE

SC_MODULE is a simple cpp[15] macro:

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

Figure 5-4. SystemC Header Snippet of SC_MODULE as #define

[15] 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
- Process member functions (processes)
- Helper functions

Of these, only the constructor is required; however, to have any useful behavior, you must have either a process or a sub-design. Let us first look at the constructor in Section 5.3, and then a simple process in Section 5.4. This sequence lets us finish in Section 5.5 with a basic example of a minimal design.

5.3 The SC_MODULE Class Constructor: SC_CTOR

The **`SC_MODULE`** constructor performs several tasks specific to SystemC. These tasks include:

- Initializing/allocating sub-designs (Chapter 10, Structure)
- Connecting sub-designs (Chapters 10, Structure and 11, Connectivity)
- Registering processes with the SystemC kernel (Chapter 7, Concurrency)
- Providing static sensitivity (Chapter 7, Concurrency)
- Miscellaneous user-defined setup

To simplify coding, SystemC provides a C-preprocessor (cpp) macro, **SC_CTOR**(). The syntax of this macro follows:

```
SC_CTOR(module_name)
 :  Initialization // OPTIONAL
 {
   Subdesign_Allocation
   Subdesign_Connectivity
   Process_Registration
   Miscellaneous_Setup
 }
```

Figure 5-5. Syntax of SC_CTOR

Let us now examine the process, and see how it fits.

5.4 The Basic Unit of Execution: SystemC Process

The process is the basic unit of execution in SystemC. From the time the simulator begins until simulation ends, all executing code is initiated from one or more processes. Processes appear to execute concurrently.

DEFINITION: A SystemC process is a member function or class method of an **SC_MODULE** that is invoked by the scheduler[16] in the SystemC simulation kernel.

The prototype of a process member function for SystemC is:

```
void PROCESS_NAME(void[17]);
```

Figure 5-6. Syntax of SystemC Process

A SystemC process takes no arguments and returns none. This syntax makes it simple for the simulation kernel to invoke. There are several kinds of processes, and we will discuss all of them eventually. For the purposes of simplification, we will look at only one process type in this chapter[18].

[16] We will look closely at the scheduler in Chapter 7, Concurrency.

[17] The keyword void is optional here, and it is typically left out.

[18] Chapter 7, Concurrency looks at the more common process types (e.g., **SC_THREAD** and **SC_METHOD)** in more detail, and Chapter 14 Advanced Topics finishes out the discussion of processes with less common types (e.g., **SC_CTHREAD** and dynamic processes).

The easiest type of process to understand is the SystemC thread, **SC_THREAD**. Conceptually, a SystemC thread is identical to a software thread. In simple C/C++ programs, there is only one thread running for the entire program. The SystemC kernel lets many threads execute in parallel, as we shall learn in Chapter 7, Concurrency.

A simple **SC_THREAD** begins execution when the scheduler calls it and ends when the thread exits or returns. An **SC_THREAD** is called only once[19], just like a simple C/C++ program. An **SC_THREAD** may also suspend itself, but we will discuss that topic in the next two chapters.

5.5 Registering the Simple Process: SC_THREAD

Once you have defined a process method, 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.

Registration of a SystemC thread is coded by using the cpp macro **SC_THREAD** inside the constructor as follows:

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

Figure 5-7. Syntax of SC_THREAD

The *process_name* is the name of the corresponding member method of the class. C++ lets the constructor appear before or after declaration of the process method. Here is a complete example of an **SC_THREAD** defined within a module:

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

Figure 5-8. Example of Simple SC_THREAD

[19] An **SC_THREAD** is similar to a Verilog **initial** block or a VHDL **process** that ends with a simple **wait;**.

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

Notice that `my_thread_process` is not implemented, but only declared. In the manner of C++, it would be 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. As an example, the implementation for the `my_thread_process` would be found in a file named `simple_process_ex.cpp`, and might contain the following:

```
//FILE: simple_process_ex.cpp
void simple_process_ex::my_thread_process(void) {
  std::cout << "my_thread_process executed within "
            << name()
            << std::endl;
 }
```

Figure 5-9. Example of Simple SC_THREAD Implementation

Using **`void`** inside the declaration parentheses is not required; however, this approach clearly states the intent, and it is a legal construct of C++[20].

Test bench code typically uses **`SC_THREAD`** processes to accomplish a series of tasks and finally stops the simulation. On the other hand, high-level abstraction hardware models commonly include infinite loops. It is a requirement that such loops explicitly hand over control to other parts of the simulation. This topic will be discussed in Chapter 7, Concurrency.

[20] Some situations in C++ using templates require the legality of this syntax.

5.6 Completing the Simple Design: main.cpp

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

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

Figure 5-10. Example of Simple sc_main

Notice the string name constructor argument "`my_instance`" in the preceding. 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.

5.7 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`**.

You can use this macro in two situations. First, use **`SC_HAS_PROCESS`** when you require constructors with arguments beyond just the 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 into 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, a memory design might allow selection of different sizes of memories with an argument:

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

Figure 5-11. Example of SC_HAS_PROCESS Instantiation

To use this alternative approach, invoke **SC_HAS_PROCESS**, and then create conventional constructors. One caveat applies. You must construct or initialize the module base class, **sc_module**, with an instance name string. That requirement is why the **SC_CTOR** needed an argument. The syntax of this style when used in the header file follows:

```
//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
   }
};
```

Figure 5-12. 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.

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

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

Figure 5-13. Syntax of SC_HAS_PROCESS Separated

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

5.8 Two Basic Styles

We finish this chapter with two templates for coding SystemC designs. First, we provide the more traditional style, which leans 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.

Use either of these templates for your coding and you'll do well. We'll visit these again in more detail in Chapter 10 when we discuss the details of hierarchy and structure.

5.8.1 The Traditional Template

The traditional template illustrated in *Figure 5-14* and *Figure 5-15* places all the instance creation and constructor definitions in header (`.h`) files. Only the implementation of processes and helper functions are deferred to the compiled (`.cpp`) file. Let's remind ourselves of the basic components in each file. First, the **`#ifndef/#define/#endif`** prevents problems when the header file is included multiple times. Using `NAME_H` definition is fairly standard. This definition is followed by file inclusions of any sub-module header files by way of **`#include`**.

Next, the **`SC_MODULE`**{...}`;` surrounds the class definition. Don't forget the trailing semicolon, which is a fairly common error. Within the class definition, ports are usually the first thing declared because they represent the interface to the module. Local channels and sub-module instances come next. We will discuss all of these later in the book.

Next, we place the class constructor, and optionally the destructor. For many cases, the **`SC_CTOR`**`()` {...} macro proves quite sufficient for this. The body of the constructor provides initializations, connectivity of sub-modules, and registration of processes. Again, all of this will be discussed in detail in following chapters.

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

```
#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
```

Figure 5-14. Traditional Style NAME.h Template

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

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

Figure 5-15. Traditional Style NAME.cpp Template

5.8.2 Recommended Alternate Template Form

For various reasons (discussed in Chapter 10, Structure), we recommend a different approach that is more conducive to independent development of modules. For now, we'll just present the template in *Figure 5-16* and *Figure 5-17* and note the differences.

First, the header contains the same **#define** and **SC_MODULE** components as the traditional style. The differences reside in how the channel/sub-module definitions are implemented and movement of the constructor into the implementation body. Notice that the channel/sub-modules are implemented in a different manner (using pointers).

```
#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
```

Figure 5-16. Recommended Style NAME.h Template

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

Figure 5-17. Recommended Style NAME.cpp Template

5.9 Exercises

For the following exercises, use the samples provided at www.EklecticAlly.com/Book/.

Exercise 5.1: Compile and run the `simple_process_ex` example from the website. Add an output statement before **`sc_start`**`()` indicating the end of elaboration and beginning of simulation.

Exercise 5.2: Rewrite `simple_process_ex` using **`SC_HAS_PROCESS`**. Compile and run the code.

Exercise 5.3: Add a second **`SC_THREAD`** to `simple_process_ex`. Be sure the output message is unique. Compile and run.

Exercise 5.4: Add a second instantiation of `simple_process_ex`. Compile and run.

Exercise 5.5: 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.]