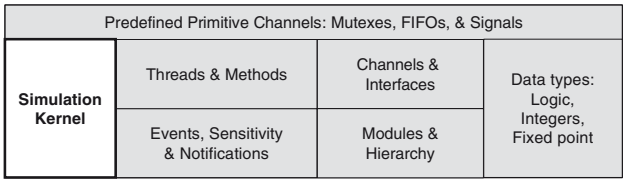
Chapter 5
A Notion of Time



As a SystemC simulation runs, there are three unique time measurements: wall-clock time, processor time, and simulated time:

- The simulation's wall-clock time is the time from the start of execution to completion, including time waiting on other system activities and applications.
- The simulation's processor time is the actual time spent executing the simulation, which will always be less than the simulation's wall-clock time.
- The simulated time is the time being modeled by the simulation, and it may be less than or greater than the simulation's wall-clock time. For example, it might take 2 seconds by your watch (wall-clock time) to simulate 15 ms (simulated time) of your design, but it may only take 1 second (processor time) of the CPU because another program was hogging the processor.

SystemC simulation performance is a combination of many factors: the host system, system load, the C++ compiler, the SystemC simulator, and the model being simulated. Of these factors, the model development team has direct control over the model being simulated in various ways including using efficient coding styles and selecting the correct level of abstraction. This and other chapters identify coding styles and techniques that help create high performance SystemC models.

The remainder of this chapter focuses on the representation and control of simulated time. The SystemC simulator kernel tracks simulated time using a 64-bit unsigned integer. This integer is set to zero as the simulation starts and is increased during simulation in response to the model's behavior.

5.1 sc_time

The data type **`sc_time`** is used by the simulation kernel to track simulated time and to specify delays and timeouts. Internal to SystemC, **`sc_time`** is represented by a minimum of a 64-bit unsigned integer and a time unit **`sc_time`** syntax is illustrated in Fig. 5.1.

D.C. Black et al., *SystemC: From the Ground Up*, Second Edition,
DOI 10.1007/978-0-387-69958-5_5, © Springer Science+Business Media, LLC 2010

```
sc_time name…; // no initialization
sc_time name(double, sc_time_unit)…;
sc_time name(const sc_time&)…;
```

Fig. 5.1 Syntax of **sc_time**

The time units are defined by the enumeration **sc_time_unit**. Table 5.1 lists the available time units:

Table 5.1 SystemC time units

enum	Units	Magnitude
SC_FS	femtoseconds	10^{-15}
SC_PS	picoseconds	10^{-12}
SC_NS	nanoseconds	10^{-9}
SC_US	microseconds	10^{-6}
SC_MS	milliseconds	10^{-3}
SC_SEC	seconds	10^{0}

Objects of **sc_time** data type are declared using the following syntax:

5.1.1 SystemC Time Resolution

All objects of **sc_time** use a single (global) time resolution that has a default of 1 picosecond. The **sc_time** class provides get and set methods to read the time resolution and to change time resolution. The get method **sc_get_time_resolution** shown in Fig. 5.2 returns time resolution as **sc_time**. Because time resolution is a global variable that is used by the simulation kernel and all objects of **sc_time**, changes to time resolution are restricted.

```
//positive power of ten for resolution
sc_set_time_resolution(double, sc_time_unit);
```

Fig. 5.2 Syntax of **sc_set_time_resolution**()

The method **sc_set_time_resolution**() may be used to change time resolution once and only once in a simulation. The change must occur before both creating objects of **sc_time** and starting the simulation. The time resolution set method requires two parameters: the first argument is a **double** that must be a positive power of ten, and the second argument is an **sc_time_unit**. This method has the following syntax:

5.1.2 Working with sc_time

Objects of **sc_time** may be used as operands for assignment, arithmetic, and comparison operations. All operations accept operands of **sc_time;** multiplication allows one of its operands to be a **double;** and division allows the divisor to be a double. Table 5.2 lists the operations supported by the **sc_time** data_type. Figure 5.3 illustrates the syntax for **sc_time**.

In addition to the operations listed above, the **sc_time** data type provides conversion methods to convert **sc_time** to a **double** (**to_double**()) or to a **double** scaled to seconds (**to_seconds**()).

Table 5.2 **sc_time** operators

Comparison	== != < <= > >=
Arithmetic	+ - * /
Assignment	= += -= *= /=

```
sc_time t_PERIOD(5, SC_NS);
sc_time t_TIMEOUT(100, SC_MS);
sc_time t_MEASURE, t_CURRENT, t_LAST_CLOCK;
t_MEASURE = (t_CURRENT-t_LAST_CLOCK);
if (t_MEASURE > t_HOLD) { error("Setup violated") }
```

Fig. 5.3 Examples of **sc_time**

Also the **sc_time** data type overloads the output stream inserter **(operator<<)** allowing a formatted version of **sc_time** to be placed in an output stream for display or printing as shown in Fig. 5.4.

```
ostream_object << desired_sc_time_object;
```

Fig. 5.4 Syntax of **ostream** << overload

5.2 sc_time_stamp()

The SystemC simulation kernel tracks simulated time using an **sc_time** object. Simulated time cannot be directly modified. The method **sc_time_stamp**() can be used to obtain the current simulated **time_value** as illustrated in Fig. 5.5. The returned value is an **sc_time** object. This return value allows **sc_time_stamp**() to be used as any other object for assignment, arithmetic, and comparison operations or to be inserted in an output stream for display or printing.

```
sc_time current_time = sc_time_stamp();
```

Fig. 5.5 Example of `sc_time_stamp()`

Figures 5.6 and 5.7 is a simple example and corresponding output:

```
cout << "  The time is now "
         << sc_time_stamp()
         << "!" << endl;
```

Fig. 5.6 Example of `sc_time_stamp()` and `ostream` << overload

```
  The time is now 0 ns!
```

Fig. 5.7 Output of `sc_time_stamp()` and `ostream` << overload

5.3 sc_start()

The method `sc_start()` is used to start simulation and the syntax is shown in Fig. 5.8. Of interest to this chapter, the `sc_start()` method takes an optional argument of type **sc_time**. This syntax allows the specification of a maximum simulation time. Without an argument to `sc_start()`, a simulation is allowed to run until it is stopped by some other method, until there is no more activity left in the simulation model, or until the simulator's time counter runs out. If you provide a time argument, simulation stops after the specified simulation time has elapsed.

```
//sim "forever"
sc_start();
//sim no more than max_sc_time
sc_start(const sc_time& max_sc_time);
//sim no more than max_time time_unit's
sc_start(double max_time, sc_time_unit time_unit);
```

Fig. 5.8 Syntax of `sc_start()`

The example in Fig. 5.9, which is based on the previous chapter's basic process example, illustrates limiting the simulation to 60 seconds.

```
//FILE: main.cpp
int sc_main(int argc, char* argv[]) { // args unused
  basic_process_ex my_instance("my_instance");
  sc_start(60.0,SC_SEC); // Limit sim to one minute
  return 0;
}
```

Fig. 5.9 Example of `sc_start()`

5.4 wait(sc_time)

Simulations use delays in simulated time to model real world behaviors, mechanical actions, chemical reaction times, or signal propagation. The **wait**() method provides a syntax to allow this delay in **SC_THREAD** processes. When a **wait**() is invoked, the **SC_THREAD** process blocks itself and is resumed by the scheduler after the requested delay in simulated time. The **SC_THREAD** processes will be discussed in detail in the next chapter and will include additional syntaxes for **wait**() .

```
wait(delay_sc_time); // wait specified amount of
                     // time
```

Fig. 5.10 Syntax of **wait**() with a timed delay

When the resolution of **sc_time** used in a wait request is finer than the current time resolution, rounding must occur. The rounding of **sc_time** is not specified. This lack of rounding specification lets different simulators implement different rounding algorithms. The algorithms may vary from vendor to vendor. For example, if the specified time resolution is 100 ps and the request wait time is 20 ps, one simulator could round to zero resulting in an effective 0 ps delay. Another simulator could round to the minimum delay possible for the time resolution resulting in an effective delay of 100 ps.

The examples in Figs. 5.11 and 5.12 use the **sc_time** data type and several of the methods discussed in this chapter.

In the Fig. 5.11 example **wait**() is used to let simulated time advance. Other methods and overloaded operators are used to modify the display of simulated time.

```
//FILE: wait_ex.cpp
void wait_ex::my_thread_process(void) {
  wait(10,SC_NS);
  cout << "Now at " << sc_time_stamp() << endl;
  sc_time t_DELAY(2,SC_MS);
  t_DELAY *= 2;
  cout << "Delaying " << t_DELAY<< endl;
  wait(t_DELAY);
  cout << "Now at " << sc_time_stamp()<< endl;
}
```

output:

```
  Now at 10 ns
  Delaying 4 ms
  Now at 4000010 ns
```

Fig. 5.11 Example of **wait**()

For the Fig. 5.12 example, we know that the simulation will not run for more than two simulated hours (or 7200 seconds) as implied from the line containing **sc_start**(7200, **SC_SEC**). The initial value of t will be between 0.000 and 7200000.000 since the resolution is in milliseconds.

```
//FILE: main.cpp
int sc_main(int argc, char* argv[]) {// args unused
  sc_set_time_resolution(1,SC_MS);
  basic_process_ex my_instance("my_instance");
  sc_start(7200,SC_SEC); // Limit simulation to 2
                         // hours (or 7200 secs.)
  double t = sc_time_stamp(); //max is 7200 x 10**3
  unsigned hours   = int(t / 3600.0);
  t -= 3600.0*hours;
  unsigned minutes = int(t / 60.0);
  t -= 60.0*minutes;
  double   seconds = t;
  cout<< hours<< " hours "
      << minutes<< " minutes "
      << seconds<< " seconds" //to the nearest ms
      << endl;
  return 0;
}
```

Fig. 5.12 Example of **sc_time** data type

5.5 Exercises

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

Exercise 5.1: Examine, compile, and run the example `time_flies`, found on the web site.

Exercise 5.2: Modify `time_flies` to see how much time you can model (days? months?). See how it changes with the time resolution.

Exercise 5.3: Copy the basic structure of `time_flies` and model one cylinder of a simple combustion engine. Modify the body of the thread function to represent physical actions using simple delays. Use **cout** statements to indicate progress.

Suggested activities include opening the intake, adding fuel and air, closing the intake, compressing gas, applying voltage to the spark plug, igniting fuel, expanding gas, opening the exhaust valves, closing the exhaust valves. Use delays representative of 800 RPM. Use time variables with appropriate names. Compile and run.