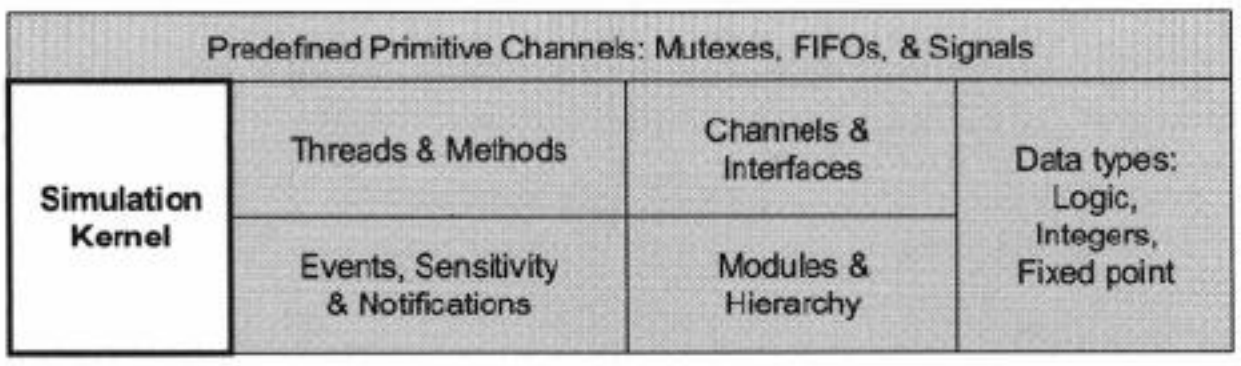

Chapter 6

A NOTION OF TIME

This chapter briefly describes the fundamental notion of time provided by SystemC. We will defer an exploration of many of the intricacies of time until after we discuss events in Chapter 7, Concurrency. Although the time data type itself is simple, the underlying mechanisms are all part of the simulation kernel also discussed in Chapter 7.

6.1 sc_time

SystemC provides the **sc_time** data type to measure time. Time is expressed in two parts: a numeric magnitude and a time unit. Possible time unit specifiers are:

```
SC_SEC  // seconds
SC_MS   // milliseconds
SC_US   // microseconds
SC_NS   // nanoseconds
SC_PS   // picoseconds
SC_FS   // femtoseconds
```

Figure 6-1. Syntax of sc_time Units

The time data type is declared with the following syntax:

```
sc_time name...;   // no initialization
sc_time name(magnitude, timeunits)...;
```

Figure 6-2. Syntax of sc_time

SystemC allows addition, subtraction, scaling, and other related operations on **sc_time** objects. Simple examples include:

```
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") }
```

Figure 6-3. Examples of sc_time

Note the convention used to identify time variables. This convention aids understanding of code. In addition, hard coding of constants is discouraged because it reduces both readability and reusability. One special constant should be noted, **SC_ZERO_TIME,** which is simply **sc_time** (0, **SC_SEC**).

6.2 sc_start()

sc_start() is a key method in SystemC. This method starts the simulation phase, which consists of initialization and execution. Of interest to this chapter, **sc_start**() takes an optional argument of type **sc_time**. This syntax lets you specify a maximum simulation time. Without an argument, **sc_start**() specifies that the simulation may run forever. If you provide a time argument, simulation stops after the specified simulation time has elapsed.

```
sc_start();                  //sim "forever"
sc_start(max_sc_time); //sim no more than max_sc_time
```

Figure 6-4. Syntax of sc_start()

The following example, which is based on the previous chapter's simple process example, illustrates limiting the simulation to sixty seconds.

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

Figure 6-5. Example of sc_start()

Note that internally, SystemC represents time with a 64-bit integer (**`uint64`**). This data type can represent a very long time but not infinite time.

6.3 sc_time_stamp () and Time Display

SystemC's simulation kernel keeps track of the current time and is accessible with a call to the **`sc_time_stamp`**() method.

```
cout << sc_time_stamp() << endl;
```


Figure 6-6. Example of sc_time_stamp()

Additionally, the **`ostream operator<<`** has been overloaded to allow convenient display of time.

```
ostream_object << desired_sc_time_object;
```

Figure 6-7. Syntax of ostream << overload

Here is a simple example and corresponding output:

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

Figure 6-8. Example of sc_time_stamp () and ostream << overload


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

Figure 6-9. Output of sc_time_stamp() and ostream << overload

A more complete example follows in the next section.

6.4 wait(sc_time)

It is often useful to delay a process for specified periods of time. You can use this delay to simulate delays of real activities (e.g., mechanical actions, chemical reaction times, or signal propagation). The `wait()` method provides syntax allowing delay specifications in **`SC_THREAD`** processes. When a **`wait`**`()` is invoked, the **`SC_THREAD`** process blocks itself and is resumed by the scheduler at the specified time. We will discuss **`SC_THREAD`** processes in further detail in the next chapter.

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

Figure 6-10. Syntax of wait () with a Timed Delay

Here are some simple examples:

```
void simple_process_ex::my_thread_process (void)  {
  wait (10,SC_NS);
  std::cout<< "Now at "<< sc_time_stamp() << std::endl;
  sc_time t_DELAY(2,SC_MS); // keyboard debounce time
  t_DELAY *= 2;
  std::cout<< "Delaying "<< t_DELAY<< std::endl;
  wait(t_DELAY);
  std::cout << "Now at " << sc_time_stamp()
            <<std::endl;
}
```

```
% ./run_example
Now at 10 ns
Delaying 4 ms
Now at 4000010 ns
```

Figure 6-11. Example of wait()

6.5 sc_simulation_time(), Time Resolution and Time Units

There are times when you may want to manipulate time in a native C++ data type and shed the time units. For this purpose, `sc_simulation_time()` returns time as a `double` in the current default time unit.

```
sc_simulation_time()
```

Figure 6-12. Syntax of sc_simulation_time ()

To establish the default time unit, call `sc_set_default_time_unit()`. You must call this routine prior to all time specifications and prior to the initiation of `sc_start()`. You may precede this call by specifying the time resolution using `sc_set_time_resolution()`.

These methods have the following syntax:

```
//positive power of ten for resolution
sc_set_time_resolution(value, tunit);
//power of ten >= resolution for default time unit
sc_set_default_time_unit(value, tunit);
```

Figure 6-13. Syntax to set time units and resolution

Rounding will occur if you specify a time constant within the code that has a resolution more precise than the resolution specified by this routine. For example, if the specified time resolution is 100 ps, then coding 20 ps will result in an effective value of 0 ps.

Because the simulator has to keep enough information to represent time to the specified resolution, the time resolution can also have an effect on simulation performance. This result depends on the simulation kernel implementation.

GUIDELINE: Do not specify more resolution than the design needs.

The following example uses the `sc_time` data type and several of the methods discussed in this chapter. For this example, we know that the simulation will not run for more than two hours (or 7200 seconds). The value of t will be between 0.000 and 7200.000 since the resolution is in milliseconds.

```
int sc_main(int argc, char* argv[]) {// args unused
  sc_set_time_resolution(1,SC_MS);
  sc_set_default_time_unit(1,SC_SEC);
  simple_process_ex my_instance("my_instance");
  sc_start(7200,SC_SEC); // Limit simulation to 2
                         // hours
  double t = sc_simulation_time();
  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;
}
```

Figure 6-14. Example of sc_time Data Type

6.6 Exercises

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

Exercise 6.1: Examine, compile, and run the example `time_flies`, found on the website.

Exercise 6.2: Modify `time_flies` to see how much time you can model (days? months?). See how it changes with the time resolution.

Exercise 6.3: 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 **`std::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.