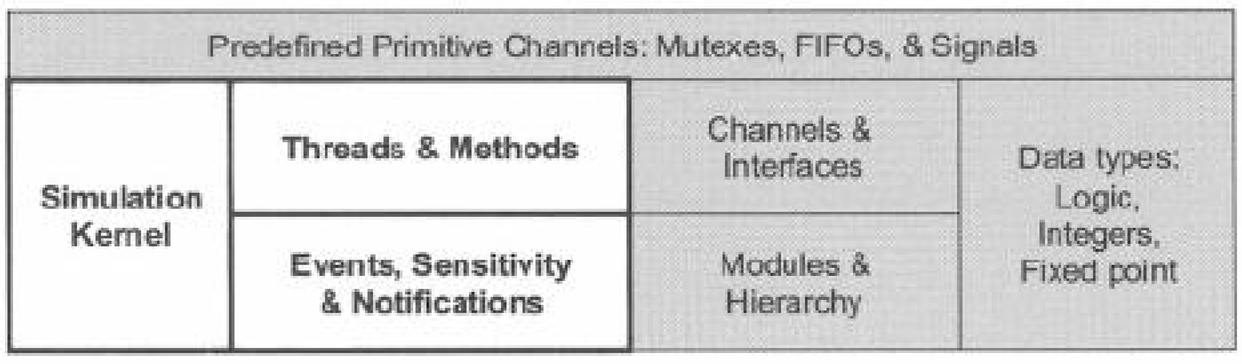

Predefined Primitive Channels: Mutexes, FIFOs, & Signals
Simulation Kernel
Threads & Methods
Events, Sensitivity & Notifications
Channels & Interfaces
Modules & Hierarchy
Data types: Logic, Integers, Fixed point

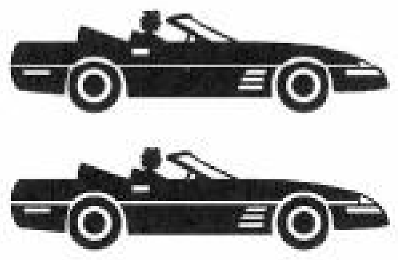
# Chapter 7

# CONCURRENCY

*Processes & Events*

Many activities in a real system occur at the same time or concurrently. For example, when simulating something like a traffic pattern with multiple cars, the goal is to model the vehicles independently. In other words, the cars operate in parallel.

Software typically executes using a single thread of activity, because there is usually only one processor on which to run, and partly because it is much easier to manage. On the other hand, in real systems many things occur simultaneously. For example, when an automobile executes a left turn, it is likely that the left turn indicator is flashing, the brakes are engaged to slow down the vehicle, engine power is decreased as the driver lets off the accelerator, and the transmission is shifting to a lower gear. All of these activities can occur at the same instant.



SystemC uses processes to model concurrency. As with most event-driven simulators, concurrency is not true concurrent execution. In fact, simulated concurrency works like cooperative multi-tasking. In other words, the concurrency is not pre-emptive. Each process in the simulator executes a small chunk of code, and then voluntarily releases control to let other processes execute in the same simulated time space.

As with any simulator, understanding how the simulator handles concurrency enables the designer to use the simulation kernel more effectively and write more efficient models. This understanding will also help avoid many traps.

The simulator kernel is responsible for starting processes and managing which process executes next. Due to the cooperative nature of the simulator model, processes are responsible for suspending themselves to allow execution of other concurrent processes.

SystemC presently provides two major process types, **SC_THREAD** processes and **SC_METHOD** processes. A third type, the **SC_CTHREAD** , is a minor variation on the **SC_THREAD** process that we will discuss separately as an advanced topic in Chapter 14.

## 7.1 sc_event

Before we can discuss how processes work in the simulator, it is necessary to discuss events. Events are key to an event-driven simulator like the SystemC simulation kernel.

An event is something that happens at a specific point in time. An event has no value and no duration. SystemC uses the **sc_event** class to model events. This class allows explicit launching or triggering of events by means of a notification method.

DEFINITION: A SystemC event is the occurrence of an **sc_event** notification and happens at a single point in time. An event has no duration or value.

Once an event occurs, there is no trace of its occurrence other than the side effects that may be observed as a result of processes that were sensitive to or waiting for the event. The following diagram illustrates an event e_rdy "firing" at three different points. Note that unlike a waveform, events have no time width.

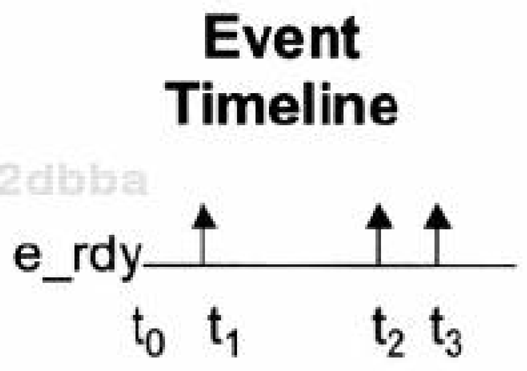


*Figure 7-1.* Graphical Representation of an Event

Because events have no duration, you must be watching to catch them. Quite a few coding errors are due to not understanding this simple rule. Let's restate it.

RULE: To observe an event, the observer must be watching for the event.

SystemC lets processes wait for an event by using a dynamic or static sensitivity that we will discuss shortly. If an event occurs, and no processes are waiting to catch it, the event goes unnoticed. The syntax to declare a named event is simple:

```
sc_event event_name₁ [, event name_i]...;
```

*Figure 7-2.* Syntax of sc_event

Remember that **sc_event**s have no value, and you can perform only two actions with an **sc_event**: wait for it or cause it to occur. We will discuss details in the next sections.

## 7.2 Simplified Simulation Engine

Before proceeding, we need to understand how event-driven simulation works. The following simplified[21] flow diagram illustrates the operation of the SystemC simulation kernel.

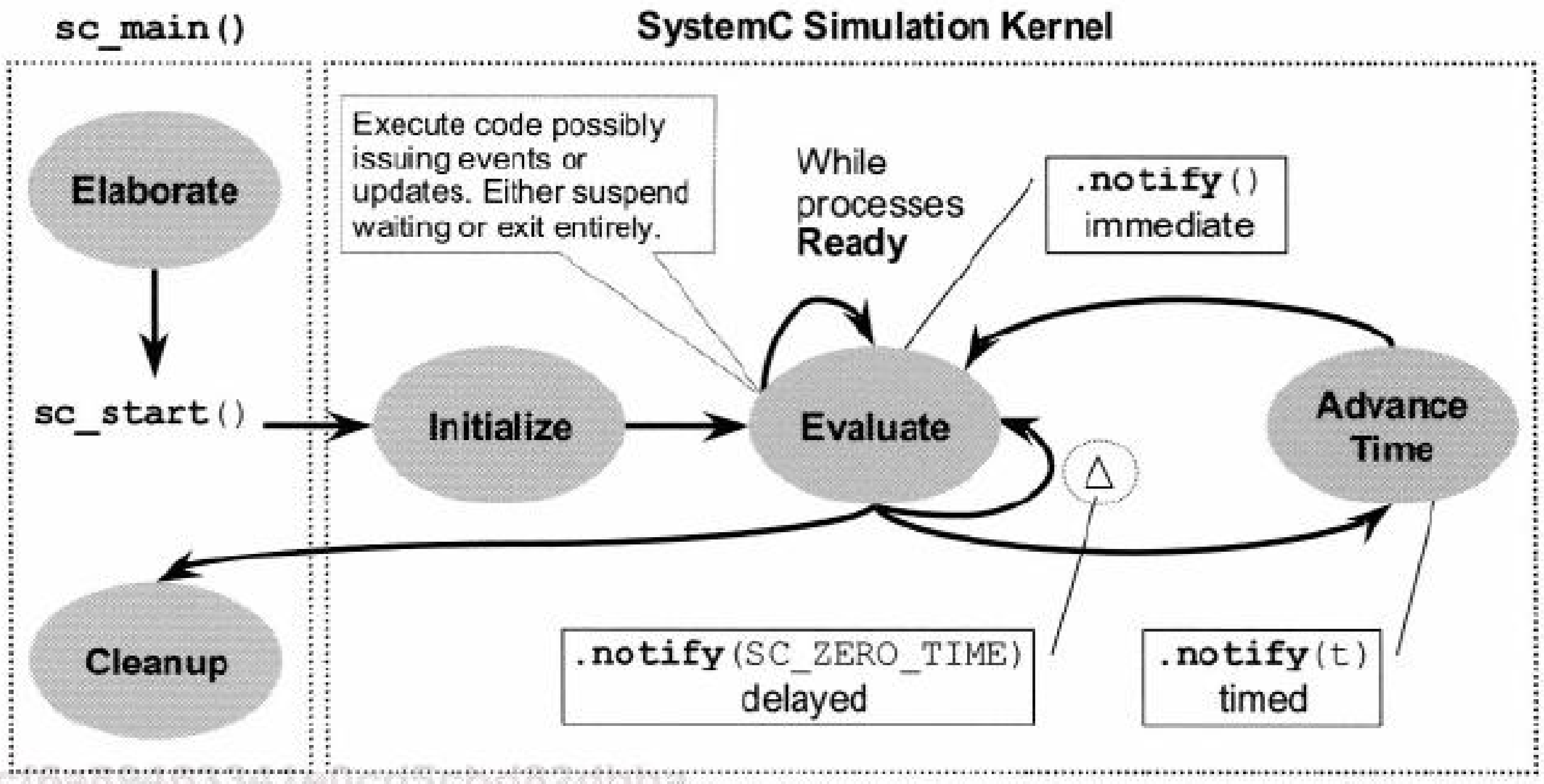

sc_main()
SystemC Simulation Kernel
Elaborate
sc_start()
Initialize
Evaluate
Advance Time
Cleanup
Execute code possibly issuing events or updates. Either suspend waiting or exit entirely.
While processes Ready
.notify() immediate
Δ
.notify(SC_ZERO_TIME) delayed
.notify(t) timed

*Figure 7-3.* **Simplified Simulation Engine**

First, elaboration occurs as previously discussed in Chapter 5, Modules. During elaboration, SystemC modules are constructed and various simulation parameters are established. This elaboration phase is followed by a call to **sc_start**(), which invokes the simulation kernel. This call begins the initialization phase. Processes (e.g., **SC_THREAD** processes) defined during elaboration need to be started. During the initialization phase, all[22] processes are placed initially into a ready pool.

DEFINITION: A process is ready whenever it is available to be executed.

[21] We discuss the remaining details in Chapter 9, Evaluate-Update Channels.
[22] Actually most, but we will discuss this in a later section of this chapter.

We sometimes say that processes are placed into the pool of processes ready to execute. The following diagram depicts process and event pools.

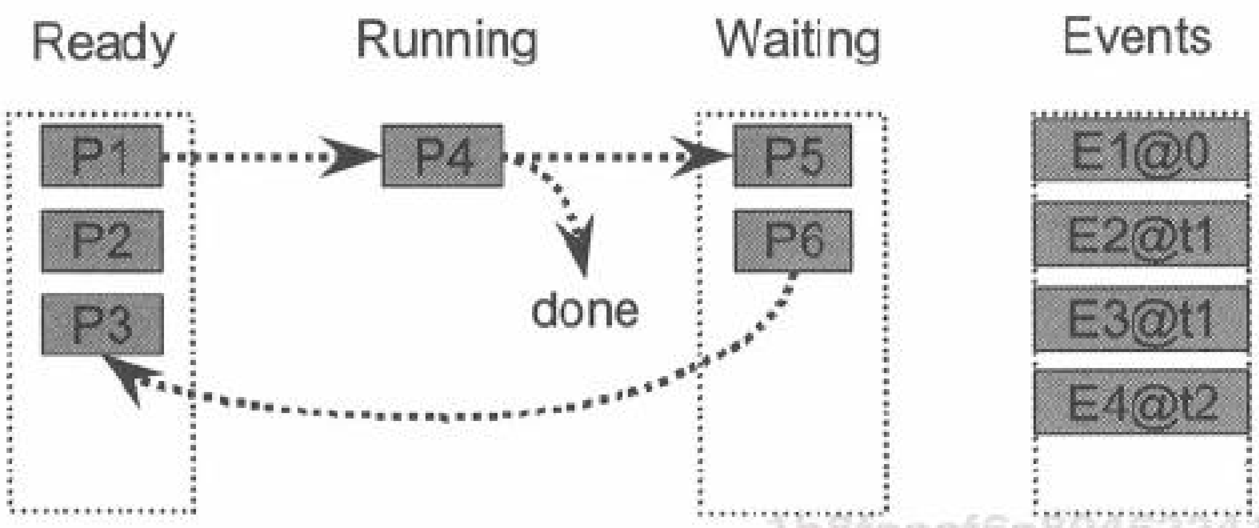


*Figure 7-4* Process and Event Pools

Simulation then proceeds into the evaluation phase. One by one processes are randomly[23] taken from the ready pool by designating them as running and invoked. Each process executes until it either completes (e.g., via a **`return`**) or suspends (e.g., calls **`wait`**`()`).

During execution, a process may invoke immediate event notification (i.e., *`event`*.**`notify`**`()`) and possibly cause one or more waiting processes to be placed in the ready state. It is also possible to generate delayed or timed event notifications as indicated by E1–E4 in the preceding figure. Completed processes are discarded. Suspended processes are placed into a waiting pool. Simulation proceeds until there are no more processes ready to run.

DEFINITION: SystemC enters the waiting state whenever it encounters an explicit **`wait`**`()`, or in some cases performs a **`return`**.

At this point, execution exits the evaluate bubble at the bottom of *Figure 7-3* with one of three possibilities: waiting processes or events that are zero time delayed, non-zero time delayed, or neither.

First, there may be processes or events (to be discussed shortly) waiting for an **`SC_ZERO_TIME`** delay (.**`notify`**(0)). This delay is known as a delta-cycle delay. In this case, waiting pool processes with zero time delays are placed back into the ready pool. Zero time events originating from delayed notifications may cause processes waiting on those events to also be placed into the ready pool. Another round of evaluation occurs if any processes have been moved into the ready pool.

[23] Although not truly random, the SystemC specification allows that different implementations may choose any ordering as is convenient for simulation.

Second, there may be processes or events, scheduled for later, waiting for a non-zero time delay to occur. In this case, time is advanced to the nearest time indicated. Processes waiting on that specific delay will be placed into the ready pool. If an event occurs at this new time, processes waiting on that event are placed into the ready pool. Another round of evaluation occurs if any processes have been moved into the ready pool.

Third, it is possible that there were no delayed processes or events. Since there are no processes in the ready pool, then the simulation simply ends by returning to `sc_main` and finishing cleanup. It is not possible to successfully re-enter `sc_start` in the current definition of SystemC.

## 7.3 SC_THREAD

`SC_THREAD` processes are started once and only once by the simulator. Once a thread starts to execute, it is in complete control of the simulation until it chooses to return control to the simulator.

SystemC offers two ways to pass control back to the simulator. One way is to simply exit (e.g., `return`), which has the effect of terminating the thread for the rest of the simulation. When an `SC_THREAD` process exits, it is gone forever, therefore `SC_THREAD`s typically contain an infinite loop containing at least one `wait`.

The other way to return control to the simulator is to invoke the module `wait` method. The `wait` suspends the `SC_THREAD` process.

Sometimes `wait` is invoked indirectly. For instance, a blocking `read` or `write` of the `sc_fifo` invokes `wait` when the FIFO is empty or full, respectively. In this case, the `SC_THREAD` process suspends similarly to invoking `wait` directly.

## 7.4 Dynamic Sensitivity for SC_THREAD::wait()

As indicated previously, `SC_THREAD` processes rely on the `wait` method to suspend their execution. The `wait` method supplied by the `sc_module` class has several syntaxes as indicated below. When `wait` executes, the state of the current thread is saved, the simulation kernel is put in control and proceeds to activate another ready process. When the suspended process is reactivated, the scheduler restores the calling context of the original thread, and the process resumes execution at the statement after the **wait**.[24]

```
wait(time);
wait(event);
wait(event1 | eventn...);  // any of these
wait(event1 & eventn...);  // all of these
wait(timeout, event);     // event with timeout
wait ( timeout, event1 | eventn...);  //any event with
                                       // timeout
wait ( timeout,event1 & eventn...);  //all events with
                                       // timeout
wait(); // static sensitivity
```

*Figure 7-5.* Syntax of SC_THREAD wait()

We have already described the first syntax in Chapter 6, A Notion of Time; this syntax provides a delay for a specified period. The next several forms specify events and suspend execution until one or all the events have occurred. The operator | is defined to mean any of these events; whichever one happens first will cause a return to **wait**. The operator & is defined to mean all of these events in any order must occur before **wait** returns. The last syntax, **wait**(), will be deferred to a joint discussion with static sensitivity later in this chapter.

Use of a *timeout* is handy when testing protocols and various error conditions. When using a *timeout* syntax, the Boolean function **timed_out**() may be called immediately after the **wait** to determine whether a time out caused execution to resume.

```
...
sc_event ack_event, bus_error_event;
...
wait(t_MAX_DELAY, ack_event | bus_error_event);
if (timed_out()) break; // path for a time out
  ...
```

*Figure 7-6.* Example of timed_out() and wait()

[24] This magic is handled by the SystemC library's scheduler. A detailed description of how context switches are managed goes beyond the scope of this book.

Notice when multiple events are or'ed, it is not possible to know which event occurred in a multiple event wait situation as events have no value (Section 7.1). Thus, it is illegal to test an event for **`true`** or **`false`**.

```
if  (ack_event)  do_something;  //  syntax  error!
```

*Figure 7-7.* Example of Illegal Boolean Compare of sc_event()

It is ok to test a Boolean that is set by the process that caused an event; however, it is problematic to clear it properly. We now need to learn how to generate event occurrences.

## 7.5 Another Look at Concurrency and Time

Let's take a look at a hypothetical example to better understand how time and execution interact. Consider a design with four processes as illustrated in *Figure 7-8*. For this discussion, assume the times, $t_1$, $t_2$, and $t_3$ are non-zero. Each process contains lines of code or statements (`stmt`$_{A1}$, `stmt`$_{A2}$, ...) and executions of wait methods (**`wait`**`(t1)`, **`wait`**`(t2)`,...)

```
Process_A() {
  //@ t0
  stmtA1;
  stmtA2;
  wait(t1);
  stmtA3;
  stmtA4;
  wait(t2);a
  stmtA5;
  stmtA6;
  wait(t3);
}
```

```
Process_B() {
  //@ t0
  stmtB1;
  stmtB2;
  wait(t1);
  stmtB3;
  stmtB4;
  wait(t2);
  stmtB5;
  stmtB6;
  wait(t3);
}
```

```
Process_C() {
  //@ t0
  stmtC1;
  stmtC2;
  wait(t1);
  stmtC3;
  stmtC4;
  wait(t2);
  stmtC5;
  stmtC6;
  wait(t3);
}
```

```
Process_D() {
  //@ t0
  stmtD1;
  stmtD2;
  wait(t1);
  stmtD3;
  wait(
    SC_ZERO_TIME);
  stmtD4;
  wait(t3);
}
```

*Figure 7-8.* Four Processes

Notice that `Process_D` skips $t_2$. At first glance, it might be perceived that time passes as shown below. The uninterrupted solid and hatched line portions indicate program activity. Vertical discontinuities indicate a wait.

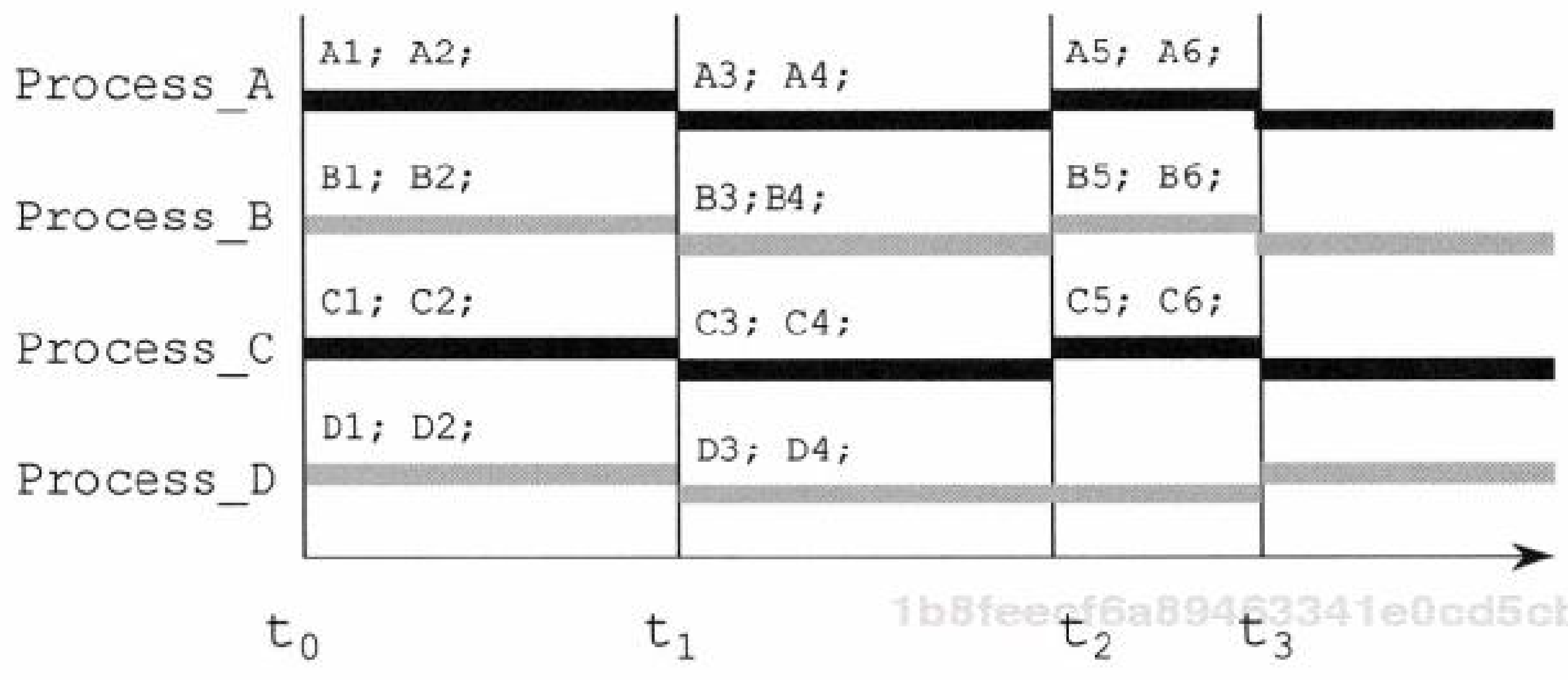


*Figure 7-9.* Simulated Activity—Perceived

Each process' statements take some time and are evenly distributed along simulation time. Perhaps surprisingly that is not how it works at all. In fact, actual simulated activity is shown in the next figure.

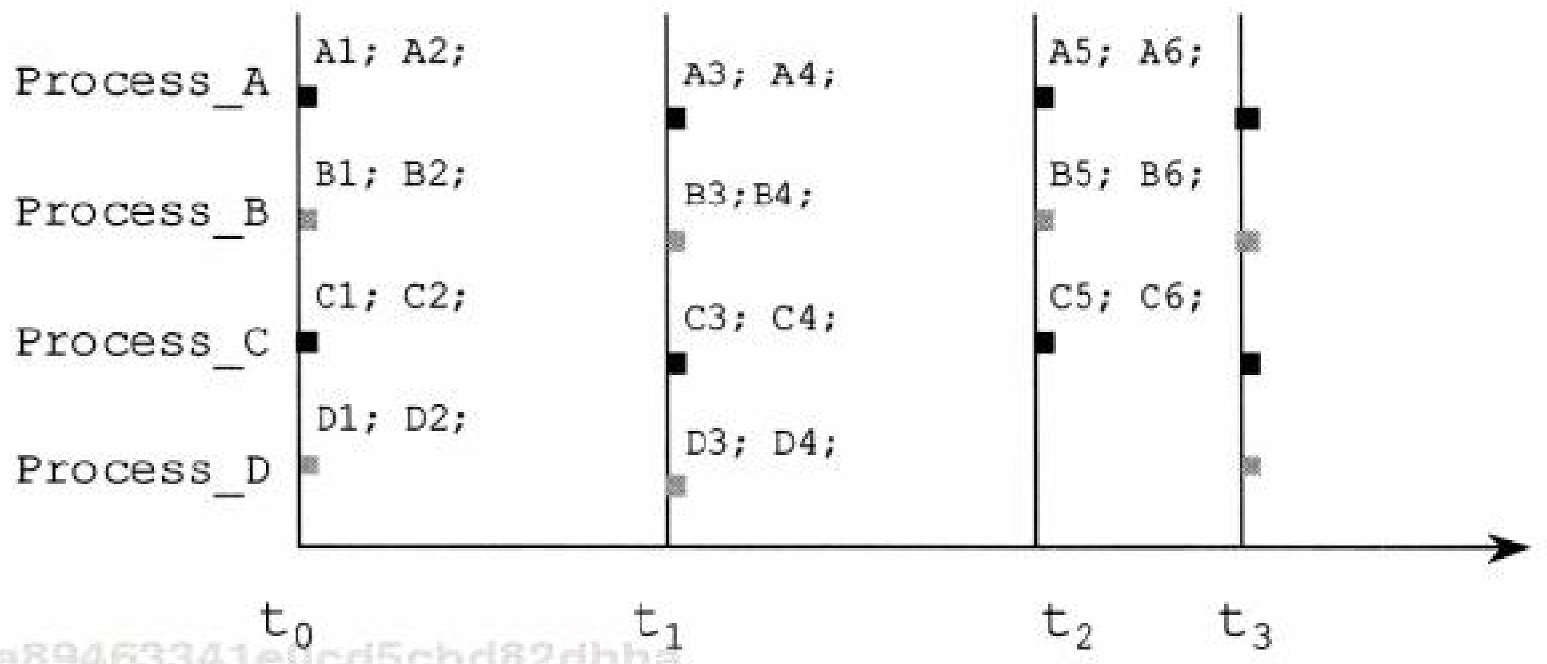


*Figure 7-10.* Simulated Activity-—Actual

Each set of statements executes in zero time! Let's expand the time scale to expose the simulator's time as well. This expansion exposes the operation of the scheduler at work.

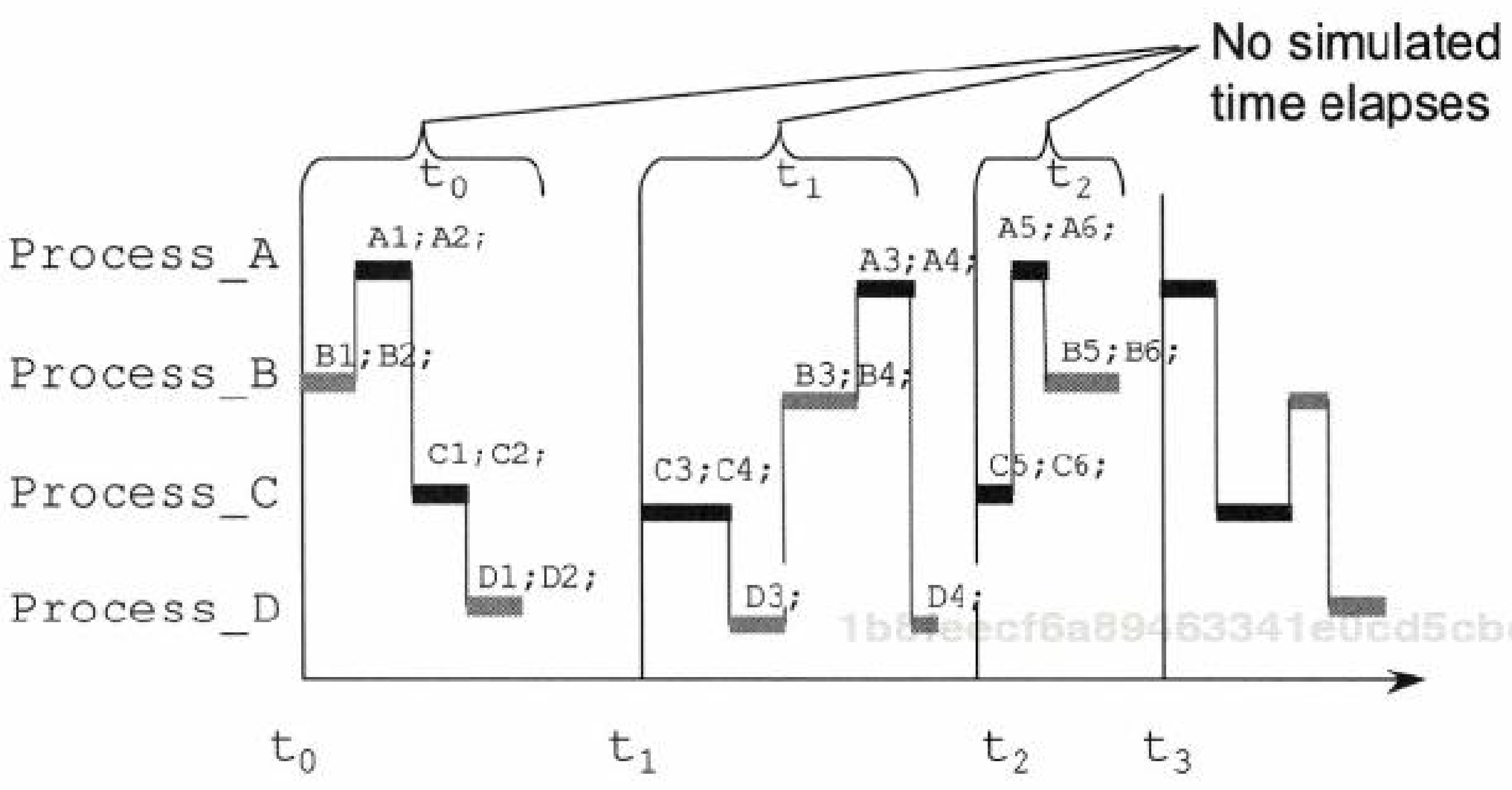


*Figure 7-11.* Simulated Activity with Simulator Time Expanded

Notice that the ordering of processes appears quite non-deterministic in this example. This non-determinism is specified by the SystemC standard. Now for any given simulator and set of code, there is also a requirement that the simulation be deterministic in the sense that one may rerun the simulation and obtain the same results.

All of the executed statements in this example execute during the same evaluate phase of a delta cycle. If any of the statements had been a delayed notification, then multiple delta cycles may have occurred during the same instant in time.

As a final consideration, the previous diagrams would be equally valid with any or all of the indicated times $t_1$, $t_2$, or $t_3$ as zero (i.e., **`SC_ZERO_TIME`**). Once you grasp these fundamental concepts, understanding SystemC behaviors will become much easier.

## 7.6 Triggering Events: .notify()

Events occur explicitly by using the **notify**() method. This method has two syntax styles. The authors prefer the object-oriented style to the function-call style.

```
// Object-oriented style  (preferred)
event_name.notify(); //immediate notification
event_name.notify(SC_ZERO_TIME);// delayed
                                // notification
event_name.notify(time); //timed notification

// Functional-call style
notify(event_name); //immediate notification
notify(event_name, SC_ZERO_TIME) ; // delayed
                                   // notification
notify(event_name, time); //timed notification
```

*Figure 7-12.* Syntax of notify()

Invoking an immediate **notify**() causes any processes waiting for the event to be immediately moved from the wait pool into the ready pool for execution.

Delayed notification occurs when a time of zero is specified. Processes waiting for a delayed notification will execute only after all waiting processes have executed or in other words executes on the next delta-cycle (after an update phase). This is quite useful as we shall see shortly. There is an alternate syntax of .**notify_delayed**(), but this syntax may be deprecated since it is redundant.

Timed notification would appear to be the easiest to understand. Timed events are scheduled to occur at some time in the future.

One confounding aspect of timed events, which includes delayed events, concerns multiple notifications. An **sc_event** may have no more than a single outstanding scheduled event, and only the nearest time notification is allowed. If a nearer notification is scheduled, the previous outstanding scheduled event is canceled.

In fact, scheduled events may be canceled with the .**cancel**() method. Note that immediate events cannot be canceled because they happen at the precise moment they are notified (i.e., immediately).

```
event_name.cancel();
```

*Figure 7-13.* Syntax of cancel() Method

The best way to understand events is by way of examples. Notice in the following example that all of the **notify**s execute at the same instant in time.

```
...
sc_event action;
sc_time now(sc_time_stamp()); //observe current time
//immediately cause action to fire
action.notify();
//schedule new action for 20 ms from now
action.notify(20, SC_MS);
//reschedule action for 1.5 ns from now
action.notify(1.5,SC_NS);
//useless, redundant
action.notify(1.5, SC_NS);
//useless preempted by event at 1.5 ns
action.notify(3.0,SC_NS);
//reschedule action for next delta cycle
action.notify(SC_ZERO_TIME);
//useless, preempted by action event at SC_ZERO_TIME
action.notify(1, SC_SEC);
//cancel action entirely
action.cancel();
//schedule new action for 1 femtosecond from now
action.notify(20,SC_FS);
...
```

*Figure 7-14.* Example of sc_event notify() and cancel() Methods

To illustrate the use of events, let's consider how one might model the interaction between switches on a steering wheel column and the remotely-located signal indicators (lamps). The following example models a mechanism that detects switching activity and notifies the appropriate indicator. For simplicity, only the stoplight interaction is modeled here.

In this model, the process `turn_knob_thread` provides a stimulus and interacts with the process `stop_signal_thread`. The idea is to have several threads representing different signal indicators, and `turn_knob_thread` directs each indicator to turn on or off via the signal_stop and signals_off events. The indicators provide their Status via the stop_indicator_on and stop_indicator_off events.

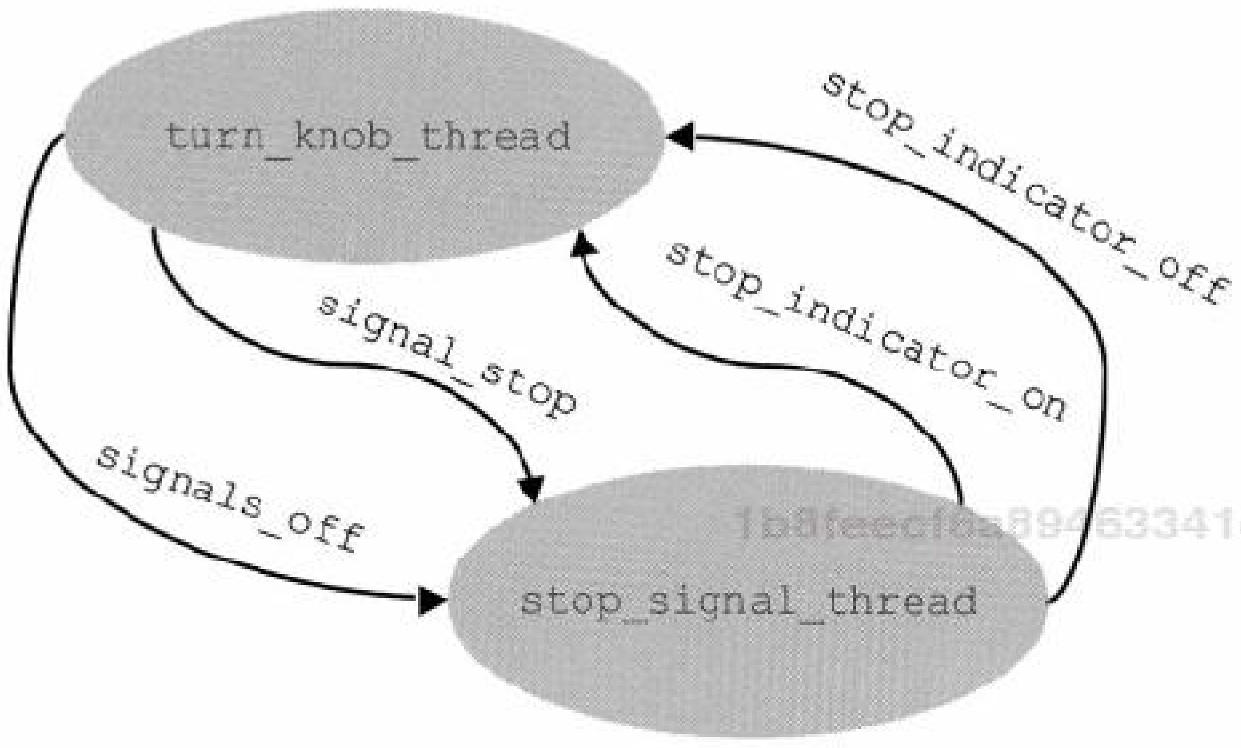


*Figure 7-15.* Turn of Events Illustration

```
//FILE: turn_of_events.h
SC_MODULE(turn_of_events)  {
  // Constructor
  SC_CTOR(turn_of_events) {
    SC_THREAD(turn_knob_thread);
    SC_THREAD(stop_signal_thread);
  }
  sc_event signal_stop, signals_off;
  sc_event stop_indicator_on, stop_indicator_off;
  void turn_knob_thread(); // stimulus process
  void stop_signal_thread(); // indicator process
};//endclass turn_of_events
```

*Figure 7-16.* Example of turn_of_events Header

An interesting aspect of the implementation shown in the following figure is consideration of process ordering effects. Recall the rule that "To see an event, a process must be waiting for it." It is because of this requirement that the turn_knob_thread implementation starts out with **wait**(**SC_ZERO_TIME**). Without that pause, if turn_knob_thread runs first, then the stop_signal_thread will never see any events because it will not have executed the first **wait**(). As a result, the simulation would starve and exit.

Similarly, consider what would happen if the signals_off event were issued before signal_stop. If an unconditional wait for acknowledgement occurred, the simulation would exit. It would exit because the turn_knob_thread would be waiting on an event that never occurs because the stop_signal_thread was not in a position to issue that event.

```
//FILE:  turn_of_events.cpp
void turn_of_events::turn_knob_thread() {
  // This process provides stimulus using stdin
  enum directions {STOP="S", OFF="F"};
  char direction; // Selects appropriate indicator
  bool did_stop = false;
  // allow other threads to get into waiting state
  wait(SC_ZERO_TIME);
  for(;;) {
    // Sit in an infinite loop awaiting keyboard
    // or STDIN input to drive the stimulus…
    std::cout << "Signal command: ";
    std::cin >> direction;
    switch (direction) {
      case STOP:
        // Make sure the other signals are off
        signals_off.notify();
        signal_stop.notify(); // Turn stop light on
        // Wait for acknowledgement of indicator
        wait(stop_indicator_on);
        did_stop = true;
        break;
      case OFF:
        // Make the other signals are off
        signals_off.notify() ;
        if (did_stop) wait(stop_indicator_off) ;
        did_stop = false;
        break;
    }//endswitch
  }//endforever
}//end turn_knob_thread()
```

*Figure 7-17.* Example of Turn of Events Stimulus

```
void turn_of_events:: stop_signal_thread() {
  for(;;) {
    wait(signal_stop);
    std::cout << "STOPPING     !!!!!!" << std::endl;
    stop_indicator_on.notify();
    wait(signals_off);
    std::cout << "Stop off     ------" << std::endl;
    stop_indicator_off.notify();
  }//endforever
}//end stop_signal_thread()
```

*Figure 7-18.* Example of Turn of Events Indicator

The preceding example, `turn_of_events`, models two processes with SystemC threads. The `turn_knob_thread` takes input from the keyboard and notifies the `stop_signal_thread`. Sample output might look as follows (user input highlighted):

```
% ./turn_of_events.x
Signal command: S
STOPPING  !!!!!!
Signal command: F
Stop off  ------…
```

*Figure 7-19.* Example of Turn of Events Output

## 7.7 SC_METHOD

As mentioned earlier, SystemC has more than one type of process. The **SC_METHOD** process is in some ways simpler than the **SC_THREAD**; however, this simplicity makes it more difficult to use for some modeling styles. To its credit, **SC_METHOD** is more efficient than **SC_THREAD**.

What is different about an **SC_METHOD**? One major difference is invocation. **SC_METHOD** processes never suspend internally (i.e., they can never invoke **wait**()). Instead, **SC_METHOD** processes run completely and return. The simulation engine calls them repeatedly based on the dynamic or static sensitivity. We will discuss both shortly.

In some sense, **SC_METHOD** processes are similar to the Verilog **always@** block or the VHDL **process**. By contrast, if an **SC_THREAD** terminates, it never runs again in the current simulation.

Because **SC_METHOD** processes are prohibited from suspending internally, they may not call the **wait** method. Attempting to call **wait** either directly or implied from an **SC_METHOD** results in a runtime error.

Implied waits result from calling SystemC built-in methods that are defined such that they may issue a **wait**. These are known as blocking methods. The **read** and **write** methods of the **sc_fifo** data type, discussed later in this book, are examples of blocking methods. Thus, **SC_METHOD** processes must avoid using calls to blocking methods.

The syntax for **SC_METHOD** processes follows and is almost identical to **SC_THREAD** except for the keyword **SC_METHOD:**

```
SC_METHOD(process_name);//Located INSIDE constructor
```

*Figure 7-20.* Syntax of SC_METHOD

A note on the choice of these keywords might be useful. The similarity of name between an **SC_METHOD** process and a regular object-oriented method betrays its name. It executes without interruption and returns to the caller (the scheduler). By contrast, an **SC_THREAD** process is more akin to a separate operating system thread with the possibility of being interrupted and resumed.

Variables allocated in **SC_THREAD** processes are persistent. **SC_METHOD** processes must declare and initialize variables each time the method is invoked. For this reason, **SC_METHOD** processes typically rely on module local data members declared within the **SC_MODULE**. **SC_THREAD** processes tend to use locally declared variables.

GUIDELINE: To differentiate threads from methods, we strongly recommend adopting a naming style. One naming style appends `_thread` or `_method` as appropriate. Being able to differentiate processes based on names becomes useful during debug.

## 7.8 Dynamic Sensitivity for SC_METHOD: next_trigger()

**SC_METHOD** processes dynamically specify their sensitivity by means of the **next_trigger**() method. This method has the same syntax as the **wait()** method but with a slightly different behavior.

```
next_trigger(time);
next_trigger(event);
next_trigger(event1 | eventi...); //any of these
next_trigger (event1 & eventi...); //all of these
                                  //required
next_trigger(timeout, event);    //event with timeout
next_trigger(timeout, event1 | eventi...) ; //any + timeout
next_trigger(timeout, event1 & eventi...); //all + timeout
next_trigger() ; //re-establish static sensitivity
```

*Figure 7-21.* Syntax of SC_METHOD next_trigger()

As with **wait,** the multiple event syntaxes do not specify order. Thus, with **next_trigger** (evt1 & evt2), it is not possible to know which occurred first. It is only possible to assert that both evt1 and evt2 happened.

The **wait** method suspends **SC_THREAD** processes; however, **SC_METHOD** processes are not allowed to suspend. The **next_trigger** method has the effect of temporarily setting a sensitivity list that affects the **SC_METHOD**. **next_trigger** may be called repeatedly, and each invocation encountered overrides the previous. The last **next_trigger** executed before a return from the process determines the sensitivity for a recall of the process. The initialization call is vital to making this work. See the **next_trigger** code in the downloads section of the website for an example.

You should note that it is critical for EVERY path through an **SC_METHOD** to specify at least one **next_trigger** for the process to be called by the scheduler. Without a **next_trigger** or static sensitivity (discussed in the next section), an **SC_METHOD** will never be executed again. A safeguard can be adopted of placing a default **next_trigger** as the first statement of the **SC_METHOD**, since subsequent **next_triggers** will overwrite any previous. A better way to manage this problem exists as we will now discuss.

## 7.9 Static Sensitivity for Processes

Thus far, we've discussed techniques of dynamically (i.e., during simulation) specifying how a process will resume (either by **`SC_THREAD`** using **`wait`** or by **`SC_METHOD`** using **`next_trigger)`**. SystemC provides another type of sensitivity for processes called static sensitivity. Static sensitivity establishes the parameters for resuming during elaboration (i.e., before simulation begins). Once established, static sensitivity parameters cannot be changed (i.e. they're static). It is possible to override static sensitivity as we'll see.

Static sensitivity is established with a call to the **`sensitive()`** method or the overloaded stream **`operator<<`** that is placed just following the registration of a process. Static sensitivity applies only to the most recent process registration. **`sensitive`** may be specified repeatedly. There are two syntax styles:

```
// IMPORTANT: Must follow process registration
sensitive << event [<< event]…;//  streaming style
sensitive (event [, event]…);   //  functional style
```

*Figure 7-22.* Syntax of sensitive

We prefer the streaming style as it feels more object-oriented, reduces typing, and keeps us focused on C++.

For the next few sections, we will examine the problem of modeling access to a gas station to illustrate the use of sensitivity coupled with events. Initially, we model a single pump station with an attendant and only two customers. The declarations for this example in *Figure 7-24* illustrate the use of the **`sensitive`** method.

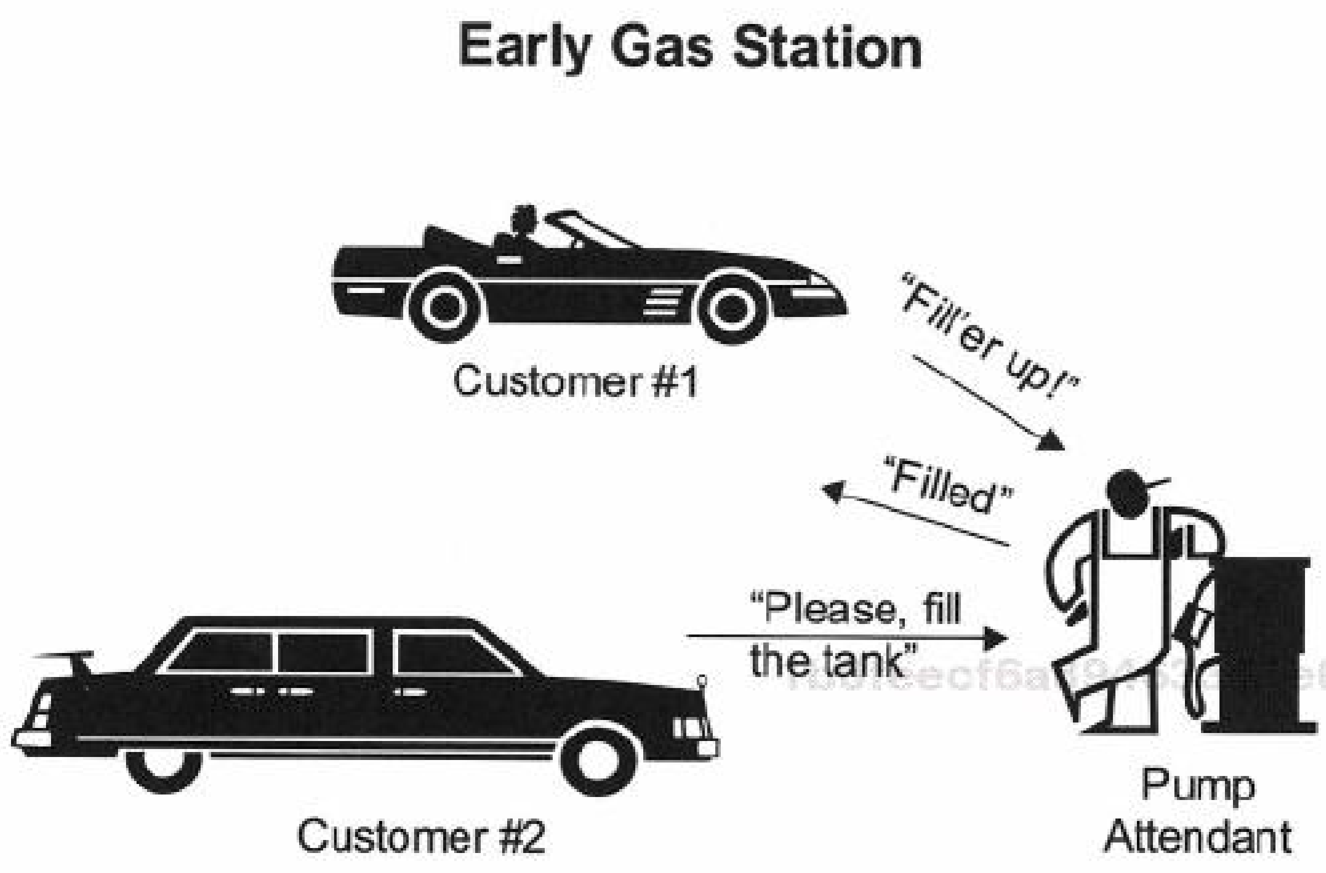


*Figure 7-23.* Initial Gas Station Illustration

```
SC_MODULE(gas_station) {
  sc_event e_request1, e_request2;
  sc_event e_tank_filled;
  SC_CTOR(gas_station) {
    SC_THREAD (customer1_thread);
      sensitive (e_tank_filled); // functional
                                 // notation
    SC_METHOD (attendant_method);
      sensitive  <<  e_request1
                 <<  e_request2; // streaming notation
    SC_THREAD (customer2_thread);
  }
  void attendant_method();
  void customer1_thread();
  void customer2_thread();
};
```

*Figure 7-24.* Example of Gas Station Declarations

The `gas_station` module has two processes with different sensitivity lists and one, `customer2_thread`, which has none. The `attendant_method` implicitly executes every time an `e_request1` or `e_request2` event occurs (unless dynamic sensitivity is invoked by the simulation process).

Notice the indentation used in *Figure 7-24*. This format helps draw attention to the sensitivity being associated with only the most recent process registration.

Here are some fragments of the implementation code focused on the elements of this chapter. You can find the full code in the downloads section of the website.

```
...
void gas_station:: customer1_thread() {
  for (;;) {
    wait(EMPTY_TIME);
    cout << "Customer1 needs gas" << endl;
    do {
      e_request1.notify();
      wait(); // use static sensitivity
    } while (m_tank1 == 0) ;
  } //endforever
}//end customer1_thread()

// omitting customer2_thread (almost identical
// except using wait(e_request2);)

void gas_station:: attendant_method() {
  if (!m_filling) {
      ...
      cout << "Filling tank" << endl;
      m_filling = true;
      next_trigger(FILL_TIME) ;
      ...
  } else {
      ...
      e_filled.notify(SC_ZERO_TIME);
      cout << "Filled tank" << endl;
    ...
    m_filling = false;
    ...
  }//endif
}//end attendant_method()
```

*Figure 7-25.* Example of Gas Station Implementation

The preceding code produces the following output:

```
...
Customer1 needs gas
Filling tank
Filled tank
...
```

*Figure 7-26.* Example of Gas Station Sample Output

## 7.10 dont_initialize

The simulation engine description specifies that all processes are executed initially. This approach makes no sense in the preceding `gas_station` model as the `attendant_method` would fill the tank before being requested.

Thus, it becomes necessary to specify some processes that are not initialized. For this situation, SystemC provides the **`dont_initialize`** method. The syntax follows:

```
// IMPORTANT: Must follow process registration
dont_initialize();
```

*Figure 7-27.* Syntax of dont_initialize()

Note that the use of **`dont_initialize`** requires a static sensitivity list; otherwise, there would be nothing to start the process. Now our **`gas_station`** module contains:

```
    ...
    SC_METHOD(attendant_method);
      sensitive(fillup_request);
      dont_initialize();
    ...
```

*Figure 7-28.* Example of dont_initialize()

## 7.11 sc_event_queue

In light of the limitation that **sc_event**s may only have a single outstanding schedule, **sc_event_queue**s have been added in SystemC version 2.1. These additions let a single event be scheduled multiple times even for the same time! When events are scheduled for the same time, each happens in a separate delta cycle.

```
sc_event_queue event_name₁("event_name₁")…;
```

*Figure 7-29.* Syntax of sc_event_queue

**sc_event_queue** is slightly different from **sc_event**. First, **sc_event_queue** objects do not support immediate notification since there is obviously no need to queue these. Second, the **.cancel()** method is replaced with **.cancel_all()** to emphasize that it cancels all outstanding **sc_event_queue** notifications.

```
…
sc_event_queue  action;
sc_time now(sc_time_stamp()); //observe current time
action.notify(20, SC_MS);//schedule for 20 ms from now
action.notify(1.5,SC_NS);//another for 1.5 ns from
                           //now
action.notify(1.5,SC_NS);//another identical action
action.notify(3.0,SC_NS);//another for 3.0 ns from
                            //now
action.notify(SC_ZERO_TIME);//for next delta cycle
action.notify(1,SC_SEC);//for 1 sec from now
action.cancel_all(); // cancel all actions entirely
…
```

*Figure 7-30.* Example of sc_event_queue

The **.cancel()** method is not currently implemented; although, an obvious extension might be to allow canceling notifications at specific times. Another extension might be obtaining information on how many outstanding notifications exist (**.pending()**).

## 7.12 Exercises

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

**Exercise 7.1:** Examine, predict the behavior, compile, and run the `turn_of_events` example.

**Exercise 7.2:** Examine, predict the behavior, compile, and run the `gas_station` example.

**Exercise 7.3:** Examine, predict the behavior, compile, and run the `method_delay` example.

**Exercise 7.4**: Examine, predict the behavior, compile, and run the `next_trigger` example.

**Exercise 7.5:** Examine, predict the behavior, compile, and run the `event_filled` example. If using SystemC version 2.1, compile the simulation a second time with the macro `SYSTEMC21` defined (-`DSYSTEMC21` command-line option for gcc).