Chapter 14
Additional Topics

Reporting, Clocks, Clocked Threads, Programmable Hierarchy, and Signal Tracing

Congratulations for keeping up to this point. This chapter begins with important discussions of reporting, configuration and programmable structure, and a basic discussion of clocks. The chapter then quickly accelerates to a discussion of the **`SC_CTHREAD`**, which is followed by a discussion on debugging and waveform tracing.

If you are able to follow this section, then you are ready to take on the world. However, if you become discouraged, come back and reread the chapter after gaining a little more SystemC coding experience.

14.1 Error and Message Reporting

Reporting information about the state and status of a simulation as it progresses is an important art. Many teams create utilities to standardize this reporting within the project because of the large volume of data from reporting. Many a project has seen thousands, if not millions of lines of output from simulations. In fact, controlling output can have a significant effect on run-time performance. At the same time, it is crucial that engineers have a solid handle on any errors that are produced and have enough information to efficiently debug the problems that arise.

Messages have classifications including informational, warning, error, and fatal. Additionally, messages usually apply to a variety of areas and need to be isolated to their source to aid debugging. For simulations, it is also important to identify the time that a message occurs. Because simulations provide a tremendous amount of output data, it is important that messages be standardized and easy to identify.

SystemC has an error reporting system that greatly simplifies this task. Throughout our examples thus far, you have seen a stylized format of error management. In this short section, we will examine a subset of the error-reporting facilities in SystemC. For more information, you are referred to the SystemC LRM and the example documentation that accompanies the release.

We need a few definitions first. Every message is associated with an identifying name. This labeling is used to keep messages from different parts of the design properly identified. It can be anything; however, we recommend something along the lines of *"/COMPANY/PROJECT_OR_IP/FUNCTIONAL_AREA"*. A message identifier is simply a character string (Fig. 14.1):

```
const char* MSGID = "UNIQUE_STRING";
```

Fig. 14.1 Syntax of message identifier

Next, all messages need to be classified. SystemC has the following classifications (Fig. 14.2):

```
SC_INFO    – informational only–this includes debug
SC_WARNING – possible problem, possibly harmless
SC_ERROR   – problem identified probably serious
SC_FATAL   – extremely serious problem probably
             ending simulation
```

Fig. 14.2 Error classifications

For each classification, a variety of actions may be taken. For the most part, defaults are sufficient. Possible actions include the following actions taken from the SystemC example documentation (Table 14.1):

Table 14.1 Error actions

Error Classification	Action
SC_UNSPECIFIED	Take the action specified by a configuration rule of a lower precedence.
SC_DO_NOTHING	Don't take any actions for the report. The action will be ignored, if other actions are given.
SC_THROW	Throw a C++ exception (**sc_exception**) that represents the report. The method **sc_exception::get_report()** can be used to access the report instance later.
SC_LOG	Print the report into the report log, which is typically a file on disk. The actual behavior is defined by the report handler function.
SC_DISPLAY	Display the report to the screen, which is typically done by writing it into the standard output channel using **std::cout**.
SC_INTERRUPT	Interrupt simulation if simulation is not being run in batch mode. Actual behavior is implementation-defined; the default configuration calls **sc_interrupt_here**(...) debugging hook and has no further side effects.
SC_CACHE_REPORT	Save a copy of the report for the current process. The report could be read later using **sc_report_handler::get_cached_report()**. The reports saved by different processes do not overwrite each other; however, the default behavior is to save only one cached report per process.
SC_STOP	Call **sc_stop()**. See **sc_stop()** manual for further detail.
SC_ABORT	The action requests the report handler to call **abort()**.

SystemC has a large class of setup that may be specified for message reporting. For basic designs, the following syntax should suffice (Fig. 14.3):

```
sc_report_handler::set_log_file_name("filename");
sc_report_handler::stop_after(SC_ERROR, MAXERRORS);
sc_report_handler::set_actions(MSGID,CLASS,ACTIONS);
```

Fig. 14.3 Syntax for basic message setup

The following code, named `report`, illustrates the basics of message handling (Figs. 14.4 and 14.5):

```
const char* MSGID = "/ESLX/Examples/mysim";
const char* sim_vers = "Version 5.2"; // Code ver-
sion
int sc_main(int argc, char* argv[]) {
  sc_report rp;
  sc_report_handler::set_log_file_name("run.log");
  sc_report_handler::stop_after(SC_ERROR, 100);
  sc_report_handler::set_actions(
    MSGID, SC_INFO, SC_DISPLAY|SC_LOG
  );
  SC_REPORT_INFO(MSGID,sim_vers);
  …/* Body of main */
  sc_start();
  if (sc_report_handler::get_count(SC_ERROR) > 0
   || sc_report_handler::get_count(SC_FATAL)
     )
  {
    cout << rp.->get_msg() << endl;
    cout << MSGID << " FAILED" << endl;
    return 1;
  } else {
    cout << MSGID << " PASSED" << endl;
    return 0;
  }
}
```

Fig. 14.4 Example of **`main.cpp`** with SystemC error reporting

```
extern char* MSGID;
void mymod::some_thread() {
  wait(2,SC_NS);
  SC_REPORT_INFO(MSGID,"Sample info");
  SC_REPORT_WARNING(MSGID,"Sample warning");
  SC_REPORT_ERROR(MSGID,"Sample error");
  SC_REPORT_FATAL(MSGID,"Sample fatal");
}
```

Fig. 14.5 Example of reporting in a module

Here is a sample of the log file output (Fig. 14.6):

```
0 s: Info: /ESLX/Examples/mysim: Version 5.2
2 ns: Info: /ESLX/Examples/mysim: Sample info
2 ns: Warning: /ESLX/Examples/mysim: Sample warning
In file: mymod.cpp:21
In process: mymod_i.some_thread @ 2 ns
2 ns: Error: /ESLX/Examples/mysim: Sample error
In file: mymod.cpp:22
In process: mymod _i.some_thread @ 2 ns
…
```

Fig. 14.6 Example of output messages

Notice that all the messages have a standard format. SystemC has added some useful information to the messages: the simulated time, filename, line number, and process identification. Also notice that the Info messages do not include module name and line number information by default. This is controlled by the **`sc_report_handler`**.

You can enhance the output by using a syntax-highlighting editor and setting up a coloring scheme for log files. A slightly more involved route involves supplanting the default report handler with your own. An advantage to this is that you can enhance the output options including perhaps providing an XML output as we have done for some of our customers.

Along the line of enhancing reporting, it is useful to have standard preludes and summaries. In the prelude, it is nice to specify such things as the versions of files, the name of the running host computer, and date of execution. In the summary, it is essential to know if the simulation passed or failed. It is also useful to know the clock wall time (i.e., how long did it take to simulate), and how many errors, warnings, etc. (i.e., statistics) were encountered.

All of this information can be reported by creating an object in the topmost design as the first instance in the module. The object should be a class that issues the prelude in the constructor and the summary in the destructor. This approach will guarantee execution at the correct times in the simulation lifetime. Other things should happen outside of simulation and they are discussed in the next section.

14.2 Elaboration and Simulation Callbacks

The SystemC **`sc_module`** class provides four routines that may be overridden, and they are executed at the boundaries of simulation. These routines provide modelers with a place to put initialization and clean-up code that has no place to live. For example, checking the environment, reading run-time configuration information and generating summary reports at the end of simulation. The member functions are as follows (Fig. 14.7).

```
void before_end_of_elaboration(void);
void end_of_elaboration(void);
void start_of_simulation(void);
void end_of_simulation(void);
```

Fig. 14.7 `sc_module` callbacks

It is important to realize these are called once for every module instance in a design. In many cases, it is desirable to execute code only once per module. A static variable may be invoked to serve this purpose. Below (Fig. 14.8) is an example using the `sc_module` callbacks.

```
void top::before_end_of_elaboration(void) {
  // Can add to elaboration here
  // Can setup reporting here
  sc_report_handler::stop_after(SC_ERROR,100);
}
void top::end_of_elaboration(void) {
  this->count++; // count instances
  static bool once(false);
  if (!once) {
    once = true;
    // possible to examine netlist here
  }
}
void top::start_of_simulation(void) {
  // report on counts talled beforehand
  // initialize channels/ports
}
void top::end_of_simulation(void) {
  static bool once(false);
  if (!once) {
    once = true;
    // provide post-processing/cleanup code here
  }
}
```

Fig. 14.8 Example using callbacks

14.3 Configuration

Configuring the design and its environment is a very important topic, and it has three techniques. Selection of these techniques affects the development time, compile time and run time. There is a time and a place for using each of these techniques.

First, there is configuration affected by the designer's choice of data types and constructors coded directly into the source code. This type of configuration is static, and it is not very easy to modify. Changes involve both editing (usually manual) and recompilation. This technique may be appropriate early in the design cycle.

Second, configuration of the design at compile time using C pre-processor (**cpp**) constructs allows for some variation provided re-compilation is acceptable. This form of configuration is limited to simple conditional forms (e.g., **#if**, **#ifdef**, **#ifndef**, **#else**) and to more complex text substitution. It has the undesirable aspect of being difficult to debug, and changes invoke recompilation, which may be lengthy for larger projects. This technique is appropriate for selections that may be affected by platform OS and tool versions (e.g., Linux vs. Windows).

The third form of configuration is the most interesting: run-time configuration. This form has the advantage of not requiring recompilation, but it has the potential disadvantage of more coding than the other two forms. Configuration information for run time can be obtained from environment variables using the **getenv**[1], from the command line, from files, from **cin**, or a combination of all the preceding (Fig. 14.9). Here is an example using an environment variable:

```
#include <cstdlib>

const char* varname = "MYVAR";
string result("*UNDEFINED*");
char* value = getenv(varname);
if (value != NULL) {
  result = value;
} else {
  if (setenv(varname,"x",0) == -1) {
    result = ""; // failed means it is defined
  } else {
    unsetenv(varname); // undo the setenv
  }
}//endif
```

Fig. 14.9 Example of run-time environment variable retrieval

One can also use the command line. To help with the command line, SystemC provides two global functions, **sc_argc()** and **sc_argv()**, that correspond to the values passed to **main()** and correspondingly **sc_main()**. These may be called anywhere in your code. In the following example (Fig. 14.10), we create a simple function that specifies an option to check for and return an optional value:

[1] See the manual page for **getenv** in a Linux environment.

```
bool uint_option(string opt, unsigned &value) {
  string arg;
  for (unsigned i=1; i!=sc_argc(); ++i) {
    arg = sc_argv()[i];
    if (arg.find(opt,0) != 0) continue;
    if (arg.length() == 0) continue;
    arg.erase(0,opt.length());
    if (isdigit(arg[0])) {
      istringstream ins(arg);
      ins >> value;
      return true;
    }//endif
  }//endfor
  return false;
}
…
// usage
if (uint_option("-n=", n)) {
  size = n;
}//endif
```

Fig. 14.10 Example of command-line run-time configuration

We will leave reading a file to establish run-time configuration as an exercise for the reader. Quite simply, this option is straightforward C++ programming. Ever since our undergraduate days, we've wanted to say this.

The run-time configuration information may be used at any time (i.e., during elaboration or simulation). The next section discusses how to use run-time configuration at elaboration.

14.4 Programmable Structure

Programmable structuring is an aspect of SystemC that may be obvious to some but not to others. Structure for SystemC occurs at elaboration time before **sc_start()** is called. The code that performs elaboration (i.e., instantiates modules, channels, and connects them) is executable C++ code in the form of one or more constructor functions. This means that it is possible to use standard C++ constructs such as **if-then-else**, **switch**, **for**, and **while** loops to dynamically establish the design's structure (e.g., connectivity).

Thus, it is conceivable to have simulations that use run-time configuration to alter their code structure. In some cases, this dynamic connectivity is a matter of convenience. For instance, configurability is appropriate for a large regular structure. In other cases, configurability may be a way to test various aspects of the design. Let's look at a couple of examples.

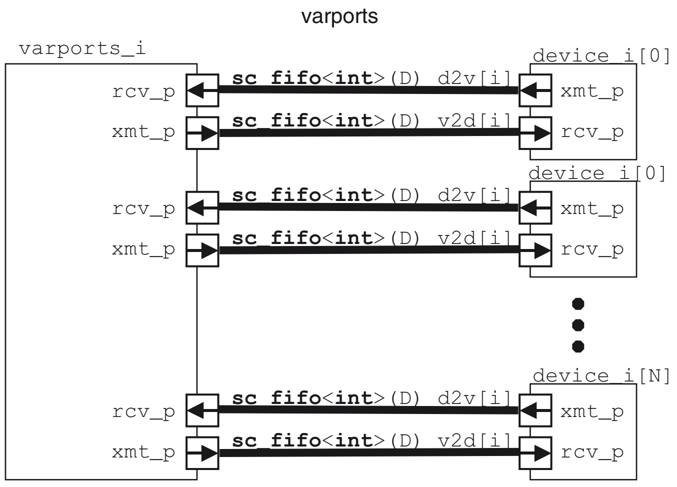

Fig. 14.11 Design with 1-N ports

First, we consider a design that supports a variable number of devices attached externally. Take for example, an Ethernet or USB port. The specification diagram looks something like the previous figure (Fig. 14.11).

To test this design, the verification team would like a single executable that can be configured at run time to handle 0 to 16 devices with varying FIFO depths. The supporting code is shown in Fig. 14.12 on the next page.

The preceding example uses arrays of pointers to both the instances and the channels connecting them. We could have dynamically set the array size; however, it would not save enough resources to justify the complexity and effort.

Our next example recognizes the importance of configuration management. A design may start out with a TLM and eventually be refined to RTL. It is desirable to be able to run simulations that easily select portions of the design to run at TLM or RTL levels. TLM portions will simulate quickly; RTL portions will represent something closer to the final implementation and will simulate more slowly. This configurability lets the verification engineer keep simulations running quickly, and he or she can focus on finding problems in a particular area.

Configurability may be achieved by using conditional code (e.g., **`if–else`**) around the areas of interest. For example, consider the hierarchical channel design of the previous chapter (`hier_chan` example). Suppose we package both the architectural model and the behavioral model within a wrapper that lets us configure the design at run time.

The supporting code is shown in Fig 14.12

```
#include <sstream>
#include "varports.h"
#include "device.h"
class testbench : public sc_module {
public:
  varports*     varports_i;
  device*       device_i[N]; //N previously set to
16
  sc_fifo<int>* v2d[N];
  sc_fifo<int>* d2v[N];
  SC_CTOR(testbench);
};

// Constructor
SC_HAS_PROCESS(testbench);
testbench::testbench(sc_module_name mdl)
: sc_module(mdl)
{
  /* Figure out N from command-line */
  unsigned nDevices, depth;
  uint_option("-n=",nDevices);//See 2 pages back
  varports_i = new varports(…init parameters…);
  for (unsigned i=0;i!=nDevices;i++) {
    stringstream nm; // for unique instance names
    // Create instances
    nm.str(""); nm << "device_name_i[" << i << "]";
    device_i[i] = new device(nm.str().c_str());
    nm.str(""); nm << "v2d[" << i << "]";
    v2d[i]=new sc_fifo<int>(nm.str().c_str(),depth);
    nm.str(""); nm << "d2v [" << i << "]";
    d2v[i]=new sc_fifo<int>(nm.str().c_str(),depth);
    // Connect devices to varports using channels
    device_i[i]->rcv_p(*v2d[i]);
    device_i[i]->xmt_p(*d2v[i]);
    varports_i->rcv_p(*d2v[i]);
    varports_i->xmt_p(*v2d[i]);
  }//endfor
}
```

Fig. 14.12 Example of configurable code with 1-*N* ports

We can read the configuration instance names into an STL `map<KEY, VALUE>`. An example of the wrapper code in a memory module and `sc_main()` is shown in the next figure (Figs. 14.13 & 14.14). The code shown defaults to an architectural implementation, mem_arch. Both an RTL and bsyn configuration are supported; although, the selection of an RTL version only produces a warning message.

```
#include <sstream>
#include <map>
extern std::map<sc_string, sc_string> cfg;

SC_MODULE(mem) {
  mem_arch*                 mem_arch_i;
  mem_bsyn*                 mem_bsyn_i;
  …
  SC_HAS_PROCESS(mem);
  explicit mem(sc_module_name nm
              ,unsigned long ba // mem base address
              ,unsigned sz)     // mem size
  : sc_channel(nm)
  {
    if (cfg[name()] == "rtl") {
      SC_REPORT_FATAL(MSGID, "RTL not supported");
    }
    if (cfg[name()] == "bsyn") {
      SC_REPORT_INFO(MSGID, "Configuring bsyn");
      mem_bsyn_i = new mem_bsyn("mem_bsyn_i",ba,sz);
      // module instantiations and connections
      …
    } else {
      SC_REPORT_INFO(MSGID, "Configuring arch");
      mem_arch_i = new mem_arch("mem_arch_i",ba,sz);
      …
    }//endif
  }
};
```

Fig. 14.13 Example of configurable code to manage modeling levels

```
#include <map>
std::map<string, string> cfg;

int sc_main(int argc, char *argv[])
{
  ifstream cf("sim.cfg");
  if (!cf) {
    SC_REPORT_FATAL("EX","Unable to read file");
  } else {
    string inst, model;
    while(cf>>inst) {
      if (cf>>model) {
        cfg[inst] = model;
      }
    }
  }
  …
};
```

Fig. 14.14 Example of configurable code in `sc_main()`

14.5 sc_clock, Predefined Processes

Clocks represent a common hardware behavior, that of a repetitive Boolean value. If you are a hardware designer, it is likely you've been concerned about the late discussion of this topic. This topic is delayed for a reason.

Clocks add many events, and much resulting simulation activity is required to update those events. Consequently, clocks can slow simulations significantly. Additionally, quite a lot of hardware can be modeled adequately without clocks. If you need to delay a certain number of clock cycles, it is much more efficient to execute a wait for the appropriate delay than to count clocks as illustrated in Fig. 14.15.

```
wait(N*t_PERIOD) // one event -> FAST!
-OR-
for(i=1;i<=N;i++) // creates many events -> slow
  wait(clk->posedge_event())
```

Fig. 14.15 Comparing wait statements to clock statements

More importantly, many designs can be modeled without any delays. It all depends on information to be derived from the model at a particular stage of a project.

A clock can be easily modeled with SystemC. Indeed, we have already seen an example of a clock modeled with just an event, namely the `heartbeat` example. More commonly, clocks are modeled with a **`sc_signal<bool>`** and the associated event.

Clocks are so common that SystemC provides a built-in hierarchical channel known as a **`sc_clock`** (Fig. 14.6). Clocks are commonly used when modeling low-level hardware where clocked logic design currently dominates.

```
sc_clock name("name",period
             [,duty_cycle=0.5
              ,start_time=0
              ,posedge_first=true]);
```

Fig. 14.16 Syntax of **`sc_clock`**

Notice the optional items indicated by their defaults.

Some caveats apply to **`sc_clock`**. First, if declared within a module, **`sc_clock`** must be declared and initialized prior to its use. Second, if you want to communicate a clock as an output to the module, you must use an **`sc_export<sc_signal_in_if<bool> >`**.

For example (Fig. 14.17):

```
SC_MODULE(clock_gen) {
  sc_export<sc_signal_inout_if<bool> > clkout_p;
  sc_port<sc_signal_inout_if<bool>   > clkdiv_p;
  sc_clock clk;
  SC_CTOR(clock_gen)
  : clk("clk",sc_time(6,SC_NS))
  {
    SC_METHOD(clk_method);
    sensitive << clk.posedge_event();
    clkout_p(clk);
  }
  void clk_method() {
    clkdiv_p->write(!clkdiv_p->read());
  }
};
```

Fig. 14.17 Example of `sc_clock` generation

The preceding example exports a clock and uses a method to produce a derived clock at half the frequency. This approach inevitably slows the simulation. This method also entails more code.

14.6 Clocked Threads, the SC_CTHREAD

SystemC has two basic types of processes: the `SC_THREAD` and the `SC_METHOD`. A variation on the `SC_THREAD` that is popular for behavioral synthesis tools is the clocked thread or `SC_CTHREAD`. This popularity is partly because synthesized logic tools currently produce fully synchronous code, and it is partly because the `SC_CTHREAD` provides some new facilities to simplify coding (Fig. 14.18).

```
SC_CTOR(module_name) {
  SC_CTHREAD(NAME_cthread, clock_name.edge());
}
```

Fig. 14.18 Syntax of `SC_CTHREAD`

One of the simpler facilities provided by this new simulation process is a new behavior of `wait`(`void`) (Fig. 14.19).

```
  wait(void); // go to start of next clock cycle
```

Fig. 14.19 Syntax of clocked wait

In versions of SystemC prior to standardization, there is another syntax for **`wait`***(N)* and a level-sensitive wait, called **`wait_until()`**. We mention this because you are likely to see this in legacy code for several years. The syntaxes are (Fig. 14.20):

```
wait(N); // delay N clock edges
wait_until(delay_expr); // until expr true @ clock
```

Fig. 14.20 Older syntax of clocked waits

The syntax for **`wait_until()`** requires the delay expression, ***`delay_expr`***, must be expressed using delayed signals. In other words, the argument for **`wait_until()`** must be of the form *signal*.**`delayed()`**. The **`delayed()`** method is a special method that provides the value at the end of a delta-cycle. Keep in mind that all of this is deprecated in the standard, and this syntax only applies to versions prior to OSCI version 2.2.

Neither of these is extremely interesting (Fig. 14.21). They are correspondingly almost equivalent to the following **`SC_THREAD`** code assuming the thread is statically sensitive to a clock edge:

```
for(i=0;i!=N;i++) wait();//similar as wait(N)
do wait() while(!expr);// sames as
                      // wait_until(dexpr)
```

Fig. 14.21 Example of code equivalent to clocked thread **`wait()`** and **`wait_until()`**

Of greater interest, **`SC_CTHREAD`** provides the concept of reset signals, which effectively changes the behavior of **`wait()`**. When a reset signal activates, execution jumps back to the start of the function upon return from **`wait()`** rather than proceeding to the next statement. The syntax is simple and follows (Fig. 14.22):

```
SC_CTOR(module_name) {
  SC_CTHREAD(NAME_cthread);
  reset_signal_is(signal,true);
}
```

Fig. 14.22 Syntax of watching

Previous versions of SystemC also included other constructs to watch signals. These constructs included calls to **`watching()`**, and the use of macros named **`W_BEGIN`**, **`W_DO`**, **`W_ESCAPE`**, and **`W_END`**. Check the documentation for the version of SystemC used with legacy code that you may need to reuse.

Here (Fig. 14.23) is an example of how to implement similar functionality in SystemC:

```
#include "processor.h"
SC_HAS_PROCESS(processor);
processor::processor(sc_module_name nm)
//Constructor
: sc_module(nm)
{
  // Process registration
  SC_CTHREAD(processor_cthread,clock_p.pos());
  reset_signal_is(reset_p, false);
}//endconstructor }}}
class Aborted {}; // used for throwing
#define WAIT_CYCLE \
  wait(); if (abort_p->read()==true) throw Aborted
void processor::processor_cthread() { //{{{
  // Initialization
  pc = RESET_ADDR;
  for(;;) {
    try {
      WAIT_CYCLE(); // use instead of wait();
      read_instr();
      switch(opcode) {
        case LOAD_ACC:
          acc = bus_p->read(operand1);
          break;
        case STORE_ACC:
          bus_p->write(operand1,acc);
          break;
        case INCR:
          acc++;
          result = (acc != 0);
          break;
      }
      …
    } catch (Aborted) {
      SC_REPORT_WARNING("Aborting");
    }//endtry
  }//endforever
}//endcthread
```

Fig. 14.23 Example code using clocked threads

We'll note one last point. Just like **SC_THREAD**, upon exiting an **SC_CTHREAD** never runs again. Normally, **SC_CTHREAD** contains an infinite loop.

There has been some discussion of deprecating **SC_CTHREAD**. However, **SC_THREAD** functionality may need to be augmented by the extra mechanisms of watching and the resulting simplified syntax before eliminating this feature.

14.7 Debugging and Signal Tracing

Until this point, we have assumed the use of standard C++ debugging techniques such as in-line print statements or using a source code debugger such as gdb. Hardware designers are familiar with using waveform viewing tools that display values graphically.

While SystemC does not have a built-in graphic viewer, it can copy data values to a file in a format compatible with most waveform viewing utilities. The format is known as VCD or Value Change Dump format. It is a simple text format.

Obtaining VCD files involves three steps. First, open the VCD file. Next, select the signals to be traced. These two steps occur during elaboration. Running the simulation (i.e., calling `sc_start()`) will automatically write the selected data to the dump file. Finally, close the trace file. Here is the syntax presented in sequence (Fig. 14.24):

```
sc_trace_file* tracefile;
tracefile =
sc_create_vcd_trace_file(tracefile_name);
if (!tracefile) cout <<"There was an error."<<endl;
...
sc_trace(tracefile,signal_name,"signal_name");
...
sc_start(); // data is collected
...
sc_close_vcd_trace_file(tracefile);
```

Fig. 14.24 Syntax to capture waveforms

It is required that the signal names being traced are defined before calling **`sc_trace`**. Also, it is possible to use hierarchical notation to access signals in submodules. It is possible to trace ordinary C++ data values and ports as well. The trace filename should not include the filename extension since the **`sc_create_vcd_trace_file`** automatically does this. Notice the error checking of the file creation using the Boolean complement operator (!).

A simple coding example using **`sc_trace`** is show in an example in Fig. 14.25.

Notice the use of a destructor to close the file. Using a destructor is the safest way to ensure the file will be closed. If additional modules are instantiated in the example above, they would need to include appropriate **`sc_trace`** syntax within their constructors.

[2] Available from http://www.cs.man.ac.uk/apt/tools/gtkwave

Another moderately complex example of signal tracing may be found in the `tracing` example from the book web site. A simple coding example (Fig. 14.25):

```
//FILE: wave.h
SC_MODULE(wave) {
  sc_signal<bool> brake;
  sc_trace_file*  tracefile;
  …
  double temperature;
};
```

```
//FILE: wave.cpp
wave::wave(sc_module_name nm) //Constructor
: sc_module(nm) {
  …
  tracefile = sc_create_vcd_trace_file("wave");
  sc_trace(tracefile,brake,"brake");
  sc_trace(tracefile,temperature,"temperature");
}//endconstructor
wave::~wave() {
    sc_close_vcd_trace_file(tracefile);
    cout << "Created wave.vcd" << endl;
}
```

Fig. 14.25 Example of simple waveform capture

Here is some sample output viewed with the open source `gtkwave`[2] viewer (Fig. 14.26):

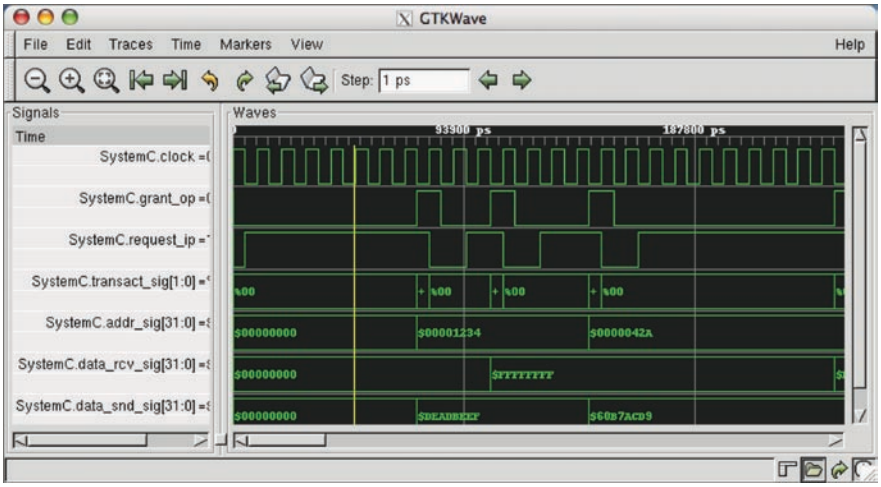

Fig. 14.26 Sample waveform display from gtkwave

This manual designation of waveforms is required when using the OSCI simulator. Many of the commercial SystemC implementations let you bypass this step and do signal tracing interactively.

14.8 Other Libraries: SCV, ArchC, and Boost

Beyond the core of SystemC, several libraries are available for the serious SystemC user to explore. These include:

- The SystemC Verification library, the SCV, has an extensive set of features useful for verification. The original set was donated by Cadence Design Systems. This library is discussed in a later chapter.
- The ArchC architecture description language is an open source architecture description language used to describe processors and create SystemC models. Several models are already available. ArchC was designed at the Computer Systems Laboratory (LSC) of the Institute of Computing of the University of Campinas (IC-UNICAMP). See www.archc.org for more information.
- The Boost web site provides free peer-reviewed portable C++ source libraries. The emphasis is on libraries that work well with the C++ Standard Library. See www.boost.org for more information.

14.9 Exercises

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

Exercise 14.1: Examine, compile, and run the `clock_gen` example. Change `clk_method` to a thread. Measure the performance difference.

Exercise 14.2: Examine, compile, and run the `processor` example. Notice the clocked thread constructs. Can you think of better ways to code this from an execution performance standpoint?

Exercise 14.3: Examine, compile, and run the varports example.

Exercise 14.4: This exercise examines design configuration. Examine, compile, and run the `manage` example. Can you think of a simpler way to manage different implementations that leverages C++?

Exercise 14.5: Examine, compile, and run the wave example. View the VCD data using a waveform viewer. Obtain `gtkwave` from http://intranet.cs.man.ac.uk/apt/projects/tools/gtkwave/ if necessary.

Exercise 14.6: Examine, compile, and run the `tracing` example.

Exercise 14.7: Examine, compile, and run the `report` example. Apply these concepts to an earlier example.