# SystemC Introduction

## Introduction

In series of labs we will introduce SystemC library and understand various features. SystemC provides a C++ based library for modeling or creating a Software executable specification of electronic systems containing Hardware and embedded SW. Below figure shows the architecture of a SystemC application.

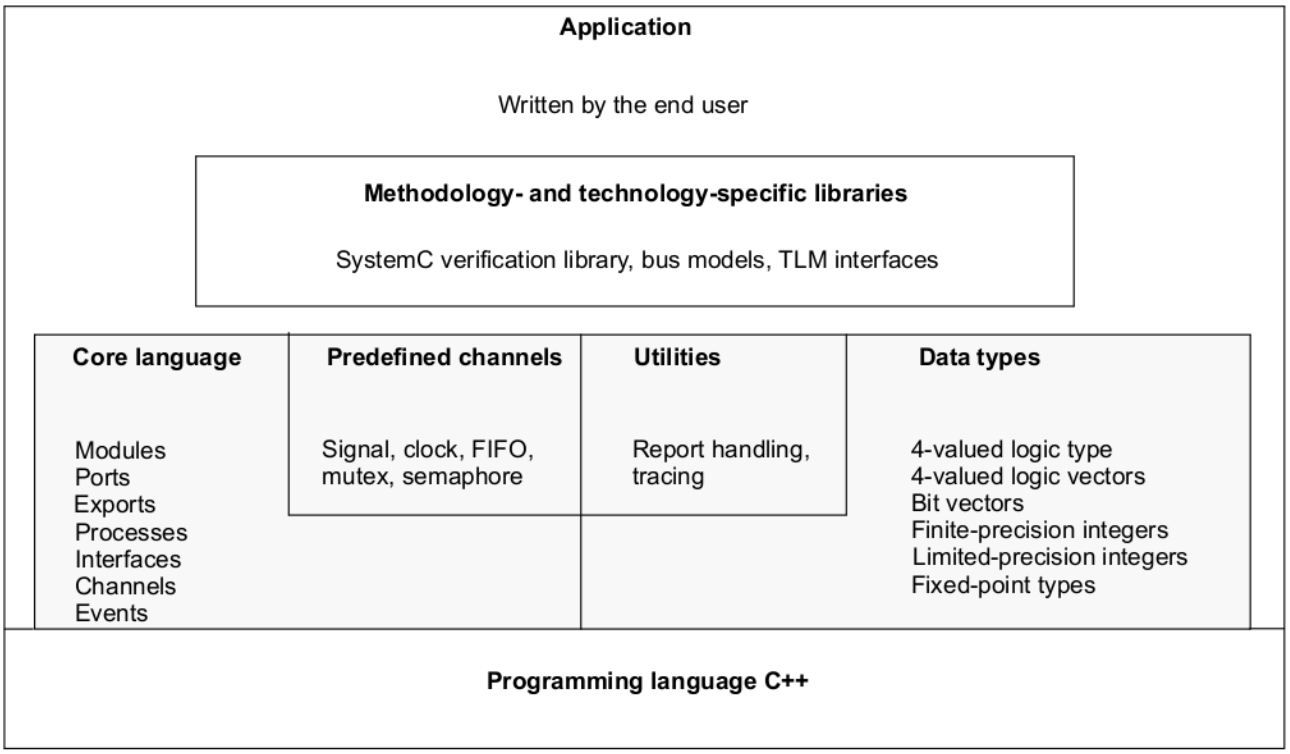


**Figure A-1—SystemC language architecture**

The diagram is referenced for “IEEE Standard of Standard SystemC Language Reference Manual.

In these labs we will try to implement small examples to discuss various features. All the suggestions, coding guidelines discussed in these labs/documents are solely my point of view. These should not be considered has industry standard nor can be quoted has reference. I will try my best to avoid mistakes but some of them may slip. Please do double check. Suggestions, bug fixes and pointing out issues are welcomed.

Prerequisites are prior HW implementation knowledge (design or verification), basic understanding of computer architecture and C++ programming language. Good knowledge of C++ language is must prerequisite to understand SystemC and follow these labs.  All the examples will use modern C++ features (C++11, C++14 and C++17).

My preferences are to avoid using Macros (SC_MODULE, etc), prefer C++ primitive data types over SystemC data types and to implement design at much higher levels of abstraction than signal level (sc_in, sc_out, etc). In general, were ever possible, I tend to use core C++ language features rather than equivalent SystemC features.

# Lab0 Desciption

In this lab we will mainly focus on installing SystemC, building and running a “Hello World!” program. The design has a clock, reset as inputs and a 4-bit counter as output. Counter is incremented by 1 on every positive edge of clock. On negative edge of reset, counter resets to 0 and holds on to the value until reset is released. Design must display “Hello World!” message when ever counter reaches max value and wraps back to 0. We will discuss about various concepts of SystemC program that are used to build SystemC program.

# Installing SystemC

Below installation steps work for Linux Desktop. For various other operating systems refer to SystemC documentation. Also, for Linux it is recommended to refer SystemC installation guidelines on Accellera website.

Steps to install SystemC (version: 2.3.3) on linux machines

- Download SystemC from https://accellera.org/downloads/standards/systemc
- Extract the tar file and cd to systemc-2.3.3
- Using CMake build scripts is easier.
- Below are the steps I followed but please refer to cmake/INSTALL_USING_CMAKE file for details
- For bash shell: export SYSTEMC_HOME=<installation path>
- For csh shell: setenv SYSTEMC_HOME <installation path>

For RELEASE version of shared library

- mkdir build
- cd build
- cmake DCMAKE_BUILD_TYPE=Release -DBUILD_SOURCE_DOCUMENTATION=ON -DCMAKE_CXX_STANDARD=14 -DCMAKE_VERBOSE_MAKEFILE=ON -DINSTALL_TO_LIB_BUILD_TYPE_DIR=ON ..
- make -j
- make -j check
- make -j install

If above cmake command fails dueto missing doxygen installation, Then either install it or get rid of -DBUILD_SOURCE_DOCUMENTATION=ON

For DEBUG version of shared library

- cmake DCMAKE_BUILD_TYPE=Release -DBUILD_SOURCE_DOCUMENTATION=ON -DCMAKE_CXX_STANDARD=14 -DCMAKE_VERBOSE_MAKEFILE=ON -DINSTALL_TO_LIB_BUILD_TYPE_DIR=ON ..
- make -j
- make -j check
- make -j install

Either you can add systemC shared library path to LD_LIBRARY_PATH so that linker is able to locate the shared library or add it to gcc commandline with switch "rpath". It adds an entry to the binary and hints the loader to search additional path. I followed the second approach, please refer to makefile for exact command.

# Environment Settings

- Operating System: Distributor ID: Ubuntu, Description: Pop!_OS 18.04 LTS
- Tested GCC versions: (GCC) 9.1.0, g++ (Ubuntu 7.4.0-1ubuntu1~18.04.1) 7.4.0

# SystemC Kernel

Execution of SystemC application consists of elaboration followed by simulation. In elaboration phase, module hierarchy is created. All the Sc_modules are created starting from top, constructors of all classes inherited from Sc_module. Hence it is important to invoke constructors of nested Sc_modules in the constructor of higher level Sc_module. Similarly, all ports, primitive channels and process are created and ports and exports must be bounded in elaboration phase. All the hierarchy of the application is built in elaboration phase and remains constant in entire simulation phase.

In Simulation phase, execution of scheduler is initiated which in-turn may execute processes within the application. The primary purpose of the scheduler is to trigger or resume the execution of the processes that the user supplies as part of the application. The scheduler is event-driven, meaning that processes are executed in response to the occurrence of events. Events occur (are notified) at precise points in simulation time. Events are represented by objects of the class sc_event, and by this class alone. Simulation time is an integer quantity. Simulation time is initialized to zero at the start of simulation and increases monotonically during simulation. Scheduler can execute a spawned process. The static sensitivity of an unspawned process instance is fixed during elaboration. The static sensitivity of a spawned process instance is fixed when the function sc_spawn is called. The dynamic sensitivity of a process instance may vary over time under the control of the process itself. A process instance is said to be sensitive to an event if the event has been added to the static sensitivity or dynamic sensitivity of the process instance. A time-out occurs when a given time interval has elapsed.

Phases of elaboration and simulation run in following phases.

- Elaboration—Construction of the module hierarchy.
    - All modules constructors are invoked (Top to Bottom).
- Elaboration—Callbacks to function before_end_of_elaboration.
- Elaboration—Callbacks to function end_of_elaboration.
- Simulation—Callbacks to function start_of_simulation.
- Simulation—Initialization phase.
- Simulation—Evaluation, update, delta notification, and timed notification phases (repeated).
- Simulation—Callbacks to function end_of_simulation.
- Simulation—Destruction of the module hierarchy.
    - All modules destructors are invoked (Bottom to Top)

# Simulation Execution

The function ‘main(int argc, char* argv[])’ is the entry point for the C++program. Common usage is that the function ‘main()’ is implemented by the SystemC kernel. Then the kernel invokes function ‘sc_main(int argc, char* argv[])’ implemented by the application (program) implemented in global namespace. This is the only entry point to the application. If the application is unable to find this

signature, then linker will fail with below error message indicating that the linker is unable to find the implementation of function ‘sc_main()’.

```
libsystemc.so: undefined reference to `sc_main'
collect2: error: ld returned 1 exit status
```

Elaboration phase consists of the execution of the sc_main function from the start of sc_main to the point immediately before the first call to the function ‘sc_start()’ implemented in sc_core namespace. Simulation phase starts with first invocation of ‘sc_start()’. When invoked for the first time, sc_start() shall start the scheduler. When function ‘sc_start()’ is invoked without any arguments, the scheduler shall run until there is no remaining activity, unless otherwise interrupted. In other words, except when sc_stop() or sc_pause() have been called or an exception has been thrown.

As mentioned above, below callbacks are invoked from kernel at respective phases.

- virtual void before_end_of_elaboration();
- virtual void end_of_elaboration();
- virtual void start_of_simulation();
- virtual void end_of_simulation();

# sc_module

Class sc_module is the base class for all modules implemented in the application. Refer to reference manual for sdditional details.

# sc_module_name

Class sc_module_name acts as a container for the string name of a module and provides the mechanism for building the hierarchical names of instances in the module hierarchy during elaboration. Refer to reference manual for additional details.

# sc_spawn and sc_spawn_options

Function ‘sc_spawn()’ is used to create a static of dynamic spawned process instance. Function ‘sc_spawn()’ may be called during elaboration or from a static, dynamic, spawned, or unspawned process during simulation.

- If function ‘sc_spawn()’ is called during elaboration, in which case the spawned process is a child of the module instance.
- If sc_spawn() is called during simulation, spawned process is the child of the process that called the function sc_spawn().
- The process or module from which sc_spawn is called is the parent of the spawned process. Thus a set of dynamic process instances may have a hierarchical relationship, similar to the module hierarchy, which will be reflected in the hierarchical names of the process instances.

Class “sc_spawn_options” is used to create an object that is passed as an argument to function “sc_spawn()” when creating a spawned process instance. Sensitivity list of the spawned process is set using this class member function.

Few Member functions of class “sc_spawn_options”

- void spawn_method()
  - Shall set a property of spawn options to indicate spawned process shall be a method process. Default is thread process
- void dont_initialize()
  - Member function shall set a property of the spawn options to indicate that the spawned process instance shall not be made runnable during the initialization phase or when it is created.
- void set_sensitivity(sc_event const* )
  - Member function ‘set_sensitivity()’ shall set a property of the spawn options to add the object passed as an argument to set_sensitivity to the static sensitivity of the spawned process. Calls to ‘set_sensitivity()’ are cumulative: each call to set_sensitivity extends the static sensitivity as set in the spawn options.

Two function overloads are provided for ‘sc_spawn()’.

```
template <typename T>
sc_process_handle sc_spawn(
   T object ,
   const char* name_p = 0 ,
   const sc_spawn_options* opt_p = 0
);

template <typename T>
sc_process_handle sc_spawn(
   typename T::result_type* r_p ,
   T object ,
   const char* name_p = 0 ,
   const sc_spawn_options* opt_p = 0
);
```

The second overload is used if return value of the function object invoked must be stored. Make sure the location where return value will be stored is valid when the function object is invoked.

## Usage Guidelines

- I personally prefer using sc_spwan instead of using SC_THREAD, SC_METHOD or SC_CTHREAD macros. Explicitly invoking ‘sc_spawn()’ gives you more control over the child process and events that trigger the process. Macros are considered bad coding style as per C++ coding guidelines.

# Program Description

Hello World module implements a 4-bit counter that prints “Hello World” whenever counter wraps back to reset value.

- Steps to execute program
  - make -f Makefile
- Default verbosity level is 0. Value can be changed at run time.
  - ./sim 2

# References

- C++ references
  - The C++ Programming Language 4th Edition
  - Effective Modern C++
  - https://github.com/isocpp/CppCoreGuidelines/blob/master/CppCoreGuidelines.md
- SystemC references
  - IEEE Standard for Standard SystemC Lanaguage Referece
    - https://accellera.org/downloads/standards/systemc
  -