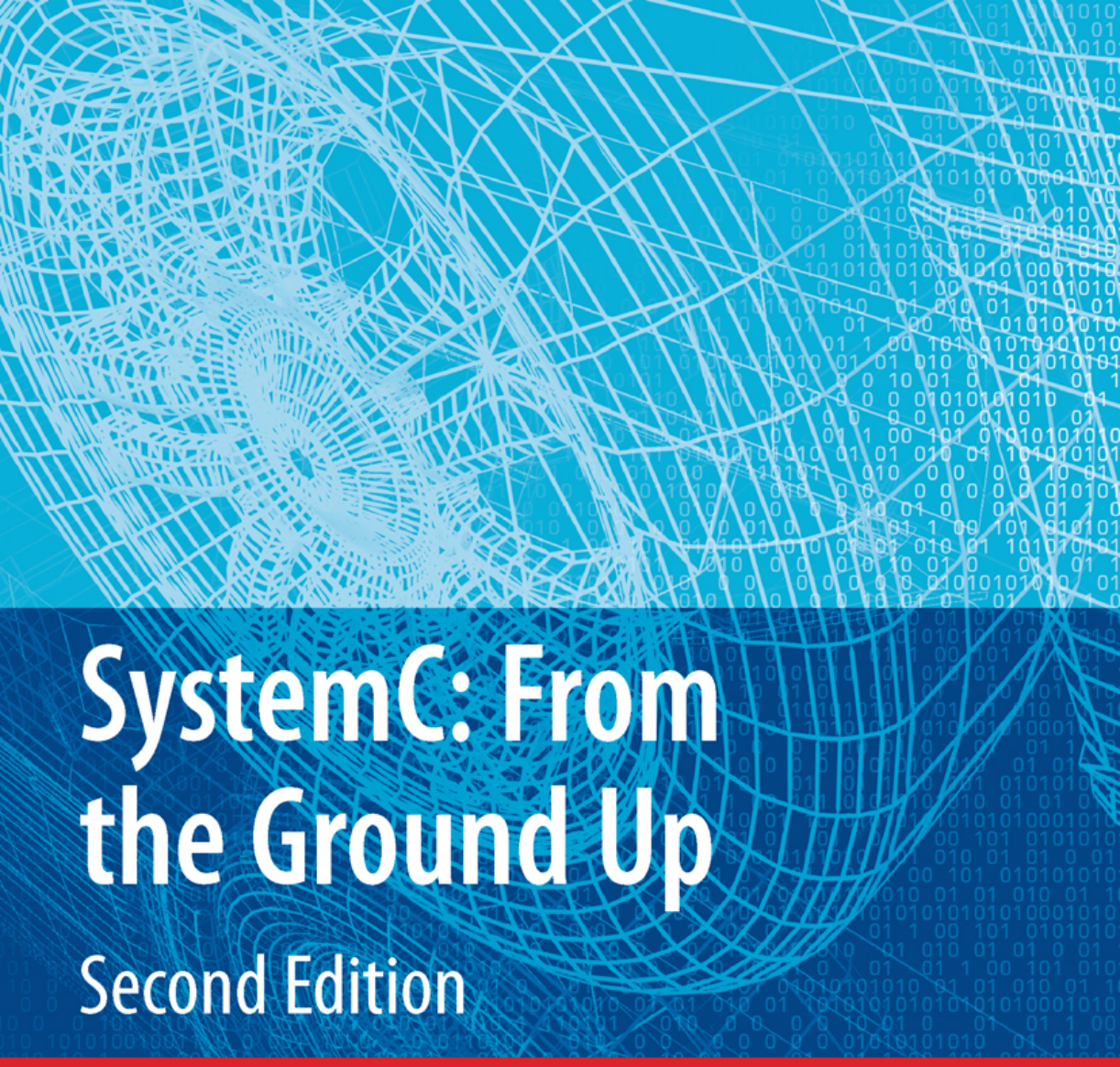


SystemC: From the Ground Up

Second Edition

David C. Black
Jack Donovan
Bill Bunton
Anna Keist

Springer

SystemC: From the Ground Up

David C. Black · Jack Donovan
Bill Bunton · Anna Keist

SystemC: From the Ground Up

Springer

David C. Black
XtremeEDA
Austin, TX 78749
USA
dcblack@eslx.com

Jack Donovan
HighIP Design Company
Round Rock, TX 78764
USA
jack@donovanweb.org

Bill Bunton
LSI Corporation
Austin, TX 78749
USA

Anna Keist
XtremeEDA
Austin, TX 78749
USA

ISBN 978-0-387-69957-8 e-ISBN 978-0-387-69958-5
DOI 10.1007/978-0-387-69958-5
Springer New York Dordrecht Heidelberg London

Library of Congress Control Number: 2009933997

Printed on acid-free paper

Springer is part of Springer Science+Business Media (www.springer.com)

This book is dedicated to our spouses Pamela Black, Carol Donovan, Evelyn Bunton, and Rob Keist and to our children Christina, Loretta, & William Black, Chris, Karen, Jenny, & Becca Donovan, John Williams & Sylvia Headrick, Alex, Madeline, and Michael Keist

2nd Edition Preface

Why the 2nd Edition?

The reader (or prospective buyer of this book) might ask about the need for a second edition. The first edition was highly successful and progressed to a second and third life after being translated to Japanese and Korean.

There are three over-arching reasons for the second edition:

- A fast changing technical landscape
- Incorporation of additional topic suggestions from readers
- Fixing of errors and improvement of confusing text segments and chapters

To address the shifting technical landscape, we have significantly updated the chapters addressing Electronic System-Level design to reflect the refinements of ESL methodology thinking in the industry. Although this is not a complete discussion of ESL, it is an overview of the industry as currently viewed by the authors.

We have added a chapter on TLM, a standard that will enable interoperability of models and a model marketplace. Although this chapter discusses TLM 1.0, we think it imparts to the reader a basic understanding of TLM. Those of you who follow the industry will note that this is not TLM 2.0. This new standard was still emerging during the writing of this edition. But not to worry! Purchasers of this edition can download an additional chapter on TLM 2.0 when it becomes available within the next six months at www.scftgu.com.

Although SystemC is now IEEE 1666 it is not immune from the shifting technical landscape, so the authors have included material on some proposed extensions to the SystemC standard related to process control.

Readers have suggested several additional topics over the last several years and we have tried to address these with an additional chapter on the SystemC Verification (SCV) Library and an appendix on C++ fundamentals.

The chapter on the SCV library is a high level introduction and points the reader to additional resources. The authors have found that many organizations have started using the SCV library after becoming familiar with SystemC and ESL methodologies. For those readers, we have added this chapter.

The authors received several suggestions asking us to add examples and comparisons to HDL languages like Verilog and VHDL. The authors have respectfully declined, as we feel this actually impedes the reader from seeing the intended uses of SystemC. After exploring these suggestions, we have found that these readers were not entirely comfortable with C++, and because C++ is fundamental to an understanding of SystemC, this edition includes a special appendix that attempts to highlight those aspects of C++ that are important prerequisites, which is most of the language.

Writing a book of this type is very humbling, as most who have journeyed on similar endeavors will confirm. Despite our best efforts at eliminating errors from the first edition, the errata list had grown quite long. We have also received feedback that certain portions of the book were "confusing" or "not clear". After reviewing many of these sections, we had to ask: What were we thinking? (a question asked by many developers when they revisit their "code" after several years)

In some cases we were obviously "not thinking", so several chapters and sections of chapters have been significantly updated or completely rewritten. The topic of concurrency proved a more challenging concept to explain than the authors first thought. This edition effectively rewrote the chapters and sections dealing with the concept of concurrency.

The authors have been quite gratified at the acceptance of the first edition and the rapid adoption of SystemC. We hope we have played at least a small part in the resulting changes to our community. We wish you good luck with SystemC and your contributions to our community.

Jack Donovan, David Black, Bill Bunton, Anne Keist
4authors@scftgu.com

Preface

Jack Donovan, David Black, Bill Bunton, and Anne Keist

Why This Book

The first question any reader should ask is "Why this book?" We decided to write this book after learning SystemC using minimal documentation to help us through the quest to deeper understanding of SystemC. After teaching several SystemC classes, we were even more convinced that an introductory book focused on the SystemC language was needed. We decided to contribute such a book.

This book is about SystemC. It focuses on enabling the reader to master the language. The reader will be introduced to the syntax and structure of the language, and the reader will also learn a few techniques for use of SystemC as a tool to shorten the development cycle of large system designs.

We allude to system design techniques and methods by way of examples throughout the book. Several books that discuss system-level design methodology are available, and we believe that SystemC is ideally suited to implement many of these methods. After reading this resource, the reader should not only be adept at using SystemC constructs, but also should have an appreciation of how the constructs can be used to create high performance simulation models.

We believe there is enough necessary information about SystemC to learn the language that a stand-alone book is justified. We hope you agree. We also believe that there is enough material for a second book that focuses on using SystemC to implement these system-level design methods. With reader encouragement, the authors have started on a second book that delves deeper into the application of the language.

Prerequisites for This Book

As with every technical book, the authors must write the content assuming a basic level of understanding; this assumption avoids repeating most of an engineering undergraduate curriculum. For this book, we assumed that the reader has a working knowledge of C++ and minimal knowledge of hardware design.

For C++ skills, we do not assume that the reader is a "wizard". Instead, we assumed that you have a good knowledge of the syntax, the object-oriented

features, and the methods of using C++. The authors have found that this level of C++ knowledge is universal to current or recent graduates with a computer science or engineering degree from a four-year university.

Interestingly, the authors have also found that this level of knowledge is lacking for most ASIC designers with 10 or more years of experience. For those readers, assimilating this content will be quite a challenge but not an impossible one.

As an aid to understanding the C++ basics, this edition includes an appendix on C++. Those who have been exposed to C++ in the past are encouraged to quickly review this appendix. For a few novices, this appendix may also work as a quick introduction to the topics, but it is unlikely to be completely sufficient.

For readers who are C++ novices or for those who may be rusty, we recommend finding a good C++ class at a community college, taking advantage of many of the online tutorials, or finding a good consulting group offering an Intro to C++ class. For a list of sources, see Appendix A. We find (from our own experience) that those who have learned several procedural languages (like FORTRAN or PL/I) greatly underestimate the difficulty of learning a modern object-oriented language.

To understand the examples completely, the reader will need minimal understanding of digital electronics.

Book Conventions

Throughout this book, we include many syntax and code examples. We've chosen to use an example-based approach because most engineers have an easier time understanding examples rather than strict Backus-Naur form[1] (BNF) syntax or precise library declarations. Syntax examples illustrate the code in the manner it may be seen in real use with placeholders for user-specified items. For more complete library information, we refer the reader to the SystemC Language Reference IEEE 1666-2005, which you can obtain for free via www.systemc.org.

Code will appear in monotype Courier font. Keywords for both C/C++ and SystemC will be shown in Courier **bold**. User-selectable syntax items are in *italics* for emphasis. Repeated items may be indicated with an ellipsis (…) or subscripts. The following is an example:

```
wait(name.posedge_event()|event_i…);
if (name.posedge()) { //previous delta-cycle
  //ACTIONS…
```

Fig. 1 Example of sample code

[1]John Backus and Peter Naur first introduced BNF as a formal notation to describe the syntax of a given language. Naur, P. (1960, May). Revised report on the algorithmic language ALGOL 60. *Communications of the ACM*, *3*(5), 299-314.

When referring to a class within the text it will appear as **class_name** or **sc_class_name**. When referring to a templated class within the text, it will appear as **template_class_name<*T*>**. When referring to a member function or method from the text it will appear as **member_name(***args***)** or **sc_member_name**(*args*). Occasionally, we will refer to a member function or method without the arguments. It will appear in the text as **member_name**() or **sc_member_name**().

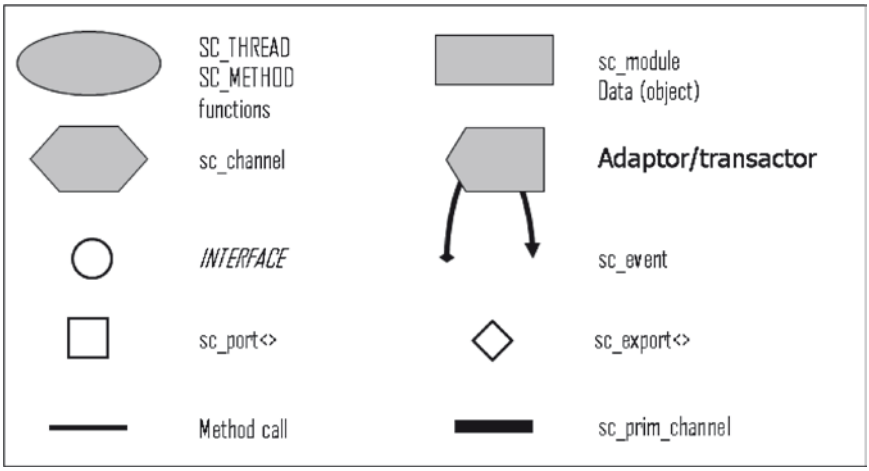


Fig. 2 Standard graphical notations

In addition, we have adopted standard graphical notations as shown in Fig 2. The terminology will be presented as the book progresses. Readers of the first edition will note that we changed the depiction of an **sc_export** from a dotted circle to a diamond. This change was the result of comments that the dotted circle was too hard to make out in some cases. We also removed arrows since in most cases, the meaning is not always clear[2].

SystemC uses a naming convention where most SystemC-specific identifiers are prefixed with **sc_** or **SC_**. This convention is reserved for the SystemC library, and you should not use it in end-user code (your code).

About the Examples

To introduce the syntax of SystemC and demonstrate its usage, we have filled this book with examples. Most examples are not real-world examples. Real examples become too cluttered too fast. The goal of these examples is to communicate

[2]Does an arrow convey calling direction (i.e., C++ function call) or direction of data flow? Since many interfaces contain a mixture of calls, some input and some output, showing data flow direction is not very useful.

concepts clearly; we hope that the reader can extend them into the real world. For the most part, we used a common theme of an automobile for the examples.

By convention, we show syntax examples stylistically as if SystemC is a special language, which it is not. We hope that this presentation style will help you apply SystemC on your first coding efforts. If you are looking for the C++ declarations, please browse the Language Reference Manual (LRM) or look directly into the SystemC Open SystemC Initiative reference source code (www.systemc.org).

It should also be noted that due to space limitations and to reduce clutter, we have omitted showing standard includes (i.e., **#include**) and standard namespace prefixes in most of the examples. You may assume something such as the following is implied in most of the examples:

```
#include <iostream>
#include <systemc>
#include <scv.h>
using namespace std;
using namespace sc_core;
using namespace sc_dt;
```

Fig. 3 Assumed code in examples

Please note that it is considered bad C++ form to include the standard namespace in header files (i.e., do not include "**using namespace** std;" in a header). We believe making the examples clear and brief warrants ignoring this common wisdom.

How to Use This Book

The authors designed this book primarily for the student or engineer new to SystemC. This book's structure is best appreciated by reading sequentially from beginning to end. A reader already familiar with SystemC will find this content to be a great reference.

Chapters 1 through 3 provide introductions and overviews of the language and its usage. Methodology is briefly discussed.

For the student or engineer new to SystemC, the authors present the language from the bottom up and explain the topics from a context of C++ with ties to hardware concepts. We provide exercises at the end of Chapters 4 through 16 to reinforce the concepts presented in the text. Chapters 16 through 18 strengthen the basic language concepts. In these chapters, readers will find discussions of areas to watch and understand when designing, writing, or using SystemC in a production environment.

For the student or engineer already familiar with SystemC, Chapters 4 through 13 will provide some interesting background and insights into the language. You can either skip or read these early chapters lightly to pick up more nuances of the language. The content here is not meant to be a complete description of the language.

For a thorough description, the reader is referred to the SystemC LRM. Chapters 14 through 18 provide intermediate to advanced material.

For the instructor, this book may provide part of an advanced undergraduate class on simulation or augment a class on systems design.

In most of the examples presented in the book, the authors show code fragments only so as to emphasize the points being made. Examples are designed to illustrate specific concepts, and as such are toy examples to simplify learning. Complete source code for all examples and exercises is available for download from www.scftgu.com as a compressed archive. You will need this book to make best use of these files.

SystemC Background

SystemC is a system design language based on C++. As with most design languages, SystemC has evolved. Many times a brief overview of the history of language will help answer the question "Why do it that way?" We include a brief history of SystemC and the Open SystemC Initiative to help answer these questions.

The Evolution of SystemC

SystemC is the result of the evolution of many concepts in the research and commercial EDA communities. Many research groups and EDA companies have contributed to the language. A timeline of SystemC is included in Table 1.

SystemC started out as a very restrictive cycle-based simulator and "yet another" RTL language. The language has evolved (and is evolving) to a true system design language that includes both software and hardware concepts. Although SystemC

Table 1 Timeline of development of SystemC

Date	Version	Notes
Sept 1999	0.9	First version; cycle-based
Feb 2000	0.91	Bug fixes
Mar2000	1.0	Widely accessed major release
Oct 2000	1.0.1	Bug fixes
Feb 2001	1.2	Various improvements
Aug 2002	2.0	Add channels & events; cleaner syntax
Apr 2002	2.0.1	Bug fixes; widely used
June 2003	2.0.1	LRM in review
Spring 2004	2.1	LRM submitted for IEEE standard
Dec 2005	2.1v1	IEEE 1666-2005 ratified
July 2006	2.2	Bug fixes to more closely implement IEEE 1666-2005

does not specifically support analog hardware or mechanical components, there is no reason why these aspects of a system cannot be modeled with SystemC constructs or with co-simulation techniques.

Open SystemC Initiative

Several of the organizations that contributed heavily to the language development efforts realized very early that any new design language must be open to the community and not be proprietary. As a result, the Open SystemC Initiative (OSCI) was formed in 1999. OSCI was formed to:

- Evolve and standardize the language
- Facilitate communication among the language users and tool vendors
- Enable adoption
- Provide the mechanics for open source development and maintenance

The SystemC Verification Library

As you will learn while reading this book, SystemC consists of the language and potential methodology-specific libraries. The authors view the SystemC Verification (SCV) library as the most significant of these libraries. This library adds support for modern high-level verification language concepts such as constrained randomization, introspection, and transaction recording. The first release of the SCV library occurred in December of 2003 after over a year of Beta testing. This edition includes a chapter devoted to the SCV from a syntactic point of view.

Current Activities with OSCI

At present, the OSCI has completed the SystemC LRM that has been ratified as a standard (IEEE 1666-2005) by the Institute of Electrical and Electronics Engineers (IEEE). Additionally, sub-committees are studying such topics as synthesis subsets and formalizing terminology concerning levels of abstraction for transaction-level modeling (TLM). This edition includes a chapter devoted to TLM and current activities.

Acknowledgments

- Our inspiration was provided by:
  Mike Baird, President of Willamette HDL, who provided
  the basic knowledge to get us started on our SystemC journey.
- Technical information was provided by:
  IEEE-1666-2005 Standard
  OSCI Proof-of-Concept Library associated information on systemc.org
  Andy Goodrich of Forte Design Systems, who provided technical insights.
- Our reviewers provided feedback that helped keep us on track:
  Chris Donovan, Cisco Systems Incorporated
  Ronald Goodstein, First Shot Logic Simulation and Design
  Mark Johnson, Rockwell-Collins Corporation
  Rob Keist, Freescale Corporation
  Miriam Leeser, Northeastern University
  Chris Macionski, Synopsys Inc.
  Nasib Naser, Synopsys Inc.
  Suhas Pai, Qualcomm Incorporated
  Charles Wilson, XtremeEDA Corporation
  Claude Cloutier, XtremeEDA Corporation
  David Jones, XtremeEDA Corporation
- The team who translated the first edition of the book into Japanese and asked us many thought provoking questions that have hopefully been answered in this edition:
  Masamichi Kawarabayashi (Kaba), NEC Electronics Corporation
  Sanae Nakanishi, NEC Electronics Corporation
  Takashi Hasegawa, Fujitsu Corporation
  Masaru Kakimoto, Sony Corporation
- The translator of the Korean version of the first edition who caught many detailed errors. We hope that we have corrected them all in this edition:
  Goodkook, Anslab Corporation
- Our Graphic Artist
  Felix Castillo
- Our Technical Editors helped us say what we meant to say:
  Kyle Smith, Smith Editing
  Richard Whitfield

- Our Readers from the First Edition:
  David Jones, Junyee Lee, Soheil Samii, Kazunari Sekigawa, Ando Ki, Jeff Clark, Russell Fredrickson, Mike Campin, Marek Tomczyk, Luke Lee, Adamoin Harald Devos, Massimo Iaculo, and many others who reported errata in the first edition.

Most important of all, we acknowledge our spouses, Pamela Black, Carol Donovan, Rob Keist, and Evelyn Bunton. These wonderful life partners (despite misgivings about four wild-eyed engineers) supported us cheerfully as we spent many hours researching, typing, discussing, and talking to ourselves while pacing around the house as we struggled to write this book over the past year.

We also acknowledge our parents who gave us the foundation for both our family and professional life.

Contents

Chapter 1
Why SYSTEMC: ESL and TLM

1.1 Introduction

The goal of this chapter is to explain why it is important for you to learn SystemC. If you already know why you are studying SystemC, then you can jump ahead to Chapter 2. If you are learning SystemC for a college course or because your boss says you must, then you may benefit from this chapter. If your boss doesn't know why you need to spend your time learning SystemC, then you may want to show your boss this chapter.

SystemC is a system design and modeling language. This language evolved to meet a system designer's requirements for designing and integrating today's complex electronic systems very quickly while assuring that the final system will meet performance expectations.

Typically, today's systems contain both application-specific hardware and software. Furthermore, the hardware and software are usually co-developed on a tight schedule with tight real-time performance constraints and stringent requirements for low power. Thorough functional (and architectural) verification is required to avoid expensive and sometimes catastrophic failures in the device. In some cases, these failures result in the demise of the company or organization designing the errant system. The prevailing name for this concurrent and multidisciplinary approach to the design of complex systems is electronic system-level design or ESL.

The drive for concurrent engineering through ESL has side effects that affect more than the design organizations of a company. ESL affects the basic business model of a company and how companies interact with their customers and with their suppliers.

ESL happens by modeling systems at higher levels of abstraction than traditional methods used in the past. Portions of the system model are subsequently iterated and refined, as needed. A set of techniques has evolved called Transaction-Level Modeling or TLM to aide with this task.

ESL and TLM impose a set of requirements on a language that is different than the requirements for hardware description languages (HDLs) or the requirements

D.C. Black et al., *SystemC: From the Ground Up*, Second Edition,
DOI 10.1007/978-0-387-69958-5_1, © Springer Science+Business Media, LLC 2010

for traditional software languages like C, C++[1], or Java. The authors believe that SystemC is uniquely positioned to meet these requirements.

We will discuss all these topics in more detail in the following sections.

1.2 ESL Overview

ESL techniques evolved in response to increasing design complexity and increasingly shortened design cycles in many industries. Systems, Integrated Circuits (ICs), and Field Programmable Gate Arrays (FPGAs) are becoming large. Many more multi-disciplinary trade-offs are required to optimize the hardware, the software, and the overall system performance.

1.2.1 Design Complexity

The primary driver for an ESL methodology is the same driver that drove the evolution of previous design methodologies: increasing design complexity.

Modern electronic systems consist of many subsystems and components. ESL focuses primarily on hardware, software, and algorithms at the architectural level. In modern systems, each of these disciplines has become more complex. Likewise, the interaction has become increasingly complex.

Interactions imply that trade-offs between the domains are becoming more important for meeting customer requirements. System development teams find themselves asking questions like:

- Should this function be implemented in hardware, software, or with a better algorithm?
- Does this function use less power in hardware or software?
- Do we have enough interconnect bandwidth for our algorithm?
- What is the minimum precision required for our algorithm to work?

These questions are not trivial and the list could go on and on. Systems are so complex, just deriving specifications from customer requirements has become a daunting task. Hence, this task brings the need for higher levels of abstraction and executable specifications or virtual system prototypes.

Figure 1.1 illustrates the complexity issues for just the hardware design in a large system-on-a-chip (SoC) design. The figure shows three sample designs from three generations: yesterday, today, and tomorrow. In reality, tomorrow's systems are being designed today. The bars for each generation imply the code complexity for four common levels of abstraction associated with system hardware design:

- Architecture
- Behavioral
- RTL
- Gates

[1] We will see later that SystemC is actually a C++ class library that "sits on top" of C++.

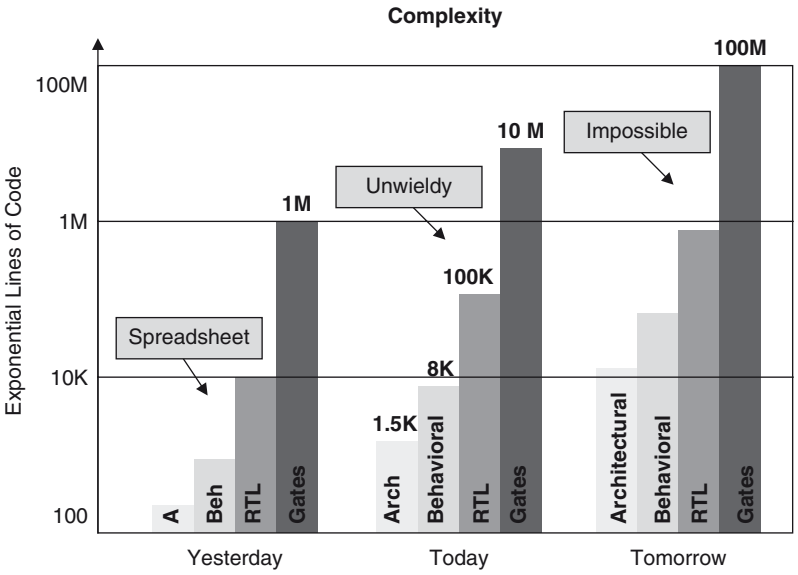


Fig. 1.1 Code complexity for four levels of abstraction

Today's integrated circuits often exceed 10 million gates, which conservatively translates to one hundred thousand lines of RTL code. Today's designs are practical because of the methodologies that apply RTL synthesis for automated generation of gates. Tomorrow's integrated circuits, which are being designed today, will exceed one hundred million gates. This size equates to roughly one million lines of RTL code, if written using today's methodologies.

Notice that Figure 1.1 considers only a single integrated circuit. It does not reflect the greater complexity of a system with several large chips (integrated circuits or FPGAs) and gigabytes of application software. Many stop-gap approaches are being applied, but the requirement for a fundamentally new approach is clear.

1.2.2 Shortened Design Cycle = Need For Concurrent Design

Anyone who has been part of a system design realizes that typical design cycles are experiencing more and more schedule pressure. Part of the reason for the drive for a shortened design cycle is the perceived need for a very fast product cycle in the marketplace. Anyone who has attempted to find a cell phone or a laptop "just like my last one", even just nine months after buying the "latest and greatest" model, will find themselves looking long and hard throughout the web for a replacement[2].

[2] This scenario describes a recent experience by one of the authors.

Many are under the misguided assumption that shorter design cycles imply reduced development expenses. If the scope of the new system is not reduced and the schedule is reduced, then additional resources are required. In this scenario, a shorter design cycle actually requires more resources (expenses). The project requires more communication between development groups (because of concurrent design), more tools, more people, and more of everything. ESL and TLM are an approach to reduce the cost of development through more efficient communication and through reduced rework.

1.2.2.1 Traditional System Design Approach

In the past, when many systems were a more manageable size, a system could be grasped by one person. This person was known by a variety of titles such as system architect, chief engineer, lead engineer, or project engineer. This guru may have been a software engineer, hardware engineer, or algorithm expert depending on the primary technology leveraged for the system. The complexity was such that this person could keep most or all of the details in his or her head. This technical leader was able to use spreadsheets and paper-based methods to communicate thoughts and concepts to the rest of the team.

The guru's background usually dictated his or her success in communicating requirements to each of the communities involved in the design of the system. The guru's past experiences also controlled the quality of the multi-disciplinary trade-offs such as hardware implementation versus software implementation versus algorithm improvements.

In most cases, these trade-offs resulted in conceptual disconnects among the three groups. For example, cellular telephone systems consist of very complex algorithms, software, and hardware. Teams working on them have traditionally leveraged more rigorous but still ad-hoc methods.

The ad-hoc methods usually consist of a software-based model. This model is sometimes called a system architectural model (SAM), written in C, Java, or a similar language. The SAM is a communication vehicle between algorithm, hardware, and software groups. The model can be used for algorithmic refinement or used as basis for deriving hardware and software subsystem specifications. The exact parameters modeled are specific to the system type and application, but the model is typically un-timed (more on this topic in the following section). Typically, each team then uses a different language to refine the design for their portion of the system. The teams leave behind the original multi-discipline system model and in many cases, any informal communication among the groups.

The traditional approach often resulted in each design group working serially with a series of paper specifications being tossed "over the wall" to the other organization. This approach also resulted in a fairly serial process that is many times described as a "waterfall schedule" or "transom engineering" by many program managers and is illustrated in Fig. 1.2.

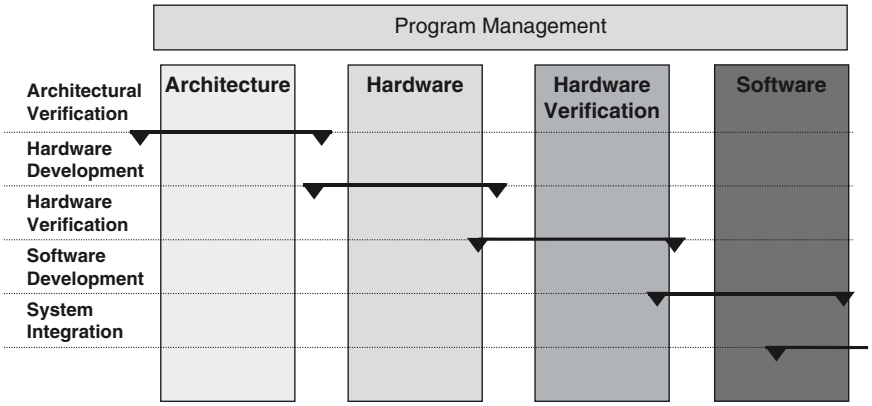


Fig. 1.2 The traditional approach of waterfall scheduling

To minimize the serialization of the design process, many techniques have been used to create some design concurrency. These techniques include processor development boards using a single-chip implementation of the processor. These implementations were used on the SoC or embedded system, FPGA prototypes of algorithms, and hardware emulation systems, just to name a few. These techniques were focused on early development of software, usually the last thing completed before a system is deployed.

The ESL approach uses these existing techniques. ESL also leverages a virtual system prototype or a TLM model of the system to enable all the system design disciplines to work in parallel. This virtual system prototype is the common specification among the groups. The resulting Gantt chart is illustrated next in Fig. 1.3.

Even though all of the electronic system design organizations will finish their tasks earlier, the primary reason for ESL is earlier development of software. Even getting a product to market a month earlier can mean tens of millions of dollars of business to a company.

Not using ESL methods will likely result in the under-design or over-design of the system. Both of these results are not good. Under-design is obviously not good. The product may be bug-free, but it doesn't necessarily meet the customer's requirements. The product may not operate fast enough, may not have long enough battery life, or just may not have the features required by the customer.

Over-design is not as obvious, but it is not good either. Over-design takes significantly more resources and time to achieve, and it adds a heavy cost to an organization. In addition, over-designed products usually are more complex, more expensive to manufacture, and are not as reliable.

The authors have significant anecdotal stories of under-design and over-design of systems. One company built an ASIC with multi-processors that made "timing closure" and paired those processors with software that made the "timing budget."

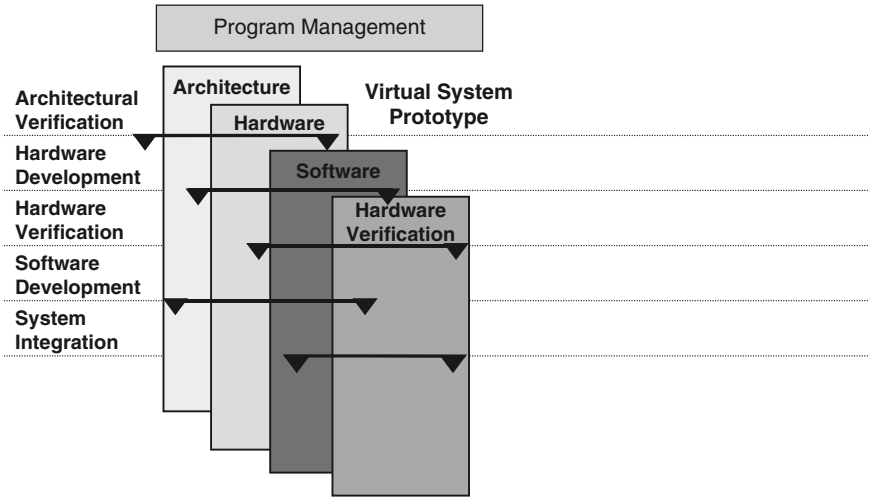


Fig. 1.3 The ESL approach of parallel schedule

Unfortunately, the ASIC didn't meet the customers requirements because of "on chip" bottlenecks. Another company related how a significant function on a chip caused weeks of schedule slip for design and verification. However, the function was later found not to be used by the software.

Things become even more interesting if a system, say a cell phone, are really a subsystem for the customer, who is a mobile phone and infrastructure provider. Now, the customer needs models very early in their design process and will be making system design trade-offs based on a model provided by the subsystem provider (previously system). In addition, the subsystem provider likely relies on third-party intellectual property. The subsystem cell phone supplier will then need a model of the intellectual property used in their subsystem very early in the development cycle to enable their design trade-offs. Customers up and down the value chain may now be making business decisions based on the type and quality of the model provided by their supplier. This hierarchy of reliance is fundamentally a different way of doing business.

The virtual system prototype may have different blocks (or components or subsystems) at different levels of abstraction for a particular task to be performed by one of the system disciplines. Initially, most of the system may be very abstract for software development until the software team is reasonably sure of the functionality. At this point, a more detailed model of the blocks that closely interact with the software can be introduced into the model.

The technique that allows this "mixing and matching" of blocks at different levels of abstraction is called Transaction-Level Modeling or TLM. We will discuss TLM in much greater detail in the following section.

1.3 Transaction-Level Modeling

TLM and the associated methodologies are the basic techniques that enable ESL and make it practical. To understand TLM, one must first have a terminology for describing abstraction levels. Secondly, one must understand the ways the models will probably be used (the use cases).

1.3.1 Abstraction Models

Several years ago, Professor Gajski from UC Irvine proposed a taxonomy for describing abstraction levels of models. The basic concept states that refinement of the interface or communication of a logical block can be refined independently of functionality or algorithm of the logical block [3]. We have significantly modified his concept and presented the result in the figure below.

- Un-Timed (UT)
- Loosely Timed (LT)
- Approximately Timed (AT)
- Register Transfer Logic (RTL)
- Pin and Cycle Accurate (PCA)

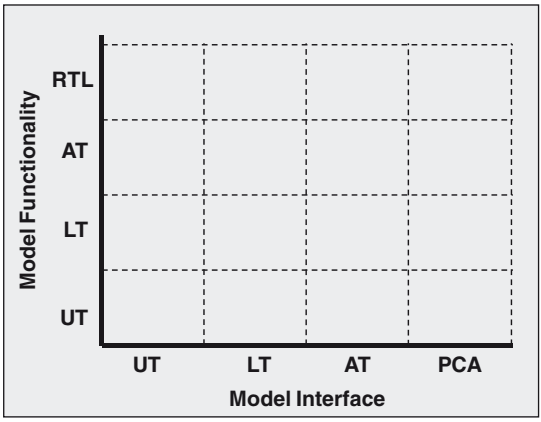


Fig. 1.4 Decoupling of abstraction refinement

[3] Gajski and L. Cai, "Transaction Level Modeling," First IEEE/ACM/IFIP International Conference on Hardware/Software Codesign and System Synthesis (CODES+ISSS 2003), Newport Beach, CA, October 1, 2003

The y-axis of this graph is the abstraction level of the model functionality. The x-axis is the abstraction level of the model or logical block interface or communication. The key thing to notice is that the functionality can be refined independent of the Interface and the reverse is also true. A logical block may be approximately timed (AT) for the model functionality. For instance, it may be a basic "c call" to a graphics algorithm with a "lump" delay for the algorithm, possibly the specified number of pipeline delays. The corresponding interface for the block may be pin and cycle accurate (PCA). The interface replicates all the "pin wiggles" of the communication between blocks. This type of block is many times referred to as a bus functional model for functional verification.

When we map some common ESL use cases to this graph, the model interface can be loosely timed (LT) or approximately timed (AT) with an occasional bus cycle accurate (BCA) representation. When communication with RTL represented blocks is required, a transactor will be used to bridge to PCA. Model functionality can be un-timed (UT), LT, or AT abstraction level. This modeling use cases are graphically represented in the following figure:

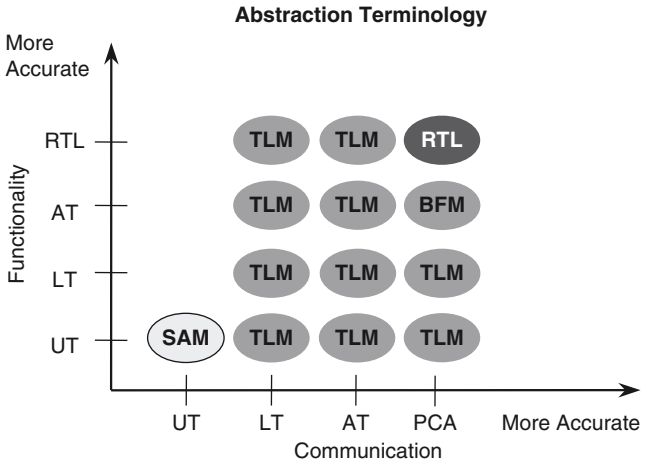


Fig. 1.5 TLM model mapping

For those from a software background, one can think of TLM style modeling as calling an application programming interface (API) for block-level model communication.

1.3.2 An Informal Look at TLM

In this section, we ease the reader into an understanding of TLM by presenting a less rigorous and more example-based discussion of TLM. We will assume a

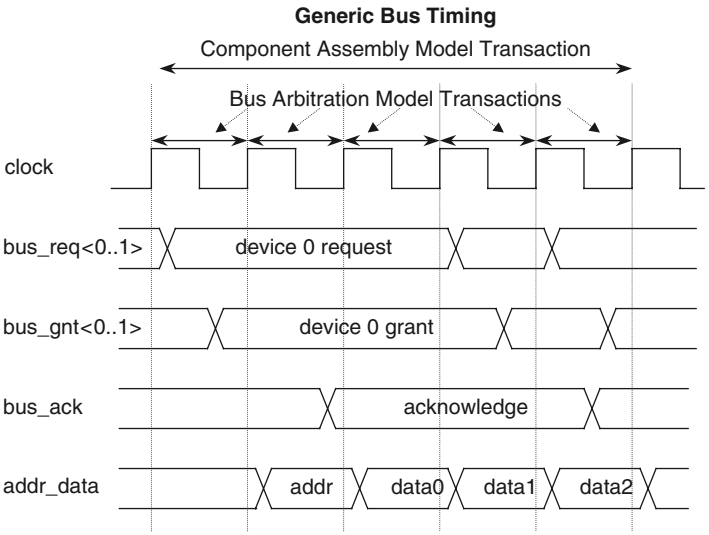


Fig. 1.6 Timing for generic bus

generic system containing a microprocessor, a few devices, and memory connected by a bus.

The timing diagram in Figure 1.6 illustrates one possible design outcome of a bus implementation. When first defining and modeling the system application, the exact bus-timing details do not affect the design decisions. All of the important information contained within the illustration is transferred between the bus devices as one event or transaction.

Further into the development cycle, the number of bus cycles may become important (to define bus cycle-time requirements, etc.). The information for each clock cycle of the bus is transferred as one transaction or event (bus-arbitration or cycle-accurate computation models).

When the bus specification is fully chosen and defined, the bus is modeled with a transaction or event per signal transition (bus functional or RTL model). Of course, as more details are added, more events occur and the speed of the model execution decreases.

In this diagram, the higher abstraction level model takes 1 "event" and the bus arbitration model takes approximately 5 "events." The RTL model takes roughly 75 "events" (the exact number depends on the number of transitioning signals and the exact simulator algorithm). This simple example illustrates the magnitude of computation required and why more system design teams are employing a TLM-based methodology.

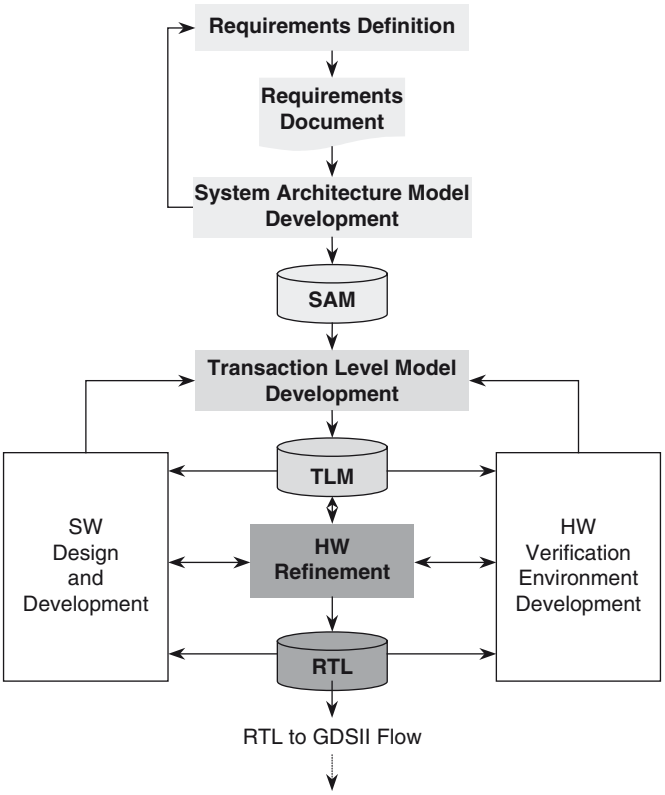


Fig. 1.7 TLM methodology

1.3.3 TLM Methodology

Now that we have discussed some of the TLM concepts, we can look more closely at a TLM-based methodology as illustrated in Figure 1.7.

In this methodology, we still start with the traditional methods used to capture the customer requirements: a paper-based Product Requirements Document (PRD). Sometimes, the product requirements are obtained directly from a customer. More likely, the requirements are captured through the research of a marketing group.

From the PRD, a high-level TLM model is developed. The TLM model development effort may cause changes or refinement of the PRD. The TLM model is usually written by an architect or architecture group. This model captures the product specification or system-critical parameters. In an algorithmic intensive system, the TLM model will be used to refine the system algorithms.

The TLM model is refined further as software design and development and hardware verification environment development progresses.

If the proper design language and techniques are used consistently throughout the flow, then the TLM can be reused and refined. The TLM has several use cases:

1. Architectural modeling
2. Algorithmic modeling
3. Virtual software development platform
4. Functional verification
5. Hardware refinement

We will further discuss these use cases in the next few sections.

At first, looked at from a particular development group's point of view, the development of the TLM appears to be a task with low return on investment (ROI). However, the TLM creates high value benefits including:

- Earlier software development, which is usually the schedule limiting task for deployment of a system
- Earlier and better hardware functional verification testbench
- A clear and unbroken path from customer requirements to detailed hardware and software specifications

After reading this book, you and your team should have the knowledge to implement TLM models quickly and effectively. The following section discusses in more detail the benefits that TLM modeling will bring to your organization.

1.3.3.1 Algorithmic Modeling

Algorithmic modeling is about the definition of application-specific algorithms such as those for cell phone receivers, encryption, video, and many forms of digital signal processing. Refining in a software environment provides a much friendlier environment for debug compared to RTL debug. In addition, a software model within the context of an ESL methodology provides the architect a platform for asking and answering a series of questions such as:

- How do we want to specify this for the hardware or software implementers?
- How sloppy can we get with our arithmetic and have the algorithm still work?
- Can I get away with a software-only solution?
- Can I use a lower clock rate for my processor (with the resulting power savings) if I implement this algorithm with a hardware co-processor?

After completing the design of the actual algorithm (getting it to work), the algorithm architect usually refines the algorithm from a floating-point implementation to a fixed-point (supported by SystemC) implementation (for hardware implementation). This architect also partitions the algorithm between hardware and software. An architect with broad experience will understand the different tradeoffs between software and hardware and will shape the algorithm implementation to be very implementable in the chosen space (for instance minimizing memory use for hardware implementation).

This activity is usually performed in SystemC, C/C++ or in MATLAB, or other commercial tools. This work is usually augmented with libraries for a particular application space such as a fixed-point library, a digital signal processing library, and others.

1.3.3.2 Architectural Modeling

Architectural modeling is about hardware and software partitioning, subsystem partitioning. It is also about bus performance analysis, and other initial refinements based on a power management concept. This modeling also includes existing intellectual property and designs as well as other product-based technical and management critical parameters and trade-offs. Some of the questions asked during this activity are:

- Is there enough performance in the bus architecture to implement our algorithm in hardware and realize performance gains?
- Is the bus performance adequate? Will we require multiple busses or a multi-tiered bus concept?
- Is the arbitration scheme sufficient?
- Is the estimated size and cost within our product goals?
- Would a different processor help performance? How would a lower clock rate or less capable processor affect performance?

This activity can be performed using SystemC or another language that supports modeling or simulation concurrency. In the past, some teams have used C or C++. These teams have been forced to develop a custom simulation kernel to support concurrency.

Many times for this activity, non-critical portions of the system are modeled using "traffic generators". These generators are based on an estimate of the bus traffic generated by the non-critical elements. As the model is refined, the blocks are refined to add the details required to start running software. There are several vendors of EDA tools that have offerings that can help accelerate this activity.

1.3.3.3 Virtual Software Development Platform

The Virtual software development platform allows very early development of system software. During this activity, the following questions are sometimes asked and answered:

- Does the hardware have the advertised features?
- Are the control and status registers supplied to the software sufficient? Does the software have the necessary control and observably to meet the customer requirements?
- Can the software meet the software timing budget with the current architecture?

SystemC is the language of choice for this set of activities. SystemC provides the necessary simulation features (simulation concurrency), ability to easily integrate with C and C++, and simulation performance.

The ability to easily integrate with C and C++ lets engineers wrap the instruction set simulators (ISS) within the model. In addition, early in the development process, C or C++ code can be wrapped and executed on the host modeling processor (versus the ISS). We call this technique direct execution. Even as the software is refined and OS calls are required, additional techniques can be used to enable direct execution.

1.3.3.4 Hardware Refinement

Depending on the nature of the system, the hardware refinement activities can be focused on the creation of an executable specification to unambiguously specify the hardware. This specification accelerates hardware development. Alternatively, activities can be focused on refinement for behavioral synthesis with even greater schedule and productivity improvements. Some of the questions asked during this set of activities are:

- What is the required gate count or size for this function?
- What is the expected latency and clock speed?
- What additional control and status is easy and inexpensive to provide to software?
- What is the estimated power consumption of this block?

Refining the specification for hardware designers eliminates potential overdesign. Many times during the development process, an RTL designer will ask an architect specific questions where the answer will significantly (or even slightly) affect performance. When the architect is asked about option A or option B, he or she will answer yes or both because of the absence of information, thus adding complexity. With a model that can be refined, the architect can model the affect of A and B and possibly come back with the answer that neither option helps. The refined model saves design time, verification time, and silicon area.

1.3.3.5 Functional and Architectural Verification

Traditional functional verification of hardware involves verifying the hardware to a functional specification. Unfortunately, as systems become more complex, the functional specs can be measured in feet (not inches) of paper. Obviously, any process requiring a natural language like English (or German, Chinese, French, etc.) is very susceptible to error (just check out the errata list for the first edition of this book).

We have added a new term (at least for us): architectural verification. The goal of this activity is to answer a few big questions:

- Does this system architecture meet the customer requirements?
- Does this system architecture work for all customer use cases?

We have purposely used the term, system architecture, because this involves integrating at least the lowest level software and a hardware model. The authors have a background in computer system design and have previously lived by the maxim "it isn't verified until the software runs." Since almost all of today's complex electronic systems have a significant component of "computer system," we have reverted to this maxim from our "youth" by emphasizing architectural verification.

Functional verification is also enhanced through the early availability of a target for testbench development and a "reference model" for newer functional verification methodologies such the Open Verification Methodology (OVM) and the methodology originally defined in the Verification Methodology Manual for SystemVerilog (VMM).

Verification teams are always complaining that they are not allowed to start until too late. Some of the reason for this trend is that management is reluctant to assign resources to project activities that will not show results or progress until much later in the project. The TLM modeling activity allows a target for early development. TLM has the added benefit of reuse of some portions of the "testbench" for the system-level TLM model and for later functional verification with the RTL.

Many of the newer verification methodologies such as OVM and VMM make heavy use of constrained-random test stimulus generation. These methodologies require the development of a "reference model" to check the results of the "randomized inputs." Given a well thought out methodology, the TLM model or a portion of the TLM model can be used by the functional testbench to check the RTL results.

1.4 A Language for ESL and TLM: SystemC

ESL and TLM impose a set of requirements upon a language. Some of these requirements are:

- Abstraction spans several levels
- Standard and open language
- Common skill set
- Proper simulation performance and features
- Productivity tool support
- Supports TLM concepts

We hope that by the end of this section we will have made the case that SystemC is the best modeling language available today, and that it is the language to launch the adoption of ESL and TLM modeling.

1.4.1 Language Comparisons and Levels of Abstraction

Strictly speaking, SystemC is not a language. SystemC is a class library within a well-established language, C++. SystemC is not a panacea that will solve every design productivity issue. However, when SystemC is coupled with the SystemC Verification Library, it does provide in one language many of the characteristics relevant to system design and modeling tasks that are missing or scattered among the other languages. In addition, SystemC provides a common language for software and hardware, C++.

Several languages have emerged to address the various aspects of system design. Although Java has proven its value, C/C++ is predominately used today for embedded system software (at least the lower software levels). The hardware description languages, VHDL and Verilog, are used for simulating and synthesizing digital circuits. Vera, *e*, and recently SystemVerilog are the languages of choice for functional verification of complex application-specific integrated circuits (ASICs). SystemVerilog is a new language that evolved from the Verilog language to address many hardware-oriented system design issues. MATLAB and several other tools and languages such as Signal Processing Workbench (SPW[4]) and CoCentric® System Studio[5] are widely used for capturing system requirements and developing signal processing algorithms.

Figure 1.8 highlights the application of these and other system design languages. Each language occasionally finds use outside its primary domain, as the overlaps in the figure illustrate.

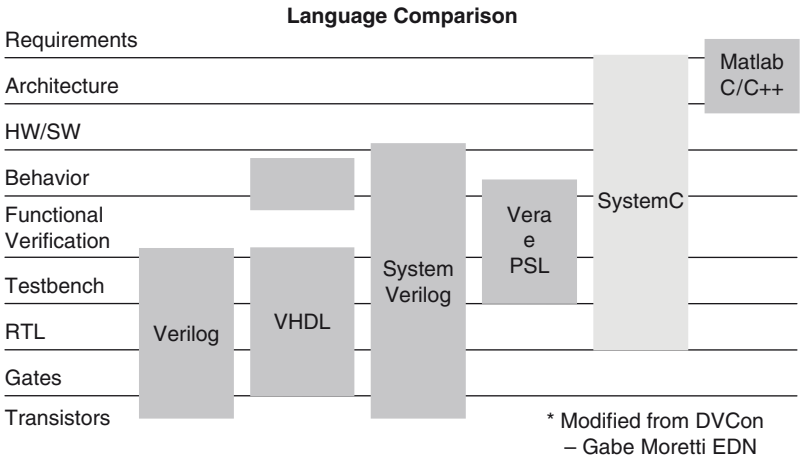


Fig. 1.8 Use of languages

[4]SPW is available from CoWare, Inc. (www.coware.com).

[5]CoCentric System Studio is available from Synopsys, Incorporated (www.synopsys.com).

1.4.2 SystemC: IEEE 1666

When a language is standard and open to the industry, then a wide array of benefits are available. These benefits include access to a set of commercial and freeware-based tools and support. SystemC was honored as a standard in December of 2006 when it was approved as IEEE 16666.

The Open SystemC Initiative (OSCI) provided much pre-standardization work to refine the SystemC language. Today, the OSCI continues this effort. The most notable effort is the standardization of a set of TLM interfaces.

OSCI also provides a proof of concept implementation of the IEEE 1666 standard. It is provided freely from the OSCI website at www.systemc.org.

1.4.3 Common Skill Set

SystemC is not yet a common skill set among all system engineers whether they be of a hardware or software orientation. However, SystemC is based on C++ and object-oriented techniques, which are a common skill set to all recent graduates of leading engineering schools. Many older engineers have also upgraded themselves by learning C++ or Java and the corresponding object-oriented techniques and thinking. These are the skills that enable a great SystemC modeler.

1.4.4 Proper Simulation Performance and Features

Obviously models need to execute swiftly. Not so obviously to some, the language needs to support concurrent simulation (we will talk about this extensively later in the book). The language must also support a series of additional productivity features. These features include the ability to manage the model complexity through hierarchy, debug features, and a list of other features discussed throughout this book.

Any model needs to execute relatively swiftly. A model is never fast enough, so what is fast enough. One can look at the requirements for model speed when used for common ESL purposes like early software development. To keep software developers from extreme frustration, the model needs to boot its operating system in just a couple of minutes or close to “real time”.

No modeling language can make up for inefficient modeling practices. The compiled nature of SystemC coupled with the availability of very good C++ compilers complemented with good modeling practices can produce a model that meets the speed requirements for all but the very largest systems.

As the reader may have guessed by now, developing a model is different than developing an application program. The difference is that in an ESL model, the

code modules need to appear as if they are executing in parallel or concurrently. When a digital system is running, then many, many computations are running at the same time or are running concurrently. SystemC provides a simulation kernel to give this illusion of concurrency. This feature, as well as several others, helps manage the complexity of the model. These features are discussed in detail in subsequent chapters.

1.4.5 Productivity Tool Support

Since SystemC is based on C++, there are a wealth of productivity tools and application-specific libraries that are available. In addition, SystemC is supported by a large and growing ecosystem of commercial suppliers.

Many of the popular C++ tools are freely available under various open source licenses. Some of these tools include integrated development environments (IDEs), performance analysis tools, and lint tools to illustrate just a few. There are many more tools available from leading software tool vendors for a small fee. These tools come with varying degrees of support.

Many of the groups using SystemC are large organizations that require ESL-specific productivity tools and technology as well as significant support. The three largest electronic design automation (EDA) vendors and an extensive list of smaller EDA companies now support SystemC. The last time we counted, the list was over 40 companies and growing.

The tools range from co-simulation with their HDL simulators to behavioral synthesis tools.

Possibly the biggest productivity boost is the availability of application libraries on the web in C or C++. The availability of graphic algorithms, DSP algorithms, and a plethora of other application libraries make the writing of initial models very easy. An extreme example is the use of an H.264 algorithm freely available on the web that is matched with about 20 lines of SystemC code to produce a graphics model in a matter of hours for an initial system model.

1.4.6 TLM Concept Support

Lastly, a language for ESL model development needs to support TLM concepts. It must support the easy substitution of one communication implementation with another without changing the interface to that implementation. As we progress through this book, we will show that C++ and the concept of interfaces implemented through a class of pure virtual functions coupled with a few coding styles enables TLM concepts efficiently.

1.5 Conclusion

We hope that we have motivated you to not only read our book, but also to study it and apply it to the examples provided at our web site. You will then be equipped to charge into the brave new world of ESL and TLM modeling and to bring about significant changes in your organization, company, and the industry.

Chapter 2
Overview of SystemC

The previous chapters gave a brief context for the application of SystemC. This chapter presents an overview of the SystemC language elements. Details are discussed in-depth in subsequent chapters.

Despite our best efforts not to use any part of the language before it is fully explained, some chapters may occasionally violate this goal due to the interrelated nature of SystemC. This chapter briefly discusses the major components of SystemC and their general usage and interactions as a way of giving context for the subsequent chapters.

The following diagram, Fig. 2.1, illustrates the major components of SystemC. As a form of roadmap, we have included a duplicate of this diagram at the beginning of each new chapter. Bolded type indicates the topics discussed within that chapter.

For the rest of this chapter, we will discuss all of the components within the figure that are outlined in bold; but first, we will discuss the mechanics of the SystemC development environment.

SystemC addresses the modeling of both hardware and software using C++. Since C++ already addresses most software concerns, it should come as no surprise that SystemC focuses primarily on non-software issues. The primary application area for SystemC is the design of electronic systems. However, SystemC also provides generic modeling constructs that can be applied to non-electronic systems[1]

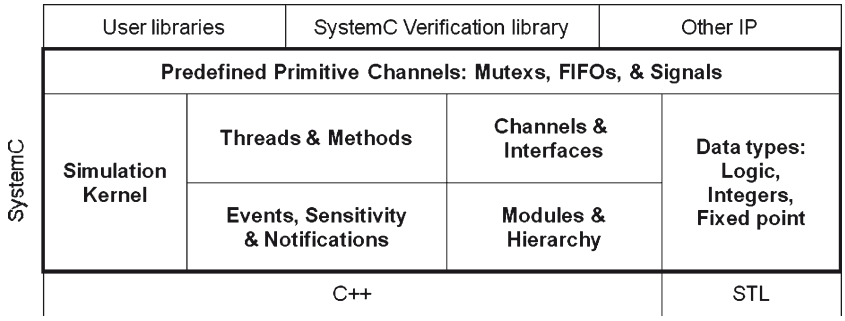


Fig. 2.1 SystemC language architecture

[1] For example, the book, *Microelectrofluidic Systems: Modeling and Simulation* by Tianhao Zhang et al., CRC Press, ISBN: 0849312760, describes applying SystemC to a non-electronic system.

D.C. Black et al., *SystemC: From the Ground Up*, Second Edition,
DOI 10.1007/978-0-387-69958-5_2, © Springer Science+Business Media, LLC 2010

2.1 C++ Mechanics for SystemC

We would like to start with the obligatory *Hello_SystemC* program; but first we will look at the mechanics of compiling and executing a SystemC program or model.

As stated before, SystemC is a C++ class library. Therefore, to compile and run a *Hello_SystemC* program, one must have a working C++ and SystemC environment.

The components of a SystemC environment include a:

- SystemC-supported platform
- SystemC-supported C++ compiler
- SystemC library (downloaded and compiled for your C++ compiler)
- Compiler command sequence make file or equivalent

The latest Open SystemC Initiative (OSCI) SystemC release (2.2 at this writing) is available for free from www.systemc.org. The download contains scripts and make files for installation of the SystemC library as well as SystemC source code, examples, and documentation. The install scripts are compatible with the supported operating systems, and the scripts are relatively straightforward to execute by carefully following the documentation.

The latest OS requirements can be obtained from the download in a ReadMe file currently called *INSTALL*. SystemC is supported on various versions of Sun Solaris, Linux, HP/UX, Windows, and Mac OS X. At this time, the OS list is limited by the use of minor amounts of assembly code that is used for increased simulation performance in the SystemC simulation kernel. The current release is also supported for various C++ compilers including GNU C++, Sun C++, HP C++, and Visual C++. The currently supported compilers and compiler versions can also be obtained from the *INSTALL* ReadMe file in the SystemC download.

For beginners, this OS and compiler list should be considered exhaustive. Notably, some hardy souls have ported various SystemC versions to other unsupported operating systems and C++ compilers. In addition to one of these platforms and compilers, you will need GNU make installed on your system to compile and quickly install the SystemC library with the directions documented in the *INSTALL* file.

The flow for compiling a SystemC program or design is very traditional and is illustrated in Fig. 2.2 for GNU C++. Most other compilers will be similar. The C++ compiler reads each of the SystemC code file sets separately and creates an object file (the usual file extension is *.o*). Each file set usually consists of two files, typically with standard file extensions. We use *.h* and *.cpp* as file extensions, since these are the most commonly used in C++. The *.h* file is generally referred to as the header file and the *.cpp* file is often called the implementation file.

Note that the compiler and linker need to know two special pieces of information. First, the compiler needs to know where the SystemC header files are located. Second, the linker needs to know the location of the compiled SystemC libraries. This information is typically passed by providing an environment variable named

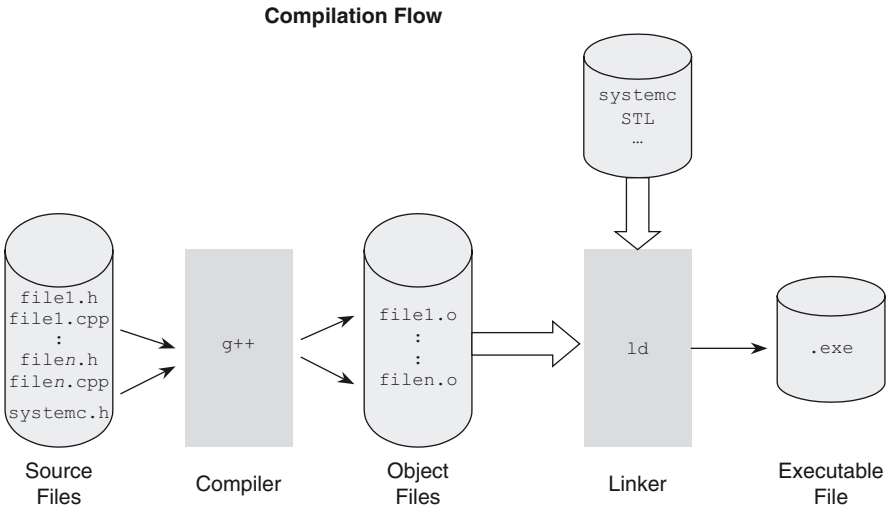


Fig. 2.2 SystemC compilation flow

SYSTEMC and by ensuring the makefile rules use the information.[2] If using gcc, the command probably looks something like Fig. 2.3.

The downloadable examples available from our web site include a makefile setup for Linux and gcc. Please refer to your C++ tool manuals for more information.

```
g++ -I$(SYSTEMC)/include \
    -L$(SYSTEMC)/lib-$(ARCH) -lsystemc \
    $(SRC)
```

Fig. 2.3 Partial gcc options to compile and link SystemC

After creating the object files, the compiler (actually the loader or linker) will link your object files and the appropriate object files from the SystemC library (and other libraries such as the standard template library or STL). The resulting file is usually referred to as an executable, and it contains the SystemC simulation kernel and your design functionality.

For the hardcore engineer types, you now have everything you need to compile and run a *Hello_SystemC* program; we have provided the obligatory program in Fig. 2.4 Keywords for both C++ and SystemC are in **bold**. The rest of you now have an overview of how to compile and run the code examples in this book as well as your own SystemC creations. Everyone is now ready to dive into the language itself.

[2] For some installations, dynamic libraries may also be referenced if using the SystemC Verification library.

```
#include <systemc>
SC_MODULE(Hello_SystemC) { // declare module class

  SC_CTOR(Hello_SystemC) { // create a constructor
    SC_THREAD(main_thread);// register the process
  }//end constructor

  void main_thread(void) {
    SC_REPORT_INFO(" Hello SystemC World!");
  }
};

int sc_main(int sc_argc, char* sc_argv[]) {

  //create an instance of the SystemC module
  Hello_SystemC HelloWorld_i("HelloWorld_i");

  sc_start(); // invoke the simulator

  return 0;
}
```

Fig. 2.4 Hello_SystemC program example

2.2 SystemC Class Concepts for Hardware

SystemC provides mechanisms crucial to modeling hardware while using a language environment compatible with software development. SystemC provides several hardware-oriented constructs that are not normally available in a software language; however, these constructs are required to model hardware. All of the constructs are implemented within the context of the C++ language. This section looks at SystemC from the viewpoint of the hardware-oriented features. The major hardware-oriented features implemented within SystemC include:

- Time model
- Hardware data types
- Module hierarchy to manage structure and connectivity
- Communications management between concurrent units of execution
- Concurrency model

The following sections briefly discuss the implementation of these concepts within SystemC.

2.2.1 Time Model

SystemC tracks time with 64 bits of resolution using a class known as **`sc_time`**. Global time is advanced within the kernel. SystemC provides mechanisms to obtain the current time and implement specific time delays. To support ease of

use, an enumerated type defines several natural time units from seconds down to femtoseconds.

For those models that require a clock, a class called **`sc_clock`** is provided. Since many applications in SystemC do not require a clock (but do require a notion of time), the clock discussion is deferred to later chapters of the book. Additionally, clocks do not add to the fundamental understanding of the language. By the later chapters, you should be able to implement the clock class yourself with the fundamentals learned throughout the book.

2.2.2 Hardware Data Types

The wide variety of data types required by digital hardware are not provided inside the natural boundaries of C++ native data types, which are typically 8-, 16-, 32-, and 64-bit entities.

SystemC provides hardware-compatible data types that support explicit bit widths for both integral and fixed-point quantities. Furthermore, digital hardware requires non-binary representation such as tri-state and unknowns, which are provided by SystemC.

Finally, hardware is not always digital. SystemC does not currently directly support analog hardware; however, a working group has been formed to investigate the issues associated with modeling analog hardware in SystemC. For those with immediate analog issues, it is reasonable to model analog values using floating-point representations and provide the appropriate behavior.

2.2.3 Hierarchy and Structure

Large designs are almost always broken down hierarchically to manage complexity, easing understanding of the design for the engineering team. SystemC provides several constructs for implementing hardware hierarchy. Hardware designs traditionally use blocks interconnected with wires or signals for this purpose. For modeling hardware hierarchy, SystemC uses the module entity interconnected to other modules using channels. The hierarchy comes from the instantiation of module classes within other modules.

2.2.4 Communications Management

The SystemC channel provides a powerful mechanism for modeling communications. Conceptually, a channel is more than a simple signal or wire. Channels can represent complex communications schemes that eventually map to significant

hardware such as the AMBA bus[3]. At the same time, channels may also represent very simple communications such as a wire or a FIFO (first-in first-out queue).

The ability to have several quite different channel implementations used interchangeably to connect modules is a very powerful feature. This feature enables an implementation of a simple bus replaced with a more detailed hardware implementation, which is eventually implemented with gates.

SystemC provides several built-in channels common to software and hardware design. These built-in channels include locking mechanisms like mutex and semaphores, as well as hardware concepts like FIFOs, signals and others.

Finally, modules connect to channels and other modules via port classes.

2.2.5 Concurrency

Concurrency in a simulator is always an illusion. Simulators execute the code on a single physical processor. Even if you did have multiple processors performing the simulation, the number of units of concurrency in real hardware design will always outnumber the processors used to do the simulation by several orders of magnitude. Consider the problem of simulating the processors on which the simulator runs.

Simulation of concurrent execution is accomplished by simulating each concurrent unit. Each unit is allowed to execute until simulation of the other units is required to keep behaviors aligned in time. In fact, the simulation code itself determines when the simulator makes these switches by the use of events. This simulation of concurrency is the same for SystemC, Verilog, VHDL, or any other hardware description languages (HDLs). In other words, the simulator uses a cooperative multitasking model. The simulator merely provides a kernel to orchestrate the swapping of the various concurrent elements, called simulation processes. SystemC provides a simulation kernel that will be discussed lightly in the last section of this chapter and more thoroughly in the rest of the book.

2.2.6 Summary of SystemC Features for Hardware Modeling

SystemC implements the structures necessary for hardware modeling by providing constructs that enable concepts of time, hardware data types, hierarchy and structure, communications, and concurrency. This section has presented an overview of SystemC relative to a generic set of requirements for hardware design. We will now give a brief overview of the constructs used to implement these requirements in SystemC.

[3] See AMBA AHB Cycle-Level Interface Specification at www.arm.com.

2.3 Overview of SystemC Components

In this section, we briefly discuss all the components of SystemC that are highlighted in Fig. 2.1 from the beginning of this chapter, that we will see at the beginning of each chapter throughout the book.

2.3.1 Modules and Hierarchy

Hardware designs typically contain hierarchy to reduce complexity. Each level of hierarchy represents a block. VHDL refers to blocks as entity/architecture pairs, which separate an interface specification from the body of code for each block. In Verilog, blocks are called modules and contain both interface and implementation in the same code.

SystemC separates the interface and implementation similar to VHDL. The C++ notion of header (`.h` file) is used for the entity and the notion of implementation (`.cpp` file) is used for the architecture.

Design components are encapsulated as "modules". Modules are classes that inherit from the `sc_module` base class. As a simplification, the `SC_MODULE` macro is provided.

Modules may contain other modules, processes, and channels and ports for connectivity.

2.3.2 SystemC Threads and Methods

Before getting started, it is necessary to have a firm understanding of simulation processes in SystemC. As indicated earlier, the SystemC simulation kernel schedules the execution of all simulation processes. Simulation processes are simply member functions of **`sc_module`** classes that are "registered" with the simulation kernel.

Because the simulation kernel is the only caller of these member functions, they need no arguments and they return no value. They are simply C++ functions that are declared as returning a void and having an empty argument list.

An **`sc_module`** class can also have processes that are not executed by the simulation kernel. These processes are invoked as function calls within the simulation processes of the **`sc_module`** class. These are normal C++ member functions or class methods.

From a software perspective, processes are simply threads of execution. From a hardware perspective, processes provide necessary modeling of independently timed circuits. Simulation processes are member functions of an **`sc_module`** that are registered with the simulation kernel. Generally, registration occurs during the elaboration phase (during the execution of the constructor for the

sc_module class) using an **SC_METHOD, SC_THREAD**, or **SC_CTHREAD**[4] SystemC macro.

The most basic type of simulation process is known as the **SC_METHOD.** An **SC_METHOD** is a member function of an **sc_module** class where time does not pass between the invocation and return of the function. In other words, an **SC_METHOD** is a normal C++ function that happens to have no arguments, returns no value, and is repeatedly and only called by the simulation kernel.

The other basic type of simulation process is known as the **SC_THREAD**. This process differs from the **SC_METHOD** in two ways. First, an **SC_METHOD** is invoked (or started) multiple times and the **SC_THREAD** is invoked only *once*. Second, an **SC_THREAD** has the option to *suspend* itself and potentially allow time to pass before continuing. In this sense, an **SC_THREAD** is similar to a traditional software thread of execution.

The **SC_METHOD** and **SC_THREAD** are the basic units of concurrent execution. The simulation kernel invokes each of these processes. Therefore, they are never invoked directly by the user. The user indirectly controls execution of the simulation processes by the kernel as a result of events, sensitivity, and notification.

2.3.3 Events, Sensitivity, and Notification

Events, sensitivity, and notification are very important concepts for understanding the implementation of concurrency by the SystemC simulator.

Events are implemented with the SystemC **sc_event** and **sc_event_queue** classes. Events are caused or fired through the event class member function, **notify**. The notification can occur within a simulation process or as a result of activity in a channel. The simulation kernel invokes **SC_METHOD** and **SC_THREAD** when they are sensitive to an event and the event occurs.

SystemC has two types of sensitivity: static and dynamic. Static sensitivity is implemented by applying the SystemC **sensitive** command to an **SC_METHOD** or **SC_THREAD** at elaboration time (within the constructor). Dynamic sensitivity lets a simulation process change its sensitivity on the fly. The **SC_METHOD** implements dynamic sensitivity with a **next_trigger(arg)** command. The **SC_THREAD** implements dynamic sensitivity with a **wait(arg)** command. Both **SC_METHOD** and **SC_THREAD** can switch between dynamic and static sensitivity during simulation.

---

[4] **SC_CTHREAD** is a special case of **SC_THREAD**. This process type is a thread process that has the requirement of being sensitive to a clock. **SC_CTHREAD** is under consideration for deprecation; however, several synthesis tools depend on it at the time of writing.

2.3.4 SystemC Data Types

Several hardware data types are provided in SystemC. Since the SystemC language is built on C++, all of the C++ data types are available. Also, SystemC lets you define new data types for new hardware technology (i.e., multi-valued logic) or for applications other than electronic system design.

These data types are implemented using templated classes and generous operator overloading, so that they can be manipulated and used almost as easily as native C++ data types. Hardware data types for mathematical calculations like **sc_fixed**<*T*> and **sc_int**<*T*> allow modeling of complex calculations like DSP functions. These data types evaluate the performance of an algorithm when implemented in custom hardware or in processors without full floating-point capability. SystemC provides all the necessary methods for using hardware data types, including conversion between the hardware data types and conversion from hardware to software data types.

Non-binary hardware types are supported with four-state logic (0,1,X,Z) data types (e.g., **sc_logic**). Familiar data types like **sc_logic** and **sc_lv**<*T*> are provided for RTL hardware designers who need a data type to represent basic logic values or vectors of logic values.

2.3.5 Ports, Interfaces, and Channels

Processes need to communicate with other processes both locally and in other modules. In traditional HDLs, processes communicate via ports/pins and signals or wires. In SystemC, processes communicate using channels or events. Processes may also communicate across module boundaries. Modules may interconnect using channels, and connect via ports. The powerful ability to have interchangeable channels is implemented through a component called an interface. SystemC uses the constructs **sc_port**<*T*>, **sc_export**<*T*>, and the base classes **sc_interface**, and **sc_channel** to implement connectivity.

SystemC provides some standard channels and interfaces that are derived from these base types. The provided channels include the synchronization primitives **sc_mutex** and **sc_semaphore**, and the communication channels **sc_fifo**<*T*>, **sc_signal**<*T*>, and others. These channels implement the SystemC-provided interfaces **sc_mutex_if**, **sc_semaphore_if**, **sc_fifo_in_if**<*T*>, **sc_fifo_out_if**<*T*>, **sc_signal_in_if**<*T*>, and **sc_signal_inout_if**<*T*>.

Interestingly, module interconnection happens programmatically in SystemC during the elaboration phase. This interconnection lets designers build regular structures using loops and conditional statements. From a software perspective, elaboration is simply the period of time when modules invoke their constructor methods.

2.3.6 Summary of SystemC Components

Now, it is time to tie together all of the basic concepts that we have just discussed into one illustration, Fig. 2.5 This illustration is used many times throughout the book when referring to the different SystemC components. It can appear rather intimidating since it shows almost all of the concepts within one diagram. In practice, a SystemC module typically will not contain all of the illustrated components.

The figure shows the concept of an **`sc_module`** that can contain instances of another **`sc_module`**. An **`SC_METHOD`** or **`SC_THREAD`** can also be defined within an **`sc_module`**.

Communication among modules and simulation processes (**`SC_METHOD`** and **`SC_THREAD`**) is accomplished through various combinations of ports, interfaces, and channels. Coordination among simulation processes is also accomplished through events.

We will now give a brief initial overview of the SystemC simulation kernel that coordinates and schedules the communications among all of the components illustrated in Fig. 2.5

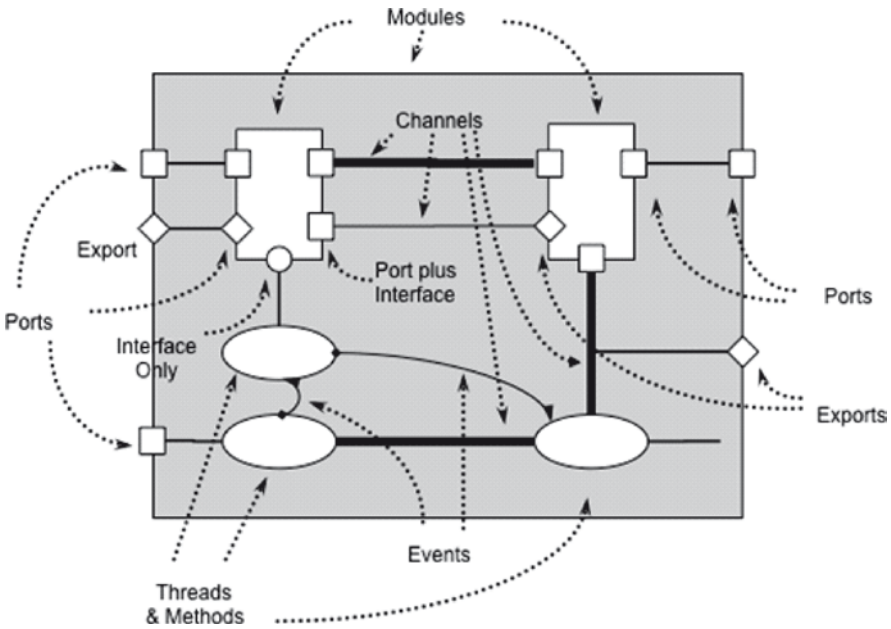


Fig. 2.5 SystemC components

2.4 SystemC Simulation Kernel

The SystemC simulator has two major phases of operation: *elaboration* and *execution*. A third, often minor, phase occurs at the end of execution; this phase could be characterized as post-processing or *cleanup*.

Execution of statements prior to the **`sc_start()`** function call are known as the elaboration phase. This phase is characterized by the initialization of data structures, the establishment of connectivity, and the preparation for the second phase, execution.

The execution phase hands control to the SystemC simulation kernel, which orchestrates the execution of processes to create an illusion of concurrency.

The illustration in Fig. 2-6 should look very familiar to those who have studied Verilog and VHDL simulation kernels. Very briefly, after **`sc_start()`**, all simulation processes (minus a few exceptions) are invoked in unspecified deterministic order[5] during initialization.

After initialization, simulation processes are run when events occur to which they are sensitive. The SystemC simulator implements a cooperative multitasking environment. Once started, a running process continues to run until it yields control. Several simulation processes may begin at the same instant in simulator time. In this case, all of the simulation processes are evaluated and then their outputs are updated. An evaluation followed by an update is referred to as a *delta cycle*.

If no additional simulation processes need to be evaluated at that instant (as a result of the update), then simulation time is advanced. When no additional simulation processes need to run, the simulation ends.

This brief overview of the simulation kernel is meant to give you an overview for the rest of the book. This diagram will be used again to explain important

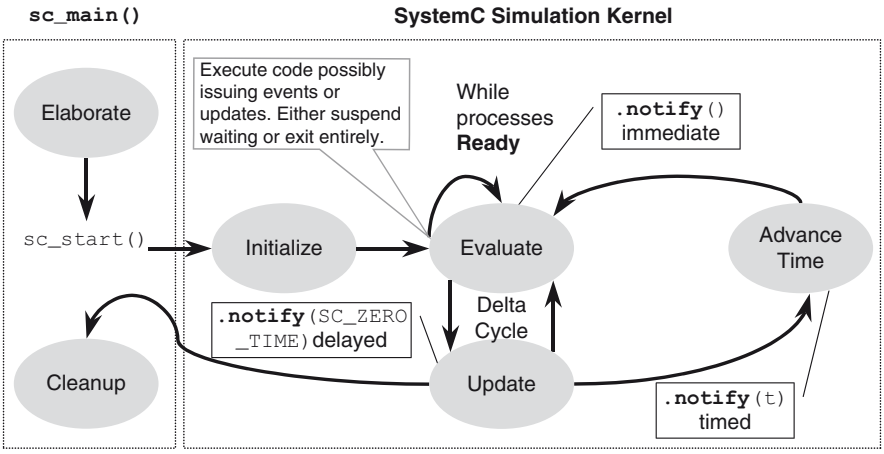


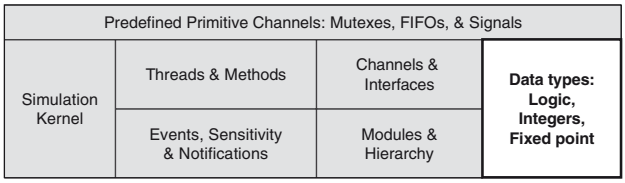
Fig. 2.6 SystemC simulation kernel

[5] Discussed later.

intricacies later. It is very important to understand how the kernel functions to fully understand the SystemC language.

We have provided an animated version of this diagram walking through a small code example at our web site, www.scftgu.com. The IEEE Standard 1666-2005 SystemC LRM (Language Reference Manual) specifies the behavior of the SystemC simulation kernel. This manual is the definitive source about SystemC. We encourage the reader to use any or all of these resources during their study of SystemC to fully understand the simulation kernel.

Chapter 3
Data Types



This chapter provides an overview of the data types available to a user creating SystemC simulation models. The SystemC Language Reference Manual (LRM) IEEE Standard 1666 uses over 190 pages to specify the SystemC data types. We have attempted to be considerably briefer.

The reader will consequently need to refer to the SystemC LRM for a full definition of the SystemC data types and other resources for the C++ data types and other libraries. The SystemC library provides integer, logic, and fixed-point data types designed for modeling hardware. In addition to the SystemC data types, SystemC simulations may use native C++ data types, other library data types, and user-defined data types.

The use of SystemC data types is not restricted to models using the simulation kernel; they may be used in non-simulation applications as other data types would be used. Though simulation models may be created using any of the available data types, the choice of data types affects simulation speed, synthesizability, and synthesis results. The use of the native C++ data types can maximize simulation performance, at the cost of hardware fidelity and synthesizability.

3.1 Native C++ Data Types

The native C++ data types available on most systems include the logic data type **`bool`,** and signed and unsigned versions of the following arithmetic data types in Table 3.1.

Table 3.1 Example of C++ built-in data type on a 32-bit architecture

Name	Description	Size
`char`	Character	1 byte
`short int (short)`	Short integer	2 bytes
`int`	Integer	4 bytes
`long int (long)`	Long integer	4 bytes
`long long int`	Long long integer	8 bytes
`float`	Floating point	4 bytes
`double`	Double precision floating point	8 bytes

D.C. Black et al., *SystemC: From the Ground Up*, Second Edition,
DOI 10.1007/978-0-387-69958-5_3, © Springer Science+Business Media, LLC 2010

```
// Example native C++ data types
const bool      WARNING_LIGHT(true);// Status
int             spark_offset; // Adjust ignition
unsigned        repairs(0);   // # of repairs
unsigned long   mileage;      // Miles driven
short int       speedometer;  // -20..0..100 MPH
float           temperature;  // Engine temp in C
double          time_of_last_request; // bus activity
string          license_plate;// license plate text
enum            Direction { N,NE,E,SE,S,SW,W,NW };
Direction       compass;
```

Fig. 3.1 Example of C++ built-in data types

The Standard Template Library (STL) has a rich set of additional data types. The beginner will want to become familiar with **`string`** from the STL. The string data type provides operators for appending (`+=`, `+`, and **`assign()`**) and comparison (`==`, `!=`, `<`, `<=`, `>`, `>=`, and **`compare()`**) as well as many others including a conversion to a c-string (**`c_str()`**).

For many SystemC models, the native C++ data type in Fig. 3.1 are more than sufficient. The native C++ data types are most efficient in terms of memory usage and simulator execution speed because they can be mapped directly to processor instructions.

3.2 SystemC Data Types Overview

The SystemC library provides data types that are designed for modeling digital logic and fixed-point arithmetic. There are two SystemC logic vector types: 2-valued and 4-valued logic; and two SystemC numeric types: integers and fixed-point. The SystemC data types are designed to be freely mixed with the native C++ data types, other SystemC data types, and C++ strings. SystemC data types may be used in C++ applications just as any other C++ library.

With the exception of the single-bit **`sc_logic`** type, all of the SystemC data types are length configurable over a range much broader than the native C++ data types. SystemC provides assignment and initialization operations with type conversions, allowing C++ data types, SystemC data types, and C++ strings to be used for initialization or in assignment operation to SystemC data types. All SystemC data types implement equality and bitwise operations.

The SystemC arithmetic data types (integer and fixed-point) implement arithmetic and relational operations. The SystemC implementations of these operations are semantically compatible with C++. SystemC also supports the conversion between SystemC data types and C++ data types.

The SystemC data types allow single-bit and multi-bit select operations. The results of these select operations may be used as the source (right-hand side) or

target (left-hand side) of assignment operations or concatenated with other bit-selects or with SystemC integers or vector data types.

All of the SystemC data type are part of the **sc_dt** namespace, which may be used with the scope operator (::) to avoid naming collisions with other C++ libraries.

3.3 SystemC Logic Vector Data Types

SystemC provides two logic vector types: **sc_bv**<*W*> (bit vector) and **sc_lv**<*W*> (logic vector), and a single-bit logic type **sc_logic**. Early versions of SystemC defined **sc_bit,** a single-bit version of **sc_bv**<*W*>, which was deprecated by the IEEE SystemC Standard. Older applications that used the **sc_bit** data type may replace instances of **sc_bit** with the C++ **bool** data type.

The SystemC logic vector types are intended to model at a very low level (near RTL) and do not implement arithmetic operations. They do implement a full range of assignment and logical operations, given some obvious restrictions. For example, an **sc_lv**<*W*> with high-z or unknown bit values cannot be assigned to an **sc_bv**<*W*> without losing some information.

3.3.1 sc_bv<W>

The SystemC bit vector data type **sc_bv**<*W*> has the same capabilities as the **sc_lv**<*W*> with the bit values restricted to logic zero or logic one. The **sc_bv**<*W*> is a templated class where *T* specifies bit width.

```
sc_bv<BITWIDTH> NAME…;
```

Fig. 3.2 Syntax of Boolean data types

SystemC bit vector *operations* include all the common bitwise-and, bitwise-or, and bitwise-xor operators (i.e., **&**, |, **^**). In addition to bit selection and bit ranges (i.e., **[]** and **range()**), **sc_bv**<*W*> also supports **and_reduce()**, **or_reduce()**, **nand_reduce()**, **nor_reduce()**, **xor_reduce()**, and **xnor_reduce()** operations. Reduction operations place the operator between all adjacent bits.

```
sc_bv<5> positions = "01101";
sc_bv<6> mask = "100111";
sc_bv<5> active = positions & mask;// 00101
sc_bv<1> all = active.and_reduce(); // SC_LOGIC_0
positions.range(3,2) = "00";// 00001
positions[2] = active[0] ^ flag;
```

Fig. 3.3 Examples of bit operations

3.3.2 sc_logic and sc_lv<W>

More interesting than the Boolean data types are the four-value data types used to represent unknown and high impedance (tri-state) conditions. SystemC uses **sc_logic** and **sc_lv**<*W*> to represent these data types (Fig. 3.4). The logic state of these data types are represented as:

- logic 0 - **SC_LOGIC_0, Log_0,** or **'0'**
- logic 1 - **SC_LOGIC_1, Log_1,** or **'1'**
- high-impedance - **SC_LOGIC_Z, Log_Z, 'Z'** or **'z'**
- unknown - **SC_LOGIC_X, Log_X, 'X'** or **'x'**

Because of their overhead, these data types are considerably slower than **bool** and **sc_bv.** The **sc_logic** data type is a single-bit version of the templated **sc_lv**<*W*> class where the single template parameter is the bit width.

SystemC does not have representations for other multi-level data types or drive strengths like Verilog's 12-level logic values or VHDL's 9-level **std_logic** values. However, you can create custom data types if truly necessary, and you can manipulate them by operator overloading in C++.

```
sc_logic         NAME[,NAME]…;
sc_lv<BITWIDTH> NAME[,NAME]…;
```

Fig. 3.4 Syntax of multi-value data types

SystemC logic vector operations (Fig. 3.5) include all the common bitwise-and, bitwise-or, and bitwise-xor operators (i.e., **&**, **|**, **^**). In addition to bit selection and bit ranges (i.e., **[]** and **range()**), **sc_lv**<*W*> also supports **and_reduce()**, **or_reduce()**, **nand_reduce()**,**nor_reduce()**, **xor_reduce()**, and **xnor_reduce()** operations. Reduction operations place the operator between all adjacent bits.

```
sc_lv<5> positions =  "01xz1";
sc_lv<6> mask      = "10ZX11";
sc_lv<5> active = positions & mask; // 0xxx1
sc_lv<1> all = active.and_reduce(); // SC_LOGIC_0
positions.range(3,2) = "00";        // 000Z1
positions[2] = active[0] ^ flag;    // !flag
```

Fig. 3.5 Examples of bit operations

3.4 SystemC Integer Types

SystemC provides signed and unsigned two's complement versions of two basic integer data types. The two data types are a limited precision integer type, which has a maximum length of 64-bits, and a finite precision integer type, which can be much longer. These integer data types provide functionality not available in the native C++ integer types. The native C++ data types have widths that are host processor and compiler dependent. The native data types are optimized for the host processor instruction set and are very efficient. These data types are typically 8, 16, 32, or 64 bits in length. The SystemC integer data types are templated and may have data widths from 1 to hundreds of bits. In addition to configurable widths, the SystemC integer data types allow bit selections, bit range selections, and concatenation operations.

3.4.1 sc_int<W> and sc_uint<W>

Most hardware needs to specify actual storage width at some level of refinement. When dealing with arithmetic, the built-in **sc_int**<*W*> and **sc_uint**<*W*> (unsigned) numeric data types (Fig. 3.6) provide an efficient way to model data with specific widths from 1- to 64-bits wide. When modeling numbers where data width is not an integral multiple of the simulating processor's data paths, some bit masking and shifting must be performed to fit internal computation results into the declared data format.

```
sc_int<LENGTH>   NAME…;
sc_uint<LENGTH>  NAME…;
```

Fig. 3.6 Syntax of arithmetic data types

3.4.2 sc_bigint<W> and sc_biguint<W>

Some hardware may be larger than the numbers supported by native C++ data types. SystemC provides **sc_bigint**<*W*> and **sc_biguint**<*W*> data types (Fig. 3.7) for this purpose. These data types provide large number support at the cost of speed.

```
sc_bigint<BITWIDTH>  NAME…;
sc_biguint<BITWIDTH> NAME…;
```

Fig. 3.7 Syntax of sc_bigint<*W*> and sc_biguint<*W*>

```
// SystemC integer data types
sc_int<5>      seat_position=3; //5 bits: 4 plus
                                // sign
sc_uint<13>    days_SLOC(4000); //13 bits: no sign
sc_biguint<80> revs_SLOC;       // 80 bits: no sign
```

Fig. 3.8 Example of SystemC integer data types

3.5 SystemC Fixed-Point Types

The SystemC fixed-point data types address the need for non-integer data types when modeling DSP applications that cannot justify the use of floating-point hardware. Early DSP models should be developed using the native C++ floating-point data types due to the much higher simulation speed. As a design evolves, fixed-point data types can provide higher fidelity modeling of the signal processing logic and can be used to provide a path to a synthesizable design.

SystemC provides multiple fixed-point data types: signed and unsigned, compile-time (templated) and run-time configurable, and fixed-precision and limited-precision (`_fast`) versions.

The SystemC fixed-point data types (Fig. 3.9) are characterized by word length (total number of bits) and their integer portion length. Optional parameters provide control of overflow and quantization modes.

Unlike other SystemC data types, fixed-point data types may be configured at compile time, using template parameters or at run time using constructor parameters. Regardless of how a fixed-point data object is created, it cannot be altered later in the execution of the program.

```
sc_fixed<WL,IWL[,QUANT[,OVFLW[,NBITS]>       NAME;
sc_ufixed<WL,IWL[,QUANT[,OVFLW[,NBITS]>      NAME;
sc_fixed_fast<WL,IWL[,QUANT[,OVFLW[,NBITS]>  NAME;
sc_ufixed_fast<WL,IWL[,QUANT[,OVFLW[,NBITS]> NAME;

sc_fix NAME(WL,IWL[,QUANT[,OVFLW[,NBITS]);
sc_ufix NAME(WL,IWL[,QUANT[,OVFLW[,NBITS]);
sc_fix_fast NAME(WL,IWL[,QUANT[,OVFLW[,NBITS]);
sc_ufix_fast NAME(WL,IWL[,QUANT[,OVFLW[,NBITS]);
```

Fig. 3.9 Syntax of fixed-point data types

```
// to enable fixed-point data types
#define SC_INCLUDE_FX
#include <systemc>
// fixed-point data types are now enabled
sc_fixed<5,3>    compass // 5-bit fixed-point word
```

Fig. 3.10 Example of fixed-point data types

Due to their compile-time overhead, fixed-point data types are omitted from the default SystemC include file. To enable fixed-point data types, **SC_INCLUDE_FX** must be defined prior to including the SystemC header file (Fig. 3.10).

The fixed-point data types have several easy-to-remember distinctions. First, those data types ending with **fast** are faster than the others, because their precision is limited to 53 bits internally; **fast** types are implemented using C++ `double`[1].

Second, the prefix **sc_ufix** indicates unsigned just as uint indicates unsigned integers. Third, the past tense **ed** suffix to **fix** indicates a templated data type that must have static parameters defined using compile-time constants.

Remember that **fixed** is past tense (i.e., already set in stone), and it cannot be changed after compilation. At run time, you may create and use new objects of the non-templated (**fix** versions) data types varying the configuration as needed.

Though the **sc_fix**, **sc_ufix**, **sc_fix_fast**, and **sc_ufix_fast** are run-time configurable, once an object of these types is created, its configuration cannot be modified.

The parameters needed for fixed-point data types are the word length (*WL*), integer-word length (*IWL*), quantization mode (*QUANT*), overflow mode (*OVFLW*), and number of saturation bits (*NBITS*). Word length (*WL*) and integer word length (*IWL*) have no defaults and must be set.

The word length establishes the total number of bits representing the data type. The integer word length indicates where to place the binary point and can be positive or negative. Figure 3.11 below shows how this works.

The preceding figure shows examples with the binary point in several positions. Consider example b in Fig. 3.11. This could be declared as **sc_fixed**<5,3>, and would represent values from –4.00 up to 3.75 in 1/4 increments.

[1] This implementation takes advantage of the linearly scaled 53-bit integer mantissa inside a 64-bit IEEE-754 compatible floating-point unit. On processors without an FPU, this behavior must be emulated in software, and there will be no speed advantage.

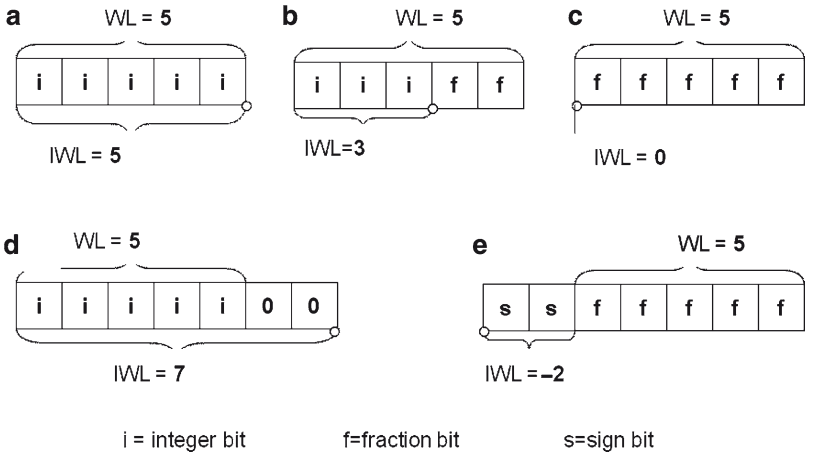


Fig. 3.11 Fixed-point formats

You can select several overflow modes from a set of enumerations that are listed in Table 3.2. A similar table for the quantization modes is also shown in Table 3.3. Overflow mode, quantization mode, and number of saturation bits all have defaults. You can modify the defaults by setting up a **`sc_fxtype_context`** object for the run-time configurable data types.

Table 3.2 Overflow mode enumerated constants

Name	Overflow Meaning
`SC_SAT`	Saturate
`SC_SAT_ZERO`	Saturate to zero
`SC_SAT_SYM`	Saturate symmetrically
`SC_WRAP`	Wraparound
`SC_WRAP_SYM`	Wraparound symmetrically

Table 3.3 Quantization mode enumerated constants

Name	Quantization Mode
`SC_RND`	Round
`SC_RND_ZERO`	Round towards zero
`SC_RND_MIN_INF`	Round towards minus infinity
`SC_RND_INF`	Round towards infinity
`SC_RND_CONV`	Convergent rounding[a]
`SC_TRN`	Truncate
`SC_TRN_ZERO`	Truncate towards zero

[a]Convergent rounding is probably the oddest. If the most significant deleted bit is one, and either the least significant of the remaining bits or at least one of the other deleted bits is one, then add one to the remaining bits.

The following examples in Fig. 3.12 should help explain the syntax for the fixed-point data types:

```
const sc_ufixed<19,3> PI("3.141592654");
sc_fix oil_temp(20,17,SC_RND_INF,SC_SAT);
sc_fixed_fast<7,1> valve_opening;
```

Fig. 3.12 Examples of fixed-point data types

Only the word length and integer word length are required parameters. If not specified, the default overflow is **SC_WRAP**, the default quantization is **SC_TRN**, and saturation bits default to one.

A special note applies if you intend to set up arrays of the **_fix** types. Since a constructor is required, but C++ syntax does not allow arguments for this situation, it is necessary to use the **sc_fxtype_context** type to establish the defaults.

3.6 SystemC Literal and String

Representation of literal data is fundamental to all languages. C++ allows for integers, floats, Booleans, characters, and strings. SystemC provides the same capability for its data types and uses C++ representations as a basis.

3.6.1 SystemC String Literals Representations

The SystemC string literals may be used to assign values to any of the SystemC data types. SystemC string literals consist of a prefix, a magnitude and an optional sign character "+" or "-". The optional sign may precede the prefix for decimal and sign and magnitude forms, but is not allowed with unsigned, binary, octal, and hexadecimal. Negative values for binary, octal, and hexadecimal may be expressed using a two's complement representation. The supported prefixes are listed in Table 3.4.

Instances of the SystemC data types may be converted to a standard C++ string using the data type's **to_string** method. The format of the resulting string is specified using **sc_numrep** enumeration shown in the left-hand column of Table 3.4. Examples of this formatting are shown in the right-most column of the above table; the values used are 13 and -13. The **to_string** method has two arguments: a numbers representation from column one of the table; and **bool** to add a representation prefix to the resulting string.

Table 3.4 Unified string representation for SystemC

sc_numrep	Prefix	Meaning	sc_int<5> = 13 sc_int<5> = -13
SC_DEC	0d	Decimal	0d13 -0d13
SC_BIN	0b	Binary	0b01101 0b10011
SC_BIN_US	0bus	Binary unsigned	0bus1101 negative
SC_BIN_SM	0bsm	Binary signed magnitude	0bsm01101 -0bsm01101
SC_OCT	0o	Octal	0o15 0o63
SC_OCT_US	0ous	Octal unsigned	0ous15 negative
SC_OCT_SM	0osm	Octal signed magnitude	0osm15 -0osm15
SC_HEX	0x	Hex	0x0d 0xf3
SC_HEX_US	0xus	Hex unsigned	0xusd negative
SC_HEX_SM	0xsm	Hex signed magnitude	0xsm0d -0xsm0d
SC_CSD	0csd	Canonical signed digit	0csd10-01 0csd-010-

```
string to_string(sc_numrep rep, bool wprefix);
```

Fig. 3.13 Syntax of **to_string**

3.6.2 String Input and Output

SystemC string literal representations and streaming IO represent data using the same formats. SystemC supports the input stream using the input extractor (**operator**>>) and output stream using the output inserter (**operator**<<).

Input streams use the literal prefixes shown in the second column of Table 3.4 to define the format of data being read from an input stream. This is the same format used when assigning a literal to a SystemC variable.

Output streams may use the C++ output stream manipulators **dec, oct**, and **hex** to control the display format of the SystemC data types. Additional display control may be obtained by using the data type's **to_string** methods.

Earlier versions of SystemC LRM defined a unique string type **sc_string**, which is now deprecated. New SystemC applications should use standard C++ string and the **to_string** method described in this section.

Below are a few examples in Fig. 3.14 that demonstrate the use of SystemC literals, the **to_string** method, and output stream:

```
//-------------------------------------------------
// sc_lv<8>      8-bit logic vectors
//-------------------------------------------------
sc_lv<8> LV1;
LV1 = 15;
cout << " LV1= " << LV1;
sc_lv<8> LV2("0101xzxz"); // literal string init
cout << " LV2= " << LV2;
cout << endl;
//-------------------------------------------------
// sc_int<5>   5-bit signed integer
//-------------------------------------------------
sc_int<5> Int1;         // 5-bit signed integer
Int1 = "-0d13";         // assign -13
cout << " Int1=" << Int1;
cout << " SC_BIN=" << Int1.to_string(SC_BIN);
cout << " SC_BIN_SM=" << Int1.to_string(SC_BIN_SM);
cout << "          " << endl;
cout << " SC_HEX=" << Int1.to_string(SC_HEX);
cout << endl;
//-------------------------------------------------
// sc_fixed<5,3> fixed 3-bit int & 2 bit fraction
//-------------------------------------------------
sc_fixed<5,3> fix1; // fixed point
fix1 = -3.3;
cout << " fix1=" <<fix1;
cout << " SC_BIN=" << fix1.to_string(SC_BIN);
cout << " SC_HEX=" << fix1.to_string(SC_HEX);
cout << endl;
```

Output:

```
LV1=00001111 LV2=0101XZXZ
Int1=-13 SC_BIN=0b10011 SC_BIN_SM=-0bsm01101
         SC_HEX=0xf3
fix1=-3.5 SC_BIN=0b100.10 SC_HEX=0xc.8
```

Fig. 3.14 Example of literals and **to_string**

3.7 Operators for SystemC Data Types

The SystemC data types support all the common operations with operator overloading as illustrated in Table 3.5.

In addition, SystemC provides special methods to access bits, bit ranges, and perform explicit conversions as illustrated in Table 3.6.

The bit and part select and concatenation operations support much like the hardware descriptions languages. This support allows the simple isolation of fields in

Table 3.5 Operators

Comparison	`== != > >= < <=`
Arithmetic	`++ -- * / % + - << >>`
Bitwise	`~ & | ^`
Assignment	`= &= |= ^= *= /= %= += -= <<= >>=`

Table 3.6 Special methods

Bit Selection	`bit(idx), [idx]`
Range Selection	`range(high,low), (high,low)`
Conversion (to C++ types)	`to_double(),to_int(), to_int64(),to_long(), to_uint(),to_uint64(), to_ulong(),to_string(type)`
Testing	`is_zero(), is_neg(), length()`
Bit Reduction	`and_reduce(),nand_reduce(), or_reduce(),nor_reduce(), xor_reduce(), xnor_reduce()`

packed data or the concatenation of multiple bit or bit-fields to create packed data objects. Several examples are shown in Fig. 3.15.

```
//-------------------------------------------------
// bit-select and part-select examples
//-------------------------------------------------
sc_uint<8>  I1 = "0x35";     // 8-bit signed integer
sc_uint<5>  I2 = "0b01010";  // 5-bit signed integer
sc_uint<4>  I3 = 0;          // 5-bit signed integer

sc_uint<16> I4 = 0;            // 16-bit signed
integer

I3    = I2.range(3,0);       // I3= 0b1010
I3[2] = true;                // I3= 0b1110
I3[0] = true;                // I3= 0b1111

I4 = (I3,I1.range(7,4),I2(3,0),I1(3,0));
// I4 = 0x0f3a5                    HEX format
// I4 = 0b01111001110100101        BIN format
```

Fig. 3.15 Bit-select, part-select, and concatenation

One often overlooked aspect of these data types (and C++ data types) is mixing types in arithmetic operations. It is OK to mix similar data types of different lengths, but crossing types is dangerous. For example, assigning the results of an operation involving two **`sc_int`**`<W>` variables to an **`sc_bigint`**`<W>` does not automatically promote the operand to **`sc_bigint`** for intermediate calculations. To accomplish that, it is necessary to have one of the arguments be an **`sc_bigint`**`<W>` or perform an explicit conversion of one of at least one of the operand arguments. Below is an example of addition.

```
sc_int<3> d(3);
sc_int<5> e(15);
sc_int<5> f(14);
sc_int<7> sum = d + e + f;// Works
sc_int<64> g("0x7000000000000000");
sc_int<64> h("0x7000000000000000");
sc_int<64> i("0x7000000000000000");
sc_bigint<70> bigsum = g + h + i; // Doesn't work
bigsum = sc_bigint<70>(g) + h + i;// Works
```

Fig. 3.16 Example of conversion issues

```
#include <vector>
int main(int argc, char* argv[]) {
  vector<int> memory(1024);
  for (unsigned i=0; i!= 1024; i++) {
    // Following checks access (safer than
    // memory[i])
    memory.at(i) = -1; // initialize to known values
  }//endfor
  …
  memory.resize(2048); // increase size of memory
  …
}//end main()
```

Fig. 3.17 Example of STL vector

3.8 Higher Levels of Abstraction and the STL

The basic C++ and SystemC data types lack structure and hierarchy. For these, the standard C++ **`struct`** and **`array`** are good starting points. However, a number of very useful data type classes are freely available in C++ libraries, which provides another benefit of having a modeling language based upon C++.

The STL is the most popular of these libraries, and it comes with all modern C++ compilers. The STL contains many useful data types and structures, including an improved character array known as string, and generic containers such as the **`vector`**<*T*>, **`map`**<*T, T*>, **`set`**<*T*>, **`list`**<*T*>, and **`deque`**<*T*>. These containers can be manipulated by STL algorithms such as **`for_each()`**, **`count()`**, **`min_element()`**, **`max_element()`**, **`search()`**, **`transform()`**, **`reverse()`**, and **`sort()`**. These are just a few of the algorithms available. This book will not attempt to cover the STL in any detail, but a brief example may stimulate you to search further.

The STL container **`vector`**<*T*> closely resembles the common C++ array, but with several useful improvements. Unlike arrays, vectors can be assigned and compared. Also, the **`vector`**<*T*> may be resized dynamically. Perhaps more impor-

tantly, accessing an element of a **vector**<*T*> can have bounds checking for safety. The example below demonstrates use of an STL **vector**<*T*>.

Your mileage will vary, but the authors are certain you will benefit from using the STL with SystemC.

3.9 Choosing the Right Data Type

A frequent question is, "Which data types should be used for this design?" The best answer is, "Choose a data type that is closest to native C++ as possible for the modeling needs at hand." Choosing native data types will always produce the fastest simulation speeds. Table 3.7 gives an idea of performance.

Table 3.7 Data type performance

Speed	Data type
Fastest	Native C/C++ Data Types (e.g., int, double and bool) sc_int<W>, sc_uint<W> sc_bv<W> sc_logic, sc_lv<W> sc_bigint<W>, sc_biguint<W> sc_fixed_fast<WL,IL,…>, sc_fix_fast, sc_ufixed_fast<WL,IL,…>, sc_ufix_fast
Slowest	sc_fixed<WL,IL,…>, sc_fix, sc_ufixed<WL,IL,…>, sc_ufix

Do not use **sc_int**<*W*> or **sc_bigint**<*W*> unless you require more detail than available by native C++ data types. It is preferred to use **sc_int**<*W*> for 64 or fewer bits over **sc_bigint**<*W*> for higher performance. In general, SystemC data types are slower than native C++ data types, and more complex SystemC types are slower than simpler smaller types.

RTL synthesis tools generally require all data to be SystemC data types. Some behavioral synthesis tools allow native C++ data types. SystemC data types may be used to guide the behavioral synthesis tool.

3.10 Exercises

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

Exercise 3.1: Examine, compile, and run the examples from the web site, `datatypes` and `uni_string_rep`. Note that although these examples include `systemc`, they only use data types.

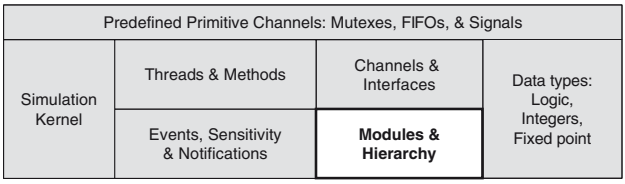
Exercise 3.2: Write a program to read data from a file using the unified string representation and store in an array of **sc_uint**<*W*>. Output the values as **SC_DEC** and **SC_HEX_SM**.

Exercise 3.3: Write a program to generate 100,000 random values and compute the squares of these values. Do the math using each of the following data types: **short, int, unsigned, long, sc_int<8>, sc_uint<19>, sc_bigint<8>, sc_bigint<100>, sc_fixed**<12,12>. Be certain to generate numbers distributed over the entire range of possibilities. Compare the run times of each data type.

Exercise 3.4: Write a program to explore data accuracy of fixed-point numbers for an application needing to add two sine waves with amplitudes of 14 and 22, where the larger of the two is also 2½ times the frequency of the smaller. There are 512 samples over the longest period. Assume you are limited to 12 bits for both input and output. Try different modes. HINT: Start modeling using **double**. Progress to using **sc_fix**.

Exercise 3.5: Examine, compile, and run the example `addition`. What would it take to fix the problems noted in the source code? Try adding various sizes of **sc_bigint**<*W*>.

Chapter 4
Modules



This chapter lays the foundation for SystemC models. Here, we explore how to write a minimal SystemC program in preparation for an exploration of time and concurrency in later chapters. With respect to hierarchy, this chapter only touches the very top level. A later chapter on structure will discuss hierarchy in more detail.

4.1 A Starting Point: sc_main

All programs need a starting point. In C/C++, the starting point is called **`main`**`()`. In SystemC, the starting point is called **`sc_main`**`()`, and this is where you will start your code. SystemC as in Verilog and VHDL, might superficially appear to start every process simultaneously. In reality, all of these simulation languages have initialization requirements that are handled here.

The top level of a C/C++ program is a function named **`main`**`()`. Its declaration is generally:

```
int main(int argc, char* argv[]) {
    BODY_OF_PROGRAM
  return EXIT_CODE; // Zero indicates success
}
```

Fig. 4.1 Syntax of C++ **`main`**`()`

In Fig. 4.1, `argc` represents the number of command-line arguments including the program name itself. `argv[]` is an array of C-style character strings representing the command line that invoked the program. Thus, `argv[0]` is the program name itself.

SystemC usurps this procedure and provides **`sc_main()`** as replacement. The SystemC library provides its own definition of **`main()`**, which in turn calls **`sc_main()`** and passes along the command-line arguments. The syntax for **`sc_main()`** is shown in Fig. 4.2.

D.C. Black et al., *SystemC: From the Ground Up*, Second Edition,
DOI 10.1007/978-0-387-69958-5_4, © Springer Science+Business Media, LLC 2010

```
int sc_main(int argc, char* argv[]) {
  ELABORATION
  sc_start(); // <-- Simulation begins & ends
              //     in this function!
  [POST-PROCESSING]
  return EXIT_CODE; // Zero indicates success
}
```

Fig. 4.2 Syntax of `sc_main()`

By convention, SystemC programmers simply name the file containing **`sc_main()`**, as main.cpp to indicate to the C/C++ programmer that this is the place where everything begins[1]. The actual **`main()`** routine is located in the SystemC library itself and is not exposed to the user.

SystemC provides access to argc and argv throughout the model hierarchy by providing the functions as shown in Fig. 4.3.

```
int         sc_argc(); //access to argc
const char* sc_argv(); //access to argv
```

Fig. 4.3 Syntax of `sc_argc()` and `sc_argv()`

Within **`sc_main()`**, code executes in three distinct major stages. We will now examine these stages, which are elaboration, simulation, and post-processing.

During elaboration, connectivity for the model is established. This includes hierarchal modules, leaf modules, channels, simulation processes, and data structures. Elaboration invokes code to register simulation processes and performs the connections between design modules.

At the end of elaboration, **`sc_start()`** invokes the simulation stage. During simulation, code representing the behavior of the model executes. Within the simulation stage, the scheduler is responsible for process execution. A following chapter on concurrency will explore this stage in greater detail.

Finally, after returning from **`sc_start()`**, the post-processing stage begins. Post-processing is mostly optional. During post-processing, code may read data created during simulation and format reports or otherwise handle the results of simulation.

Post-processing finishes with the return of an exit status from **`sc_main()`**. A nonzero return status indicates failure that can be a computed result of post-processing. A zero return should indicate success (i.e., confirmation that the model correctly passed all tests). Many developers neglect this aspect and simply return zero by default. We recommend that you explicitly confirm that the model passed all tests.

[1] This naming convention is not without some controversy in some programming circles; however, most groups have accepted it and deal with the name mismatch.

Callbacks may be used to intercept some points in the preceding stages. For instance, **end_of_elaboration()**, **start_of_simulation()**, and **end_of_simulation()**. These are discussed later in the book.

We now turn our attention to the components used to create a system model.

4.2 The Basic Unit of Design: SC_MODULE

Complex systems consist of many independently functioning components. These components may represent hardware, software, or any physical entity. Components may be large or small, and often contain hierarchies of smaller components. The smallest components represent behaviors and state. In SystemC, we use a concept known as the **SC_MODULE** to represent components.

DEFINITION: A SystemC module is the smallest container of functionality with state, behavior, and structure for hierarchical connectivity.

A SystemC module is simply a C++ class definition. For convenience, the macro **SC_MODULE** is used to declare the class in Fig. 4.4.

```
#include <systemc>
SC_MODULE(module_name) {
  MODULE_BODY
};
```

Fig. 4.4 Syntax of **SC_MODULE**

where **SC_MODULE** is a simple cpp[2] macro as shown in Fig. 4.5. Don't forget the trailing semicolon, which is a fairly common error.

```
#define SC_MODULE(module_name) \
        struct module_name: public sc_module
```

Fig. 4.5 **SC_MODULE** macro definition

We prefer the following method for coding an **SC_MODULE** as illustrated in Fig. 4.6.

```
#include <systemc>
class module_name : public sc_module {
public:
  MODULE_BODY
};
```

Fig. 4.6 Syntax without the **SC_MODULE** macro

[2] cpp is the C/C++ pre-processor that handles # directives such as # **define**.

Within this derived module class, a variety of elements make up the *MODULE BODY:*

- Ports
- Member channel instances
- Member data instances
- Member module instances (sub-designs)
- Constructor
- Destructor
- Simulation process member functions (processes)
- Other methods (i.e., member functions)

Of these, only the constructor is required. However, to contain any useful behavior in your design, you must include either a process or a sub-design. We will first look at the constructor, followed by a simple process. This sequence lets us finish with a basic example of a minimal design and a few alternatives.

4.3 The SC_MODULE Class Constructor: SC_CTOR

Since **SC_MODULE** is a C++ class, it requires a constructor. The **SC_MODULE** constructor performs several tasks specific to SystemC. These tasks include:

- Initializing/allocating sub-designs
  - – Chapter 10 Structure
- Connecting sub-designs
  - – Chapter 10 Structure
  - – Chapter 11 Connectivity
- Registering processes with the SystemC kernel
  - – Chapter 6 Concurrency
- Providing static sensitivity
  - – Chapter 6 Concurrency
- Miscellaneous user-defined setup

To simplify coding, SystemC provides the macro, **SC_CTOR**(). The syntax using this macro follows in Fig. 4.7.

In a simple design, you may only require process registration and setup, whereas in complicated designs, you may need to include multiple designs as well as multiple processes. Let us now examine processes, and see how they fit into the design concept.

```
SC_MODULE(module_name) {
  SC_CTOR(module_name)
  : Initialization // C++ initialization list
  {
    Subdesign_Allocation
    Subdesign_Connectivity
    Process_Registration
    Miscellaneous_Setup
  }
};
```

Fig. 4.7 Syntax of **SC_CTOR**

4.4 The Basic Unit of Execution: Simulation Process

The SystemC simulation process is the basic unit of execution. All simulation processes are registered with the SystemC simulation kernel and are called by the kernel, and *only* from the SystemC simulation kernel. We discuss the SystemC simulation kernel in excruciating detail in a following chapter on concurrency. From the time the simulator begins until simulation ends, all executing code is initiated from one or more processes. Simulation processes appear to execute concurrently.

DEFINITION: A SystemC simulation process is a method (member function) of an **SC_MODULE** that is invoked by the scheduler in the SystemC simulation kernel.

The prototype of a basic simulation process for SystemC is:

```
void PROCESS_NAME(void);
```

Fig. 4.8 Syntax of SystemC process

There are several kinds of simulation processes, and we will discuss all of them eventually. For the purpose of simplification, we will look only at the most basic simulation process type in this chapter.

The most straightforward type of process to understand is the SystemC thread, **SC_THREAD**. Conceptually, a SystemC thread is similar to a single software thread.

A SystemC simulation is a simple C/C++ program. There is only one thread running for the entire program. The SystemC simulation kernel, on the other hand, allows the illusion that many SystemC simulation threads are executing in parallel, as we shall learn in the chapters on concurrency.

A simple **SC_THREAD** begins execution when the scheduler calls it. An **SC_THREAD** may also suspend itself, but we will discuss that topic in the next two chapters.

4.5 Registering the Basic Process: SC_THREAD

Once you have defined a process method in the module, you must identify and register it with the simulation kernel. This step allows the thread to be invoked by the simulation kernel's scheduler. The registration occurs within the module class constructor, SC_CTOR, as previously indicated in Fig. 4.7.

To register a SystemC thread, use the cpp macro **SC_THREAD** inside the constructor as shown in Fig. 4.9.

```
SC_THREAD(process_name);//Must be INSIDE constructor
```

Fig. 4.9 Syntax of **SC_THREAD**

The *process_name* is the name of the corresponding method of the class and also needs to be declared as a method within the **SC_MODULE** class. Figure 4.10 is a complete example of an **SC_THREAD** defined within a module:

```
//FILE: basic_process_ex.h
SC_MODULE(basic_process_ex) {
  SC_CTOR(basic_process_ex) {
    SC_THREAD(my_thread_process);
  }
  void my_thread_process(void);
};
```

Fig. 4.10 Example of basic **SC_THREAD**

Traditionally, the code above is placed in a header file that has the same name as the module and has a `.h` filename extension. Thus, the preceding example could appear inside a file named `basic_process_ex.h`.

Notice that `my_thread_process` is not implemented in the module definition, but only declared. In the manner of C++, it is legal to implement the member function within the class, but implementations are traditionally placed in a separate file, the `.cpp` file.

It is also possible to place the implementation of the constructor in the `.cpp` file, as we shall see in the next section. In the following example, we show an implementation for the `my_thread_process` in the implementation file `basic_process_ex.cpp`.

```
//FILE: basic_process_ex.cpp
void basic_process_ex::my_thread_process(void) {
  cout << "my_thread_process executed within "
       << name() //returns sc_module instance name
       << endl;
}
```

Fig. 4.11 Example of basic **SC_THREAD** implementation

Testbench code typically uses **SC_THREAD** processes to accomplish a series of tasks and to eventually stop the simulation. On the other hand, high-level abstraction hardware models commonly include infinite loops to model hardware logic. It is a requirement that such loops explicitly hand over control to other parts of the simulation. This topic will be discussed in a later chapter on concurrency.

4.6 Completing the Simple Design: main.cpp

Now we complete the design with an example of the top-level file for `basic_process_ex`. The top-level file for a SystemC model is placed in the traditional file, `main.cpp` as illustrated in Fig. 4.12.

Notice the string name constructor argument "my_instance" in the preceding example. The reason for this apparent duplication is to store the name of the instance internally for use when debugging. The **SC_MODULE** class member func-

```
//FILE: main.cpp
int sc_main(int argc, char* argv[]) { // args unused
  basic_process_ex my_instance("my_instance");
  sc_start();
  return 0; // unconditional success (not
            // recommended)
}
```

Fig. 4.12 Example of simple **sc_main**()

tion **name**() may be used to obtain the name of the current instance to print information or debug messages.

4.7 Alternative Constructors: SC_HAS_PROCESS

Before leaving this chapter on modules, we need to discuss an alternative approach to creating constructors. The alternative approach uses a cpp macro named **SC_HAS_PROCESS**. An explanation of the name will become clear in the chapter on concurrency.

You can use this macro in two situations. First, use **SC_HAS_PROCESS** when you require constructors with arguments beyond just the SystemC module instance name string passed into **SC_CTOR** (e.g., to provide configurable modules). Second, use **SC_HAS_PROCESS** when you want to place the constructor in the implementation (i.e., `.cpp`) file.

You can use constructor arguments to specify sizes of included memories, address ranges for decoders, FIFO depths, clock divisors, FFT depth, and other configuration information. For instance in Fig. 4.13, a memory design might allow selection of different sizes of memories with an argument:

```
My_memory instance("instance", 1024);
```

Fig. 4.13 Example of **SC_HAS_PROCESS** instantiation

To use this alternative approach, invoke **SC_HAS_PROCESS**, just prior to the definition of your conventional constructor. One caveat applies. You must construct or initialize the module base class, **sc_module**, with an instance name string. This requirement is why **SC_CTOR** has an argument.

There are alternate forms using **SC_HAS_PROCESS**. We will first describe in Fig. 4.14 a prevalent style with all of the constructor code defined in the header file. We will then present our preferred approach, which cleanly separates declaration from definition.

```
//FILE: module_name.h
SC_MODULE(module_name) {
  SC_HAS_PROCESS(module_name);
  module_name(sc_module_name
              instname[, other_args…])
  : sc_module(instname)
  [, other_initializers]
  {
    CONSTRUCTOR_BODY
  }
};
```

Fig. 4.14 Syntax of **SC_HAS_PROCESS** in the header

The syntax for using **SC_HAS_PROCESS** in a separate implementation (i.e., separate compilation situation) is similar as shown in Fig. 4.15 and Fig. 4.16. **SC_HAS_PROCESS** can also reside in additional locations but the authors prefer to keep it in the module class definition and close to the constructor or constructor declaration.

In the preceding examples, the *other_args* are optional.

```
//FILE: module_name.h
SC_MODULE(module_name) {
  module_name(sc_module_name
              instname[,other_args…]);
};
```

Fig. 4.15 Syntax of **SC_HAS_PROCESS** separated

```
//FILE: module_name.cpp
SC_HAS_PROCESS(module_name);
module_name::module_name(
  sc_module_name instname[, other_args…])
: sc_module(instname)
[, other_initializers]
{
  CONSTRUCTOR_BODY
}
```

Fig. 4.16 Syntax of **SC_HAS_PROCESS** in the implementation file

4.8 Two Styles Using SystemC Macros

We finish this chapter with two styles for coding SystemC designs. First, we provide the more traditional style, which depends heavily on headers. Second, our recommended style places more elements into the implementation. Creating a C++ templated module usually precludes this style due to C++ compiler restrictions.

You may use either one of these styles for your project, though separating your implementation code in a different file will have certain advantages. We'll visit these topics again in more detail when we discuss the details of hierarchy and structure.

4.8.1 The Traditional Coding Style

The traditional style illustrated in Fig. 4.17 and Fig. 4.18 places all the instance creation and constructor definitions in the header (`.h`) files. Only the implementation of processes and helper functions are coded in the compiled (`.cpp`) file.

```
#ifndef NAME_H
#define NAME_H
#include "submodule.h"
…
SC_MODULE(NAME) {
    Port declarations
    Channel/submodule instances
    SC_CTOR(NAME)
    : Initializations
     {
        Connectivity
        Process registrations
    }
    Process declarations
    Helper declarations
};
#endif
```

Fig. 4.17 Traditional style ***NAME*.h**

```
#include <systemc>
#include "NAME.h"
NAME::Process {implementations }
NAME::Helper {implementations }
```

Fig. 4.18 Traditional style ***NAME*.cpp**

Let's remind ourselves of the basic components in each file. First, the **#ifndef/#define/#endif** preprocessor directives prevent compile problems when the header file is included in multiple files. Using *NAME*_H definition is a standard name for the conditional directive. This definition is followed by file inclusions of any submodule header files by way of **#include**.

Next, the **SC_MODULE**{...}; defines the class definition. Within the class definition, ports are usually the first constructs declared because they represent the interface to the module. Local channels and submodule instances come next.

Next, we place the class constructor, and optionally the destructor. In most cases, using the **SC_CTOR**() {...}; macro proves sufficient in declaring the constructor. Note that **SC_CTOR**() {...}; implies **SC_HAS_PROCESS** and can only be used when no additional constructor arguments are needed. The body of the constructor usually includes initializations, connectivity of submodules, and registration of processes. The constructs mentioned will be discussed in greater detail in the following chapters.

The header finishes out with the declarations of processes, helper functions and possibly other private data. Note that C++ and SystemC do not dictate the ordering of the elements within the class declaration.

The body of a traditional style for the implementation simply includes the SystemC header file, and the corresponding module header described above. The rest of this file simply contains external function member implementations of the processes and functions. Note that it is possible to have no implementation file if there are no processes or helper functions in the module.

4.8.2 Recommended Alternate Style

Here is another style that has some advantages over the preceding style and is illustrated in Fig. 4.19 and Fig. 4.20.

First, the header contains the same **#define** and **SC_MODULE** components as the traditional style. The differences reside in how the channel and submodule definitions are implemented and how the constructor is placed into the implementation body. Notice in the first case that the channel and submodules are implemented using pointers instead of direct instantiation. This method allows for dynamic design configuration that is not possible with direct instantiation. In the second case, placing the constructor code in the implementation file hides the details from potential users.

```
#ifndef NAME_H
#define NAME_H
Submodule forward class declarations
SC_MODULE(NAME) {
    Port declarations
    Channel/Submodule* definitions
    // Constructor declaration:
    SC_CTOR(NAME);
    Process declarations
    Helper declarations
};
#endif
```

Fig. 4.19 Recommended style ***NAME*.h**

```
#include <systemc>
#include "NAME.h"
SC_HAS_PROCESS(NAME);
 NAME::NAME(sc_module_name nm)
: sc_module(nm)
, Initializations
{
    Channel allocations
    Submodule allocations
    Connectivity
    Process registrations
}
NAME::Process {implementations }
NAME::Helper {implementations }
```

Fig. 4.20 Recommended style **NAME.cpp**

4.9 Exercises

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

Exercise 4.1: Compile and run the `basic_process_ex` example from the web site. Add an output statement before **`sc_start()`** indicating the end of elaboration and beginning of simulation.

Exercise 4.2: Rewrite `basic_process_ex` using **`SC_HAS_PROCESS`**. Compile and run the code.

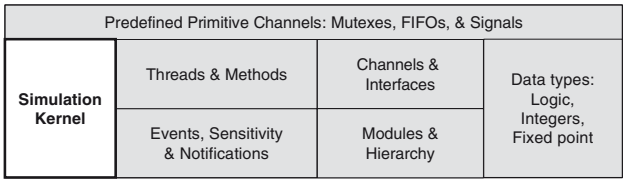
Exercise 4.3: Create two concurrent threads by adding a second **`SC_THREAD`** to `basic_process_ex`. Be sure the output message is unique. Compile and run.

Exercise 4.4: Create two design instances (and hence two concurrent threads) by adding a second instantiation of `basic_process_ex`. Compile and run.

Exercise 4.5: Write a module from scratch using what you know. The output should count down from 3 to 1 and display the corresponding words “Ready”, “Set”, “Go” with each count. Compile and run.

Try writing the code without using **SC_MODULE**. What negatives can you think of for not using **SC_MODULE**? [HINT: Think about EDA vendor-supplied tools that augment SystemC.]

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.

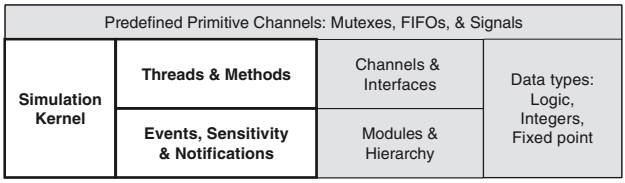
Exercise 5.2: Modify `time_flies` to see how much time you can model (days? months?). See how it changes with the time resolution.

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.

Chapter 6
Concurrency

Processes and Events

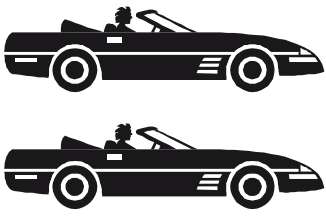


This chapter examines the topic of concurrency, which is fundamental to simulating with SystemC. We will first take a look at some types of concurrency, and then we will visit the simulation kernel used by SystemC. This examination will let us discuss SystemC thread processes, events, and sensitivity.

6.1 Understanding Concurrency

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 a thread 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 **simulation processes** to model concurrency. As with most event-driven simulators, concurrency is not true concurrent execution. In fact, simulated concurrency works like cooperative multitasking. When a simulation process runs, it is expected to execute a small segment of code and then return control to the simulation kernel. The SystemC simulator depends on a cooperative multitasking (non-preemptive) kernel that cannot force a running process to return control. This feature is unlike many operating systems that preemptively interrupt running processes to switch to a different process. A poorly behaved process that hogs control and

D.C. Black et al., *SystemC: From the Ground Up*, Second Edition,
DOI 10.1007/978-0-387-69958-5_6, © Springer Science+Business Media, LLC 2010

doesn't yield control to the simulation kernel will cause SystemC simulation to hang. We'll look closer at this subject of control after we explain how SystemC uses C++ to model the process.

SystemC simulation processes are simply C++ functions designated by the programmer to be used as processes. You simply tell the simulation kernel which functions are to be used as simulation processes. This action is known as process registration. The simulation kernel then schedules and calls each of these functions as needed.

We've already seen how to register one type of process back in the chapter on SystemC modules using the **SC_THREAD**. Its syntax was simple as shown below (Fig. 6.1).

```
SC_THREAD(MEMBER_FUNCTION);
```

Fig. 6.1 Syntax of **SC_THREAD** macro

SC_THREAD has a few restrictions. First, it can be used only within a SystemC module; hence, the function must be a member function of the module class. Second, it must be used only during the elaboration stage, which we will talk about shortly. Suffice it to say that placing **SC_THREAD** in the module's constructor meets this requirement. Lastly, the member function must exist and the function can take no arguments and return no values. In other words, the function argument list is **void**, and the return value is **void**.

Let go back to the issue of control. We stated that processes must voluntarily yield control. Yielding control may take one of two forms. For one form, simulation processes yield control by executing a **return**. For the processes registered with **SC_THREAD**, executing **return** is uninteresting because it means the process has ended permanently.

The more interesting form is calling SystemC's **wait()** function. The **wait()** function suspends the process temporarily while SystemC proceeds to execute other processes, and then the function resumes by returning.

We've already seen one syntax for **wait(***delay_sc_time***)** in the last chapter. There are several more syntaxes, but we will only introduce one more for the moment (Fig. 6.2), which relates to time delays.

```
wait(TIME_DELAY,TIME_UNITS); // Convenience
```

Fig. 6.2 Another syntax of **wait()**

While we're on the subject of waiting for a time delay, note that waiting on a negative time does not make sense and is not defined in the standard. In any case, you should avoid using negative time delays. Simulators should probably issue a fatal error for this situation.

Let's see how normal time-out waits affect simulation. Here (Fig. 6.3) is a simple example with two processes. Both processes uses **wait**(*TIME_DELAY*) to simulate the passage of time. One process models a windshield wiper. The other process models the emergency blinkers in a similar manner.

```
//FILE: two_processes.h
SC_MODULE(two_processes) {
  void wiper_thread(void); // process
  void blinker_thread(void); // process
  SC_CTOR(two_processes) {
    SC_THREAD(wiper_thread); // register process
    SC_THREAD(blinker_thread); // register process
  }
};
```

```
//FILE: two_processes.cpp
void two_processes::wiper_thread(void) {
  while (true) {
    wipe_left();
    wait(500,SC_MS);
    wipe_right();
    wait(500,SC_MS);
  }//endwhile
}

void two_processes::blinker_thread(void) {
  while (true) {
    blinker = true;
    cout << "Blink ON" << endl;
    wait(300,SC_MS);
    cout << "Blink OFF" << endl;
    blinker = false;
    wait(300,SC_MS);
  }//endwhile
}
```

Fig. 6.3 Two processes using **wait()**

It is instructive to compile and run this code. You should download the examples that accompany this book and run the *two_processes* example. Notice how both processes use infinite loops. The loops are useful because if the processes ever return, they will never run again.

SystemC thread simulation processes typically begin execution at the start of simulation and continue in an endless loop until the simulation ends. Thread processes are started once; if they terminate, they cannot be restarted. Thread processes are required to periodically return control to the simulation kernel, allowing other processes to run. A thread process returns control to the simulation kernel by executing a **wait()** method call specifying an event or a time-out. Each time a thread returns control to the simulation kernel, its execution state is saved, which lets the process be resumed when the **wait()** returns.

6.2 Simplified Simulation Engine

Before continuing into the details of the SystemC syntax, we should examine the operation of the SystemC simulator. In this section, we explain thread processes using a simplified version (Fig. 6.4) of the simulation kernel. The remaining details will be covered later in this and other chapters. SystemC simulations consist of four major stages: elaboration, initialization, simulation, and post-processing.

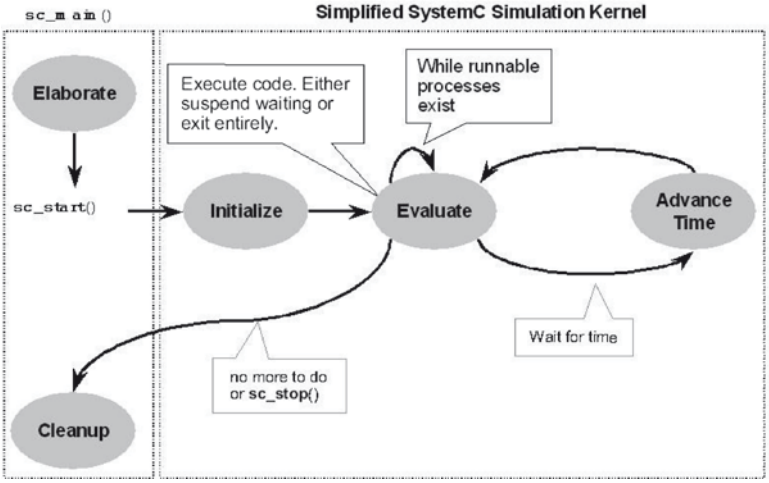


Fig. 6.4 Simplified SystemC simulation engine

In the previously described elaboration stage, SystemC components are instantiated and connected to create a model ready for simulation. This stage is also where process registration occurs. The elaboration stage ends with a call to `sc_start()`, which invokes the simulation kernel and begins the initialization stage.

In the initialization stage, the simulation kernel identifies all simulation processes and places them in either the runnable or waiting process set as illustrated in the next figure. By default, most simulation processes are placed into the set of runnable processes. Those processes explicitly requesting no initialization are placed into the set of waiting processes.

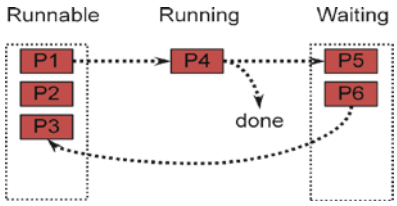


Fig. 6.5 Process sets

The simulation stage is commonly described as a state machine that schedules processes to run and advances simulation time. In this simplified simulation stage there are two internal phases: **`evaluate`** and **`advance-time`**. Simulation begins with evaluate.

During **`evaluate`**, all runnable processes are run one at a time. Each process runs until either it executes a **`wait`**(*`TIME_DELAY`*) or a return. If a **`wait`**(*`TIME_DELAY`*) is executed, then the time information is stored as an upcoming time event in the scheduling priority queue shown in the next (figure 6.6). The evaluate continues until there are no runnable processes left.

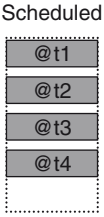


Fig. 6.6 Scheduling priority queue

It is important to note that the ordering of processes to run is unspecified in the standard. This means that when selecting which process to run from the set of runnable processes, any process may be chosen. Remember, we are simulating concurrency, which means that ideally we would run all of the processes simultaneously; however, we have only one processor to actually execute the simulation and so one process at a time is chosen. Different implementations of the simulator may run processes in different orderings; however, a given implementation must run processes in the same order to allow repeatability.

Once the set of all runnable processes has been emptied, then control passes to the advance-time phase to advance simulation time. Simulated time is moved forward to the closest time with a scheduled event. Time advancement moves processes waiting for that particular time into the runnable set, allowing the evaluation phase to continue.

This progression from evaluate to advance-time continues until one of three things occurs:

- All processes have yielded (i.e., there's nothing in the runnable set)
- A process has executed the function **`sc_stop()`**
- Internal 64-bit time variable runs out of values (i.e., maximum time is reached)

At this point, simulation stops and we proceed into the post-processing stage (AKA cleanup).

We'll now turn our attention to some details of the processes.

6.3 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 Fig. 6.7. 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(**t_1**)**, **wait(**t_2**)**,…)

```
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);
}
```

Fig. 6.7 Four processes

Notice that `Process_D` skips t_2. At first glance, it might be perceived that time passes as shown below in Fig. 6.8. The uninterrupted solid and gray line portions indicate program activity. Vertical discontinuities indicate a wait.

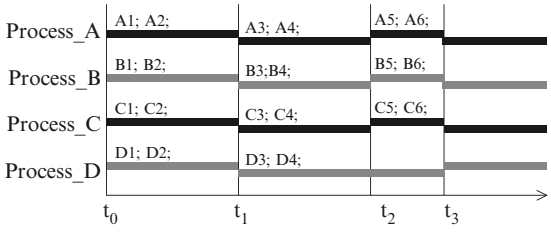


Fig. 6.8 Simulated activity—perceived

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 Fig. 6.9.

Each set of statements executes in zero time! Let's expand the time scale to expose the simulator's internal activity as well. This expansion exposes the operation of the scheduler at work (Fig. 6.10).

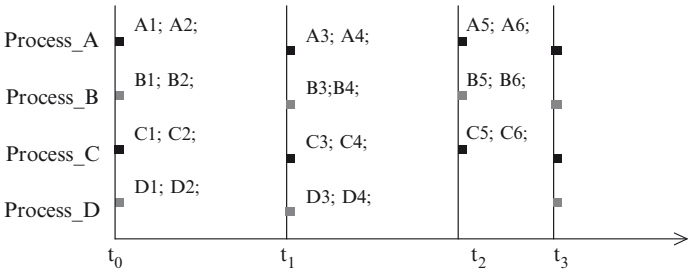


Fig. 6.9 Simulated activity—actual

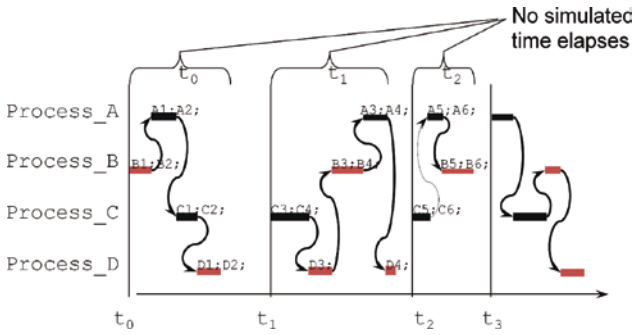


Fig. 6.10 Simulated activity with simulator time expanded

Notice that process execution order appears random in this example. This apparent random sequencing is allowed 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.

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.

6.4 The SystemC Thread Process

SystemC has two basic types of simulation processes: thread processes and method processes. Thread processes are the easiest to code and are the most popular for SystemC applications. They are named thread processes because their behavior most closely models the usual software connotation of a thread of execution. We look at the SystemC method processes later in this chapter.

SystemC thread processes begin execution at the start of simulation and typically continue in an endless loop until the simulation ends. SystemC thread processes are started once and only once by the simulator.

As we have noted, a running thread process has complete control of the simulation until it decides to yield control to the simulator kernel. A thread process can return control in two ways. First, by waiting, which suspends the process to be resumed later, and second, by simply exiting (returning). A thread that exits cannot be resumed and will not be restarted for the duration of the simulation.

Thread processes that implement endless loops must have at least one explicit or implicit call to the **`wait()`** function. One or more calls to **`wait()`** may be included in the endless loop; these calls are explicit waits. A wait call may be contained in a function called from the threads; this is called an implicit wait. For instance, a blocking **`read()`** call to an instance of the **`sc_fifo`**<*T*> invokes **`wait()`** when the FIFO is empty.

6.5 SystemC Events

Before discussing threads more extensively, it is necessary to discuss events. SystemC is an event-driven simulator. In fact, we have already discussed one type of event: the time-out event that occurs implicitly with the **`wait`***(TIME_DELAY)* syntax.

An event is something that happens at a specific instant in time. An event has no value and no duration. SystemC uses the **`sc_event`** class to model events. This

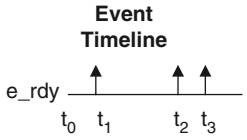


Fig. 6.11 Graphical representation of an event

class allows explicit triggering (launching, firing, causing) 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 instant 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 waiting for the event. We will explain how processes wait a few sections ahead. The next diagram (Fig. 6.11) illustrates an event **`e_rdy`** triggering at three different instants. Note that unlike a waveform, events have no time width.

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 prior to its notification.

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 and shown in Fig. 6.12.

Remember that an **`sc_event`** has no value, and you can perform only two actions with a **`sc_event`**: wait for it or cause it to occur.

```
sc_event event_name1[,event_namei]…;
```

Fig. 6.12 Syntax of **sc_event**

```
event_name.notify(void);          // Immediate
event_name.notify(SC_ZERO_TIME);// Delayed
event_name.notify(sc_time);       // Timed (time>0)
event_name.notify(double,units);// Convenience
```

Fig. 6.13 Syntax of **sc_event::notify()**

6.5.1 Causing Events

Events are explicitly caused using the **notify()** method of an **sc_event** object. Here (Fig. 6.13) is the syntax:

Invoking an immediate **notify(void)** causes any processes waiting for the event to be immediately moved from the waiting set into the runnable set for execution. This topic will be examined in more detail in an upcoming section.

Delayed notification occurs when a time of zero is specified. The predefined constant **SC_ZERO_TIME** is simply **sc_time(0,SC_SEC)** Processes waiting for a delayed notification will execute only after all runnable processes in the current evaluation state have executed. For now, it is sufficient to consider that delayed notification is treated the same as a timed notification with a time of zero. This feature is quite useful as we will see later.

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. For example, consider the following (Fig. 6.14):

```
sc_event A_event;
A_event.notify(10,SC_NS);
A_event.notify( 5,SC_NS); // only this one stays
A_event.notify(15,SC_NS);
```

Fig. 6.14 Syntax of **notify()** method

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

Fig. 6.15 Syntax of **cancel()** method

Outstanding scheduled events may be canceled with the **cancel()** method (Fig. 6.15). Note that immediate events cannot be canceled because they happen at the precise instant they are notified (i.e., immediately).

6.6 Catching Events for Thread Processes

Thread processes rely on the **wait**() method to suspend their execution. The **wait**() method supplied by SystemC has several syntaxes shown in the next figure, Fig 6.16.

When the wait function is called, control is returned to the simulator kernel, the state of the current thread process is saved, and eventually a new process is allowed to run.

When a suspended thread process is selected to run, the simulation kernel restores the calling context, and the process resumes execution at the statement following the call to **wait**().

```
wait(time);                  // timeout is the event
wait(double,time_unit);      // convenience
wait(event);                 // single event
wait(event1 | eventn…);      // any of these
wait(event1 & eventn…);      // all of these
wait(time,event);            // event or timeout
wait(time,event1 | eventn…);// any event or timeout
wait(time,event1 & eventn…);// all events or timeout
wait(); // static sensitivity - discussed later
```

Fig. 6.16 Syntax of **SC_THREAD wait()**

The first two syntaxes for **wait** (*time*) provide a delay for a period of simulation time as described in the Chapter 5.

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.

The three forms that have time with a second argument constitute a time-out. This result is really just the logical **or** of a time event with other events. Use of a

```
…
sc_event ack_event, bus_error_event;
…
sc_time start_time(sc_time_stamp());
wait(t_MAX_DELAY, ack_event | bus_error_event);
if (sc_time_stamp()-start_time == t_MAX_DELAY) {
  break; // path for a time out
  …
```

Fig. 6.17 Example using time-out variation of **wait**()

time-out is handy when testing protocols and various error conditions and an example is given in Fig. 6.17.

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. Thus (Fig. 6.18), it is illegal to test an event for **`true`** or **`false`**.

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

Fig. 6.18 Example of illegal Boolean compare of **`sc_event`**

It is legal to test a flag that is set by the process that caused an event; however, it is problematic to clear this flag properly.

6.7 Zero-Time and Immediate Notifications

Two concepts, zero-time delays and immediate notification, bear special treatment. Consider again the simulation engine diagram below Fig. 6.19 (reproduced for convenience).

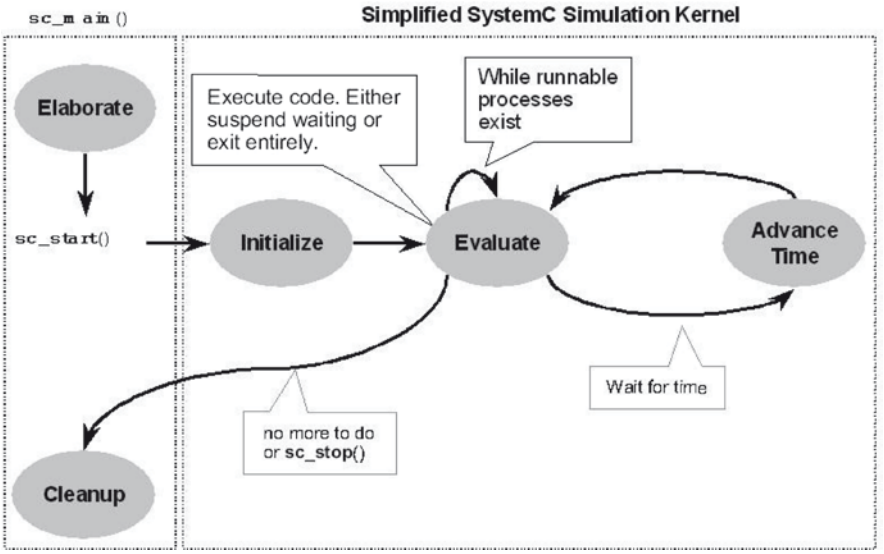


Fig. 6.19 Simplified SystemC simulation engine

Consider what it means to execute **`wait(SC_ZERO_TIME)`**. What does it mean to advance time by zero? The primary effect is that a process waiting for zero-time will resume after all the runnable processes have yielded. Since zero is

always closer than any other time, then all processes waiting for zero-time will be next in line to become runnable.

```
SC_MODULE(missing_event) {
  SC_CTOR(missing_event) {
    SC_THREAD(B_thread); // ordered to cause
    SC_THREAD(A_thread); // problems
    SC_THREAD(C_thread);
  }
  void A_thread() {
    a_event.notify(); // immediate!
    cout << "A sent a_event!" << endl;
  }
  void B_thread() {
    wait(a_event);
    cout << "B got a_event!" << endl;
  }
  void C_thread() {
    wait(a_event);
    cout << "C got a_event!" << endl;
  }
  sc_event a_event;
};
```

Fig. 6.20 Example of zero-time

This feature can be very useful. Recall the rule that to observe an event, the observer must be watching for the event prior to its notification. Now imagine we have three processes, `A_thread, B_thread,` and `C_thread`. The implementation code is shown in Fig. 6.20.

Suppose they execute in the order `A_thread, B_thread, C_thread`. Furthermore, notice that `A_thread` does an immediate notification of an event, `a_event`, which `B_thread and C_thread` are going to wait for.

If either `B_thread` or `C_thread` have not issued the **`wait`**`(a_event)` call prior to `A_thread` notifying the event, then they will miss the event. If the event never happens again, then when `B_thread` or `C_thread` issue the **`wait`**`(a_event)` call, they will wait forever.

If the event happened a second time, then `B_thread` or `C_thread` would continue, but they would have missed one of the events. Missing an event can be devastating to a simulation. This situation can be avoided by use of the zero-time delayed notification, **`notify(SC_ZERO_TIME)`**. The reason is that delayed notifications are issued only after completing all runnable processes.

Lest you think that you can simply fix the problem by ordering the processes appropriately, recall that SystemC implementations are free to choose processes from

the runnable set in any order. This unpredictability of selection is because we are simulating concurrency. Also, consider that in a real-world simulation, there may be hundreds of processes. The bottom line is that you must write your code so that it does not depend on the order of process execution except by design (e.g., using an event to force ordering). Consider the example in Fig. 6.21 that correctly handles this case.

```
SC_MODULE(ordered_events) {
  SC_CTOR(ordered_events) {
    SC_THREAD(B_thread); // ordered to cause
    SC_THREAD(A_thread); // problems
    SC_THREAD(C_thread);
  }
  void A_thread() {
    while (true) {
      a_event.notify(SC_ZERO_TIME);
      cout << "A sent a_event!" << endl;
      wait(c_event);
      cout << "A got c_event!" << endl;
    }//endwhile
  }
  void B_thread() {
    while (true) {
      b_event.notify(SC_ZERO_TIME);
      cout << "B sent b_event!" << endl;
      wait(a_event);
      cout << "B got a_event!" << endl;
    }//endwhile
  }
  void C_thread() {
    while (true) {
      c_event.notify(SC_ZERO_TIME);
      cout << "C sent c_event!" << endl;
      wait(b_event);
      cout << "C got b_event!" << endl;
    }//endwhile
  }
  sc_event a_event, b_event, c_event;
};
```

Fig. 6.21 Example of properly ordered events

Here is another difficulty that can arise when using immediate notification. When a process executes **`notify(void)`**, it may cause one or more processes in the waiting set to be moved into the runnable set. Consider the example in Fig. 6.22.

```
SC_MODULE(event_hogs) {
  SC_CTOR(event_hogs) {
    SC_THREAD(A_thread);
    SC_THREAD(B_thread);
    SC_THREAD(C_thread);
    // Following ensures sc_stop() works
    sc_set_stop_mode(SC_STOP_IMMEDIATE);
  }
  sc_event a_event, b_event;
  void A_thread() {
    while(true) {
      cout << "A@" << sc_time_stamp() << endl;
      a_event.notify(); // immediate!
      wait(b_event);
    }
  }
  void B_thread() {
    int count(8); // limit execution
    while(true) {
      cout << "B@" << sc_time_stamp() << endl;
      b_event.notify(); // immediate!
      wait(a_event);
      if (count-- == 0) sc_stop();
    }
  }
  void C_thread() {
    while(true) {
      cout << "C@" << sc_time_stamp() << endl;
      wait(1,SC_NS);
    }
  }
};
```

Fig. 6.22 Example of improper use of events

6.8 Understanding Events by Way of Example

The best way to understand events is by way of example. Notice in the code of Fig. 6.23 that all notifications execute at the same instant.

```
…
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 evaluate 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 femto sec from now
action.notify(20,SC_FS);
…
```

Fig. 6.23 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 example illustrated in Fig. 6.24 models a mechanism that detects switching activity and notifies the appropriate indicator. For simplicity, only the stop light 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. The `turn_knob_thread` process 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.

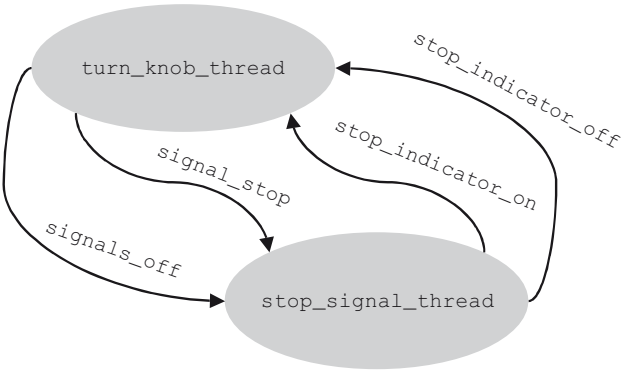


Fig. 6.24 Turn of events illustration

An interesting aspect of the example shown in Fig. 6.25 through Fig. 6.27 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.

```
//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
```

Fig. 6.25 Example of **`turn_of_events`** header

```
//FILE: turn_of_events.cpp
void turn_of_events::turn_knob_thread() {
  // This process provides stimulus to the design
  // by way of standard I/O.
  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);
  while(true) {
    // Sit in an infinite loop awaiting keyboard
    // or STDIN input to drive the stimulus…
    cout << "Signal command: ";
    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()
```

Fig. 6.26 Example of **`turn_of_events`** stimulus

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

Fig. 6.27 Example of **turn_of_events** indicator

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

Fig. 6.28 Example of **turn_of_events** output

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.

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`.

6.9 The SystemC Method Process

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** may be slightly more efficient in some situations 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.

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 indirectly from an **SC_METHOD** will result in a run-time error.

Implicit waits result from calling functions that are defined such that they may issue a **wait()**. These are known as blocking methods. As discussed later in this book, the **read()** and **write()** methods of the **sc_fifo**<*T*> data type are examples of blocking methods. Thus, **SC_METHOD** processes must avoid using calls to blocking methods.

The syntax for **SC_METHOD** processes follows (Fig. 6.29) and is identical to **SC_THREAD** except for the keyword **SC_METHOD**:

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

Fig. 6.29 Syntax of **SC_METHOD**

A note on the choice of these keywords might be useful. The similarity of names between an **SC_METHOD** process and a regular object-oriented method betrays its name. An **SC_METHOD** executes without interruption and returns to the caller (the simulation kernel). By contrast, an **SC_THREAD** process is more akin to an operating system thread, which suspends (waits) returning control to the simulation kernel and later the kernel can resume that thread process.

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.

```
next_trigger(time);
next_trigger(timeout,time_unit); //convenience
next_trigger(event);
next_trigger(event1 | event_i…); //any of these
next_trigger(event1 & event_i…); //all of these
                                 //required
next_trigger(timeout,event); //event with timeout
next_trigger(timeout,event1 | event_i…);//any + timeout
next_trigger(timeout,event1 & event_i…);//all + timeout
next_trigger(void); //re-establish static sensitivity
```

Fig. 6.30 Syntax of **SC_METHOD next_trigger()**

6.10 Catching Events for Method Processes

SC_METHOD processes dynamically specify their sensitivity by means of the **next_trigger**() method as shown in Fig 6.30. This method has the same syntax as the **wait**() method but with a slightly different behavior.

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**. The **next_trigger**() method 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 or of the process. The initialization call is vital to making this work. See the **next_trigger()** code in the downloads section of the web site 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.

You might be tempted to place a default **next_trigger**() as the first statement of the **SC_METHOD**, since subsequent calls to **next_trigger**() will overwrite any previous calls. For fairly simple designs, this approach may work; however, there is a potential problem since it could mask problem where you intended a different trigger. It might be better to both allow the possibility of hanging and insert some code to determine if the process is active. An even better solution is to use a tool that can statically analyze your code for correctness. Having said that, the next section discusses a technique that guarantees a default **next_trigger**().

6.11 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**(). The concept of actively determining what will cause a process to resume is often called **dynamic sensitivity**.

SystemC provides another type of sensitivity for processes called **static sensitivity**. Static sensitivity establishes the parameters for resumption during elaboration (i.e., before simulation begins). Once established, static sensitivity parameters cannot be changed (i.e., they're static).

The purpose of specifying static sensitivity is simply a convenience. Some processes have a single list of items to which they are sensitive. To preserve code and make this clearer to the reader, use a static list rather than dynamically specifying the same thing over and over again. It is possible to override static sensitivity with

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

Fig. 6.31 Syntax of sensitive

dynamic sensitivity, but this is rarely used. Static sensitivity is most commonly used with the RTL level of coding.

Static sensitivity is established for each process immediately after its process registration. Events may be added to a processes static sensitivity list, which is initially empty, using the insertion operator (<<) on a special **sc_module** data member called **sensitive**. The syntax is shown in Fig. 6.31.

The event object on the right has several variations that we need to defer until we have a more concrete notion of the concepts of channels, ports, and event finders. Suffice it to say that some objects can return events to be added to the sensitivity list.

For method processes, simply not specifying **next_trigger**() at all implies that static sensitivity, if any exists, will be used. Thus static sensitivity is most commonly used with method processes; however, use of the **wait(void)** syntax lets thread processes also use static sensitivity. In fact, **next_trigger**() was originally invented as a technique to alter the method process sensitivity.

```
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();
};
```

Fig. 6.32 Example of gas station declarations

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 as illustrated in Fig. 6.33. The declarations for this example in Fig. 6.32 illustrate the use of the **sensitive** method.

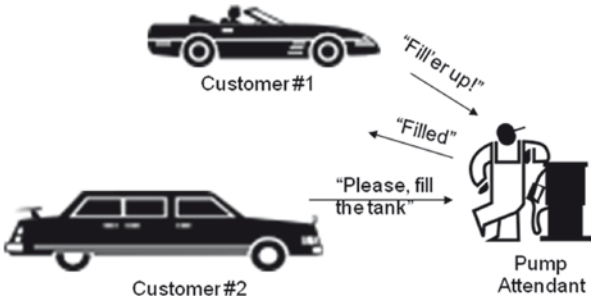


Fig. 6.33 Initial Gas Station Illustration

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 Fig. 6.32. This format helps draw attention to the sensitivity being associated with only the most recent process registration.

Figure 6.34 has 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 web site.

```
…
void gas_station::customer1_thread() {
  while (true) {
    wait(EMPTY_TIME);
    cout << "Customer1 needs gas" << endl;
    m_tank1 = 0;
    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()
```

Fig. 6.34 Example of gas station implementation

```
…
Customer1 needs gas
Filling tank
Filled tank
…
```

Fig. 6.35 Example of gas station sample output

6.12 Altering Initialization

The simulation engine description specifies that processes are executed at least once initially by placing processes in the runnable set during the initialization stage. This approach makes no sense in the preceding `gas_station` model as the `attendant_method` would fill the tank before being requested.

Thus, it may be necessary to specify that some processes should not be made runnable at initialization. For this situation, SystemC provides the **`dont_initialize`**`()` method. The syntax follows:

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 looks like Fig. 6.37.

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

Fig. 6.36 Syntax of **`dont_initialize`**`()`

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

Fig. 6.37 Example of **`dont_initialize`**`()`

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

Fig. 6.38 Syntax of **`sc_event_queue`**

6.13 The SystemC Event Queue

In light of the limitation that **sc_event** may only have a single outstanding event scheduled, the **sc_event_queue** was added with the syntax shown in Fig. 6.38. This addition allows a single event to be scheduled multiple times even for the same instant in time! When events are scheduled for the same instant in time, each happens in a separate evaluation.

There are certain situations in which one may wish to catch multiple events; however, often it may indicate a misunderstanding of the model. Most modeling only requires the **sc_event**. Furthermore, **sc_event_queue** has a performance impact. Since events impact a huge portion of simulation time, performance in this area needs to be carefully considered.

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.

The following diagram may clarify what the preceding code does:

```
sc_event_queue action;
wait(10,SC_NS)//assert time=10ns
sc_time now1(sc_time_stamp());//observe current time
action.notify(20,SC_NS);//schedule for 20ns from now
action.notify(10,SC_NS);//schedule for 20ns from now
action.cancel_all();//cancel all actions entirely
action.notify(8,SC_NS);//schedule for 8 ns from now
action.notify(1.5,SC_NS);// 1.5 ns from now
action.notify(1.5,SC_NS);// another identical action
action.notify(3.0,SC_NS);// 3.0 ns from now
action.notify(SC_ZERO_TIME);//after all runnable
action.notify(SC_ZERO_TIME);//and yet another
action.notify(12,SC_NS);// 12 ns from now
sc_time now2(sc_time_stamp());//observe current time
```

Fig. 6.39 Example of **sc_event_queue**

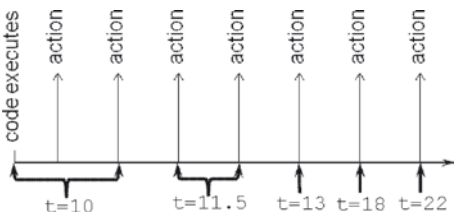


Fig. 6.40 Example of **sc_event_queue**

Notice that the first two events were completely cancelled as was explicitly stated and that all seven remaining events occurred. In fact, the evaluation phase may be entered seven distinct times, if there are processes waiting on the event. Finally, observe that time never passes between now1 and now2 because the code never yielded to the simulation kernel in that section of code. Seven events then remain to be waited upon.

6.14 Exercises

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

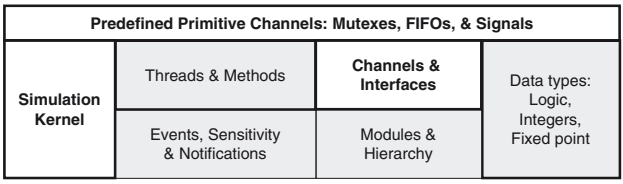
Exercise 6.1: Examine and run the `turn_of_events` example. Remove the **`wait(SC_ZERO_TIME)`** in the `turn_knob_thread` process. Is this guaranteed to give different behavior? If your implementation of SystemC does not create different behavior, insert a **`wait`** (**`SC_ZERO_TIME`**) at the top of **`stop_signal_thread`**. What is the behavior that you observe? Why?

Exercise 6.2: Learn how to delay time using an **`SC_METHOD`**. Examine, predict the behavior, compile, and run the `method_delay` example from the downloaded examples.

Exercise 6.3: This exercise illustrates some of the complexities of **`SC_METHOD`** processes. Examine, predict the behavior, compile, and run the **`next_trigger`** example.

Exercise 6.4: This exercise will clarify the differences between **`sc_event`** and **`sc_event_queue`** constructs. Examine, predict the behavior, compile, and run the `event_filled` example.

Chapter 7
Dynamic Processes



In this chapter, we will focus on a discussion of dynamic threads and **`SC_FORK`**/**`SC_JOIN`** constructs. Dynamic processes can be particularly useful in testbench scenarios to track transaction completion or to spawn traffic generators dynamically. In the following sections, we will cover the syntax required to generate dynamic processes as well as some application examples.

7.1 Introduction

Thus far, all the process types discussed have been static. In other words, once the elaboration phase completes, all **`SC_THREAD`** and **`SC_METHOD`** processes have been established. SystemC 2.1 introduced the concept of dynamically spawned processes.

Dynamic processes are important for several reasons. At the top of the list is the ability to perform temporal checks such as those supported by PSL Sugar, Vera, and other verification languages. For instance, consider a bus protocol with split transactions and timing requirements. Once a request is issued, it is important to track the completion of that transaction for verification of that transaction. Since transactions may be split, each transaction will require a separate thread to monitor. Without dynamic process support, it would be necessary to pre-allocate a number of statically defined processes to accommodate the maximum number of possible outstanding requests.

7.2 sc_spawn

Let's look at the syntax and requirements that enable dynamic processes. First, to enable dynamic processes, it is necessary to use a pre-processor macro prior to the invocation of the SystemC header file. Figure 7.1 is one way to do this:

D.C. Black et al., *SystemC: From the Ground Up*, Second Edition,
DOI 10.1007/978-0-387-69958-5_7, © Springer Science+Business Media, LLC 2010

```
#define SC_INCLUDE_DYNAMIC_PROCESSES
#include <systemc>
```

Fig. 7.1 Syntax to enable dynamic threads

Other mechanisms involve the C++ compilation tools. For example, GNU gcc has a –D option (e.g., -DSC_INCLUDE_DYNAMIC_PROCESSES), which predefines the macro.

Next, declare the functions to be spawned as processes. Functions may be either normal functions or methods (i.e., member functions of a class). The dynamic facilities of SystemC allow for either **SC_THREAD** or **SC_METHOD** style processes. Unlike static processes, dynamic processes may have up to eight arguments and a return value. The return value will be provided via a reference variable in the actual spawn function. For example, consider the declarations in Fig. 7.2.

```
// Ordinary function declarations
void inject(void); // no args or return
int count_changes(sc_signal<int>& sig);

// Method function declarations
class TestChan : public sc_module {
  …
  bool Track(sc_signal<packet>& pkt);
  void Errors(int maxwarn, int maxerr);
  void Speed(void);
  …
};
```

Fig. 7.2 Example functions used as dynamic processes

Having declared a function (possibly a member function) to be used as spawned process, you need to define the implementation and register the function with the kernel. You can register the dynamic processes within an **SC_THREAD** or with restrictions within an **SC_METHOD**. The basic syntax is shown in Fig. 7.3 and Fig. 7.4.

```
sc_process_handle hname = // ordinary function
  sc_spawn(
    sc_bind(&funcName, ARGS…)//no return value
   ,processName
   ,spawnOptions
  );

sc_process_handle hname = // member function
  sc_spawn(
    sc_bind(&methName, object, ARGS…)//no return
   ,processName
   ,spawnOptions
  );
```

Fig. 7.3 Syntax to register dynamic processes with void return

```
sc_process_handle hname = // ordinary function
  sc_spawn(
    &returnVar
   ,sc_bind(&funcName, ARGS…)
   ,processName
   ,spawnOptions
  );

sc_process_handle hname = // member function
  sc_spawn(
    &returnVar
   ,sc_bind(&methodName, object, ARGS …)
   ,processName
   ,spawnOptions
  );
```

Fig. 7.4 Syntax to register dynamic processes with return values

Note in the preceding that `object` is a reference to the calling module, and normally we just use the C++ keyword **`this`**, which refers to the calling object itself.

By default, arguments are passed by value. To pass by reference or by constant reference, a special syntax is required. This syntax is required to make **`sc_bind()`** practical.

```
sc_ref(var)  // reference
sc_cref(var) // constant reference
```

Fig. 7.5 Syntax to pass process arguments by reference

The `processName` and `spawnOptions` are optional; however, if `spawnOptions` are used, then a `processName` is mandatory. All processes should have unique names. Fortunately, uniqueness of a process name includes the hierarchical instance as a prefix (i.e., **`name()`**). If a process spawns a process, then its name is used to prefix the spawned process name.

7.3 Spawn Options

Spawn options are determined by creating an **`sc_spawn_option`** object and then invoking one of several methods that set the options. Figure 7.6 is the syntax:

```
sc_spawn_option objname;
objname.spawn_method();// register as SC_METHOD
objname.dont_initialize();
objname.set_sensitivity(event_ptr);
objname.set_sensitivity(port_ptr);
objname.set_sensitivity(interface_ptr);
objname.set_sensitivity(event_finder_ptr);
objname.set_stack_size(value); // experts only!
```

Fig. 7.6 Syntax to set spawn options

By default, spawned processes are thread processes. To specify a method process, you must call **`spawn_method()`** as shown above.

One last comment before we look at an example. The method **`sc_get_current_process_handle()`** may be used by the spawned process to reference the calling object. In particular, it may be useful to access **`name()`**.

7.4 A Spawned Process Example

That's a lot of syntax. Fortunately, you don't need to use all of it. Let's take a look at an example of the simplest case in Fig. 7.7. This example is an **`SC_THREAD`** that contains no parameters and returns no result. In other words, it looks like an **`SC_THREAD`** that just happens to be dynamically spawned. We highlight the important points.

```
#define SC_INCLUDE_DYNAMIC_PROCESSES
#include <systemc>
…
void spawned_thread() {// This will be spawned
  cout << "INFO: spawned_thread "
       << sc_get_current_process_handle().name()
       << " @ " << sc_time_stamp() << endl;
  wait(10,SC_NS);
  cout << "INFO: Exiting" << endl;
}

void simple_spawn::main_thread() {
  wait(15,SC_NS);
  // Unused handle discarded
  sc_spawn(sc_bind(&spawned_thread));
  cout << "INFO: main_thread " << name()
       << " @ " << sc_time_stamp() << endl;
  wait(15,SC_NS);
  cout << "INFO: main_thread stopping "
       << " @ " << sc_time_stamp() << endl;
}
```

Fig. 7.7 Example of a simple thread spawn

If you keep a handle on the spawned process, then it is also possible to await the termination of the process via the **sc_process_handle::terminated_event()** method. For example:

```
sc_process_handle h =
  sc_spawn(sc_bind(&spawned_thread));
// Do some work
…
// Wait for spawned thread to return
wait(h.terminated_event());
```

Fig. 7.8 Example of waiting on a spawned process

Be careful not to **wait()** on an **SC_METHOD** process; currently, there is no way to terminate an **SC_METHOD**.[1]

An interesting observation about **sc_spawn** is that it may also be used within the constructor, and it may be used with the same member function multiple times as long as the process name is unique. This capability also means there is now a way to pass arguments to **SC_THREAD** and **SC_METHOD** as long as you are willing to use the longer syntax.

A dangerous aspect of spawned threads relates to the return value. If you pass a function or method that returns a value, it is critical that the referenced return location remains valid throughout the lifetime of the process. The result will be written without respect to whether the location is valid upon exit, possibly resulting in a really nasty bug.

The creation and management of dynamic processes is not for the faint of heart. On the other hand, learning to manage dynamic processes has great rewards. One of the simpler ways to manage dynamic processes is discussed in the next section on fork/join.

7.5 SC_FORK/SC_JOIN

Another use of dynamic threads is dynamic test configuration. This feature is exemplified with a verification strategy sometimes used by Verilog suites using fork/join. Although this technique does not let you create new modules or channels dynamically (because processes may choose to stimulate ports differently on the fly), you can reconfigure tests. Let's see how this might be done.

[1]However, see the last section of this chapter for a preview of upcoming features.

Consider the DUT in the following figure:

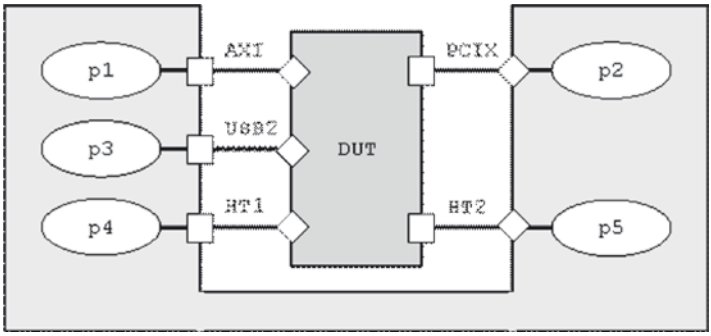


Fig. 7.9 High-level testbench

For each interface (AXI, USB2, etc.), an independent process can be created either to send or receive information likely to be generated in a real system.

Using these processes and fork/join, a high-level test might look like Fig. 7.10 (Note: Syntax will be explained shortly.):

```
DataStream d1, d2;
SC_FORK
   sc_spawn(
     sc_bind(&dut::AXI_xmt,this,sc_ref(d1)), "p1")
  ,sc_spawn(
     sc_bind(&dut::PCIX_rcv,this,sc_ref(d1)),"p2")
  ,sc_spawn(
     sc_bind(&dut::USB2,this,sc_ref(d1)),    "p3")
  ,sc_spawn(
     sc_bind(&dut::HT1_xtm,this,sc_ref(d2)), "p4")
  ,sc_spawn(
     sc_bind(&dut::HT2_rcv,this,sc_ref(d2)), "p5")
SC_JOIN
```

Fig. 7.10 Example of fork/join application

The syntax for the SystemC fork/join is shown in Fig. 7.11.

```
SC_FORK
  COMMA_SEPARATED_LIST_OF_SPAWNS
SC_JOIN
```

Fig. 7.11 Syntax for fork/join

Let's look at an example that involves a number of syntax elements discussed thus far. First, let's inspect the header for this module in Fig. 7.12.

```
//FILE: Fork.h
SC_MODULE(Fork) {
  …
  sc_fifo<double> wheel_lf, wheel_rt;
  SC_CTOR(Fork);// Constructor
  // Declare processes to be used with fork/join
  void fork_thread();
  bool road_thread(sc_fifo<double>& which);
};
```

Fig. 7.12 Example header for fork/join example

Notice that we pass a FIFO channel by reference so that `road_thread` can possibly access the FIFO channel. Passing ports or signals by reference would be a natural extension of this idea.

Now, let's inspect the rest of the code in Fig. 7.13.

```
//FILE: Fork.cpp
#define SC_INCLUDE_DYNAMIC_PROCESSES
#include <systemc>
#include "Fork.h"
…
Fork::Fork(sc_module_name nm) //{{{
: sc_module(nm)
{
  SC_THREAD(fork_thread);
  …
}
void Fork::fork_thread() { //{{{
  bool lf_up, rt_up; // use for return values
  SC_FORK
    sc_spawn(
       &lf_up
      ,sc_bind(
          &Fork::road_thread
         ,this
         ,sc_ref(wheel_lf)
       )
      ,"lf" // process name
    )
    ,sc_spawn(
       &rt_up
      ,sc_bind(
          &Fork::road_thread
         ,this
         ,sc_ref(wheel_rt)
       )
      ,"rt" // process name
    )
  SC_JOIN
}
bool Fork::road_thread(sc_fifo<double>& which) { //
  // Do some work
  return (road > 0.0);
}
```

Fig. 7.13 Example of fork/join

The full example may be found in the downloaded examples as `Fork`. This downloaded example also illustrates the use of **`sc_spawn`** instead of **`SC_THREAD`**. Using a capitalized word *fork* avoids collision with the Unix system call fork, which has nothing to do with SystemC's **`SC_FORK`**. Recall that SystemC is a cooperative multitasking system. You should not confuse Unix's fork facilities with these concepts.

7.6 Process Control Methods

The following information is being considered for addition to the SystemC standard. We include it here because we are fairly certain these extensions will be part of the standard within the lifetime of this book's publication. You may not be able to use these until you get access to OSCI version 2.3 or later. It is also possible that syntax could change slightly.

SystemC process control constructs are proposed as methods in the **`sc_process_handle`** class. Since spawning processes could occur in a hierarchical manner, there are options for the controls to affect either just the specified process, or the process and all of its descendants. This is accomplished with the enumeration shown in Fig. 7.14.

```
enum sc_descendant_inclusion_info {
  SC_NO_DESCENDANTS,
  SC_INCLUDE_DESCENDANTS
};
```

Fig. 7.14 Specifying descendants

Given the above, there are nine control constructs as shown in Fig. 7.15. Each of these takes a descendants argument that defaults to **`SC_NO_DESCENDANTS`**.

```
// Add "& resume" to sensitivity while suspended
void sc_process_handle::suspend(descend);
void sc_process_handle::resume(descend);

// Ignore sensitivity while disabled
void sc_process_handle::disable(descend);
void sc_process_handle::enable(descend);

// Complete remove process
void sc_process_handle::kill(descend);

// Asynchronously restart a process
void sc_process_handle::reset(descend);

// Reset process on every resumption event
void sc_process_handle::sync_reset_on(descend);
void sc_process_handle::sync_reset_off(descend);

// Throw an exception in the specified process
template<typename T>
  void sc_process_handle::throw_it(
                          const T&,descend);
```

Fig. 7.15 Process control constructs

Suspend and **resume** are used in situations where it is desirable to continue to collect events that the process is sensitive to even while suspended. This is likely to be used for high-level modeling. Keep in mind that the process might start evaluating immediately after resuming.

Disable and **enable** cause the process to ignore events in the sensitivity until re-enabled. For example, one might disable a process sensitive to the clock rather than resume it. Disabling is done because **resume** will start processing immediately if an event occurred while suspended. For clock synchronized processes **resume** is probably not desirable since **resume** might happen at a time not on the clock boundary.

The reset and synchronous reset methods will finally make it possible to deprecate the **SC_CTHREAD** process[2]. Resetting simplifies SystemC by reducing the number of process types. The idea is that a reset is issued whenever returning from a **wait** and the **sync_reset_on** is in effect.

Lastly, the **throw_it** method enables the concept of an interrupt. This of course requires use of the C++ exception handling mechanism **try**{CODE}**catch** (TYPE){HANDLER}.

Look in the downloadable examples for this edition of the book for more information.

7.7 Exercises

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

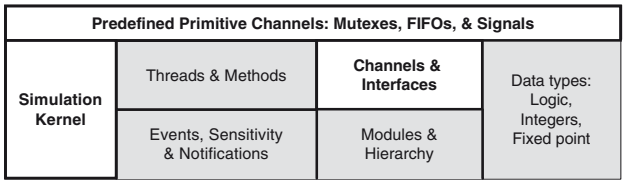
Exercise 7.1: Rewrite the *turn_of_events* example in Chapter 6 using dynamic simulations processes for *turn_knob_thread* and *stop_signal_thread*.

Exercise 7.2: Modify your dynamic simulation process version of the *turn_of_events* example in Chapter 6 so that the turn_knob_thread and stop_signal_thread do not rely on the event names *signal_stop*, *signal_off*, *stop_indicator_on*, and *stop_indicator_off*. (Hint: Pass a reference to the appropriate events to the *turn_know_thread* and the *stop_signal_thread* simulation processes.)

Exercise 7.3: As an advanced exercise, see if you can devise a classical software interrupt handler for a hardware environment that has two interrupts, a 1 us timer, and an incoming data packet. Assume data arrives randomly with separations of 500 ns to 2 us. This exercise is as much an exercise in planning as it is an exercise in using the syntax. Feel free to use the process controls if you have access to a SystemC version that supports it.

[2] **SC_CTHREAD** is discussed in a later chapter on Additional Topics.

Chapter 8
Basic Channels



Thus far, we have communicated information between concurrent processes using events and using ordinary module member data. Within one instance of time (a delta cycle), the order of execution is not defined; therefore, we must be extremely careful when sharing data.

Events let us manage simulation concurrency, but they require careful coding. Because the code may miss capturing an event, it is important to update a handshake variable indicating when a request is made, and clear it when the request is acknowledged. This mechanism also allows safe data communication between concurrent simulation processes.

Let's consider the gas station example again (Fig. 8.1). The customer notices an empty tank and arrives at the pump ①. The attendant has to be watching when the customer requests a fill-up ②, and has to make note of it if in the middle of filling up another customer ③. In the case of two arriving customers, if the attendant waits on either customer's request (i.e., **`wait`**`(e_request1|e_request2)`), the **`sc_event`** semantics do not allow the attendant to know which customer made the request. In other words, the attendant does not know which request triggered the **`wait()`** to return. This is why the `gas_station` model uses the status of the gas tank as an indicator to choose whether to fill the tank. Similarly, the customer must watch to see if the tank was actually filled when the attendant yells done ④.

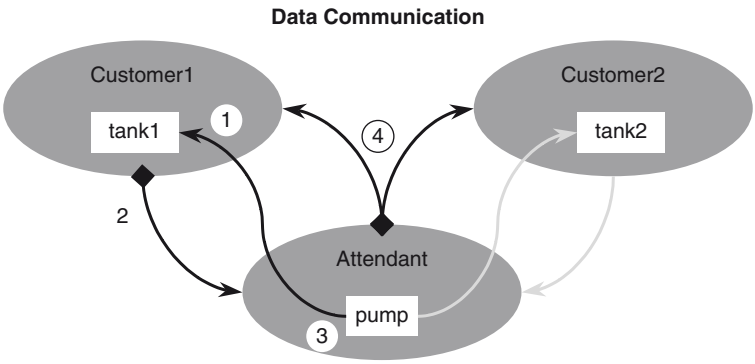


Fig. 8.1 Gas station processes and events

D.C. Black et al., *SystemC: From the Ground Up*, Second Edition,
DOI 10.1007/978-0-387-69958-5_8, © Springer Science+Business Media, LLC 2010

SystemC has built-in mechanisms, known as channels, to reduce the tedium of these chores, to aid communications, and to encapsulate complex communications. SystemC has two types of channels: primitive and hierarchical. This chapter covers the topic of primitive channels. Hierarchical channels are the subject matter of the chapter on Custom Channels.

8.1 Primitive Channels

SystemC's primitive channels are known as primitive because they contain no hierarchy, no simulation processes, and are designed simple to be very fast. All primitive channels inherit from the base class **sc_prim_channel**. Because they are SystemC channels, they must also inherit from and implement one or more SystemC interface classes.

The SystemC library contains several built-in primitive channels. This chapter focuses on the simplest channels, **sc_mutex**, **sc_semaphore**, and **sc_fifo**<*T*>. Additional primitive channel topics will be discussed in later chapters.

8.2 sc_mutex

Mutex is short for mutually exclusive text. In computer programming, a mutex is a program object that lets multiple program threads share a common resource, such as file access, without colliding.

During elaboration, a mutex is created with a unique name. Subsequently, any process that needs the resource must lock the mutex to prevent other processes from using the shared resource. The process should unlock the mutex when the resource is no longer needed. If another process attempts to access a locked mutex, that process is prevented from doing so until the mutex becomes available (unlocked).

SystemC provides the mutex function via the **sc_mutex** channel. The **sc_mutex** class implements the **sc_mutex_if** interface class (Fig. 8.2). This class contains several access methods including both blocked and unblocked styles. Remember that blocking methods can only be used in **SC_THREAD** processes.

```
sc_mutex NAME;

NAME.lock();     // Lock the mutex,
                 // wait until unlocked if in use
int NAME.trylock() // Non-blocking, returns success

NAME.unlock();  // Free a previously locked mutex
```

Fig. 8.2 Syntax of **sc_mutex_if** and **sc_mutex**

The example of the gas station attendant is a good example of a resource that needs to be shared. Our gas station only has a single pump, so only one car at a time is filled.

Another example of a resource requiring a mutex is automobile controls. Only one driver at a time can sit in the driver's seat. In a simulation modeling the interaction of drivers across town with a variety of vehicles, having one driver per set of controls might be interesting to model (Fig. 8.3).

```
class car : public sc_module {
  sc_mutex drivers_seat;
public:
  void drive_thread(void);
  …
};

void car::drive_thread(void) {
  drivers_seat.lock(); // sim driver acquires seat
  start();
  … // operate vehicle
  stop();
  drivers_seat.unlock(); // sim driver leaves
                         // vehicle
   …
}
```

Fig. 8.3 Example of **sc_mutex**

An electronic design application of an **sc_mutex** is arbitration for a shared bus. Here the ability of multiple masters to access the bus must be controlled. In lieu of an arbiter design, the **sc_mutex** might be used to manage the bus resource quickly until an arbiter can be designed. In fact, the mutex might even be part of the class implementing the bus model as illustrated in the following example (Fig. 8.4):

```
class bus : public sc_module {
  sc_mutex bus_access;
  …
  void write(int addr, int data) {
    bus_access.lock();
    // perform write
    bus_access.unlock();
  }
  …
};
```

Fig. 8.4 Example of **sc_mutex** used in bus class

Used with an **SC_METHOD** process, access might look like this (Fig. 8.5):

```
void grab_bus_method() {
  if (bus_access.trylock() == 0) {
    // access bus
    …
    bus_access.unlock();
  }
}
```

Fig. 8.5 Example of **sc_mutex** with an **SC_METHOD**

One downside to the **sc_mutex** is the lack of an event that signals when an **sc_mutex** is freed. This drawback necessitates using **trylock()** repeatedly based on some other event or time-based delay to use the shared resource. Assuming one process has locked a resource using **sc_mutex**, the second process wanting the same resource must call **trylock()** in conjunction with a **wait()** or return. The second process must use **wait()** or return between calls to **trylock()** to allow the first process some simulation cycles to finish and unlock the resource. Otherwise, the simulation will hang with infinite calls to **trylock()**.

8.3 sc_semaphore

For some resources, you may want to model more than one copy or owner. A good example of this would be parking spaces in a parking lot.

To manage this type of resource, SystemC provides the **sc_semaphore** class (Fig. 8.6). The **sc_semaphore** class inherits from and implements the **sc_semaphore_if** class. When creating an **sc_semphore** object, it is necessary to specify how many are available. In a sense, a mutex is merely a semaphore with a count of one. An **sc_semaphore** access consists of waiting for an available resource and then posting notice when finished with the resource.

```
sc_semaphore NAME(COUNT);

NAME.wait();       // Lock one semaphore
                   // Wait until available if in use
int NAME.trywait()  // Non-blocking, return success

int NAME.get_value() // Returns available semaphores

NAME.post();       // Free one previously locked
                   // semaphore
```

Fig. 8.6 Syntax of **sc_semaphore_if** and **sc_semaphore**

It is important to realize that the **sc_semaphore::wait()** is a distinctly different method from the **wait()** method previously discussed in conjunction with **SC_THREAD**. In fact, under the hood, the **sc_semaphore**::**wait()** is implemented with the **wait**(event).

A modern gas station with self-service would be a good example (Fig. 8.7) for using semaphores. Gas pumps can be represented with a semaphore where the count is set to the number of available pumps.

```
SC_MODULE(gas_station) {
  sc_semaphore pump(12);
  void customer1_thread {
    for(;;) {
      // wait till tank empty
      …
      // find an available gas pump
      pump.wait();
      // fill tank & pay
  }
};
```

Fig. 8.7 Example of **sc_semaphore**—gas_station

A multiport memory model is a good example (Fig. 8.8) of an electronic system-level design application using an **sc_semaphore**. You might use the semaphore to indicate the number of concurrent read or write accesses allowed.

```
class multiport_RAM {
  sc_semaphore read_ports(3);
  sc_semaphore write_ports(2);
  …
  void read(int addr, int& data) {
    read_ports.wait();
    // perform read
    read_ports.post();
  }
  void write(int addr, const int& data) {
    write_ports.wait();
    // perform write
    write_ports.post();
  }
  …
};//endclass
```

Fig. 8.8 Example of **sc_semaphore**—multiport_RAM

Other examples might include allocation of timeslots in a TDM (time division multiplex) scheme used in telephony, controlling tokens in a token ring, or perhaps even switching information to obtain better power management.

8.4 sc_fifo

Probably the most popular channel for modeling at the architectural level is the **sc_fifo**<*T*> channel (Fig. 8.9). First-in first-out queues (i.e., FIFOs) are a common data structure used to manage data flow. FIFOs are some of the simplest structures to manage.

In the very early stages of architectural design, the unbounded[1] STL **list**<*T*> (singly linked list) provides an easy implementation of a FIFO. Further in the design process, when FIFO depths are determined and SystemC elements come into stronger play, the **sc_fifo**<*T*> may be used to model at a higher level of detail.

The **sc_fifo**<*T*> class inherits from and implements two interface classes: **sc_fifo_in_if**<*T*> and **sc_fifo_out_if**<*T*>. It may not be intuitive at first, but the "in" interface is used for reading from the FIFO, and the "out" interface is for writing to the FIFO.

By default, an **sc_fifo**<*T*> has a depth of 16. The data type (i.e., **typename**) of the FIFO elements also needs to be specified. An **sc_fifo**<*T*> may contain any data type including large and complex structures (e.g., a TCP/IP packet or a disk block).

For example, FIFOs may be used to buffer data between an image processor and a bus, or a communications system might use FIFOs to buffer information packets as they traverse a network.

```
sc_fifo<ELEMENT_TYPENAME> NAME(SIZE);

NAME.write(VALUE);
NAME.read(REFERENCE);
… = NAME.read() /* function style */
if (NAME.nb_read(REFERENCE)) { // Non-blocking
                               // true if success
  …
}
if (NAME.num_available() == 0)
  wait(NAME.data_written_event());
if (NAME.num_free() == 0)
  next_trigger(NAME.data_read_event());
```

Fig. 8.9 Syntax of **sc_fifo**<*T*>—abbreviated

[1] This queue is limited only by the resources of the simulation machine itself.

Some architectural models are based on Kahn process networks[2] for which unbounded FIFOs provide the interconnect fabric. Although **sc_fifo**<*T*> is not unbounded, because reads and writes are blocking, it is possible to use them for this purpose given an appropriate depth. The depth needs to be set such that consumers and producers don't end up in a deadlock. This is illustrated in the next very simple example (Fig. 8.10).

```
SC_MODULE(kahn_ex) {
  …
  sc_fifo<double> a, b, y;
  …
};
// Constructor
kahn_ex::kahn_ex() : a(24), b(24), y(48)
{
  …
}
void kahn_ex ::stim_thread() {
  for (int i=0; i!=1024; ++i) {
    a.write(double(rand()/1000));
    b.write(double(rand()/1000));
  }
}
void kahn_ex::addsub_thread() {
  while(true) {
    y.write(kA*a.read() + kB*b.read());
    y.write(kA*a.read() - kB*b.read());
  }//endforever
}
void kahn_ex::monitor_method() {
  cout << y.read() << endl;
}
```

Fig. 8.10 Example of **sc_fifo<double>** kahn_ex

Software uses for FIFOs are numerous and include such concepts as mailboxes and other types of queues.

Note that when considering efficiency, passing pointers to large objects is most efficient. Be sure to consider using a safe pointer object if using a pointer. The **shared_ptr**<*T*> of the GNU publicly licensed Boost library (http://www.boost.org) makes implementation of smart pointers very straightforward.

Generally speaking, the STL may be more suited to software FIFOs. The STL **list**<*T*> might be used to manage an unknown number of stimulus data from a testbench.

In theory, an **sc_fifo**<*T*> could be synthesized at a behavioral level. It currently remains for a synthesis tool vendor to provide the functionality.

[2]Kahn, G. (1974). The semantics of a simple language for parallel programming. In J.L. Rosenfeld (Ed.), *Proceedings of IFIP Congress 74 (*pp.471–475). Amsterdam: North-Holland.

8.5 Exercises

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

Exercise 8.1: Examine, predict the output, compile, and run `mutex_ex`.

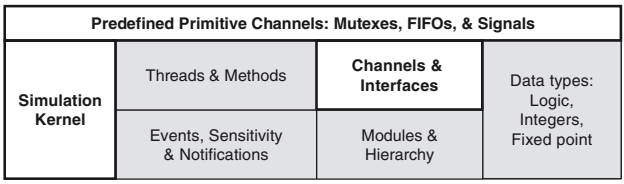
Exercise 8.2: Examine, compile, and run `semaphore_ex`. Add another family member. Explain discrepancies in behavior.

Exercise 8.3: Examine, compile, and run `fifo_fir`. Add a second filter stage to the network.

Exercise 8.4: Examine, compile, and run `fifo_of_ptr`. Discuss how one might compensate for the simulated transfer of a large packet.

Exercise 8.5: Examine, compile, and run `fifo_of_smart_ptr`. Notice the absence of **`delete`**.

Chapter 9
Evaluate-Update Channels



SC_SIGNALS

The preceding chapter considered high-level synchronization mechanisms. This chapter delves into electronic hardware[1].

Electronic signals behave in a manner approaching instantaneous activity. Generally, electronic signals have a single source (producer), but multiple sinks (consumer). It is quite important that all sinks "see" a signal update at the same time.

The easiest way to understand this concept is to consider the common hardware shift register. This model has a number of registers or memory elements as indicated in the diagram (Fig. 9.1) below.

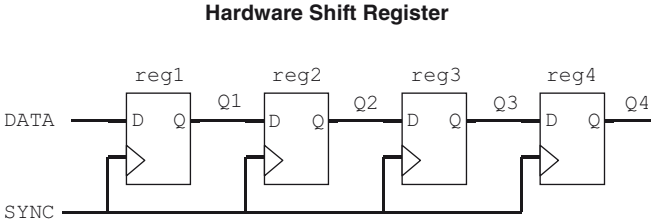


Fig. 9.1 Shift register

Data moves from left to right synchronous to the clock labeled `SYNC`. In software (e.g., C/C++), this flow would be modeled with four ordinary assignments (Fig. 9.2):

```
Q4 = Q3;
Q3 = Q2;
Q2 = Q1;
Q1 = DATA;
```

Fig. 9.2 Example of modeling a software-only shift register

[1]It is unclear whether the concepts discussed here have any application outside of electrical signals.

D.C. Black et al., *SystemC: From the Ground Up*, Second Edition,
DOI 10.1007/978-0-387-69958-5_9, © Springer Science+Business Media, LLC 2010

For this register to work, ordering is very important. In hardware, things are more difficult. Each register (reg1…reg4) is an independent concurrent process. Recall that the simulator places no order requirements on the processes. Below (Fig. 9.3) is a diagram from Chapter 6 Concurrency. Consider each process to represent a register from the preceding design.

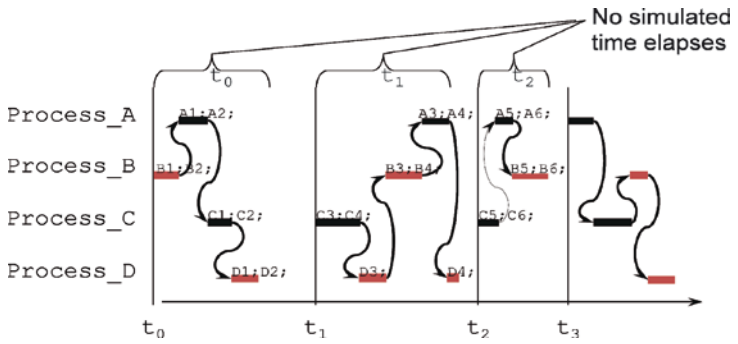


Fig. 9.3 Simulated activity with four concurrent processes

Since there is no guarantee that one assignment will take place before the other, we need to find some other solution. One idea might be to use events to force an ordering. This design would have `Process_A` wait for an event from `Process_B` before assigning its register, `Process_B` wait for an event from `Process_C` before assigning its register, and so on. This design requires coding both a **`wait`** and **`notify`** for each register, and it can become quite tiresome.

Another solution involves representing each register as a two-deep FIFO. This approach seems unnecessarily complex requiring two storage locations for the data and two pointers/counters to manage the state of the FIFO.

9.1 Completed Simulation Engine

To solve this problem, simulators have a feature known as the evaluate-update paradigm. The next diagram (Fig. 9.4) is the complete SystemC simulation kernel with the added update state. It is possible to go from evaluate to update and back. This cycle is known as the delta-cycle. Even when the simulator moves from evaluate to update to advance time, we say that at least one delta-cycle has occurred. Let's see how the delta-cycle is used.

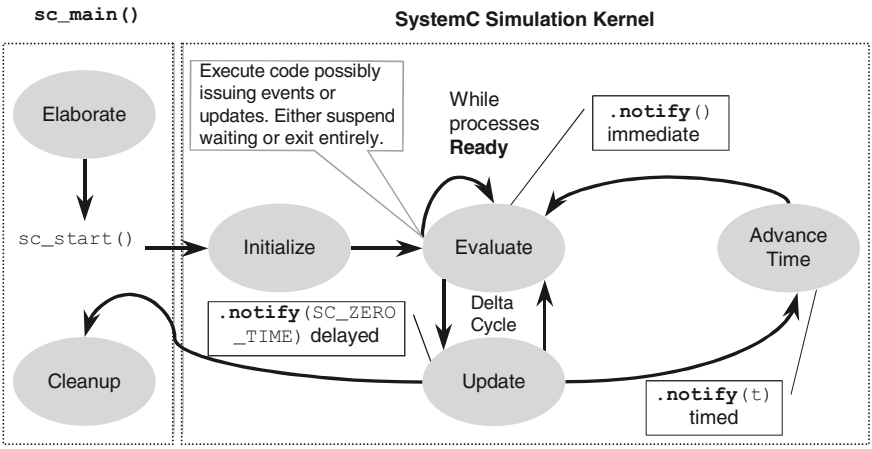


Fig. 9.4 Full SystemC simulation engine

Referring back to Chapter 6 on Concurrency, you'll recall that we discussed the notions of immediate notification, delayed notification, and timed notification. In the preceding diagram (Fig. 9.4) you will notice we've annotated the state bubble to indicate where each of these event notifications actually occur. It is important to realize that in the case of **`notify(SC_ZERO_TIME)`**, the notification occurs in the update phase after evaluation has completed. This means that processes waiting on events notified in this manner, will all see the event at the same instant. This contrasts starkly with immediate notification that uses the **`notify(void)`** syntax. Immediate notification may cause some processes to miss the event if they are not already waiting for it. These processes may miss the event because they have not run in the current evaluate phase.

Special channels, known as signal channels, use this update phase as a point of data synchronization. To accomplish this synchronization, every channel has two storage locations: the current value and the new value. Visually, there are two sets of data: new and current.

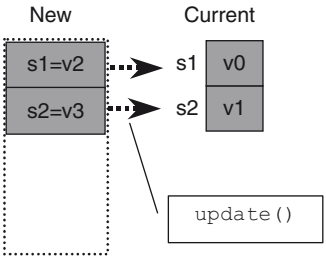


Fig. 9.5 Signal channel data storage

Referring to the preceding figure (Fig. 9.5), when a process writes to a signal channel, the process stores into the new value rather than into the current value. The process then calls **`request_update()`** to notify the simulation kernel. When the update phase occurs, the simulation kernel calls the **`update()`** methods of all channels that requested update during the preceding evaluate phase.

The **update()** method may do more than simply copy the new value into the current value. It may resolve contentions and, most importantly, notify **sc_event**'s (e.g., to indicate a change and wake up a process in the waiting state).

An important aspect of this paradigm is the current value remains unchanged until the end of the update phase. If a process writes to an evaluate-update channel and then immediately (i.e., without suspending) accesses the channel, it will find the current value unchanged.

This behavior is a frequent cause for confusion for simulation neophytes. Those familiar with HDLs should not be surprised. Shortly, we will discuss techniques to make this behavior less of a surprise.

Another consideration is that the new value will contain only the last value written to a signal channel during the evaluate phase. Thus, writing repeatedly overwrites the previous new value.

9.2 SystemC Signal Channels

The **sc_signal**<*T*> primitive channel and its close relative, **sc_buffer**<*T*>, both use the evaluate-update paradigm. Here (Fig. 9.6) is the syntax for declaration, reading, and writing:

The **sc_signal::write()** method contains the evaluate phase portion of the evaluate-update behavior. The **write()** method includes a call to the protected **sc_prim_channel::request_update()** method. The call to **sc_signal::update()** is hidden, and the call occurs during the update phase when the kernel calls it as a result of the **request_update()**.

```
  sc_signal<datatype> signame[, signame_i]…;//define
…
  signame.write(newvalue);
  varname  = signame.read();
  wait(signame.value_changed_event()|...);
  wait(signame.default_event()|...);
  if (signame.event() == true) {
    // occurred in previous delta-cycle
```

Fig. 9.6 Syntax of **sc_signal**<*T*>

The **sc_signal::read()** method returns the current value. If the signal has been written to in the current evaluate phase, the new value is *not* reflected in the returned value. This behavior may come as a surprise to some folks (especially to non-hardware background folks).

The **sc_signal::value_changed_event()** method returns a reference to an sc_event. This event is notified any time the update causes a change in value. This behavior lets code wait on changes in the channel.

The **sc_signal::default_event()** method is simply an alias for the **via** method. This method is used by the **sensitive** class to allow use of the simplified syntax shown (Fig. 9.7) below (recall that this aliasing must be done

during elaboration, normally in the constructor). Several other channels include this same aliasing to an event referencing. In fact, **sc_event_queue** returns this same method to allow it to be used in static sensitivity.

```
sensitive << signame;
```

Fig. 9.7 Static sensitivity to sc_signal<T>

The **event()** method is special. Normally, it is impossible to determine which event caused a return from **wait()**; however, for **sc_signal** channels (including other derivatives mentioned in this chapter), the **event()** method may be called to see if the channel issued an event in the immediately previous delta-cycle. This ability to determine which event caused the last return from **wait()** does not preclude the occurrence of other events in the previous cycle.

It should be noted that **sc_signal**<*T*> is essentially identical to VHDL's **signal**. For Verilog, the analogy is a **reg** that uses the Verilog non-blocking assignment operator exclusively.

```
// Declare variables
int               count;
string            message_temp;
sc_signal<int>    count_sig;
sc_signal<string> message_sig;

cout << "Initialize during 1st delta cycle" << endl;
count_sig.write(10);
message_sig.write("Hello");
count = 11;
message_temp = "Whoa";
cout << "count is " << count << " "
     << "count_sig is " << count_sig << endl
     << "message_temp is '" << message_temp << "' "
     << "message_sig is '" << message_sig << "'"
     << endl << "Waiting" << endl << endl;
wait(SC_ZERO_TIME);

cout << "2nd delta cycle" << endl;
count = 20;
count_sig.write(count);
cout << "count is " << count << ", "
     << "count_sig is " << count_sig << endl
     << "message_temp is '" << message_temp << "', "
     << "message_sig is '" << message_sig << "'"
     <<endl << "Waiting" << endl << endl;
wait(SC_ZERO_TIME);

cout << "3rd delta cycle" << endl;
message_sig.write(message_temp = "Rev engines");
cout << "count is " << count << ", "
     << "count_sig is " << count_sig << endl
     << "message_temp is '" << message_temp << "', "
     << "message_sig is '" << message_sig << "'"
     << endl << endl << "Done" << endl;
```

Fig. 9.8 Example of **sc_signal**<*T*>

An example of usage and the slightly surprising results[2] are in order.

The example in Fig. 9.8 produces the result shown in Fig. 9.9. Notice how the current value of the signals, `count_sig` and `message_sig`, remain unchanged until a delta-cycle has occurred. On the other hand, the non-signal values, `count` and `message_temp`, get immediately updated.

```
Initialize during 1st delta cycle
count is 11, count_sig is 0
message_temp is 'Whoa', message_sig is ''
Waiting

2nd delta cycle
count is 20, count_sig is 10
message_temp is 'Whoa', message_sig is 'Hello'
Waiting

3rd delta cycle
count is 20, count_sig is 20
message_temp is 'Rev engines', message_sig is
'Hello'

Done
```

Fig. 9.9 Example of `sc_signal<T>` output

Because the code uses a naming convention (i.e., appended `_sig` to the signals), it is relatively easy to spot the evaluate-update signals and make the mental connection to the behavior. Without the naming convention, one might wonder if the identifiers represent some other channel (e.g., `sc_fifo<T>`).

In addition to the preceding syntax, SystemC has overloaded the assignment and copy operators to allow the following *dangerous* syntaxes:

```
signame = newvalue; // implicit .write() dangerous
varname = signame; // implicit .read() mild danger
```

Fig. 9.10 Syntax of **sc_signal**<*T*> (dangerous)

The reason we consider these syntaxes dangerous relates to the issue of the evaluate-update paradigm. Consider the following example, assuming that ***r*** is an **`sc_signal<int>`**:

```
// Convert rectangular to polar coordinates
r = x;
if ( r != 0 && r != 1 ) r = r * r;
if ( y != 0 ) r = r + y*y;
cout << "Radius is " << sqrt(r) << endl;
```

Fig. 9.11 Dangerous **`sc_signal`**<*T*>

[2]This usage may surprise non-HDL experienced folks. HDL-experienced users should understand the VHDL or Verilog analogy.

Without sufficient context, the casual reader would be quite surprised at the results shown below. Assume on entry x=3, y=4, r=0.

```
Radius is 0
```

Fig. 9.12 Example of `sc_signal` output (dangerous)

Even when using what might be considered the safer syntax, you must be careful. We strongly suggest that you use a naming style.

One beneficial aspect of **`sc_signal`**<*T*> and **`sc_buffer`**<*T*> channels is a restriction that only a single process may write to a given signal during a specific delta-cycle. This restriction avoids the potential danger of two processes non-deterministically asserting a value and creating a race condition.

For the OSCI simulator, the run-time error is flagged in this situation if and only if you have defined the compile-time macro **`DEBUG_SYSTEMC`**. This macro is not part of the IEEE-1666 standard, but it was defined for the OSCI implementation to reduce simulation overhead. We recommend you define this macro early in the project, and only remove it once you are certain your code doesn't violate the signal process writer rule and need more performance. You can find an example of this danger in the `danger_ex` in the downloaded examples.

9.3 Resolved Signal Channels

There are times when it is appropriate to have multiple writers. One of these situations involves modeling buses that have the possibility of high impedance (i.e., Z) and contention (i.e., X).

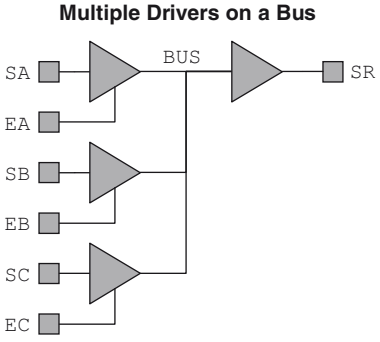


Fig. 9.13 Tri-state bus

SystemC provided the specialized channels **sc_signal_resolved** and **sc_signal_rv**<*T*> (Fig. 9.14). The _rv means resolved vector.

```
sc_signal_resolved name;
sc_signal_rv<WIDTH> name;
```

Fig. 9.14 Syntax of **sc_signal_resolved** and **sc_signal_rv**

The base functionality has identical semantics to **sc_signal**<***sc_logic***>; however, it allows for multiple writers and provides built-in resolution functionality as follows (Table 9.1):

Table 9.1 Resolution functionality for sc_signal_resolved

A\B	'0'	'1'	'X'	'Z'
'0'	'0'	'X'	'X'	'0'
'1'	'X'	'1'	'X'	'1'
'X'	'X'	'X'	'X'	'X'
'Z'	'0'	'1'	'X'	'Z'

One minor failing of SystemC is the lack of direct support for several common system-level bus concepts. Specifically, SystemC has no mechanisms for pull-ups, pull-downs, nor various open-source or open-drain variations.

For these, you have to create your own channels, which is not too difficult. The easiest way is to create a class derived from an existing class that almost works. Here is the resolution table (Table 9.2) for a pull-up functionality:

Table 9.2 Resolution functionality for eslx_pullup

A\B	'0'	'1'	'X'	'Z'
'0'	'0'	'X'	'X'	'0'
'1'	'X'	'1'	'X'	'1'
'X'	'X'	'X'	'X'	'X'
'Z'	'0'	'1'	'X'	'1'

Notice that there is only one difference in the table (shaded). The custom channel in Fig. 9.15 implements this resolution for a single-bit pull-up functionality.

```
class eslx_pullup
  : public sc_core::sc_signal_resolved {
public:
  // constructors
  eslx_pullup()
  : sc_signal_resolved(sc_gen_unique_name("pullup"))
  {}
  explicit eslx_pullup(const char* nm)
  : sc_signal_resolved(nm)
  {}
  const sc_dt::sc_logic& read() const {
    const sc_dt::sc_logic& result
      (sc_core::sc_signal_resolved::read());
    static const sc_dt::sc_logic
      ONE(sc_dt::SC_LOGIC_1);
    if (result == sc_dt::SC_LOGIC_Z) {
      return ONE;
    } else {
      return result;
    }//endif
  }
};
```

Fig. 9.15 Example of `eslx_signal_pullup`

9.4 Template Specializations of sc_signal Channels

SystemC has several template specializations that bear discussion. A template specialization occurs when a definition is provided for a specific template value. If there is more than one template variable involved, we call it a partial specialization.

For example, **`sc_signal<`***T*`>`** has a single template variable representing the **`typename`**. SystemC defines some additional behaviors for **`sc_signal<bool>`** that are not available for the general case. Thus, an **`sc_signal<char>`** does not support the concept of a **`posedge_event()`**.

The specialized templates **`sc_signal<bool>`** and **`sc_signal<sc_logic>`** have the following (Fig. 9.16) extensions:

```
  sensitive << signame.posedge_event()
            << signame.negedge_event();
  wait(signame.posedge_event()
      |signame.negedge_event());
  if (signame.posedge_event()
     |signame.negedge_event()) {
```

Fig. 9.16 Syntax of specializations **posedge** and **negedge**

For **`sc_logic`**, a **`posedge_event`** occurs on any transition to **`SC_LOGIC_1`**, which includes **`SC_LOGIC_X`** and **`SC_LOGIC_Z`**. The same is true of transitions to **`SC_LOGIC_0`** and the **`negedge_event`**.

The Boolean **posedge()** and **negedge()** methods apply similarly to the **event()** method, and they only apply to the immediately previous delta-cycle.

It is notable that **sc_buffer** does *not* support these specializations.

9.5 Exercises

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

Exercise 9.1: Examine, compile, and run `signal_ex`. Does this type of channel lend itself to higher level modeling? Why or why not?

Exercise 9.2: Examine, compile, and run `buffer_ex`. Contrast this with the previous example. In what situations might you prefer **sc_buffer<*T*>** over **sc_signal<*T*>** or visa versa?

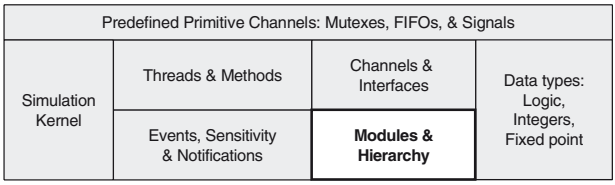
Exercise 9.3: Examine, compile, and run `danger_ex`. Change the code to make it less dangerous.

Exercise 9.4: Examine, compile, and run `resolved_ex`. Observe the definition of `DEBUG_SYSTEMC` in `../Makefile.defs`. See if you can measure the performance difference.

Exercise 9.5: Examine the interactive simulation illustrating the simulation engine model in the downloaded examples. Notice how the **update()** method was used to extract information. Would you consider this an acceptable use for normal modeling? Why?

Chapter 10
Structure

Design Hierarchy



This chapter describes SystemC's facilities for implementing structure, sometimes known as design hierarchy. Design hierarchy concerns both the hierarchical relationships of modules discussed here and the connectivity that lets modules communicate in an orderly fashion. Connectivity will be discussed in the next chapter.

10.1 Module Hierarchy

Thus far, we have examined modules containing only a single level of hierarchy with all processes residing in a single module. This level of complexity might be acceptable for small designs. However, larger system designs require partitioning and hierarchy to enable understanding and project management. We consider project management to include documentation and practical issues such as integration of third-party intellectual property (IP).

Design hierarchy in SystemC uses instantiations of modules as member data of parent modules. In other words, to create a level of hierarchy, create an **`sc_module`** object within a parent **`sc_module`**. The new **`sc_module`** is the submodule within the parent module.

Consider the hierarchy in Fig. 10.1 for the following discussions and examples. In this case, we have a parent module named `Car`, with submodules named `Engine` and `Body`. To obtain the hierarchical relationship, we create an `Engine` and `Body` object within the definition of the `Car` class.

D.C. Black et al., *SystemC: From the Ground Up*, Second Edition,
DOI 10.1007/978-0-387-69958-5_10, © Springer Science+Business Media, LLC 2010

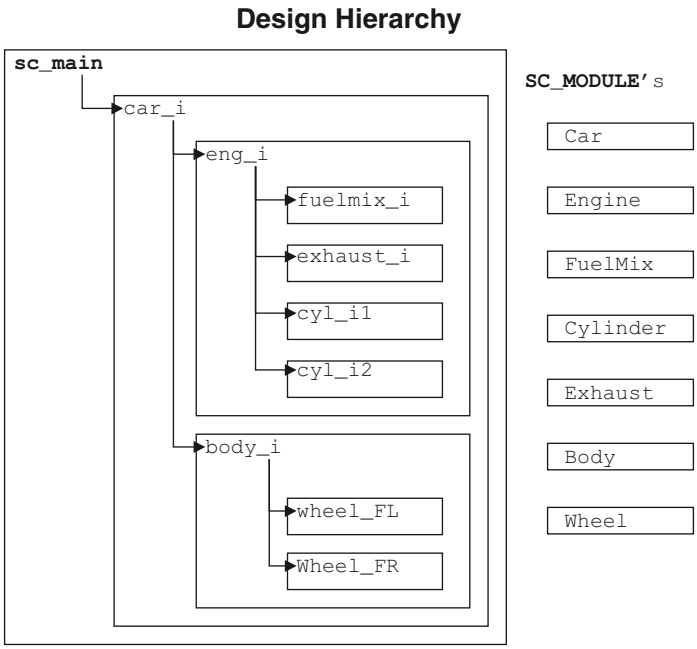


Fig. 10.1 Design hierarchy

C++ offers two basic ways to create submodule objects within the definition of a parent module or object. First, a submodule object may be created directly by declaration just like any simple member data of the module class. Second, a submodule object may be indirectly referenced by means of a pointer in combination with dynamic allocation.

When submodules are created outside of **`sc_main`**, the submodule objects must be, at minimum, initialized within the module constructor. When using indirect reference by means of a pointer, the pointer must be initialized within the constructor using `new`. Constructors can be implemented within the header file (`.h` file) or within the implementation file (`.cpp` file). Since hierarchy reflects an internal implementation decision, the authors prefer to see the constructor defined in the implementation file. The only time construction cannot be done within the implementation file is when you are defining a templated module, which triggers compiler restrictions[1]. Templated modules are a relatively rare occurrence.

Creation of hierarchy at the top level (**`sc_main`**) is slightly different from instantiation within modules. This difference results from differences in C++ syntax requirements for initialization outside of a class definition.

[1]Future versions of C++ compiler/linker tool sets may fix this restriction.

Since it is likely you may see any combination of these approaches, we will illustrate all six approaches:

- Direct top-level (**sc_main**)
- Indirect top-level (**sc_main**)
- Direct submodule header-only
- Direct submodule
- Indirect submodule header-only
- Indirect submodule

There are likely a few more variants, but understanding these should suffice. For easy reference, Table 10.1 at the end of this chapter lists the pros and cons of these approaches.

10.2 Direct Top-Level Implementation

First, we illustrate (Fig. 10.2) top-level implementation with direct instantiation, which has already been presented in `Hello_SystemC` and is used in all the succeeding discussions in the book. It is simple and to the point. Sub-design instances are simply instantiated and initialized in one statement. The name given via the constructor is the hierarchical instance name used by the SystemC kernel and is very useful during debug. The SystemC instance string is the name used by SystemC when printing error or informational messages. Normally the C++ instance name and the SystemC string name are defined to be identical. This comment applies to all SystemC module instantiation styles.

```
//FILE: main.cpp
  #include <systemc>
  #include "Car.h"
  int sc_main(int argc, char* argv[]) {
    Car car_i("car_i");
    sc_start();
    return 0;
  }
```

Fig. 10.2 Example of `main` with direct instantiation

10.3 Indirect Top-Level Implementation

A minor variation on this approach is top-level implementation with indirect instantiation (Fig. 10.3). This approach adds two lines of syntax with both a pointer declaration and an instance creation via `new`. This variation takes more code; however, it adds the possibility of dynamically configuring the design with the addition of **if-else** and looping constructs.

A design with a regular structure might even construct an array of designs and programmatically instantiate and connect them. We have used the `suffix_iptr` to indicate that `car_iptr` is a pointer to an instantiation.

```
//FILE: main.cpp
  #include <systemc>
  #include "Car.h"
  int sc_main(int argc, char* argv[]) {
    Car* car_iptr;               // pointer to Car
    car_iptr = new Car("car_i"); // create Car
    sc_start();
    delete car_iptr;
    return 0;
  }
```

Fig. 10.3 Example of `main` with indirect instantiation

10.4 Direct Submodule Header-Only Implementation

When dealing with submodules (i.e., beneath or within a module), things become mildly more interesting because C++ semantics require the use of an initializer list for the direct approach. Remember that **`SC_CTOR`** is a macro that hides the C++ constructor syntax. The constructor implemented in **`SC_CTOR`** requires initialization with a SystemC instance name, therefore the requirement to initialize the submodules. The next figure (Fig. 10.4) shows an example of direct instantiation in the header.

```
//FILE:Car.h
  #include "Body.h"
  #include "Engine.h"
  SC_MODULE(Car) {
    Body   body_i;
    Engine eng_i ;
    SC_CTOR(Car)
    : body_i("body_i") //initialization
    , eng_i("eng_i")   //initialization
    {
      // other initialization
    }
  };
```

Fig. 10.4 Example of module with direct instantiation in header

10.5 Direct Submodule Implementation

One disadvantage of the preceding approach is that it exposes the complexities of the constructor body to all potential users, even those who just want to use your module and don't care to know about your hierarchy partitioning choices. Moving the constructor into the implementation (i.e., the *module*.cpp file) requires the use of **`SC_HAS_PROCESS`**. The next figure (Fig. 10.5) shows an example of using direct instantiation.

```
//FILE:Car.h
  #include "Body.h"
  #include "Engine.h"
  SC_MODULE(Car) {
    Body   body_i;
    Engine eng_i;
    Car(sc_module_name nm);
  };
```

```
//FILE:Car.cpp
  #include <systemc>
  #include "Car.h"
  // Constructor
  SC_HAS_PROCESS(Car);
  Car::Car(sc_module_name nm)
  : sc_module(nm)
  , body_i("body_i")
  , eng_i("eng_i")
  {
    // other initialization
  }
```

Fig. 10.5 Example of module with direct instantiation and separate compilation

10.6 Indirect Submodule Header-Only Implementation

Use of indirection renders the instantiation a little bit easier to read for the submodule header-only case; however, no other advantages are clear. The next figure (Fig. 10.6) shows an example of indirect instantiation in the header.

```
//FILE:Body.h
  #include "Wheel.h"
  SC_MODULE(Body) {
    Wheel* wheel_FL_iptr;
    Wheel* wheel_FR_iptr;
    SC_CTOR(Body) {
      wheel_FL_iptr = new Wheel("wheel_FL_i");
      wheel_FR_iptr = new Wheel("wheel_FR_i");
      // other initialization
    }
  };
```

Fig. 10.6 Example of module with indirect instantiation in header

10.7 Indirect Submodule Implementation

Moving the module indirect approach into the implementation file has the advantage of possibly supplying pre-compiled object files making this approach good for IP distribution. This advantage is in addition to the possibility of dynamically determining the configuration discussed previously. The figure below (Fig. 10.7) shows an example of indirect instantiation with separate compilation.

```
//FILE:Engine.h
  class FuelMix;
  class Exhaust;
  class Cylinder;
  SC_MODULE(Engine) {
    FuelMix*  fuelmix_iptr;
    Exhaust*  exhaust_iptr;
    Cylinder* cyl1_iptr;
    Cylinder* cyl2_iptr;
    Engine(sc_module_name nm); // Constructor
  };
```

```
//FILE: Engine.cpp
  #include <systemc>
  #include "FuelMix.h"
  #include "Exhaust.h"
  #include "Cylinder.h"
  // Constructor
  SC_HAS_PROCESS(Engine);
  Engine::Engine(sc_module_name nm)
  : sc_module(nm)
  {
    fuelmix_iptr = new FuelMix("fuelmix_i");
    exhaust_iptr = new Exhaust("exhaust_i");
    cyl1_iptr    = new Cylinder("cyl1_i");
    cyl2_iptr    = new Cylinder("cyl2_i");
    // other initialization
  }
```

Fig. 10.7 Example of module with indirect separate compilation

Notice the absence of # include `FuelMix.h`, `Exhaust.h`, and `Cylinder.h` in `Engine`.h of the preceding example. This omission could be a real advantage when providing `Engine` for use by another group. You need to provide only `Engine.h` and a compiled object or library (e.g., `Engine.o` or `Engine.a`) files. You can then develop your implementation independently. This approach is good for both internal and external IP distribution.

10.8 Contrasting Implementation Approaches

The following table (Table 10.1) contrasts the features of the six approaches.

Some groups have the opinion that the top-level module should instantiate a single design with a fixed name (e.g., Design_top) and then apply a consistent approach for all the submodules. Some EDA tools perform all this magic for you.

Table 10.1 Comparison of hierarchical instantiation approaches

Level	Allocation	Pros	Cons
Main	Direct	Least code	Inconsistent with other levels
Main	Indirect	Dynamically configurable	Involves pointers
Module	Direct header only	All in one file Easier to understand	Requires submodule headers
Module	Indirect header only	All in one file Dynamically configurable	Involves pointers Requires submodule headers
Module	Direct with separate compilation	Hides implementation	Requires submodule headers
Module	Indirect with separate compilation	Hides submodule headers and implementation Dynamically configurable	Involves pointers

10.9 Exercises

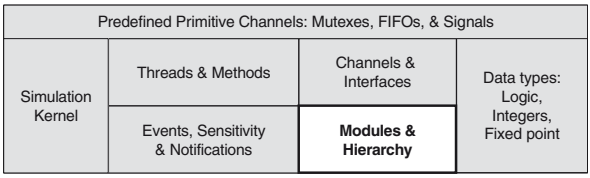
For the following exercises, use the samples provided at www.scftgu.com.

Exercise 10.1: Examine, compile, and run the **sedan** example. Which styles are simplest?

Exercise 10.2: Examine, compile, and run the **convertible** example. Notice the forward declarations of **Body** and **Engine**. How might this be an advantage when providing IP?

Chapter 11
Communication

Ports



This chapter describes SystemC's facilities for implementing connectivity, which enables orderly communication between modules.

11.1 Communication: The Need for Ports

Hierarchy without the ability to communicate between modules is not very useful, but what is the best way to communicate? There are two concerns: safety and ease of use. Safety is a concern because all activity occurs within processes, and care must be taken when communicating between processes to avoid race conditions. Events and channels are used to handle this concern.

Ease of use is more difficult to address. Let us dispense with any solution involving global variables, which are well known as a poor methodology. Another possibility is to have a process in an upper-level module. This process would monitor and manage events defined in instantiated modules. This mechanism is awkward at best.

SystemC takes an approach that lets modules use channels inserted between the communicating modules. SystemC accomplishes this communication with a concept called a port. Basically, a port is a pointer to a channel outside the module.

For simplicity, this chapter only covers the **`sc_port`**<*T*>. We will cover an alternate and related concept, the **`sc_export`**<*T*>, in a later chapter. An **`sc_export`**<*T*> is a pointer to a channel inside another module.

Consider the following example (Fig. 11.1):

D.C. Black et al., *SystemC: From the Ground Up*, Second Edition,
DOI 10.1007/978-0-387-69958-5_11, © Springer Science+Business Media, LLC 2010

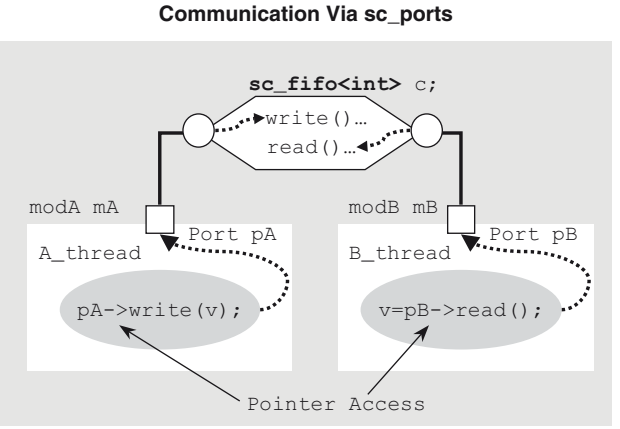


Fig. 11.1 Communication via ports

The process `A_thread` in module `modA` communicates a value contained in local variable v by calling the **`write`** method of the parent module's channel c. Process `B_thread` in module `modB` may retrieve a value via the **`read`** method of channel `c`.

Notice that access is accomplished via pointers `pA` and `pB`. Notice also that the two processes really only need access to the methods **`write`** and **`read`**. More specifically, `modA` only needs access to **`write`**, and **`modB`** only needs access to **`read`**. This separation of access is managed using a concept known as an interface, which is described in the next section.

11.2 Interfaces: C++ and SystemC

C++ defines a concept known as an abstract class. An abstract class is a class that is never used directly, but it is used only via derived subclasses. Partly to enforce this concept, abstract classes usually contain pure virtual functions. Pure virtual functions are not allowed to provide an implementation in the abstract class where they are defined as pure. This restriction in turn compels any class derived from the abstract class to override all the pure virtual functions, or in other words, the class derived from the abstract class must provide an implementation for all the pure virtual functions.

The following (Fig. 11.2) diagram illustrates the concept. Pure virtual functions are identified by 1) the keyword **`virtual`** and 2) the `=0;` to indicate they're pure.

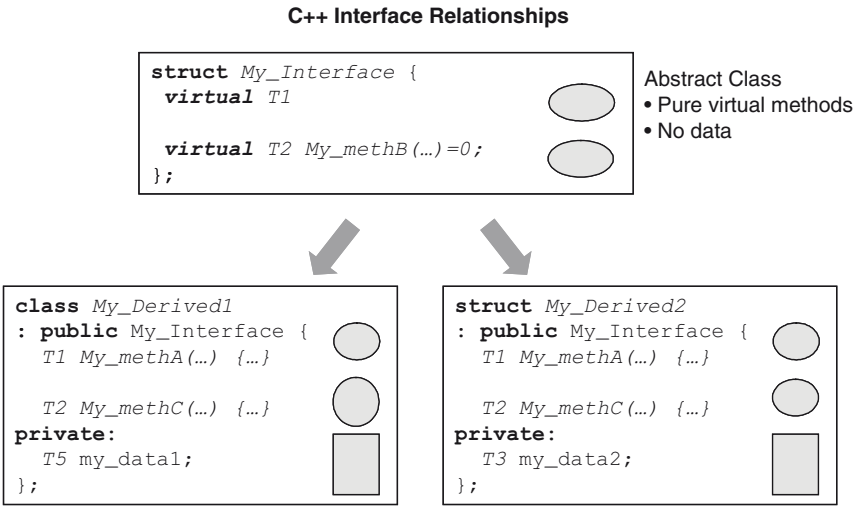


Fig. 11.2 C++ interface class relationships

If a class contains no data members and only contains pure virtual methods, it is known as an interface class. Here is a short example of an interface class:

The concept of interfaces has a powerful property when used with polymorphism. Recall from C++ that polymorphism is the following idea: A derived class can be processed by a function referencing the parent class.

```
class my_interface {
public:
  virtual void write(unsigned addr, int data) = 0;
  virtual int read(unsigned addr) = 0;
};
```

Fig. 11.3 Example of C++ interface

Consider the preceding figure (Fig. 11.3) of C++ interface class relationships. A function using `My_Interface` might access `My_methA()`. If the current object is of class My_Derived2, then the actual `My_methA()` call results in `My_Derived2::My_methA()`.

If an object is declared as a pointer to an interface class, it may be used with multiple derived classes. Suppose we define two derived classes as follows (Fig. 11.4):

```
class multiport_memory_arch: public my_interface {
public:
  void write(unsigned addr, int data) {
    mem[addr] = data;
  }// end write
  int read(unsigned addr) ) {
    return mem[addr];
  }//end read
private:
  int mem[1024];
};
```

```
class multiport_memory_RTL: public my_interface {
public:
  void write(unsigned addr, int data) {
    // complex details of RTL memory write
  }// end write
  int read(unsigned addr) ) {
    // complex details of RTL memory read
  }// end read
private:
  // complex details of RTL memory storage
};
```

Fig. 11.4 Example of two derivations from interface class

Now we write some C++ code to access the aforementioned derived classes.

```
void memtest(my_interface& mem) {
  // complex memory test
}

multiport_memory_arch fast;
multiport_memory_RTL  slow;
memtest(fast);
memtest(slow);
```

Fig. 11.5 Example of C++ interface

As seen in the preceding example (Fig. 11.5), the same code may access multiple variations of a design. You can think of an interface as the application programming interface (API) to a set of derived classes. This same concept is used in SystemC to implement ports.

DEFINITION: A SystemC interface is an abstract class that inherits from **`sc_interface`** and provides only pure virtual declarations of methods referenced by SystemC channels and ports. No implementations or data are provided in a SystemC interface.

We now provide the concise definition of the SystemC channel.

DEFINITION: A SystemC channel is a class that inherits from either **`sc_channel`** or from **`sc_prim_channel`**, and the channel should[1] inherit and implement one or more SystemC interface classes. A channel implements all the pure virtual methods of the inherited interface classes.

By using interfaces to connect channels, we can implement modules independent of the implementation details of the communication channels.

Consider the following diagram (Fig. 11.6):

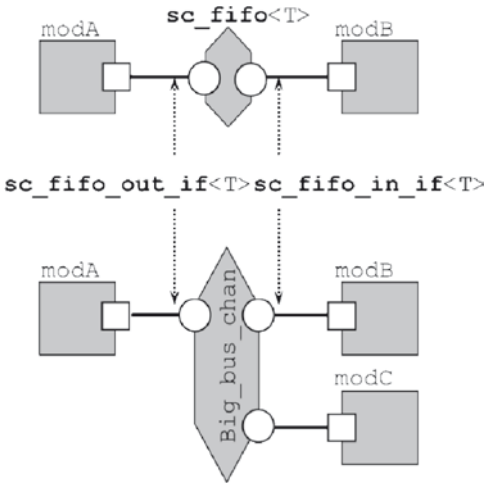


Fig. 11.6 The power of interfaces

In one design, modules `modA` and modB are connected via a FIFO. With no change to the definition of modA or modB, we can swap out the FIFO for a different channel. All that matters is for the interfaces used to remain constant. In this example, the interfaces are **`sc_fifo_out_if`**<*T*> and **`sc_fifo_in_if`**<*T*>. In the next few sections, the mechanics of using interfaces are described.

11.3 Simple SystemC Port Declarations

Given the definition of an interface, we now present the definition of a port.

DEFINITION A SystemC port is a class templated with and inheriting from a SystemC interface. Ports allow access of channels across module boundaries.

Specifically, the syntax of a simple SystemC port follows (Fig. 11.7):

[1]However, without an interface, a SystemC channel cannot be used with a SystemC port.

```
sc_port<interface> portname;
```

Fig. 11.7 Syntax of basic **sc_port**

SystemC ports are always defined within the module class definition. Here is a simple example (Fig. 11.8):

```
SC_MODULE(stereo_amp) {
  sc_port<sc_fifo_in_if<int> >  soundin_p;
  sc_port<sc_fifo_out_if<int> > soundout_p;
  …
};
```

Fig. 11.8 Example of defining ports within module class definition

Notice the extra blank space following the greater-than symbol (>). This is required C++ syntax when nesting templated classes.

11.4 Many Ways to Connect

Given the declaration of a port, we now address the issue of connecting ports, channels, modules, and processes. The following diagram (Fig. 11.9) illustrates the types of connections that are possible with SystemC:

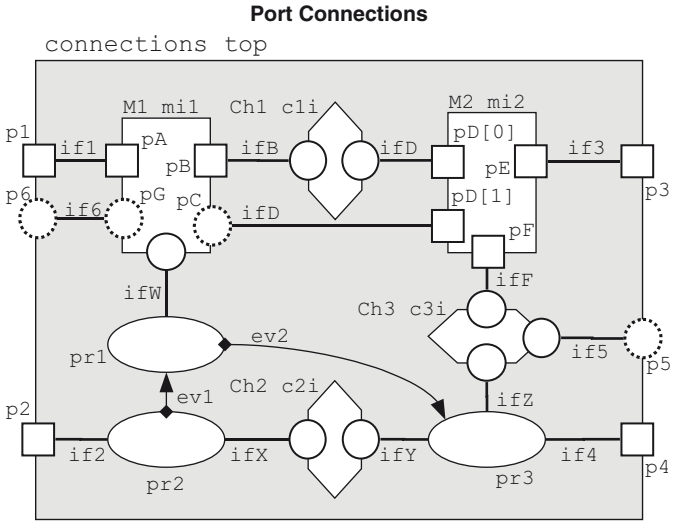


Fig. 11.9 Connectivity possibilities

This diagram (Fig. 11.9) is quite busy. Let's examine the pieces by name, and then discuss the rules of interconnection.

First, there are three modules represented with rectangles. The enclosing module instance is named `top`. The two submodule instances within top are named `mi1` and `mi2`.

Each of the modules has one or more ports represented with squares. Directional arrows within the ports indicate the primary flow of information. The ports for top are `p1`, `p2`, `p3`, `p4`, `p5`, and `p6`, which use interfaces named `if1`, `if2`, `if3`, `if4`, `if5`, and `if6`, respectively.

The ports for `mi1` are `pA`, `pB`, `pC`, and `pG`, which are connected to interfaces named `if1`, `ifB`, `ifD`, and `if6`, respectively.

Module `M1` also provides interfaces `ifW` and `if6`.

The ports for `mi2` are `pD[0]`, `pD[1]`, `pE`, and `pF`, which are connected to interfaces named `if3`, `ifD`, and `ifF`, respectively.

Next, three instances of channels represented with hexagonal shapes exist within `top`. These are named `c1i`, `c2i`, and `c3i`.

Each channel implements one or more interfaces represented by circles with a bent arrow. The arrow is intended to indicate the possibility of a call returning a value. It is possible for a channel to implement only a single interface. Channel `c1i` implements interfaces `ifB` and `ifD`. Channel `c2i` implements interfaces `ifX` and `ifY`. Finally, channel `c3i` implements interfaces `if5`, `ifF`, and `ifZ`.

Last, there are three processes named `pr1`, `pr2`, and `pr3`. For this description, we don't need to know what type of processes (i.e., threads vs. methods). There are two explicit events, `ev1` and `ev2` used for signaling between processes.

From this diagram, several rules may be observed. As we already know, processes may communicate with processes:

- At the same level either via channels or synchronized via events
- Outside the local design module through ports bound to channels by way of interfaces
- In submodule instances via interfaces to channels connected to the submodule ports or by way of interfaces through the module itself of an ***`sc_export`***

Any other attempt at inter-process communication is either forbidden or dangerous.

Ports may connect via interfaces only to local channels, ports of submodules, or to processes indirectly.

There are a few interesting features that will be discussed later. First, module instance `mi1` implements an interface `ifW`. Second, port `pD` appears to be an array of size 2. This is known as a port array. Finally, port `p5` and port `pC` illustrate the **`sc_export`**.

As a summary, let's view this information in a tabular format (Table 11.1).

Table 11.1. Ways to interconnect

From	To	Method
Port	Submodule	Direct connect via **sc_port**
Process	Port	Direct access by process
Submodule	Submodule	Local channel connection
Process	Submodule	Local channel connection or via **sc_export or** interface implemented by sub-module[a]
Process	Process	Events or local channel
Port	Local channel	Direct connect via **sc_export**

[a]In this case, the module is also known as a hierarchical channel, which will be discussed later.

11.5 Port Connection Mechanics

Modules are connected to channels after both the modules and channels have been instantiated. There are two syntaxes for connecting ports: by name and by position. Due to the error-prone nature of positional notation (especially since the number of ports on modules tends to be large and changes), the authors strongly prefer connection by name. Here are both syntaxes (Fig. 11.10):

```
mod_inst.portname(channel_instance); // Named
mod_instance(channel_instance,…); // Positional
```

Fig. 11.10 Syntax of port connectivity

An example should help greatly. We'll use a simple video mixer example with a color space transformation. For this example, we will use two standard SystemC interface classes, **sc_fifo_in_if** and **sc_fifo_out_if,** which support `read`() and `write`(value), respectively. First, we introduce the module definitions (Fig. 11.11, 11.12 & 11.13).

```
//FILE: Rgb2YCrCb.h
SC_MODULE(Rgb2YCrCb) {
  sc_port<sc_fifo_in_if<RGB_frame> >    rgb_pi;
  sc_port<sc_fifo_out_if<YCRCB_frame> > ycrcb_po;
};
```

Fig. 11.11 Example of port interconnect setup (1 of 3)

```
//FILE: YCRCB_Mixer.h
SC_MODULE(YCRCB_Mixer) {
  sc_port<sc_fifo_in_if<float> >        K_pi;
  sc_port<sc_fifo_in_if<YCRCB_frame> >  a_pi, b_pi;
  sc_port<sc_fifo_out_if<YCRCB_frame> > y_po;
};
```

Fig. 11.12 Example of port interconnect setup (2 of 3)

```
//FILE: VIDEO_Mixer.h
SC_MODULE(VIDEO_Mixer) {
  // ports
  sc_port<sc_fifo_in_if<YCRCB_frame> >  dvd_pi;
  sc_port<sc_fifo_out_if<YCRCB_frame> > video_po;
  sc_port<sc_fifo_in_if<MIXER_ctrl> >   control;
  sc_port<sc_fifo_out_if<MIXER_state> > status;
  // local channels
  sc_fifo<float>       K;
  sc_fifo<RGB_frame>   rgb_graphics;
  sc_fifo<YCRCB_frame> ycrcb_graphics;
  // local modules
  Rgb2YCrCb   Rgb2YCrCb_i;
  YCRCB_Mixer YCRCB_Mixer_i;
  // constructor
  VIDEO_Mixer(sc_module_name nm);
  void Mixer_thread();
};
```

Fig. 11.13 Example of port interconnect setup (3 of 3)

Now, let's look at interconnection of the preceding modules using both named (Fig. 11.14) and positional (Fig. 11.15) syntaxes.

```
SC_HAS_PROCESS(VIDEO_Mixer);
VIDEO_Mixer::VIDEO_Mixer(sc_module_name nm)
: sc_module(nm)
, Rgb2YCrCb_i("Rgb2YCrCb_i")
, YCRCB_Mixer_i("YCRCB_Mixer_i")
{
  // Connect
  Rgb2YCrCb_i.rgb_pi(rgb_graphics);
  Rgb2YCrCb_i.ycrcb_po(ycrcb_graphics);
  YCRCB_Mixer_i.K_pi(K);
  YCRCB_Mixer_i.a_pi(dvd_pi);
  YCRCB_Mixer_i.b_pi(ycrcb_graphics);
  YCRCB_Mixer_i.y_po(video_po);
}
```

Fig. 11.14 Example of port interconnect by name

Although slightly more code than the positional notation, the named port syntax is more robust, and tools exist to reduce the typing tedium.

```
SC_HAS_PROCESS(VIDEO_Mixer);
VIDEO_Mixer::VIDEO_Mixer(sc_module_name nm)
 : sc_module(nm)
 {
   // Instantiate
   Rgb2YCrCb_iptr   = new Rgb2YCrCb(
                            "Rgb2YCrCb_i"
                          );
   YCRCB_Mixer_iptr = new YCRCB_Mixer(
                            "YCRCB_Mixer_i"
                          );
   // Connect
   (*Rgb2YCrCb_iptr)( rgb_graphics
                     ,ycrcb_graphics
                    );
   (*YCRCB_Mixer_iptr)( K
                       ,dvd_pi
                       ,ycrcb_graphics
                       ,video_po
                      );
 }
```

Fig. 11.15 Example of port interconnect by position

The problem with positional connectivity is that of keeping the ordering correct. In large designs, middle- and upper-level modules frequently have a large number of ports (potentially multiple 10s), and it is common to add or remove ports late in the design. Using a positional notation can quickly lead to debug problems. That is why we recommend avoiding the positional syntax entirely, and always using a named port approach.

GUIDELINE: Whenever possible, use the named port interconnection style.

How does it work? Whereas the complete details require an extensive investigation of the SystemC library code, we can provide a short answer. When the code instantiating an **`sc_port`** executes, the **`operator()`** is overloaded to take a channel object by reference and saves a pointer to that reference internally for later access by the port. Thus, we recall a port is an interface pointer to a channel that implements the interface.

11.6 Accessing Ports From Within a Process

Connecting ports between modules and channels is of no great value unless a process somewhere in the design can initiate activity over the channels. This section will show how to access ports from within a process. The **`sc_port`**

overloads the C++ **operator->()**, which allows a simple syntax (Fig. 11.16) to access the referenced interface.

```
portname->method(optional_args);
```

Fig. 11.16 Syntax of port access

Continuing the previous example, we now illustrate (Fig. 11.17) port access in action. In the following, `control` and `status` are the ports; whereas, K is a local channel instance. Notice use of the **operator->** when accessing ports.

```
void VIDEO_Mixer::Mixer_thread() {
  …
  switch (control->read()) {
    case MOVIE: K.write(0.0f); break;
    case MENU:  K.write(1.0f); break;
    case FADE:  K.write(0.5f); break;
    default:    status->write(ERROR); break;
  }
  …
}
```

Fig. 11.17 Example of port access

Ports feel and behave as if they were pointers. Indeed that is a good way to think of them even though this is not precisely correct. A mnemonic may help here. P is for port and P is for pointer. When accessing channels through ports, always use the pointer operator (i.e., ->).

11.7 Exercises

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

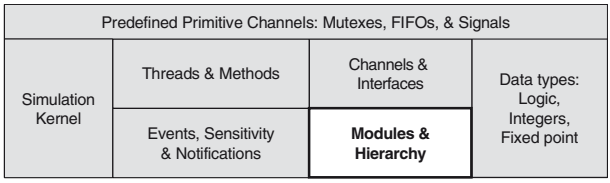
Exercise 11.1: Examine, compile, and run the `sedan` example. Which styles are simplest?

Exercise 11.2: Examine, compile, and run the `convertible` example. Notice the forward declarations of `Body` and `Engine`. How might this be an advantage when providing IP?

Exercise 11.3: Examine, compile, and run the `VIDEO_Mixer` examples. Change the port ordering, and insert a new port (with no functionality). What problems does this cause?

Exercise 11.4: In the VIDEO_Mixer port interconnect by name example, change the code from direct to indirect submodule instantiation.

Chapter 12
More on Ports & Interfaces



Specialized & sc_export

This chapter continues our discussion of ports as we go beyond the basics and explore more advanced concepts. We start out with a look at some standard interfaces that can be used to build ports. Next, we discuss built-in specialized ports and their conveniences, especially with regard to static sensitivity. Finally, we present the concept of port arrays, and finish the chapter with a different type of port, **sc_export**<*T*>.

12.1 Standard Interfaces

SystemC provides a variety of standard interfaces that go hand in hand with the built-in channels discussed previously. This section describes these interfaces. This section is a more precise definition of the interface syntax, and it provides a basis for creating custom channels that will be discussed in the following chapter.

12.1.1 SystemC FIFO Interfaces

Two interfaces, **sc_fifo_in_if**<*T*> and**sc_fifo_out_if**<*T*>, are provided for the **sc_fifo**<*T*>channel. Together, these interfaces provide all of the methods implemented by **sc_fifo**<*T*>. In fact, the interfaces were defined prior to the creation of the channel. The channel simply becomes the place to implement the interfaces and holds the data implied by the functionality of a FIFO.

The interface, **sc_fifo_out_if**<*T*>, partially shown in the following figure, provides all the methods for output from a module into an **sc_fifo**<*T*>. The module pushes data onto the FIFO using **write()** or **nb_write()** . The **num_free()** indicates how many locations are free. The **data_read_event()** method may be used dynamically to wait for free space. We've discussed all of these methods in Chapter 8, Basic Channels, in the **sc_fifo**<*T*>channel discussion.

Notice in the following figure (Fig. 12.1) that the interface itself is templated on a class name just like the corresponding channel.

D.C. Black et al., *SystemC: From the Ground Up*, Second Edition,
DOI 10.1007/978-0-387-69958-5_12, © Springer Science+Business Media, LLC 2010

```
// Definition of sc_fifo<T> output interface
template <class T>
class sc_fifo_out_if: virtual public sc_interface {
public:
  virtual void write(const T& ) = 0;
  virtual bool nb_write(const T& ) = 0;
  virtual int num_free() const = 0;
  virtual const sc_event&
                  data_read_event() const = 0;
};
```

Fig. 12.1 **sc_fifo** output interface definitions—abbreviated

The other interface, **sc_fifo_in_if**<*T*>, provides all the methods for input to a module from an **sc_fifo**<*T*>. The module pulls data from the FIFO using **read()** or **nb_read()**. The **num_available()** indicates how many locations are occupied, if any. The **data_written_event()** method may be used to dynamically wait for a new value to become available.

Again, all of these methods were discussed in Chapter 8 in the **sc_fifo**<*T*> channel discussion.

Here (Fig. 12.2) is the corresponding portion of the **sc_fifo_in_if**<*T*> interface definition:

```
// Definition of sc_fifo<T> input interface
template<class T>
class sc_fifo_in_if: virtual public sc_interface{
public:
  virtual void read( T& ) = 0;
  virtual T read() = 0;
  virtual bool nb_read( T& ) = 0;
  virtual int num_available()const = 0;
  virtual const sc_event&
            data_written_event() const = 0;
};
```

Fig. 12.2 **sc_fifo** input interface definitions—abbreviated

Something interesting to notice about the **sc_fifo**<*T*> interfaces is that if you use the **read()** and **write()** methods in your module and do not rely on the other methods, then your use of these interfaces is very compatible with the corresponding **sc_signal**<*T*> interfaces, which we will discuss next. In other words, you might want to simply swap out the interfaces and the channels; however, doing so would be dangerous. Remember **sc_fifo**<*T*>::**read()**[1] and **sc_fifo**<*T*>::**write()** are blocks waiting for the FIFO to empty; whereas, **sc_signal**<*T*>::**read()** and **sc_signal**<*T*>::**write()** arenon-blocking. This likely result is undesirable behavior.

[1] If you are having a hard time with this syntax, refer to the C++ scope resolution operator in Appendix A.

12.1.2 SystemC Signal Interfaces

Similar to sc_**fifo**<*T*>, two interfaces, **sc_signal_in_if**<*T*> and **sc_signal_inout_if**<*T*>, are provided for the **sc_signal**<*T*>channel. These two interface classes provide all of the methods provided by **sc_signal**<*T*>. Again, the interfaces were defined prior to the creation of the channel. The channel simply becomes the place to implement the interfaces and provides the request-update behavior implied for a signal.

Here (Fig. 12.3) is a portion of the **sc_signal_inout_if**<*T*> interface

```
// Definition of sc_signal<T> input/output interface
template<class T>
class sc_signal_inout_if: public sc_signal_in_if<T>
{
public:
  virtual void write( const T& ) = 0;
};
```

Fig. 12.3 **sc_signal** input/output interface definitions—abbreviated

definition:

There are two rather interesting things to notice in the preceding interface. First, if you need an **sc_signal**<*T*> output interface, use the **sc_signal_inout_if**<*T*>. This interface lets a module have access to the value of an output signal directly through a read, rather than being forced to keep a local copy in the manner required by VHDL.

Previous versions of SystemC included an **sc_signal_out_if**<*T*>interface that was type defined to **sc_signal_inout_if**<*T*>. That interface has been deprecated, and so you may see this in older code.

The update portion of the behavior is provided as a result of a call to **request_update()** that is provided indirectly as a result of a call from **sc_signal**<*T*>::**write()**. The update is implemented with the protected **sc_signal**<*T*>::**update()** method call. The **sc_signal_in_if**<*T*> interface (Fig. 12.4) provides access to the results through **sc_signal**<*T*>::**read()**.

```
// Definition of sc_signal<T> input interface
template<class T>
class sc_signal_in_if: virtual public sc_interface {
public:
  virtual const sc_event&
                  value_changed_event() const = 0;
  virtual const T& read() const = 0;
  virtual bool event() const = 0;
};
```

Fig. 12.4 **sc_signal** input interface definitions

12.1.3 sc_mutex and sc_semaphore Interfaces

The two channels, **sc_mutex** and **sc_semaphore**, also provide interfaces (Figs. 12.5 & 12.6) for use with ports. It is interesting to note that neither interface provides any event methods for sensitivity. If you require event sensitivity, you must write your own channels and interfaces as discussed in the next chapter.

```
// Definition of sc_mutex_if interface
class sc_mutex_if: virtual public sc_interface {
public:
  virtual int lock() = 0;
  virtual int trylock() = 0;
  virtual int unlock() = 0;
};
```

Fig. 12.5 **sc_mutex** interface definitions

```
// Definition of sc_semaphore_if interface
class sc_semaphore_if: virtual public sc_interface
{
public:
  virtual int wait() = 0;
  virtual int trywait() = 0;
  virtual int post() = 0;
  virtual int get_value() const = 0;
};
```

Fig. 12.6 **sc_semaphore** interface definitions

12.2 Sensitivity Revisited: Event Finders and Default Events

Recall from Chapter 6, Concurrency, that processes can be made sensitive to events. Also recall from Chapter 8, Basic Channels, that standard channels often provide methods that provide references to events (e.g., **sc_fifo**::**data_written_event()**). Since ports are defined on interfaces to channels, it is only natural to want sensitivity to events defined on those channels.

For example, it might be nice to create an **SC_METHOD** process statically sensitive to the **data_written_event()** or perhaps to monitor an **sc_signal**<*T*> for any change in the data using the **value_changed_event()**. You might even want to monitor a subset of possible events such as a positive edge transition (i.e., **false** to **true**) on a **sc_signal<bool>**.

Using static sensitivity for the situations above has a hidden complexity. Ports are pointers that become initialized during elaboration, and they are undefined at the time when the **sensitive** method needs to know about them. SystemC provides a solution for this difficulty in the form of a special class, the **sc_event_finder**.

The **sc_event_finder** defers the determination of the actual event until after elaboration. Unfortunately, the **sc_event_finder** has a minor complication. An **sc_event_finder** must be defined for each event defined by the interface. Thus, it is commonplace to define template specializations of port/interface combinations to instantiate a port and to include an **sc_event_finder** in the specialized class.

Suppose you want to create a port with sensitivity to the positive edge event of a Boolean signal port using the **sc_signal_in_if<bool>::posedge_event()** member function as shown in the following example (Fig. 12.7):

```
 classeslx_port
   :public sc_port<sc_signal_in_if<bool>,1>
 {
public:
// Use a typedef to shorten syntax below
   typedef sc_signal_in_if<bool>if_type;
   sc_event_finder& ef_posedge_event() const {
     return *newsc_event_finder_t<if_type>(
                  *this,
                  &if_type::posedge_event
                );
   }//end ef_posedge_event
 };
```

Fig. 12.7 Example of specialized port[2] implementing an event finder

Let's examine the preceding example. First, our custom event finder is inheriting from a specialized port on the second line. Second, to save some typing, we've created a **typedef** called if_type, which refers to the interface specialization. The new method, ef_posedge_event**()**[3], creates a new **event_finder** object and returns a reference to it. The constructor for an **sc_event_finder** takes two arguments: a reference to the port being found (***this**), and a reference to the member function, (**posedge_event()**) that returns a reference to the event of interest. The preceding example returns a reference to the event finder, which can be used by **sensitive**.

Now, the preceding specialization may be used as follows (Fig. 12.8):

```
SC_MODULE(my_module) {
  eslx_port my_p;
  …
  SC_CTOR(…) {
    SC_METHOD(my_method);
    sensitive<< my_p.ef_posedge_event();
  }
  void my_method();
  …
};
```

Fig. 12.8 Example of event finder use

[2] See next section for a discussion of specialized ports.

[3] The prefix ef_ is a convention. Some groups might prefer a suffix, _ef. In any case, a convention should be adopted.

A related and useful concept for sensitivity lists in SystemC is the ability to be sensitive to a port. The idea is that a process sensitive to a port, typically an **SC_METHOD**, is concerned with any change on that port. Obviously, this may be coded similar to Fig. 12.7 using **value_changed_event()** instead of **posedge_event()**.

As a syntactical simplification, SystemC also allows specifying a port name if and only if the associated interface provides a method called **default_event()** that returns a reference to an **sc_event**. The standard interfaces for **sc_signal**<*T*> and **sc_buffer**<*T*> provide this method. If you design your own interfaces, you will need to supply this method yourself.

12.3 Specialized Ports

Event finders are not particularly difficult to code; however, they are additional coding and understanding the implementation is challenging. To reduce that burden, SystemC provides a set of template specializations that provide port definitions on the standard interfaces and include the appropriate event finders.

It is important to know the port specializations for two reasons. First, you will doubtless have need for the common event finders at some point. Second, you will encounter their use in code from other engineers.

Let's take a look at the syntax of FIFO specializations (Fig. 12.9):

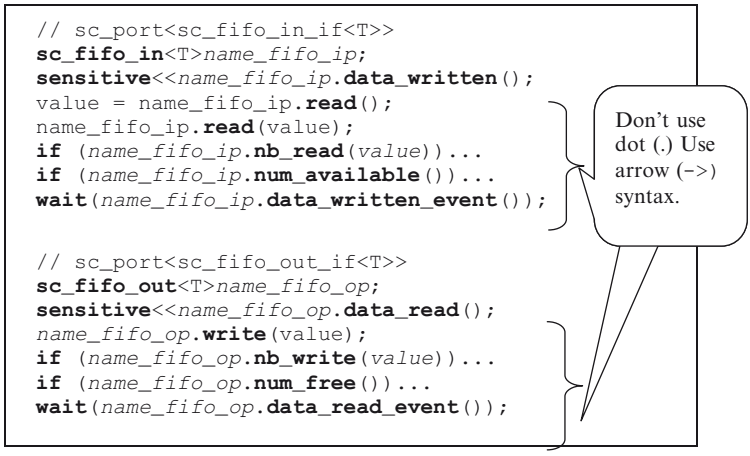


Fig. 12.9 Syntax of FIFO port specializations

These specializations have a minor downside that has to do with how ports are to be referenced. Notice in the following syntax figures that methods such as **read()** are defined. Recall from the last chapter that processes invoke port methods using the pointer operator (->). With specialized ports, you may also use dot (e.g., my_sig.**read()**). This syntax has the unfortunate effect of creating bad habits that could cause you to stumble later.

You may still use the pointer methods in processes. With exception to the new sc_event_finder methods and initialization, we recommend you use the arrow form whenever possible.

GUIDELINE: Use dot (.) in the elaboration section of the code, but use arrow (->) in processes.

This style will help you differentiate port accesses from local channel accesses and reduce confusion.

Let's look at an example (Fig. 12.10) using the FIFO port specializations using the guideline:

Now we'll examine specialized ports (Fig. 12.11) for evaluate-update channels such as **`sc_signal`**<*T*>. We left out the obvious duplication of member functions

```
// Equalizer.h
SC_MODULE(Equalizer) {
  sc_fifo_in<double> raw_fifo_ip;
  sc_fifo_out<double> equalized_fifo_op;
  void equalizer_thread();
  SC_CTOR(Equalizer) {
    SC_THREAD(equalizer_thread);
      sensitive<< raw_fifo_ip.data_written();
  }
};
```

Only available in port specialization.

```
// Equalizer.cpp
void Equalizer::equalizer_thread() {
  for(;;) {
    double sample; result;
    wait(); // uses static sensitivity
    raw_fifo_ip->nb_read(sample);
    ... /* process data */
    equalized_fifo_op->write(result);
  }//endforever
}
```

Fig. 12.10 Example using FIFO port specializations

such as **`read()`** that are better handled using the pointer operator (->).

In the **`sc_signal speicalized port`** syntax (Fig. 12.11), there are several features to note. First, there is the **`initialize()`** method. This method may be used at elaboration to establish the initial values of signal ports. This approach models start-up conditions properly, rather than waiting a delta-cycle and synchronizing processes at the start.

We also included one additional syntax that we especially don't like[4], the assignment operator (=), on the last line. When used, this operator can be especially confusing. Unless you realize the name on the left is a signal port, the behavior will seem bizarre. Remember that signals have an evaluate-update behavior; therefore the

[4] Some of this syntax was provided for backwards compatibility with earlier versions of SystemC (specifically 1.x).

```
// sc_port<sc_signal_in_if<T>>
 sc_in<T> name_sig_ip;
 sensitive << name_sig_ip.value_changed();

 // Additional sc_in specializations...
 sc_in<bool> name_bool_sig_ip;
 sc_in<sc_logic >name_log_sig_ip;
 sensitive << name_sig_ip pos();
 sensitive << name_sig_ip neg();

 // sc_port<sc_signal_out_if<T>>
 sc_inout<T> name_sig_op;
 sensitive << name_sig_op.value_changed();
 sc_inout_resolved<N> name_rsig_op;
 sc_inout_rv<N> name_rsig_op;
 sc_inout<T> name_rsig_op;
 sc_inout_resolved<T> name_rsig_op;
 sc_inout_rv<T> name_rsig_op;
 // everything under sc_in<T> plus the following...
 name_sig_op.initialize(value);
 name_sig_op = value; // <-- DON'T USE!!!
```

Fig. 12.11 Syntax of signal port specializations

changes from this assignment are not reflected until the next delta-cycle. This is quite different from ordinary assignment, and very confusing to most programmers.

GUIDELINE: To avoid confusion, never use the assignment operator with **`sc_signal<T>`** or **`sc_port<sc_signal_inout_if<T> >.`** Instead, use the **`write()`** method.

Let's look at an example using signal port specializations (Fig. 12.12 & 12.13). This example is a typical hardware block, a 32-bit linear feedback shift register (LFSR) commonly used with built-in self-test (BIST). Notice the use of **`pos()`** and **`initialize()`**.

```
//FILE: LFSR_ex.h
SC_MODULE(LFSR_ex) {
  // Ports
  sc_in<bool> sample;
  sc_out<sc_int<32>> signature;
  sc_in<bool> clock;
  sc_in<bool> reset;
  // Constructor
  SC_CTOR(LFSR_ex) {
    // Register process
    SC_METHOD(LFSR_ex_method);
    sensitive<< clock.pos() << reset;
    signature.initialize(0);
  }
  // Process declarations & Local data
  void LFSR_ex_method();
  sc_int<32> LFSR_reg;
};
```

Fig. 12.12 Example of signal port specializations—header

```
//FILE: LFSR_ex.cpp
#include "LFSR.h"
void LFSR_ex::LFSR_ex_method() {
  if (reset->read() == true) {
  LFSR_reg = 0;
  signature->write(LFSR_reg);
  }
else {
    bool lsb =LFSR_reg[31]^LFSR_reg[25]^LFSR_reg[22]
             ^LFSR_reg[21]^LFSR_reg[15]^LFSR_reg[11]
             ^LFSR_reg[10]^LFSR_reg[ 9]^LFSR_reg[ 7]
             ^LFSR_reg[ 6]^LFSR_reg[ 4]^LFSR_reg[ 3]
             ^LFSR_reg[ 1]^LFSR_reg[ 0]
             ^ sample->read();
    LFSR_reg.range(31,1) = LFSR_reg.range(30,0);
    LFSR_reg[0] = lsb;
    signature->write(LFSR_reg);
  }//endelse
}
```

Fig. 12.13 Example of signal port specializations—implementation

12.4 The SystemC Port Array and Port Policy

The **sc_port**<*T*> provides additional template parameters we have not yet discussed: the array size parameter and the port policy parameter. The array size parameter allows the creation of a number of identical ports. This construct is referred to as a multi-port or port array. The port policy specifies whether zero, one, or all ports must be connected.

For example, a communication system might have a number of T1 interfaces all with the same connectivity. Another example might be an abstract hierarchical communication that may have any number of devices connected to it. The full **sc_port**<*T*> syntax follows (Fig. 12.14).

```
sc_port<interface[,N[,POL]]> portname;
// N=0..MAX Default N=1
// POL is of type sc_port_policy
// POL defaults to SC_ONE_OR_MORE_BOUND
```

Fig. 12.14 Syntax of **sc_port**<> declaration complete

N indicates the number of channels to be connected to the port. When N = 0, we have a special case that allows an almost unlimited number of ports. In other words, you may connect any number of channels to the port.

POL is of type sc_port_policy and it is an enumerated type and has three legal values:

- **SC_ONE_OR_MORE_BOUND**
- **SC_ZERO_OR_MORE_BOUND**
- **SC_ALL_BOUND**

The value of POL enables different checking regarding the connectivity to the port. The default is **SC_ONE_OR_MORE_BOUND**. **SC_ZERO_OR_MORE_BOUND** allows a port with no connections. **SC_ALL_BOUND** requires that there are N and only N channels connected to the port (unless N=0 and then the checking associated with **SC_ONE_OR_MORE_BOUND** is used), which was the only implementation in earlier versions of SystemC.

An example with a drawing may help with understanding the use of multiports. In the figure (Fig. 12.15), four distinct channels connected to four ports T1_ip[0…3] on the left, and on the right, nine separate channels connect to nine ports request_op[0…8]. The block with the ports represents an imaginary switch.

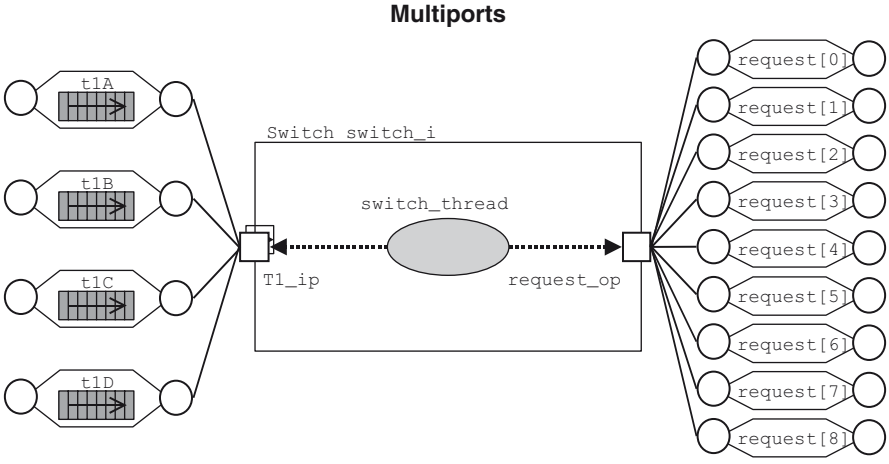


Fig. 12.15 Illustration of **sc_port<*T*>** array connectivity

Here is the header code (Fig. 12.16) for the switch in the above drawing:

```
//FILE: Switch.h
  SC_MODULE(Switch) {
    sc_port<sc_fifo_in_if<int>
           ,5
           ,SC_ONE_OR_MORE_BOUND
           > T1_ip;
    sc_port<sc_signal_inout_if<bool>
           ,0
           > request_op;
    ...
  };
```

Fig. 12.16 Example of **sc_port** array declaration

Channels are connected to port arrays the same way ordinary ports are connected, except port arrays have more than one connection. In fact, the basic port

syntax simply relies on the default that N = 1. Each connection is assigned a position in the array on a first-connected first-position basis.

Here (Fig. 12.17) is the corresponding example for the connections:

```
//FILE: Board.h
#include "Switch.h"
SC_MODULE(Board) {
  Switch switch_i;
sc_fifo<int> t1A, t1B, t1C, t1D;
sc_signal<bool> request[9];
  SC_CTOR(Board): switch_i("switch_i")
  {
    // Connect 4 T1 channels to the switch
    switch_i.T1_ip(t1A);
    switch_i.T1_ip(t1B);
    switch_i.T1_ip(t1C);
    switch_i.T1_ip(t1D);
    // Connect 9 request channels to the
    // switch request output ports
    for (unsigned i=0;i!=9;i++) {
      switch_i.request_op(request[i]);
    }//endfor
    ...
  }//end constructor
  ...
};
```

From preceding example.

Fig. 12.17 Example of **sc_port** array connections

The preceding example illustrates several things. First, a fixed port array of size 4 is connected directly to four FIFO channels. Second, an unbounded array is connected to an array of channels using a for-loop.

Access to port arrays from within a process is accomplished using the array syntax. This class also provides a method, **size()**, that may be used to examine the declared port size. This method is useful for situations where the array bounds are unknown (i.e., N = 0 or using **SC_ONE_OR_MORE_BOUND** or **SC_ZERO_OR_MORE_BOUND**).

Here (Fig. 12.18) is the code implementing the process accessing the multiports from within the Switch module:

Notice that the **size()** method requires the dot operator because it's defined in the specialized port class (e.g., request_op or T1_ip), rather than in the external channel (e.g., request[i] or t1A, t1B, t1C, t1D). On the other hand, port access to the channel uses the arrow operator as would be expected.

One current syntactical downside to **wait()** syntax may be seen in the preceding syntax. If you need to use "any event in the attached channels," current syntax requires an explicit listing.

```
//FILE: Switch.cpp
void Switch::switch_thread() {
  // Initialize requests
  for (unsigned i=0;i!=request_op.size();i++) {
    request_op[i]->write(true);
  }//endfor
  // Startup after first port is activated
  wait(T1_ip[0]->data_written_event()
      |T1_ip[1]->data_written_event()
      |T1_ip[2]->data_written_event()
      |T1_ip[3]->data_written_event()
  );
  while(true) {
    for (unsigned i=0;i!=T1_ip.size();i++) {
      // Process each port...
      int value = T1_ip[i]->read();
    }//endfor
  }//endwhile
}//end Switch::switch_thread
```

Fig. 12.18 Example of **sc_port** array access

There is an alternate possibility with dynamic threads. One could create and launch a separate thread to monitor each port and provide communication back via a shared local variable. We will examine this feature in the Advanced Topics chapter.

12.5 SystemC Exports

There is a second type of port called the **sc_export**<*T*>. The export is similar to standard ports in that the declaration syntax is defined on an interface. However, this port differs in connectivity. The idea of an **sc_export**<*T*> is to move the channel inside the defining module, thus hiding some of the connectivity details and using the port externally as though it were a channel. The following figure (Fig. 12.19) illustrates this concept:

Contrast this concept with Fig. 11.9 where we originally investigated ports.

The observant programmer might ask, Why use **sc_export** at all? After all, one could just access the internal **sc_channel** instance name directly using a hierarchical access. That approach works only if the interior channel is publicly accessible. For an IP provider, it may be desirable to export only specific channels and keep everything else private. Thus, **sc_export**<*T*> allows control over the interface.

Another reason for using **sc_export**<*T*> is to provide multiple interfaces at the top level. For example, consider the situation where you wish to create a channel that has two distinct interfaces for **sc_signal**<*T*>. Normally, it is not possible to inherit more than once from the same base class. However, with **sc_export**<*T*>, it is now possible to do this. Without **sc_export**<*T*>, we are

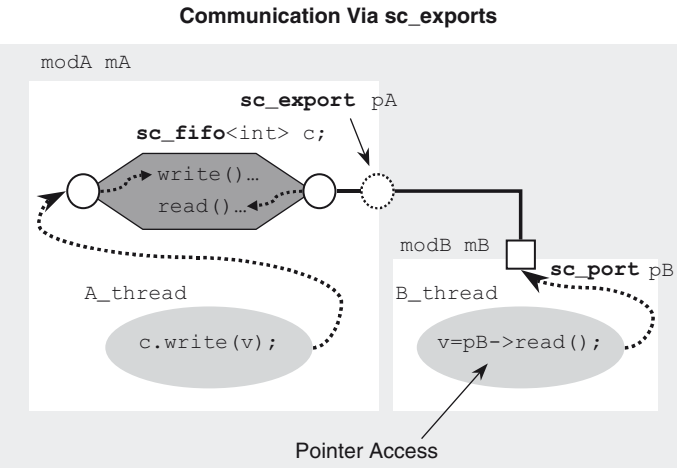


Fig. 12.19 How **sc_export** works

left to using the hierarchical channel, which allows for only a single top-level set of interfaces.

With an export, each **sc_export**<*T*> contains a specific interface. Since a connection is not required, **sc_export**<*T*> also allows creation of hidden interfaces. For example, a debug or test interface might be used internally by an IP provider, but not documented for the end user. Additionally, an export lets a development team or IP provider develop an instrumentation model. This model can be used extensively during architectural exploration and then dropped during regression runs when visibility is needed less and performance is key.

A hidden interface has the benefit of making the channel simpler to read and understand in the module where the IP is instantiated. The channel is easier to read since any programmatic instantiation of channels is hidden as well as much of the connectivity.

Another reason for using **sc_export**<*T*> is communications efficiency down the SystemC hierarchy. The **sc_export**<*T*> provides symmetry to the direction of C++ calls. Without **sc_export**<*T*>, SystemC ports would require additional channels and processes to allow an external process to pull information from a lower point in the model hierarchy. Efficiency is gained because **sc_export**<*T*> allows direct access to information (data) without intermediate channels.

The **sc_export**<*T*> syntax shown following (Fig. 12.20) **sc_port**<*T*>, but without the additional template parameters. This means that you cannot currently have an array for **sc_export**<*T*>.

```
sc_export<interface> portname;
```

Fig. 12.20 Syntax of **sc_export** declaration

Connectivity to an **sc_export**<*T*> requires some slight changes since the channel connections have now moved inside the module. Thus, we have (Fig. 12.21):

```
SC_MODULE(modulename) {
  sc_export<interface> portname;
  channel cinstance;
  SC_CTOR(modulename) {
    portname(cinstance);
  }
};
```

Fig. 12.21 Syntax **sc_export** internal binding to channel

Let's look at a simple example (Fig. 12.22). This example provides a process that is toggling an internal signal periodically. The **sc_export**<*T*> in this case is simply the toggled signal. For compactness, this example includes the entire module definition in the header.

```
SC_MODULE(clock_gen) {
  sc_export<sc_signal<bool>> clock_xp;
  sc_signal<bool> oscillator;
  SC_CTOR(clock_gen) {
    SC_METHOD(clock_method);
    clock_xp(oscillator); // connect sc_signal
                          // channel
                          // to export clock_xp
    oscillator.write(false);
  }
  void clock_method() {
    oscillator.write(!oscillator.read());
    next_trigger(10,SC_NS);
  }
};
```

Fig. 12.22 Example of simple **sc_export** declaration

To use the above **sc_export**<*T*>, we provide (Fig. 12.23) the corresponding instantiation of this simple module.

```
#include "clock_gen.h"
…
clock_gen clock_gen_i("clock_gen_i");
collision_detector cd_i("cd_i");
// Connect clock
cd_i.clock(clock_gen_i.clock_xp);
…
```

Fig. 12.23 Example of simple **sc_export** instantiation

Another powerful possibility with **sc_export**<*T*> is to let interfaces be passed up the design hierarchy as illustrated in the next figure (Fig. 12.24).

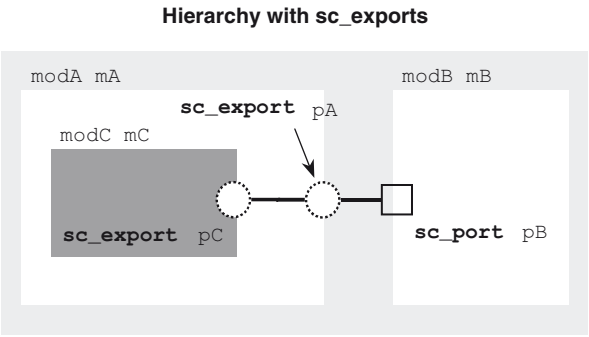


Fig. 12.24 **sc_export** used with hierarchy

Just like **sc_port**<*T*>, the **sc_export**<*T*> can be bound directly to another **sc_export**<*T*> in the hierarchy. Here (Fig. 12.25) is how to accomplish this binding:

```
SC_MODULE(modulename) {
  sc_export<interface> xportname;
  module minstance;
  SC_CTOR(modulename)
  , minstance("minstance")
  {
    xportname(minstance.subxport);
  }
};
```

Fig. 12.25 Syntax of **sc_export** internal binding to submodule

sc_export<*T*> has some caveats that may not be obvious. First, it is not possible to use **sc_export**<*T*> in a static sensitivity list. On the other hand, you can access the interface via the pointer operator (->). Thus, one can use **wait**(xportname->event()) on suitably defined interfaces accessed within an **SC_THREAD** process.

Second, as previously mentioned, it is not possible to have an array of **sc_export**<*T*> in the same manner as **sc_port**<*T*>. On the other hand, suitable channels may allow multiple connections, which may make this issue moot.

The following is an example of how an **sc_export**<*T*> might be used to model a complex bus including an arbiter to be provided as an IP component. First, let's look at the customer view (Fig. 12.26):

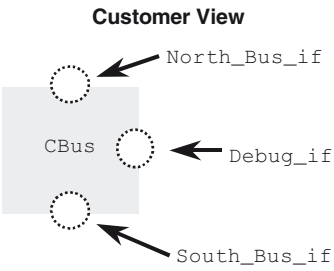


Fig. 12.26 Example of customer view of IP

This view would be defined using the following (Fig. 12.27) header file:

```
//CBus.h
#include "CBus_if.h"
class North_bus; // Forward declarations
class South_bus;
class Debug_bus;
class CBus_rtl;
SC_MODULE(CBus) {
  sc_export<CBus_North_if> north_p;
  sc_export<CBus_South_if> south_p;
  SC_CTOR(CBus);
private:
  North_bus* nbus_ci;
  South_bus* sbus_ci;
  Debug_bus* debug_ci;
  CBus_rtl*  rtl_i;
};
```

Fig. 12.27 Example of **`sc_export`** applied to a bus

Notice how the preceding code is independent of the implementation, and the end user is not compelled to hook up either bus. In addition, the debug interface is not provided in this example header. Here is the implementation view (Fig. 12.28):

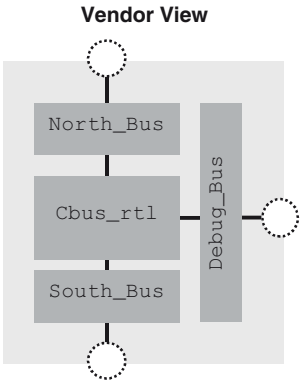


Fig. 12.28 Example of vendor view of IP

Here (Fig. 12.29) is the implementation code, which may be kept private:

```
//FILE: CBus.cpp
#include "CBus.h"
#include "North_bus.h"
#include "South_bus.h"
#include "Debug_bus.h"
#include "CBus_rtl_bus.h"
CBus::CBus(sc_module_name nm): sc_module(nm) {
  // Local instances
  nbus_ci  = new North_bus("nbus_ci");
  sbus_ci  = new South_bus("sbus_ci");
  debug_ci = new Debug_bus("debug_ci");
  rtl_i    = new CBus_rtl("rtl_i");
  // Export connectivity
  north_p(*nbus_ci);
  south_p(*sbus_ci);
  // Implementation connectivity
  …
}
```

Fig. 12.29 Example of **`sc_export`** applied to a bus constructor

In the preceding code, notice that the debug interface is not provided to the customer. Providing this interface would be an optional aspect of the IP that could be connected at compile time using an **`#ifdef`** or at run time using a "switch" from a control file or the command line.

12.6 Connectivity Revisited

Let's review port connectivity. The following diagram (Fig. 12.30) is copied from the previous chapter. It should become second nature to understand how to accomplish all the connections illustrated.

All of the possible connections are illustrated in this one figure. This figure is a handy reference when reviewing the SystemC connection rules, which are listed below:

1. A process may communicate with another process in the same module using a channel. For example, process `pr2` to process `pr3` via interface `ifX` on channel `c2i`.
2. A process may communicate with another process in the same module using an event to synchronize exchanges of information through data variables instantiated at the module level (e.g., within the module class definition). For example, process `pr2` to process `pr1` via event `ev1`.

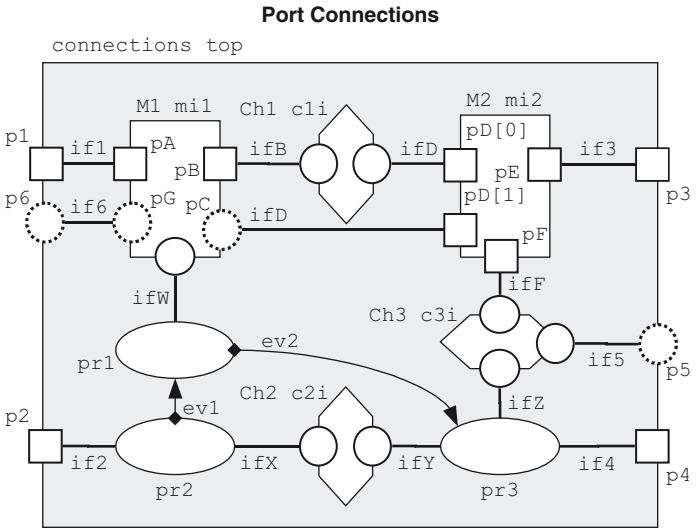


Fig. 12.30 Connectivity possibilities

3. A process may communicate with a process upwards in the design hierarchy using the interfaces accessed via **sc_port**<*T*>. For example, process `pr3` via port `p4` using interface `if4`.
4. A process may communicate with processes in submodule instances via interfaces to channels connected to the submodule ports. For example, process `pr3` to module `mi2` via interface `ifZ` on channel `c3i`.
5. An**sc_export**<*T*> may connect to another **sc_export**<*T*> via interfaces to local channels. For example, port `p5` to channel `c3i` using interface `if5`.
6. An **sc_port**<*T*> may connect directly to an **sc_port**<*T*> of submodules. For example, port `p1` is connected to port `pA` of submodule `mi1`.
7. An **sc_export**<*T*>may connect directly to an **sc_export**<*T*> of a submodule. For example, port `p6` is directly connected to port `pG` of submodule `mi1`.
8. An **sc_port**<*T*> may connect indirectly to a process by letting the process access the interface. This is just a process accessing a port described previously. For example, process `pr1` communicates with submodule `mi1` through interface `ifW`.
9. An **sc_port**<*T,N*> array may be used to create multiple ports using the same interface. For example, `pD[0]` and `pD[1]` of submodule `mi2`.

Finally, we present an equivalent diagram (Fig. 12.31) to the preceding. In this diagram, channels appear as slightly thickened lines. An **sc_port**<*T*> is represented with a square containing a circle to indicate the presence of an interface. This style is often used to simplify the schematic representation at the expense of slightly hiding the underlying functionality. In the next chapter, we will investigate more complex channels known as hierarchical channels.

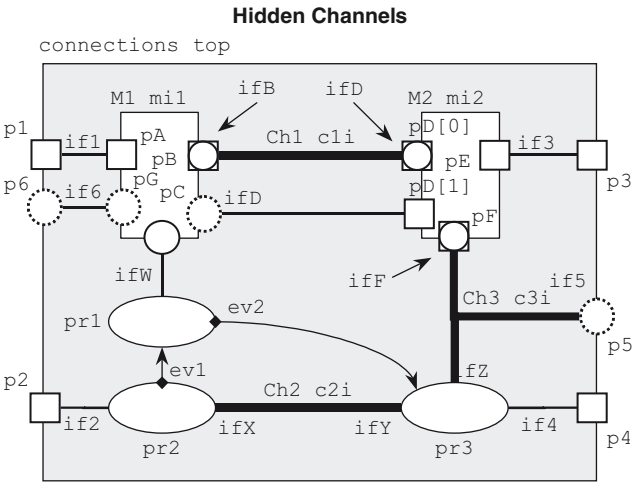


Fig. 12.31 Hidden channels

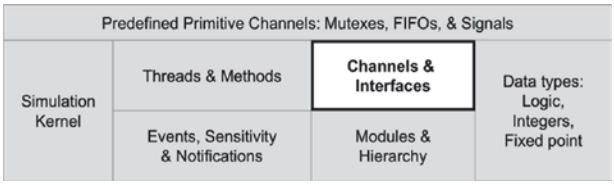
12.7 Exercises

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

Exercise 12.1: Examine, compile, and run the `static_sensitivity` example.

Exercise 12.2: Examine, compile, and run the `connections` example. See if you can identify all the connections shown in the figures in this chapter.

Chapter 13 Custom Channels and Data



Primitive & Hierarchical

We've already covered much of the syntax of SystemC. Now, we will focus on some of the more abstract concepts from which SystemC derives much of its power. This chapter illustrates how to create a variety of custom channels including: primitive channels, custom signals, custom hierarchical channels, and custom adaptors. Of these, custom signals and adaptors are probably the most commonly encountered.

13.1 A Review of SystemC Channels and Interfaces

In this section, we will review two of the four most important aspects of SystemC, channels and interfaces. The other two, modules and ports, have already been discussed in detail. Remember, SystemC channels implement communication between modules. SystemC interfaces provide an API and a means to allow independence of modules from the mechanisms of communication channels.

The basic structure of a channel is a class that inherits from one or more interfaces and a SystemC base class. The interface makes a channel usable with ports. Channels come in two flavors: primitive and hierarchical. Channels must inherit either from **`sc_prim_channel`** or **`sc_channel`**, which defines them as either primitive or hierarchical, respectively. This distinction in these latter two SystemC base classes is one of distinct capabilities and features. In other words, **`sc_prim_channel`** has capabilities not present in **`sc_channel`** and vice versa.

Primitive channels are intended to provide very simple and fast communications. They contain no hierarchy and no ports; primitive channels do not contain simulation processes. Primitive channels have the ability to use the evaluate-update paradigm as they inherit some specialized methods. These channels are discussed in the following section.

By contrast, hierarchical channels can have their own ports and processes, and they can contain hierarchy as the name suggests. In fact, hierarchical channels are really just modules that implement one or more interfaces. Hierarchical channels are intended to model complex communications buses such as PCI, HyperTransport™, AMBA™, or AXI™. Custom hierarchical channels are discussed later in this chapter.

D.C. Black et al., *SystemC: From the Ground Up*, Second Edition,
DOI 10.1007/978-0-387-69958-5_13, © Springer Science+Business Media, LLC 2010

Channels are important in SystemC because they enable several concepts:

- Appropriate channels enable safe communication among concurrent simulation processes.
- Channels in conjunction with ports clarify and delineate the relationships of communication (producer vs. consumer, master vs. slave, initiator vs. target).

Interfaces are important in SystemC because they enable the separation of communication from processing and allow independent refinement of the communication and functionality of a system.

13.2 The Interrupt, a Custom Primitive Channel

We discussed events in Chapter 6, Concurrency, and we saw how processes can use events to coordinate activities. We introduced hierarchy and ports in Chapter 10, Structure. This section answers the question of how we can provide a simple event or interrupt between processes located in different modules. Obviously, this interrupt is not the usual preemptive interrupt as used in software; instead, it is a handshake signal between modules or processes.

One approach might take a channel that has an event and simply use the side effect. For example, this approach could use sensitivity to **`sc_signal<bool>`** by use of the **`value_changed_event()`** method. However, using side effects is unsatisfying. Let us see how we might create a custom channel just for this purpose.

A proper channel must have one or more interfaces to implement. The ideal interface provides only the methods required for a particular purpose. For our channel, we'll create two interfaces: one interface for sending events, `eslx_interrupt_gen_if`, and another interface for receiving events, `eslx_interrupt_evt_if`. To allow and simplify use in static sensitivity lists, we'll specify a **`default_event()`**.

The interfaces are shown in the next figure (Fig. 13.1). Notice that interfaces are required to inherit from the **`sc_interface`** base class. Also, notice in `eslx_interrupt_evt_if` that **`default_event()`** has a specific calling signature. This signature is required for **`default_event()`** to be recognized by **`sensitive`**.

```
class eslx_interrupt_gen_if: public sc_interface {
public:
  virtual void notify() = 0;
  virtual void notify(sc_time t) = 0;
};
```

```
class eslx_interrupt_evt_if: public sc_interface {
public:
  virtual const sc_event& default_event() const = 0;
};
```

Fig. 13.1 Examples of custom channel interfaces

Next, we look at the implementation of the primitive channel shown in Fig. 13.2. The implementation has four features of interest.

First, the channel must inherit from **sc_prim_channel** and both of the interfaces we defined previously.

Second, the constructor for the channel has similar requirements to an **sc_module**; the constructor must construct the base class **sc_prim_channel**. This class has a single constructor that requires an instance name string.

Third, the channel must implement the methods compelled by the pure virtual functions in the interfaces from which it inherits. Thus, this channel must implement two versions of the **notify()** method and one **default_event()** method.

Fourth, we specify a private implementation of the copy constructor to prevent its use. Simply put, a channel should never be copied. This feature of the implementation provides a compile-time error if copying is attempted.

```
#include "eslx_interrupt_evt_if.h"
#include "eslx_interrupt_gen_if.h"

class eslx_interrupt
: public sc_prim_channel
, public eslx_interrupt_evt_if
, public eslx_interrupt_gen_if
{
public:
  // Constructors
  explicit eslx_interrupt()
  :sc_prim_channel(
             sc_gen_unique_name("eslx_interrupt"))
  {}//end constructor
  explicit eslx_interrupt(sc_module_name nm)
  :sc_prim_channel(nm)
  {} //end constructor
  // Methods
  void notify() { m_interrupt.notify(); }
  void notify(sc_time t) { m_interrupt.notify(t); }
  const sc_event& default_event() const
                 { return m_interrupt; }
private:
  sc_event m_interrupt;
  // Copy constructor so compiler won't create one
  eslx_interrupt( const eslx_interrupt& rhs)
  {} //end copy constructor
};
```

Fig. 13.2 Example of custom interface implementation (AKA channel)

13.3 The Packet, a Custom Data Type for SystemC

Creating custom primitive channels is not very common; however, instantiating an **sc_signal**<*T*> channel or an **sc_fifo**<*T*> is very common. SystemC defines all the necessary features for both of these channels when used with built-in data types.

For custom data types, SystemC requires you to define several methods for your data type.

The reasons for the required methods are easy to understand. As an example, both channels support read and write methods, which involve copying the custom data type. For this reason, SystemC requires the definition of the assignment operator (i.e., **operator=()**). Also, **sc_signal**<*T*> supports the method **value_changed_event()**, which implies the use of comparison. In this case, SystemC requires the definition of the equality operator (i.e., **operator==()**).

Finally, there are two other methods required by SystemC, streaming output (i.e., **ostream& operator<<()**) and **sc_trace()**. Streaming output allows for a pleasant printout of your data structure during debug. The trace function allows all or parts of your data type to be used with the SystemC trace facility. This function enables viewing of trace data with a waveform viewer. We'll explain waveform data tracing in the next chapter.

Consider the following C/C++ custom data type (Fig. 13.3), which might be used for PCI-X transactions:

```
struct eslx_pcix_trans {
  int devnum;
  int addr;
  int attr1;
  int attr2;
  int cmnd;
  int data[8];
  bool done;
};
```

Fig. 13.3 Example of user-defined data type

This structure or record contains all the information necessary to fully communicate a PCI-X transaction; however, it is not usable with an **sc_signal**<*T*> channel or an **sc_fifo**<*T*>. Let's add the necessary methods (Fig. 13.4) to support this usage.

Note the **friend** declarations, which also need to be declared outside the class definition in the header file, are required because **ostream** and **sc_trace** are globally defined.

We provide example implementations of the latter two methods here (Fig. 13.5):

It should be noted that the **sc_trace()** method is optional; however, best practices suggest that you should always provide this method. Observe that this method is always expressed in terms of other traces that are already defined (e.g., the built-in ones).

In some cases, it may be difficult to determine an appropriate representation. For **sc_trace()** as an example, **char*** or **string** have no real logical equivalent.

```
//FILE: eslx_pcix_trans.h
class eslx_pcix_trans {// previously as struct
  int devnum; // May need get and set methods
  int addr;   // and could rename member data to
  int attr1;  // match m_ naming conventions.
  int attr2;
  int cmnd;
  int data[8];
  bool done;
public:
  // Required by sc_signal<> and sc_fifo<>
  eslx_pcix_trans& operator =(
    const eslx_pcix_trans& rhs
  ) {
    devnum = rhs.devnum;  addr   = rhs.addr;
    attr1  = rhs.attr1;   attr2  = rhs.attr2;
    cmnd   = rhs.cmnd;    done   = rhs.done;
    for (unsigned i=0;i!=8;i++) data[i]=rhs.data[i];
    return *this;
  }
  // Required by sc_signal<>
  bool operator==(const eslx_pcix_trans& rhs)
                  const {
    return (
        devnum ==rhs.devnum && addr   ==rhs.addr
     && attr1  ==rhs.attr1  && attr2  ==rhs.attr2
     && cmnd   ==rhs.cmnd   && done   ==rhs.done
     && data[0]==rhs.data[0]&& data[1]==rhs.data[1]
     && data[2]==rhs.data[2]&& data[3]==rhs.data[3]
     && data[4]==rhs.data[4]&& data[5]==rhs.data[5]
     && data[6]==rhs.data[6]&& data[7]==rhs.data[7]
     );
  }
  friend ostream& operator<<(ostream& file,
                    const eslx_pcix_trans& trans);
  friend void sc_trace(sc_trace_file*& tf,
              const eslx_pcix_trans& trans,
              string nm);
};
```

Fig. 13.4 Example of SystemC user data type

In these cases, you may either convert to an unsigned fixed-bit-width vector (e.g., **`sc_bv`**), or omit it completely. However, remember that converting these representations is for ease of debug operations. Providing a complete and clear **`sc_trace`** is usually of much more value than you might originally think. The same can be said of appropriate representation for **`ostream`**.

You may also want to implement **`ifstream`** or **`ofstream`** to support verification needs.

As you can see, the added support is really quite minimal, and it is only required for custom data types.

```
//FILE: eslx_pcix_trans.cpp
#include "eslx_pcix_trans.h"
ostream& operator<<(ostream& os,
                    const eslx_pcix_trans& trans)
{
  os << "{" << endl << "  "
     << "cmnd: " << trans.cmnd   << ", "
     << "attr1:" << trans.attr1  << ", "
     …
     << "done:"  << (trans.done?"true":"false")
     << endl << "}";
  return os;
} // end
// trace function, only required if actually used
void sc_trace(sc_trace_file*& tf,
              const eslx_pcix_trans& trans,
              string nm)
{
  sc_trace(tf, trans.devnum,  nm + ".devnum");
  sc_trace(tf, trans.addr,    nm + ".addr");
  …
  sc_trace(tf, trans.data[7], nm + ".data[7]");
  sc_trace(tf, trans.done,    nm + ".done");
} // end trace
```

Fig. 13.5 Example of SystemC user data type implementation

13.4 The Heartbeat, a Custom Hierarchical Channel

Hierarchical channels are interesting because they're really hybrid modules. Technically, a hierarchical channel must inherit from **`sc_channel`**; however, **`sc_channel`** is really just a **`typedef`** for **`sc_module`**. Hierarchical channels must also inherit from an interface to let them be connected to an external **`sc_port`**<*T*>.

Why would you define a hierarchical channel? One use of hierarchical channels is to model complex buses such as PCI, AMBA, HyperTransport, or AXI. Another common use of hierarchical channels, adaptors, and transactors will be discussed in the next section.

To keep things simple, we'll model a basic clock or heartbeat. This clock will differ from the standard hardware concept that typically uses a Boolean signal. Instead, our heartbeat channel will issue a simple event. This usage would correspond to the **`posedge_event()`** used by so many hardware designs.

Because it's basic, the heartbeat is more efficient simulation-wise than a Boolean signal. Here (Fig. 13.6) is the header for our simple interface:

```
class eslx_heartbeat_if: public sc_interface {
public:
  virtual const sc_event& default_event() const = 0;
  virtual const sc_event& posedge_event() const = 0;
};
```

Fig. 13.6 Example of hierarchical interface header

It's no different than a primitive channel interface. Notice that we use method names congruent with **sc_signal**<*T*>. This convention will simplify design refinement. The careful design of interfaces is key to reducing work that is done later.

Let's look at the corresponding channel header (Fig. 13.7), which inherits from **sc_channel** instead of **sc_prim_channel** and has a process, **SC_METHOD**.

```
include "eslx_heartbeat_if.h"
class eslx_heartbeat
 :public sc_channel
 ,public eslx_heartbeat_if {
public:
  SC_HAS_PROCESS(eslx_heartbeat);
  // Constructor (only one shown)
  explicit eslx_heartbeat(sc_module_name nm
                          ,sc_time _period)
  :sc_channel(nm)
  ,m_period(_period)
  {
    SC_METHOD(heartbeat_method);
    sensitive << m_heartbeat;
  }
  // User methods
  const sc_event& default_event() const
  { return m_heartbeat; }
  const sc_event& posedge_event() const
  { return m_heartbeat; }
  void heartbeat_method(); // Process
private:
  sc_event m_heartbeat; // *The* event
  sc_time  m_period;    // Time between events
  // Copy constructor so compiler won't create one
  eslx_heartbeat( const eslx_heartbeat& );
};
```

Fig. 13.7 Example of hierarchical channel header

Let's see how it's implemented (Fig. 13.8):

```
#include <systemc>
#include "eslx_heartbeat.h"

void eslx_heartbeat::heartbeat_method(void) {
  m_heartbeat.notify(m_period);
}
```

Fig. 13.8 Example of hierarchical channel interface header

In the next chapter, we'll see the built-in SystemC clock, which has more flexibility at the expense of performance.

13.5 The Adaptor, a Custom Primitive Channel

Also known in some circles as transactors, adaptors are a type of channel specialized to translate between modules with different interfaces. Adaptors are used when moving between different abstractions. For example, an adaptor is commonly used between a testbench that models communications at the transaction level (i.e., TLM), and an RTL implementation that models communications at the pin-accurate level. Transaction-level communications might have methods that transfer an entire packet of information (e.g., a PCI-X transaction). Pin-accurate level communications use Boolean signals with handshakes, clocks, and detailed timing.

To make it easy to understand, we're going to investigate two adaptors. In this section, we'll see a simple primitive channel that uses the evaluate-update mechanism. In the following section, we'll investigate a hierarchical channel. For many of the simpler communications, an adaptor needs nothing more than some member functions and a handshake to exchange data. This setup often meets the requirements of a primitive channel. Many of the simpler adaptors could be of either type, since they don't require an evaluate-update mechanism.

We will now discuss an example design. The example design includes a stimulus (`stim`) and a response (`resp`) that is connected via an `eslx_interrupt` channel described in an earlier section. We now would like to replace `resp` with a refined RTL version, `resp_rtl`, which requires a **`sc_signal<bool>`** channel interface. The before and after example design is graphically shown in Fig. 13.9.

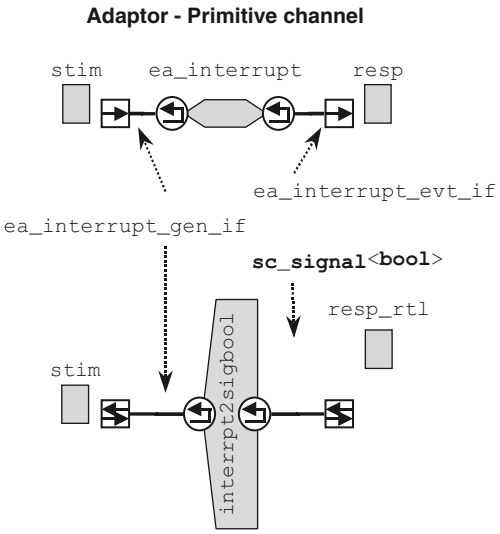


Fig. 13.9 Before and after adaptation

Here (Fig. 13.10) is the adaptor's header:

```
#include "eslx_interrupt_gen_if.h"
class interrupt2sigbool
: public sc_prim_channel
, public eslx_interrupt_gen_if
, public sc_signal_in_if<bool>
{
public:
  // Constructors
  explicit interrupt2sigbool()
  : sc_prim_channel(
      sc_gen_unique_name("interrupt2sigbool")) {}
  explicit interrupt2sigbool(sc_module_name nm)
  : sc_prim_channel(nm) {}
  // Methods for eslx_interrupt_gen_if
  void notify() {
    m_delay = SC_ZERO_TIME; request_update(); }
  void notify(sc_time t) {
    m_delay = t; request_update(); }
  // Methods for sc_signal_in_if<bool>
  const sc_event& value_changed_event() const
  { return m_interrupt; }
  const sc_event& posedge_event() const
  { return value_changed_event(); }
  const sc_event& negedge_event() const
  { return value_changed_event(); }
  const sc_event& default_event() const
  { return value_changed_event(); }
  // true if last delta cycle was active
  const bool& read() const {
    m_val = event(); return m_val;
  }
  // Did value change in the previous delta cycle?
  bool event() const {
    return (sc_delta_count() == m_delta+1);
  }
```

```
  bool posedge() const { return event(); }
  bool negedge() const { return event(); }
protected:
  // every update is a change
  void update() {
    m_interrupt.notify(m_delay);
    m_delta = sc_delta_count();
  }
private:
  sc_event m_interrupt;
  mutable bool m_val;
  sc_time  m_delay;
  uint64   m_delta;    // delta of last event
  // Copy constructor so compiler won't create one
  interrupt2sigbool( const interrupt2sigbool& );
};
```

Fig. 13.10 Example of primitive adaptor channel header

The first thing to notice is all the methods. Most of these are forced upon us because we are inheriting from the **sc_signal_in_if<bool>** class. Fortunately, most of them may be expressed in terms of others for this particular adaptor. Another way to handle excess methods is to provide stubbed **SC_FATAL**[1] messages with the assumption that nobody will use them.

The second feature of interest is the manner in which evaluate-update is handled. In the **notify()** methods, we update the delay and make a **request_update()** call to the scheduling kernel. When the delta-cycle occurs, the kernel will call our **update()** function that issues the appropriately delayed notification.

For the most part, this adaptor was simple. The hard part was obtaining a list of all the routines that needed to be implemented as a result of the interface. Listing the routines is accomplished easily enough by simply examining the interface definition in the Open SystemC Initiative library source code.

A third feature to note is the use of **sc_delta_count()**. It is used to determine that the interrupt event has occurred in the previous delta-cycle. The value returned by **sc_delta_count()** is incremented by one for each delta-cycle while the simulator is running.

Finally, for those not completely up on their C++, a comment on the **mutable bool**. The keyword **mutable** means changeable even if const. The **read()** method is defined in **sc_signal_in_if<bool>** interface, so we have to implement it. The **read()** method is defined as **const**, and it is required to return a **const** reference (&). We are using the member function **event()** to obtain a value, which is not a reference. So, we create a member data `m_val` to store the return value temporarily. Because the value is mutable, we are able to change it (even though the method is **const**) and return it as a **const** reference.

13.6 The Transactor, a Custom Hierarchical Channel

When a more complex communications interface is encountered; such as one that requires processes, hierarchy, or ports; then a hierarchical channel solution is required. The following processor interface problem demonstrates this type of channel.

Suppose we have a testbench connected to an abstract model of a memory, and wish to replace the abstract memory with an RTL model. On one side, we have a testbench that needs to use simple transaction calls to verify the functionality of the memory. On the other side, we have a peripheral, an 8K x 16 memory. To not change the testbench, we insert an adaptor between the testbench and the RTL memory. This adaptor allows the testbench to convert transactions into pin-level stimulus.

[1] SC_FATAL is discussed in Chapter 14, Additional Topics.

Graphically, here (Fig. 13.11) are the elements of the design:

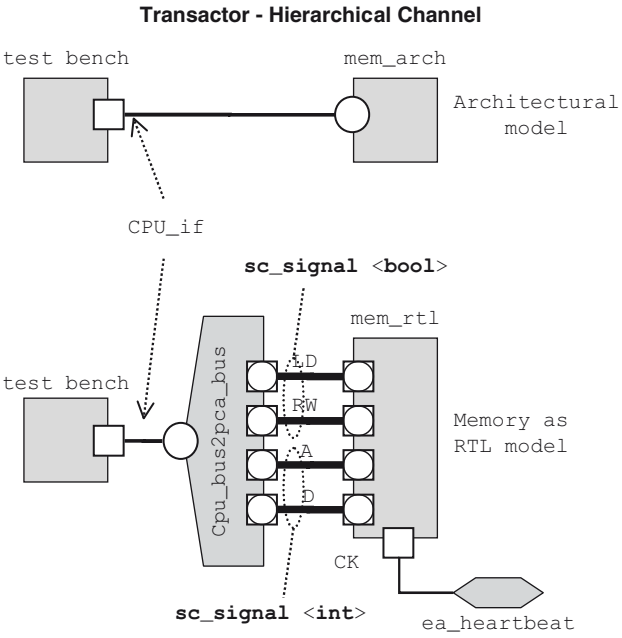


Fig. 13.11 Testbench adaptation using hierarchical channels

This figure actually has two hierarchical channels. The architectural model of the memory is a module implementing an interface, in this case the `CPU_if`. Our memory was designed to hang directly off the CPU.

Let's take a look at the CPU interface (Fig. 13.12):

```
class CPU_if: public sc_interface {
public:
  virtual void write(unsigned long addr
                     ,long data) = 0;
  virtual long read(unsigned long  addr) = 0;
};
```

Fig. 13.12 Example of simple CPU interface

The corresponding memory implementation is a straightforward channel (Fig. 13.13):

```
//FILE: mem_arch.h
#include "CPU_if.h"
class mem
: public sc_channel
, public CPU_if
{
public:
  // Constructors & Destructor
  explicit mem(sc_module_name nm
               ,unsigned long ba
               ,unsigned sz)
  :sc_channel(nm)
  ,m_base(ba)
  ,m_size(sz)
  { m_mem = new long[m_size]; }
  ~mem() { delete [] m_mem; }
  // Interface Implementations
  virtual void write(unsigned long addr
                     ,long data) {
    if (m_start <= addr && addr < m_base+m_size) {
      m_mem[addr-m_base] = data;
    }
  }//end write
  virtual long read(unsigned long  addr) {
    if (m_base <= addr && addr < m_base+m_size) {
      return m_mem[addr-m_base];
    } else {
      cout << "ERROR:"<<name()<<"@"<<sc_time_stamp()
           << ": Illegal address: " << addr << endl;
      sc_stop(); return 0;
    }
  }//end read
private:
  unsigned long m_base;
  unsigned      m_size;
  long*          m_mem[];
  mem_arch(const mem_arch&); // Disable
};
```

Fig. 13.13 Example of hierarchical channel memory implementation

Now, suppose we have the following timing diagram (Fig. 13.14) for the pin-cycle accurate interface:

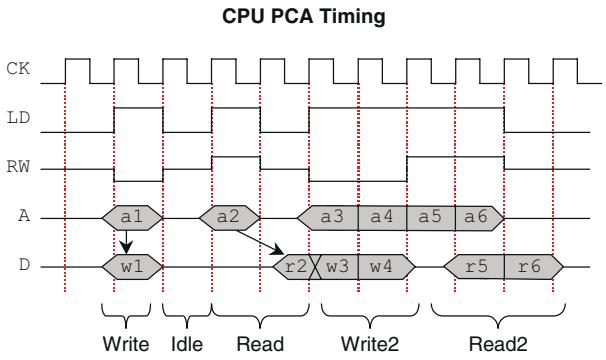


Fig. 13.14 CPU pin-cycle accurate timing

Notice that write transactions take place in a single clock cycle; whereas, the read transaction has a one-cycle delay for the first read in a burst. Also, this interface assumes a bidirectional data bus. Address and read/write have a non-asserted state. For this design, we'll allow this setup.

Here is the transactor's header (Fig. 13.15):

```
#include "CPU_if.h"
#include "eslx_heartbeat_if.h"
class cpu2pca
  :public sc_module
  ,public CPU_if
{
public:
  // Ports
  sc_port<eslx_heartbeat_if> ck; // clock
  sc_out<bool>               ld; // load/exec cmd
  sc_out<bool>               rw; // read high
                                 // write low
  sc_out<unsigned long>      a;  // address
  sc_inout_rv<32>            d;  // data
  // Constructor
  SC_CTOR(cpu2pca):FLOAT("ZZZZZZZZ") {}
  // Interface implementations
  void write(unsigned long addr
             ,long data);
  long read(unsigned long addr);
  // Useful constants
  const sc_lv<32> FLOAT;
private:
  cpu2pca(const cpu2pca&); // Disable
};
```

Fig. 13.15 Example of hierarchical transactor channel header

Clearly, with the preceding example, the basics of a module are present. Inheriting from `CPU_if` simply adds a few methods to be implemented, namely **`read()`** and **`write()`**.

An interesting point to ponder with channels (especially adaptors) is the issue of member function collisions. What if two or more interfaces that need to be implemented have identically named member functions with identical argument types?

There are two solutions. One solution is to modify the interface method in a renamed interface. This solution is ugly. Another solution is to isolate each interface to an **`sc_export`**<*T*>. This improved solution lets you use the implementation in a locally instantiated channel to complete your implementation.

Here (Fig. 13.16) is the implementation code for the transactor:

```
#include "cpu2pca.h"
enum operation {WRITE=false, READ=true};
void cpu2pca::write(unsigned long addr
                    ,long data) {
  wait(ck->posedge_event());
  ld->write(true);
  rw->write(WRITE);
  a->write(addr);
  d->write(data);
  wait(ck->posedge_event());
  ld->write(false);
}
long cpu2pca::read(unsigned long addr) {
  wait(ck->posedge_event());
  ld->write(true);
  rw->write(READ);
  a->write(addr);
  d->write(FLOAT);
  wait(ck->posedge_event());
  ld->write(false);
  return d->read().to_long();
}
```

Fig. 13.16 Example of hierarchical transactor channel implementation

The code for an adaptor can be very straightforward. For more complex applications; such as a PCI, AMBA, or AXI; the design of an adaptor may be more complex.

Because adaptors allow high-level abstractions to interface with lower-level implementations, they are very common in SystemC designs. Sometimes these hybrids are used as part of a design refinement process. At other times, they merely aid the development of verification environments. There are no fixed rules defining abstraction levels or how to use them.

13.7 Exercises

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

Exercise 13.1: Examine, compile, and run the `interrupt` example. Write a specialized port for this channel to support the method **`pos()`**.

Exercise 13.2: Examine, compile, and run the `pcix` example. Could this process of converting a **`struct`** to work with an **`sc_signal`** be automated? How?

Exercise 13.3: Examine, compile, and run the `heartbeat` example. Extend this channel to include a programmable time offset.

Exercise 13.4: Examine, compile, and run the `adapt` example. Notice the commented-out code from the adaptation of `resp` to `resp_rtl`.

Exercise 13.5: Examine, compile, and run the `hier_chan` example. Examine the efficiency of the calls. Extend the design to allow back-to-back reads and writes while using cycles efficiently.

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

D.C. Black et al., *SystemC: From the Ground Up*, Second Edition,
DOI 10.1007/978-0-387-69958-5_14, © Springer Science+Business Media, LLC 2010

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.

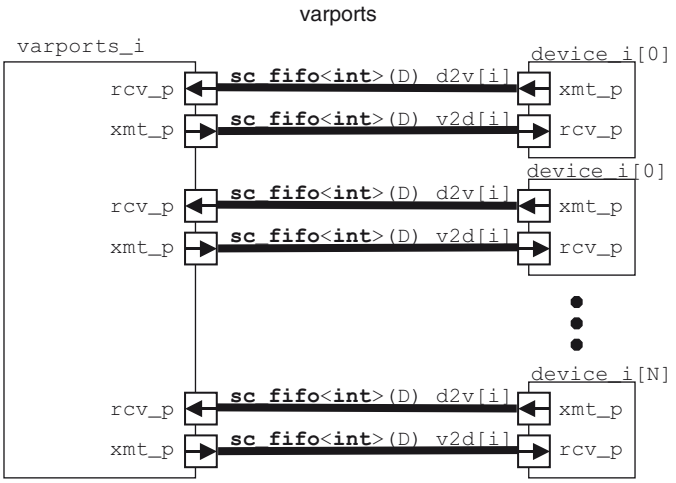


Fig. 14.11 Design with 1-N ports

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):

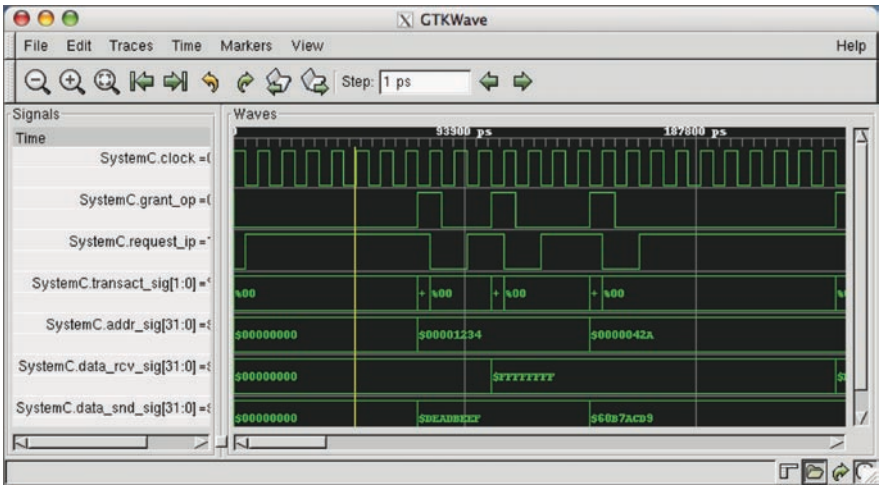


Fig. 14.26 Sample waveform display from gtkwave

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.

Chapter 15
SCV

SystemC Verification Library

15.1 Introduction

In the course of this book, we have covered the SystemC language and its many uses. We have explored the SystemC constructs that let us model hardware easily, including clocks, hardware data types, concurrency constructs, threads, etc. With this knowledge, you are equipped with the ability to construct a sophisticated system model and most of the features required to develop a robust testbench. There is an additional library, the SystemC Verification Library (SCV), which provides much of the features required to implement a robust reusable testbench without having to develop these on your own. This library is described in detail in the downloadable PDF document "SystemC Verification Standard Specification [Version 1.0b]" from <www.systemc.org>. Please note that the document references some aspects of SystemC that have changed from version 2.0.1 upon which it was originally based.

The SCV library includes many add-on features to SystemC including data introspection, extended data types, random data types, transaction monitoring, and transaction recording.

It is beyond the scope of this book to cover specific verification methodologies. We will, however, lightly touch on the topic of developing transaction-based verification and how this allows for higher levels of abstraction test cases, promotes reusable verification IP, and shortens the overall verification cycle.

15.2 Data Introspection

Data introspection is one of the key features of SCV. Introspection allows for variable manipulation without compile-time knowledge of the variable type. SCV implements this feature using partial template specialization. From the user point of view, SCV provides a standard abstract interface, **`scv_extensions_if`**, through which the user can access and manipulate the desired data.

D.C. Black et al., *SystemC: From the Ground Up*, Second Edition,
DOI 10.1007/978-0-387-69958-5_15, © Springer Science+Business Media, LLC 2010

The **scv_extensions_if** class is an abstract interface. This means that the class does not implement any methods directly. Instead, this interface contains five components, which in turn provide the methods to manipulate the data. The component interfaces are:

- **scv_extension_util_if**
- **scv_extension_type_if**
- **scv_extension_rw_if**
- **scv_extension_rand_if**
- **scv_extension_callbacks_if**

These components can be further categorized (Fig. 15.1) into two groups: static and dynamic extensions. Static extensions do not require additional data to be associated with the extended data type. The **type** and **rw** interfaces provide the methods for static extensions by letting the user extract data type information and to read and write to the data object. On the other hand, dynamic extensions require additional data to be associated with the extended data object for the purpose of storing constraints for randomization, and for storing callback function pointers.

We will present each of the **sc_extension** interface components, but will not

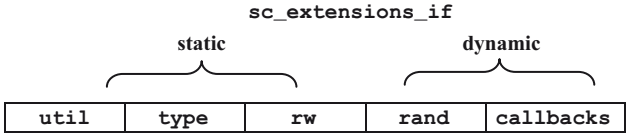


Fig. 15.1 **sc_extensions_if** components

give an exhaustive coverage of all the methods, which are covered in detail in the SCV specification.

15.2.1 Components for scv_extension Interface

As mentioned above, there are five components in the **scv_extension** interface. This section will give a quick overview of the five components, and then we will develop some of the details in later sections.

The **util** component provides the utility methods to obtain information about the data, including the name of the data object, whether dynamic extensions are supported, and whether the data object is a valid extension. SCV supports both C/C++ and SystemC built-in data types. In the following example (Figs. 15.2 & 15.3), the code tests whether the user-defined type has valid extensions.

In the example (Figs. 15.1 & 15.2), the test for valid extensions will fail unless the Packet **struct** is extended. We will cover user-defined data type extensions in the next section.

```
//File: Packet.h
struct Packet { // user-defined type
  enum type { SIMPLE, EXTENDED };
  type        mode;
sc_uint<16> address;
sc_uint<32> data;
};
ostream&operator<<(ostream& os, const Packet& p) {
  os<< "{"
    <<((p.mode==p.SIMPLE)?"SIMPLE":"EXTENDED")
    <<hex
    << ":addr ess=0x"
    <<p.address
    <<",data=0x" << p.data
    <<"}";
return os
}
```

Fig. 15.2 User-defined type

```
#include "Packet.h"
if (scv_get_extensions(Packet).has_valid_extensions())
{
  // Tests to see if
  // Packet has a valid
  // extension
   ….
}
```

Fig. 15.3 Test for extensions on user-defined type

Another **sc_extension** component, **type**, provides methods to extract data type information, including type name and bit width. SCV provides the data types shown in Fig. 15.4:

```
enum data_type {
   BOOLEAN,
   ENUMERATION,
   INTEGER,
   UNSIGNED,
   FLOATING_POINT_NUMBER,
   BIT_VECTOR,
   LOGIC_VECTOR,
   FIXED_POINT_INTEGER,
   UNSIGNED_FIXED_POINT_INTEGER,
   RECORD,
   POINTER,
   ARRAY,
   STRING
}
```

Fig. 15.4 Supported data types for **sc_extensions_if**

The next component of the **sc_extension** interface is **rw,** which provides methods to read and write to the data object. Using this interface might seem cumbersome, when all you want is to obtain or set the data, but it lets your code be written more generically and thus, it is more maintainable. The example in Fig. 15.5 shows how the **type** and **rw** components might be used.

```
void print_data(scv_extensions_if* data_ptr) {
  switch(data_ptr->get_type()) {
    case scv_extensions_if::BOOLEAN:
        cout<<data_ptr->get_type_name()
            <<"_value is: "
            <<data_ptr ->get_bool();
…
  }//end switch
}// end print_data
```

Fig. 15.5 Using **type** and **rw** components

The last two components are **rand** and **callback,** which provide the methods to perform random operations on the data and register for callbacks. Recall that **rand** and **callback** are dynamic components that require additional data associated with the object. These components will add some overhead to your code execution.

As the name implies, the **rand** component provides the interface to generate a random data stream for your extended object. Used with the SCV constraint class and weighted randomization techniques, this interface becomes a powerful tool for building a full-featured testbench. We will cover this interface in more detail later in the chapter.

As for the **callback** interface, the methods provided let the user register for callback if the value of the extended object has either changed or been deleted. You will need to write your own callback function and implement the actions required upon such a change. We will present more on callbacks later in this chapter.

15.2.2 Built-In scv_extensions

As we mentioned before, SCV provides extensions for all the built-in C/C++ and SystemC data types.In addition to the extension methods discussed so far, the template extensions also include operators that let you manipulate the extended objects as you would a built-in type. For example, you can use +=, <<=, *=, etc. on the extended data objects to perform simple operations. The template extensions also provide **read**() and **write**() functions to access the data.

15.2.3 User-Defined Extensions

In some cases, you may want data introspection on your user-defined type. To extend the user data object, you must implement the partial template specialization of the **scv_extensions**. Let us demonstrate in Fig. 15.6 by using the example shown earlier.

```
//File: Packet_ext.h
#include "Packet.h"

// Extend above user-defined type as follows:
SCV_EXTENSIONS(Packet) {
public:
  scv_extensions<sc_uint<16>> address;
  scv_extensions<sc_uint<32>> data;
  SCV_EXTENSIONS_CTOR(Packet) {
    SCV_FIELD(address);
    SCV_FIELD(data);
  }
bool has_valid_extensions() { return true; }
};
```

Fig. 15.6 Extending user-defined types

Fortunately, Cadence Design provided an open source Perl script, **tb_wizard_ext**, that takes your user-defined type as input and generates the appropriate partial template specialization. We have included the script in the examples that go with this book.

15.3 scv_smart_ptr Template

In most cases, it is easier to use a template provided by SCV to handle the **scv_extensions** pointer. The **scv_smart_ptr**class acts just like a C++ pointer to the **scv_extensions**object. Under the hood, the template incorporates both **scv_extensions** and **scv_shared_ptr**objects:

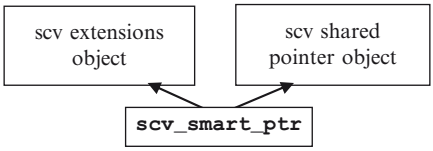


Fig. 15.7 **scv_smart_ptr** template

The **scv_shared_ptr** lets multiple threads share data objects by implementing the necessary memory management. To use this template, instantiate the object with the appropriate data type.

```
scv_smart_ptr<Packet> pPkt;

pPkt->address = 0;
pPkt->data = 0;
```

Fig. 15.8 Using **scv_smart_ptr** on user-defined type

An important feature of the smart pointer is that it implements both the static and dynamic components of the **scv_extensions_if**. This feature means that you can use the smart pointers to implement randomization and register callbacks. We will revisit the smart pointers later in the chapter.

15.4 Randomization

Traditionally, hardware designs have been verified using directed testing methodology. In recent years, there has been more focus on using random testing methodology to achieve wider test coverage. In many situations, verification engineers implement a combination of directed and random tests to achieve their testing goals.

In directed testing, one creates certain scenarios to test each feature of the design. If you want to vary sequences of reads/writes, packet size, address, block sizes, or data values, you must do so manually. When one creates tests using randomization, the stimulus is created using constrained randomization. The expected results can be calculated with a reference model, possibly your system model.

In a unit test environment, it may be desirable to use directed testing and make sure your design block is behaving correctly as you walk through all the scenarios. Some others advocate doing directed unit testing by way of a fully constrained (constrained to one value) random testbench.

In system-level verification, this approach becomes tedious and time consuming. It becomes difficult for the human to consider all possible combinations of test vectors that are required to test all features in normal and boundary conditions. Using random testing methodology, you can easily create many different scenarios for the test stimulus.

SCV provides the infrastructure for you to create basic randomization, constrained randomization, and weighted randomization tests. We will cover the basics of these random concepts and provide some demonstration of how to use the SCV library using the standard API. We will also cover two important **sc_interface_if** templates used for randomization: **scv_smart_ptr**, and **scv_bag**.

15.4.1 Global Configuration

SCV provides a random class, which provides the basis for the random stream generator. Before generating a random number for your data variable, you may

want to set some global configuration parameters to define the desired random distribution or seed value. By default, SCV uses the **jrand48()** algorithm from the standard C library, but you may choose a different algorithm or customize your own by using the **set_default_algorithm** method. The following is a list of possible algorithms to choose from:

```
enum  value_generation_algorithm {
   RAND,
   RAND32,
   RAND48,    // default
   CUSTOM     // requires further configuration setup
}
```

Fig. 15.9 List of randomization algorithms

In addition to the algorithm, you can also specify the method by which the random values are generated. By default, the mode is set to **RANDOM**, which enables the data to be generated across all possible values for that data. For example, randomizing an **sc_uint**<8> with this mode allows 2^8 possible values to be generated in the data stream. On the other hand, you can specify regions of values to keep in or out from the generated data by specifying the **DISTRIBUTION** mode. Specify the mode by calling the **set_mode**method and selecting one of the four modes described next:

```
enum mode_t {
      RANDOM,
      SCAN,
      RANDOM_AVOID_DUPLICATE,
      DISTRIBUTION
}
```

Fig. 15.10 Supported modes using the **set_mode** method

- **RANDOM**—uniform distribution across all legal values
- **SCAN**—maintains the history; starts with the smallest legal value
- **RANDOM_AVOID_DUPLICATE**—maintains the history; avoids duplicated values until all legal values have been generated, at which point it resets the history
- **DISTRIBUTION**—uses specified constraints to affect generated values

Another important configuration is the seed value. Through the **scv_random::set_global_seed** method, the user can change the default seed at the start of simulation. If unspecified, SCV defaults the seed to 1. You may also reset the seed during the simulation by calling **scv_random::set_current_seed**, which will set the seed for the next generated random number. While most

users will set the global seed, you may want to consider generating a unique seed for each process thread. This implementation allows for better repeatability that is independent of your SystemC simulator, and therefore independent of any scheduler differences.

Lastly, you can further control your randomization by enabling or disabling the random feature on a per-data-object basis. If you have a composite type, you can also choose to turn off randomization for just one of the data members. The methods to do this are **enable_randomization**() and **disable_randomization**().

15.4.2 Basic Randomization

When you have decided to use randomization in your testbench, the most basic feature you need is to generate random sequences for your data object. If you have extended your object using **scv_extensions**, you can generate a random sequence by calling the **next** method in the **scv_extensions_if** random component interface as shown in Fig. 15.11.

```
#include "Packet.h"
scv_smart_ptr<Packet> pPkt;
pPkt->next();//creates random values for address &
data
```

Fig. 15.11 Generating random stream on user-defined type

In the example, **next**() returns one of 2^{32} possible integer values for address and data. In a composite type such as Packet above, you can opt to disable selective data members for randomization. For instance, the code in Fig. 15.12 allows only the data to be randomized:

```
scv_smart_ptr<Packet> pPkt;
pPkt->address.disable_randomization();
pPkt->next();
```

Fig. 15.12 Disable randomization

Alternatively, you can call **next**() on just the data member you want to randomize as shown in Fig. 15.13.

```
scv_smart_ptr<Packet> pPkt;
pPkt->address.next();
```

Fig. 15.13 Selective randomization

If you need to constrain the randomization to a range of values, or weigh small packets to occur more often, you will need to use constrained random or weighted random methods as described in the following sections.

15.4.3 Constrained Randomization

Constrained randomization restricts the random generated data stream by letting you specify allowable regions or values. SCV provides a constraint class, **scv_constraint_base**, with convenient macros to specify each constraint rule. The following is a list of the macros:

- **SCV_CONSTRAINT_CTOR**—constructor for the constraint class
- **SCV_CONSTRAINT**—defines a hard constraint
- **SCV_SOFT_CONSTRAINT**—defines a soft constraint
- **SCV_BASE_CONSTRAINT**—defines a base constraint

The constraint class uses **scv_smart_ptr** to implement the randomized data. To define a constraint, first create a constraint class to define rule(s) for constraining your data type. This class must inherit from the base constraint class, **scv_constraint_base**, and implement the constructor.

```
class Pkt_constraint
      :virtual public scv_constraint_base {
public:
scv_smart_ptr<Packet> pPkt;
SCV_CONSTRAINT_CTOR(Pkt_constraint) {
    // define constraints
SCV_CONSTRAINT(
      (pPkt->address() != 0x00000000) &&
      (pPkt->address()  < 0x00000800)
    );
SCV_CONSTRAINT(pPkt->data() >= 0x00001000);
}
};
```

Fig. 15.14 Constraint class

An important point to note in the constraint macros concerns the use of the parenthesis operator to construct Lamda[1] expressions. Notice that every reference to a data member is followed by an empty parenthesis pair (i.e., "`()`"). This syntax is needed because the class creates an internal representation of the constraint equations, which are used with a solver. Detailed explanation of how this is accomplished goes beyond the scope of this book, but suffice it to say that these parenthesis pairs are required. It should also be noted this it is a common error to forget to supply these pairs when coding.

[1] The inquisitive reader may choose to research Lamda calculus to understand the reasons for this. Basically, the SCV needs to store equations to allow it to do constraint solving.

To use the constraint, instantiate as you would any class as illustrated in Fig. 15.15:

```
Pkt_constraint cPkt;

cPkt.next();
cout<< "data = " << cPkt.pPkt->data <<endl;
```

Fig. 15.15 Using constraint class

By default, **SCV_CONSTRAINT** specifies hard constraints. Using hard constraints means that if SCV cannot find a legal value for this constraint, it will generate an error and ignore the constraint. You can specify a soft constraint, **SCV_SOFT_CONSTRAINT**, and SCV will generate only a warning if it cannot find a legal value.

Since constraints are captured in a constraint class, you can build hierarchical constraints using inheritance. Start with a basic constraint class defining fundamental constraints and add in more complicated constraints using hierarchy to define more specific or complicated constraints.

15.4.4 Weighted Randomization

While constraints are used to define legal values of the random data stream, weighted randomizations are used to define frequency of certain generated values. SCV provides some methods to define simple distribution through **scv_extensions**:

- **keep_only**—define a value or range of values to include in distribution
- **keep_out**—define a value or range of values to exclude in distribution

When you call these methods, SCV automatically sets the randomization mode to **DISTRIBUTION** and disregards any previous **set_mode** distributions for that particular data object. Likewise, any previous constraints defined for the data object are also disregarded. Note that you can define multiple **keep_only** and **keep_out** ranges for the data object, and the result is a cumulative effect of the defined ranges.

```
scv_smart_ptr<Packet> pPkt;
pPkt->address.keep_only(1,9999);
pPkt->data.keep_out(0);
pPkt->data.keep_out(10000U,(1U<<30));
```

Fig. 15.16 Using **keep_only** and **keep_out** to define random distribution

In some cases, you may want to define more complicated distribution rules for your data. In this case, SCV provides a templated class, **scv_bag**, to define relative weight of particular values that is illustrated in Fig. 15.17.

```
// define a bag
scv_bag<int> intBag;

intBag.add(0, 25); //add 25 objects of value 0 to bag

intBag.add(1, 25); //add 25 objects of value 1 to bag

intBag.add(2, 50); //add 50 objects of value 2 to bag


scv_smart_ptr<int> smart_int;
smart_int->set_mode(intBag);   //set smart_int
                               //distribution
```

Fig. 15.17 Using **scv_bag** to define random distribution

In dealing with multiple smart pointer variables within a constraint, you can specify distributions for each data member by using the **set_mode** method as shown in Fig. 15.18.

```
class Pkt_constraint
: virtual public scv_constraint_base
{
public:
 scv_smart_ptr<sc_uint<16>> address;
 scv_smart_ptr<sc_uint<32>> data;
 SCV_CONSTRAINT_CTOR(Pkt_constraint) {
    // define constraints
SCV_CONSTRAINT(
      (address() != 0x00000000) &&
      (address() < 0x00001000));
SCV_CONSTRAINT(data() >= 0x1000);
  }
};

void test() {

  typedef pair <sc_uint<32> sc_u int<32>> data_range;
  scv_bag<data_range> data_dist;
  //set range distribution for data
  //data range (0x1000, 0xffff) occurs 30%
  //data range (0x10000, 0x20000) occurs 70%)
  data_dist.add(data_range(0x1000, 0xffff),30);
  data_dist.add(data_range(0x10000, 0x20000), 70);

  Pkt_constraint cPkt;
  cPkt.next();          //generate addr and data using
                        //constraints
  cPkt.data->set_mode(data_dist);
  cPkt.next();          //generate addr using
                        //constraints and generate
                        //data using 'data_dist'
                        //distribution
}
```

Fig. 15.18 Changing randomization modes

15.5 Callbacks

Callbacks present a powerful mechanism for monitoring variables (Fig 5.19). For that purpose, the SCV provides two main methods, **register_cb** and **remove_cb**. Once a function has been registered as a callback, it will be called anytime the referenced object changes. It should be obvious that if abused, this usage can result in a lot of overhead for the simulation. Therefore, you should use caution when selecting which variables to use callbacks on.

Let's look at a simple example in Fig. 15.20 of how to use a callback. For this example, we will use the previously used Packet type.

This example illustrates the concept that more than one callback may be registered on an object.

```
enum callback_reason {
  VALUE_CHANGE,
  DELETE
}
// Register a simple callback function
callback_h register_cb (
void (*f)(
scv_extensions_if& OBJ,
callback_reason    REASON
)
);
// Template method registers a callback function
// with an extra argument in a type-safe manner
template<typename T>
callback_h register_cb(
void (*f)(
scv_extensions_if& OBJECT,
callback_reason REASON,
    T ARG
  ),
  T arg
);
// Remove existing callback
virtual void remove_cb( callback_h HANDLE);
```

Fig. 15.19 Interface for callbacks

```
#include "Packet.h"

static unsigned changes;
// A function to monitor changes on a Packet
void Packet_cbA(
scv_extensions_if& obj,
scv_extensions_if::callback_reason reason
) {
if (reason == scv_extensions_if::VALUE_CHANGE)  {
cout<< "Packet " << obj.get_name()
<< " value change to " << obj.get_unsigned()
<<endl;
  } else {
cout<< "Packet " << obj.get_name()
<< " deleted." <<endl;
  }
}
void Packet_cbB(
scv_extensions_if& obj,
scv_extensions_if::callback_reason reason
) {
  if (reason == scv_extensions_if::VALUE_CHANGE)  {
   changes++;
  } else {
cout<< changes << " distinct values"
<<endl;
  }
}
scv_smart_ptr<Packet> pPkt1("pPkt1"),pPkt2("pPkt2");
scv_extensions_if::callback_h
  h1A(pPkt1->register_cb(Packet_cbA)),
h1B(pPkt1->register_cb(Packet_cbB)),
  h2A(pPkt2->register_cb(Packet_cbA));

for (int i=0; i!=10; ++i) {
  pPkt1->next(); pPkt2->next();
}
pPkt1->remove_cb(h1A);
```

Fig. 15.20 Example callback

15.6 Sparse Arrays

Almost a seemingly unrelated topic, a model for a sparse array is included in the SCV, but it is not completely unrelated. Memories and large memories are a part of almost every electronic system today. When simulating memories in a system, it is not possible to simulate a 4 GB memory while running on a 2 GB simulation computer without some compromises.

Fortunately, most of the time, simulations only use a tiny fraction of a large memory. For that reason, it makes sense to model memories as sparsely populated. Although, one could use a standard STL **map<>** container for this purpose; there are several useful extensions that make the **scv_sparse_array** a better choice. For one, reading an

unwritten location returns a default value. Another aid is the definition of memory bounds (i.e., upper and lower limits for the address). Fig. 15.21 is the constructor syntax.

```
scv_sparse_array<T1,T2> NAME(
const char * name,
const T2& default_value,
const T1& indexLB = 0,
const T1& indexUB = INT_MAX
);
```

Fig. 15.21 Creating a sparse array

The first typename, T1, designates the type of the index for the sparse array. The second typename, T2, designates the data value types. Accessing the memory is straightforward. Here is an example:

```
scv_sparse_array<unsigned,short> mem("mem",0,0,1e6);
scv_smart_ptr<unsigned> a_ptr;
scv_smart_ptr<short> d_ptr;
a_ptr->keep_only(0,1e6);
d_ptr->keep_out(0);
for (unsigned count=0; count!=30; ++count) {
  a_ptr->next(); d_ptr->next();
  mem[*a_ptr] = *d_ptr;
}
for (unsigned count=0; count!=30; ++count) {
  a_ptr->next();
  *d_ptr = mem[*a_ptr];
cout<< *d_ptr <<endl;
}
```

Fig. 15.22 Sparse array example

One thought that comes to mind when modeling a system that may use consistently sized chunks of memory (e.g., a 256 or 1024 block of data), suggests deriving a custom sparse array that contains blocks of data (e.g., a **vector**<*T*>) sized to contain the data. Using a custom sparse array may prove more efficient when treating small groups of locations repeatedly.

15.7 Transaction Sequences

In many systems today, test teams are faced with the obstacle of verifying complex designs. In some cases, the testbench itself requires a great deal of work, including code partitioning, design for reusability, and flexibility. In many cases, you can use the concept of transaction-based testing to help achieve these goals. Throughout the book, you have been exposed to various SystemC constructs that let you design a transaction-based testbench.

A transaction is a set of activities defined by a start and finish and lasting a certain duration. For example, a read or write to memory is considered a transaction, including arbitration and transfer of data.

15.8 Transaction Recording

The SCV 1.0 library provides a set of APIs that lets the user record transactions. According to the OSCI documentation, this set of APIs is not an official part of the standard yet; however, many companies are using the interface. So, it is unlikely to go away. Furthermore, several commercial EDA tools are available to help visualize the results.

The APIs are categorized into three classes:

- **scv_tr_db**—transaction database containing a collection of transaction streams
- **scv_tr_stream**—transaction stream containing a collection of transactions
- **scv_tr_generator**—transaction generator for a specific transaction type

In a given simulation, you can instantiate one or more collections of transaction streams. This instantiation is usually done in **sc_main**. Then, within your modules, instantiate the **scv_tr_stream** and **scv_tr_generator** objects. A simple data recording example follows is shown in Fig. 15.23.

```
// note, scv_tr_db instantiated in sc_main.

class simple_transactor : public simple_ports {
scv_tr_stream read_stream;
scv_tr_generator<sc_uint<8>, sc_uint<8>> read_tr;
SC_CTOR(simple_transactor)
    // assign a name and type to your read_stream
    :read_stream("read_stream", "transactor")
    // assign name to this transaction generator
    // associated with the stream, and assign names
    // to the associated attributes.
    ,read_tr("read", read_stream, "addr", "data")
    {…}

sc_uint<8> read(sc_uint<8>&* addr) {
    // signals the start of a transaction
    scv_tr_handle xactionHandle =
            read_tr.begin_transaction(addr);
sc_uint<32> data;
    //process read
    …
    // signals the end of the transaction
    read_tr.end_transaction(xactionHandle
                                  ,data);
return data;
  }
};
```

Fig. 15.23 Transaction recording

Every time the read function is called, it records the transaction to the stream with a unique handle, and the transaction is terminated when the read transaction is done.

We encourage the reader to explore more advanced features of transaction recording in the SCV documentation. Be sure to see the exercises at the end of this chapter.

15.9 SCV Tips

Some simple observations are in order:

1. SCV extensions carry overhead both in execution speed and size. Be judicious in their use.
2. When designing constraints, remember to use the **`operator()`** to reference values.
3. Consider the option of overriding the **`next()`** method as a means to controlling the randomization.
4. Understand the distribution of your data when using a sparse array. For small arrays, a normal vector may be sufficient.
5. Don't abuse callbacks. Be strategic.
6. There is no substitute for a well thought out design.

15.10 Exercises

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

Exercise 15.1: Randomize 1000 **`struct`**s containing two **`int`**s and display a histogram of their distribution. Divide the space into approximately 100 buckets.

Exercise 15.2: Using the same base code as in exercise 15.1, add restrictions to keep the random numbers in the ranges of 0–10 and 16–100.

Exercise 15.3: Create a distribution of $-\pi$ to $+\pi$ using **`float`**s. Using **`scv_bag`**, create a sinusoidal distribution.

Exercise 15.4: Create a custom structure to represent a configuration register space for a cell phone. Include fields for transmit/receive frequencies, display types (LCD, plasma, none), supported email and the phone number. Be sure the phone numbers are the correct number digits for your region. Restrict randomization of phone numbers to legal values for your region (e.g., USA uses athree-digit area code and a seven-digit number and neither group may begin with 0 or 1). Randomize 16 times and display. Use constraints to ensure numbers are not duplicates.

Exercise 15.5: Use callbacks to implement functional coverage with an STL map.

Exercise 15.6: Try using a sparse memory to model automobiles on a grid. Try using an STL `pair` as the index to model the x,y coordinate system.

Exercise 15.7: Explore transaction recording by reviewing, compiling and running the recording_ex code in the downloads.

Chapter 16
OSCI TLM

In Chapter 2, we introduced the concept of transaction-level modeling-based (TLM) methodology. In the chapter, we talked about some of the reasons why one would use a TLM methodology, including design exploration, early hardware/software integration, and early verification development. Clearly, any one of the above is a good reason to develop a TLM model.

To implement a TLM methodology, there needs to be a precise definition of a TLM standard. This standard should be flexible enough to work at different levels of abstraction and be protocol-independent. Currently, there are various working groups defining a TLM specification. In this chapter, we will focus only on the Open SystemC Initiative (OSCI) TLM 1.0 standard, and we will provide a glimpse of objectives planned for the next release at the time of this writing.

16.1 Introduction

The OSCI TLM 1.0 specification was created in response to a need to standardize a TLM definition. The core component of the specification offers a set of TLM interfaces that define the APIs required for unidirectional blocking interface, unidirectional non-blocking interface, and bidirectional blocking interface. The specification also defines a set of channels that implement the above interfaces. While it is possible to define your own channels, the supplied example channels should be sufficient for the user to connect most system components.

In addition to a set of APIs, the OSCI TLM standard offers other benefits including abstraction, speed, and reuse. Using the TLM components, you can design a very high-level abstract model, refine portions of the design, or mix a variety of models at different abstraction levels. Of course, there are speed trade-offs to consider as you develop your model. Earlier in this book, we discussed the different abstraction levels including un-timed, approximate-timed, and cycle-timed. At the most abstract level, the un-timed models give the best performance in terms of speed. In many cases, this model can be refined further to an approximate-timed model to achieve better accuracy, though with some simulation performance

D.C. Black et al., *SystemC: From the Ground Up*, Second Edition,
DOI 10.1007/978-0-387-69958-5_16, © Springer Science+Business Media, LLC 2010

degradation. Lastly, we need to mention the benefit of reuse in developing TLM models. As with any standardized API, one of the benefits is being able to reuse the components in different projects. In cases where there is a mixture of model abstractions, you can develop specific transactors or adaptors and still model the components together.

In the remainder of the chapter, we talk about the architecture of the TLM 1.0 standard, including the interface and channels, as well as offer an example of a TLM-based design.

16.2 Architecture

The TLM 1.0 specification provides a definition for the TLM architecture, including the transport layer, protocol layer, and user layer as shown below:

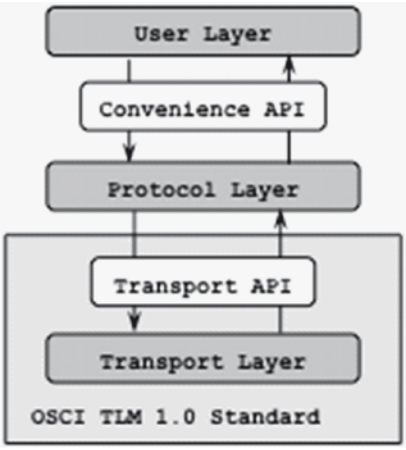


Fig. 16.1 TLM 1.0 Architecture

The transport layer is the heart of the TLM 1.0 specification. This layer focuses on the fundamental APIs that define the transfer of generic data, using either blocking or non-blocking low-level calls. The SystemC components here include the TLM interfaces that define the unidirectional **`put`**`()` and **`get`**`()` functions, as well as examples of the channels that implement them. As we mentioned previously, the user has the option to use the TLM channels supplied with the standard, or to create their own custom channels that implement the underlying TLM interfaces. Some of the components in the transport layer may include some generic system blocks such as routers and arbiters to assist in the transport of data.

The protocol layer consists of the classes that interface directly with the transport layer by calling the appropriate APIs. The OSCI standard provides simple initiator and target example classes that you may use, but in most cases, you will need to develop protocol-dependent code to interface with specific model components.

Lastly, the user layer contains convenience methods that let the user access data through simple function calls, such as reads, writes, block reads, and block writes. Since user class definitions inherit from the protocol class definitions, the user classes normally don't need to change when the underlying protocols change. The following diagram shows the interactions among the layers.

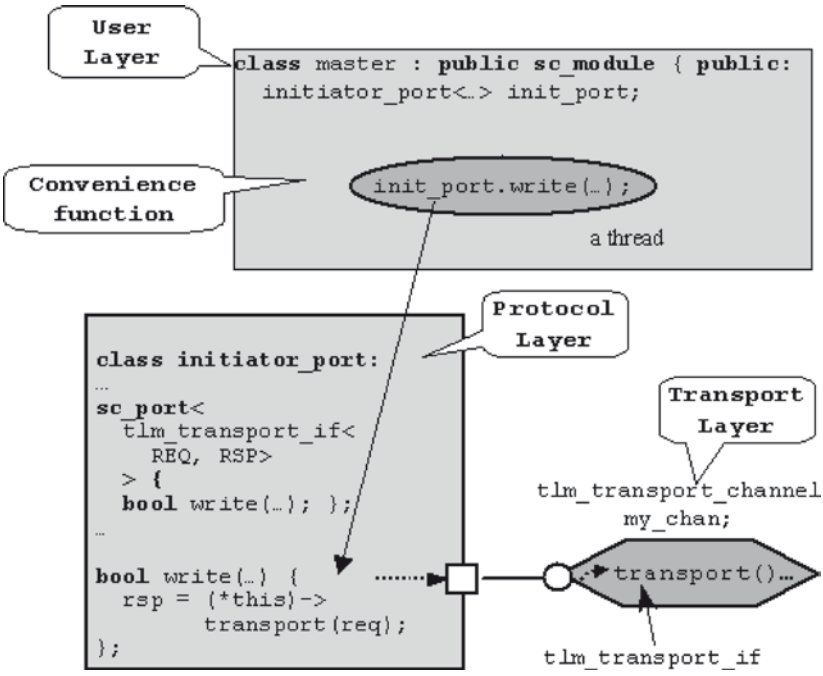


Fig. 16.2 OSCI TLM approach

In the preceding diagram, the user layer defined by master, instantiates a specialized port, the `init_port`, and calls the **`write`**`()` method in this class from an **`SC_THREAD`** process. The `initiator_port` class implements the protocol layer and calls the appropriate **`transport`**`()` function in the TLM channel, `my_chan`. This example illustrates one of many possibilities. In an alternative example, the **`transport`**`()` function could have been implemented in the target via an **`sc_export`**`<if>` connection, which we will explore later.

16.3 TLM Interfaces

The OSCI TLM 1.0 Standard is focused on specifying interfaces. There are three interface categories defined in the OSCI TLM standard:

- Unidirectional blocking interface
- Unidirectional non-blocking interface
- Bidirectional blocking interface

For abstract models, the **SC_THREAD** process style of coding is easier and more convenient to use when implementing. Thread processes are allowed to use the blocking calls, which simplify the use of the TLM channels. Method processes may use TLM channels, but are required to use the non-blocking functions. Using the non-blocking interface will tend to be more complex than using the blocking style. Though more tedious, some components may need to use this non-blocking style. The blocking and non-blocking interfaces are supplied to allow the user flexibility in using either **SC_THREAD** or **SC_METHOD** to their design modules. The following figure illustrates the class hierarchy that implements all of the TLM 1.0 interfaces. The interfaces are broken into **get** and **put** categories. As you can see, the standard divides the interfaces into quite a few fine-grained classes to provide maximum flexibility.

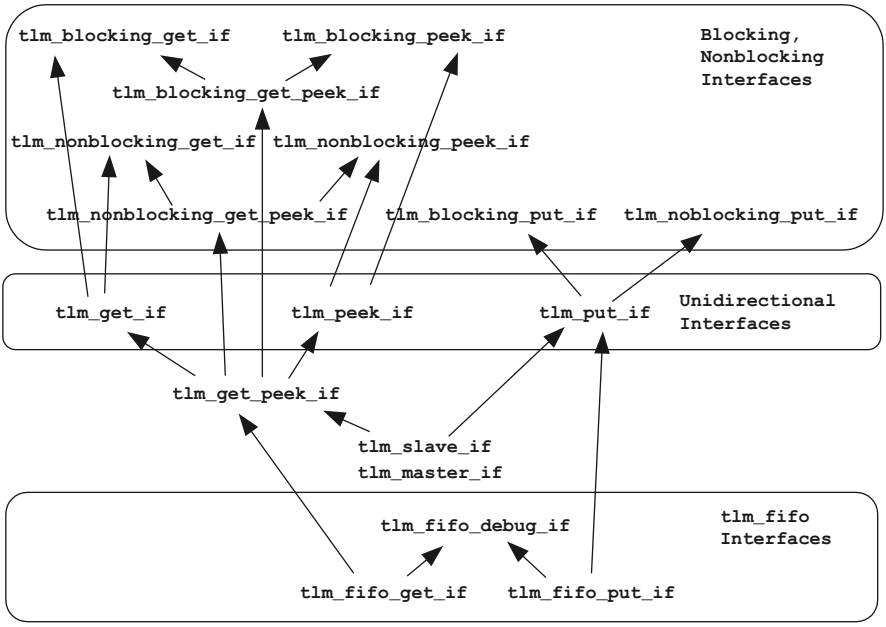


Fig. 16.3 TLM 1.0 class hierarchy

Let us look at each of the interface categories in more detail. Because all the methods are in the base classes, the methods are really not as daunting as the

preceding figures might imply. The derived interface classes represent convenience groupings.

16.3.1 Unidirectional Blocking Interfaces

The following code shows the APIs defined in the OSCI TLM standard for a unidirectional blocking interface, which loosely has behavior matching a FIFO.

```
tlm_uni_channel<T> instance("instance");
instance.get(result_ptr);
result = get();
instance.put(value);
```

Fig. 16.4 Unidirectional blocking **get** interface syntax

The functions **get**() and **put**() correspond to the **sc_fifo**<*T*> **read**() and **write**(). It should be no surprize that the **get**() and **put**() functions can contain waits and only return when data is transferred. Thus, these functions can only be called from an **SC_THREAD** process because **SC_METHOD** processes are not allowed to wait.

The **get** interface contains an additional API that uses the **tlm_tag**<*T*>, which allows a target to implement multiple versions of an interface.

```
tlm_uni_channel<T> instance("instance");
instance.peek(result_ptr);
instance.peek(result_ptr, offset);
result = peek();
result = peek(offset);
instance.poke(value);
instance.poke(value, offset);
```

Fig. 16.5 Unidirectional blocking interface syntax

The blocking interfaces also include classes used to check for data availability without actually consuming the data. This is useful when designing a distributed decode and each target needs to determine if the value presented is for the interogating target before consuming the data.

16.3.2 Unidirectional Non-Blocking Interfaces

Methods in the non-blocking category all have the prefix **nb_**. The following code shows the syntax for the basic unidirectional non-blocking interfaces:

```
tlm_uni_channel<T> instance("instance");
if (not instance.nb_get(variable) {
  next_trigger(instance.ok_to_get());
}
if (instance.nb_can_get()) {
  cout << "instance available for get" << endl;
}
if (not instance.nb_put(value) {
  next_trigger(instance.ok_to_put());
}
if (instance.nb_can_put()) {
  cout << "instance available for put" << endl;
}
```

Fig. 16.6 Unidirectional non-blocking syntax

As with the blocking counterparts discussed above, these APIs include **nb_get**() and **nb_put**() functions.

These methods let users check whether data transfer will succeed before actually calling the transfer functions. This set includes functions with return values or events that indicate it is ok to proceed with the data transfer. Depending on the FIFOs and number of requestors, the user is not guaranteed the success of the data transfer even after waiting on the **ok_to_put**() and **ok_to_get**() events. You can verify the success of your transfer by the fail/success return value from the transfer functions.

Note that the non-blocking functions return a **bool** type. This return data type indicates whether the data transfer actually occurred. These non-blocking interfaces can be called from within both **SC_METHOD** and **SC_THREAD** processes.

There are also non-blocking debug interface methods that provide additional features used primarily for verification.

```
tlm_uni_channel<T> instance("instance");
if (!instance.nb_peek(variable, offset) {
  next_trigger(instance.ok_to_peek())&
}
if (instance.nb_can_peek(offset)) {
  cout << "instance available for get" << endl;
}
if (!instance.nb_poke(value) {
  next_trigger(instance.ok_to_poke())&
}
if (instance.nb_can_poke()) {
  cout << "instance available for poke" << endl;
}
```

Fig. 16.7 TLM non-blocking debug interfaces

Debug interfaces never consume time, because they are intended for debug. The idea of the `offset` in the peek and poke routines is that the interface has some depth (e.g., a FIFO). An offset of zero would indicate the topmost piece of

information (i.e., what is retrieved with **get**( )). Other values would probe deeper into the interface. For a FIFO, the `offset` is somewhat obvious. For other constructs, it is not obvious and would be documented by the channel creators.

16.3.3 Bidirectional Blocking Interface

The bidirectional blocking interfaces are provided for cases where there is a one-to-one relationship between the request and response. This interface is provided as a convenience, since the underlying implementation can call the unidirectional blocking **put**( ) and **get**( ) functions. In fact, the OSCI TLM standard implements the transport channel example in this manner.

```
tlm_transport<REQ,RSP> instance;
result = instance.transport(request);
```

Fig. 16.8 Bidirectional blocking syntax

The request and response must be different classes to avoid ambiguities within the class. This structure can be accomplished by deriving one function from the other. This structure avoids a C++ ambiguity to differentiate the signatures of two **get**( ) functions and **put**( ) functions within the class implementation.

Keep in mind that the unidirectional and bidirectional interfaces we just covered are OSCI TLM's core interfaces. The specification provides additional interfaces that bundle some of the core interface APIs in addition to providing a debug interface.

16.4 TLM Channels

As we mentioned before, channels implement the APIs of the interfaces they inherit. The OSCI TLM standard provides three different example channels:

- **tlm_fifo**<*T*> - implements unidirectional interfaces
- **tlm_req_rsp_channel**<*Req,Rsp*> - implements two unidirectional interfaces
- **tlm_transport_channel**<*Req,Rsp*> - implements bidirectional interface

Note that these channels are not part of the TLM specification. These channels are provided as supplementary communication components. In some cases, you may need to develop protocol-specific channels as required by your design.

The **tlm_fifo**<*T*> channel is modeled after the SystemC FIFO class, and this channel implements the unidirectional interfaces, including both blocking

and non-blocking interfaces. The FIFO depth can be defined as any size from zero to infinite depth.

The **tlm_req_rsp_channel**<*Req,Rsp*> implements two unidirectional interfaces, using TLM FIFOS. One FIFO is used for request and the other is used for response. From the connectivity point of view, the **tlm_req_rsp_channel**<*Req,Rsp*> exports the **put** request FIFO interface and **get** response FIFO interface to the master. Likewise, the slave is connected to the **get** request FIFO interface and the **put** response interface.

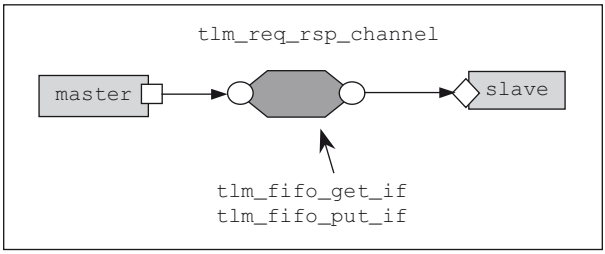


Fig. 16.9 **tlm_req_resp_channel**

The last example channel provided by TLM 1.0 is the **tlm_transport_channel**<*Req,Rsp*>. Internally, this channel is implemented by a **tlm_req_rsp_channel**<*Req,Rsp*> with a FIFO size of 1. The master is connected via a **tlm_transport_if**<*Req,Rsp*> **sc_export**<*T*>, while the slave is connected via the unidirectional interfaces.

16.5 Auxiliary Components

Thus far in this chapter, we have focused on the core TLM 1.0 APIs that define the communication between design components. We also covered some basic OSCI TLM channels that are provided as optional components to the TLM standard. In this section, we focus on the components you will need to model your designs.

In many systems, designs typically contain multiple components including masters, slaves, routers, and arbiters. Regardless of function, each component is connected via a port/export pair, or through an intermediate channel.

The OSCI TLM 1.0 kit provides documentation, working examples, and example components masters, slaves, an arbiter, and a router. These components and example configurations provide a base for developing your own TLM components. Developers new to SystemC should study these examples closely; some features are subtle such as model connectivity and the use of specialized ports for detail abstraction. Developers would be well served, if they created their own block diagrams to trace model connectivity, and review the previous chapter on specialized ports.

16.5.1 TLM Master

The figure below shows a master instantiating a port using the **tlm_transport_if**<*Req*,*Rsp*> interface.

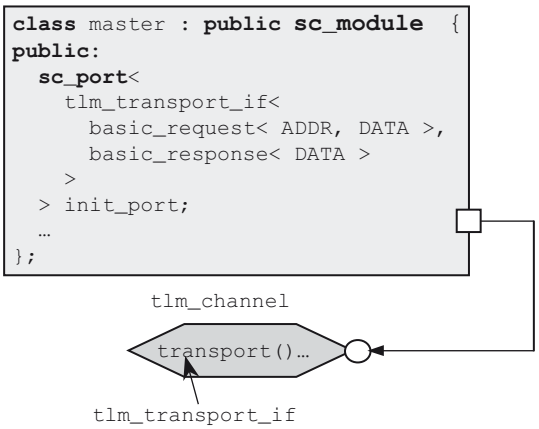


Fig. 16.10 Master connected to TLM channel

Notice that the master can be easily modified to use a different interface by changing its port specification. Of course, the channel will need to match the corresponding interface. In this example, we have defined a single port master, but you may find it necessary to instantiate multiple ports, each of which may be connected via different interfaces.

Next, note that the request and response types are user-defined and can be customized to your design requirements. Lastly, the example shows a master connected to a channel. In some cases, you may want to directly connect the master to a memory target, bypassing channels entirely. This structure is possible if the target also defines the same port TLM interface and implements the associated APIs, as described in the next section.

16.5.2 TLM Slave

A TLM slave contains similar components. The following figure shows a slave target connected to a master via an **sc_export**<*T*> and **sc_port**<*I*> connection, respectively. In the example, both ports are declared with the bidirectional interface, **tlm_transport_if**<*Req*,*Rsp*>. Subsequently, the slave must implement the interface API, **transport**() function.

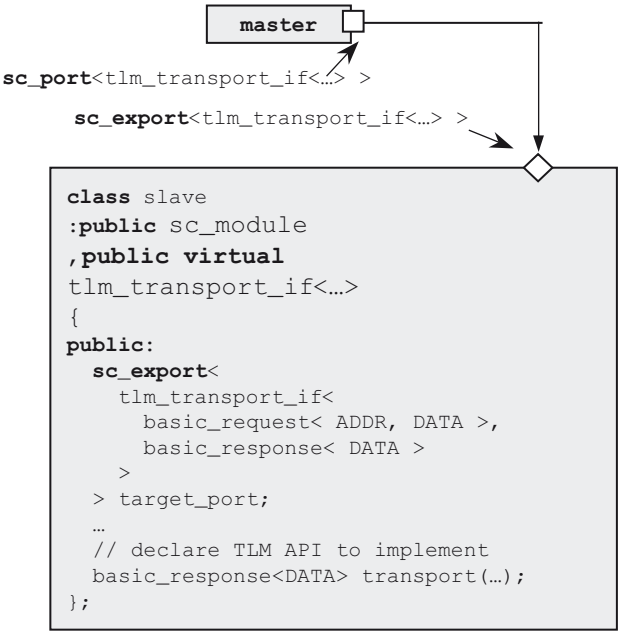


Fig. 16.11 TLM slave and TLM master connection

The designer is free to choose any of the TLM interface, transfer data type, and number of ports for the module. Just make sure the interface is the same between connections and the data type is consistent between modules.

16.5.3 Router and Arbiter

Routers and arbiters are often used in designs to model real systems. As components in TLM modeling, they transfer data in the same manner as master and slave components we just discussed. The router uses the **`sc_export`**`<I>` construct for connection, and the router master interface uses the **`sc_port`**`<I>` construct for connection as shown below. Note that the TLM 1.0 kit examples have some interesting names.

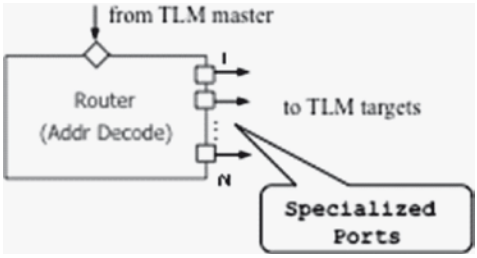


Fig. 16.12 TLM router

The OSCI TLM provides an example of a basic router utility that uses a specialized port to communicate to TLM targets.

The OSCI TLM arbiter uses polling to get requests from its master interface ports as shown below. When the arbiter finds a request on the master ports, it forwards the request onto the TLM target. The OSCI arbiter uses a starving priority algorithm, which you may want to consider modifying based on your design.

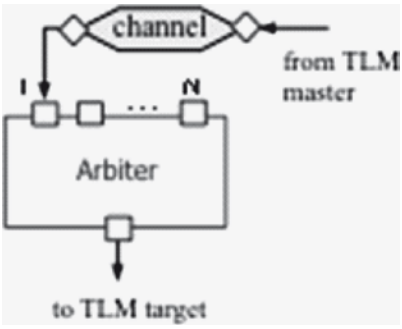


Fig. 16.13 TLM arbiter

As with the TLM channels, the master, slave, router, and arbiter components are not part of the OSCI TLM 1.0 specification. However, these utilities can be used to start off a high-level abstract design with very little effort.

16.6 A TLM Example

In this section, we will discuss using the TLM methodology to design a realistic system. When creating a system-level design, it is important to ask yourself what the purpose of your model is. As we mentioned earlier in the chapter, a TLM model can be used to aid architectural exploration, verify system performance, assist in early software development, or aid in functional verification.

In some cases, your model may be used for multiple purposes, which means your system-level model architecture should also facilitate modeling at different levels of abstraction. The example in this section will focus on a high-level abstract system model that can be used for architectural exploration and software development. In addition, we will present some simulation performance statistics that show significant incentives for using TLM modeling.

The figure below shows the system block diagram for a Voice-Over-IP (VoIP) design. This system uses many typical SoC components, including a processor, flash memory, RAM, and IO devices. All system components are connected using TLM.

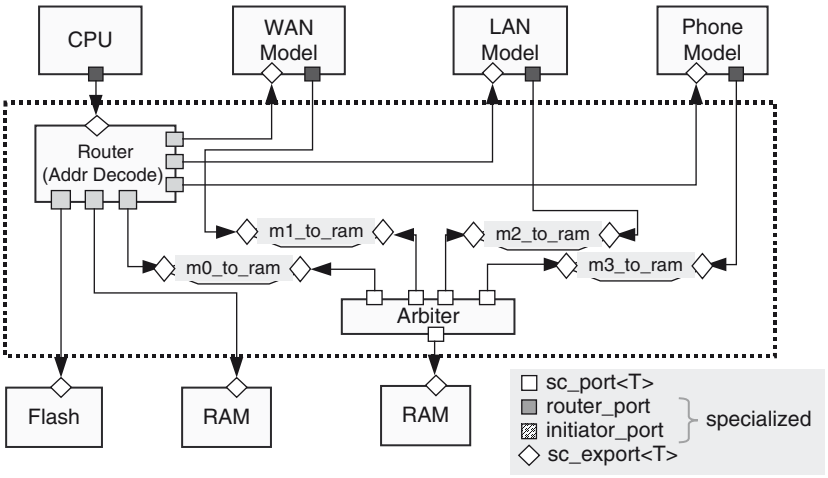


Fig. 16.14 VoIP system

This system model simulates a home network with multiple PCs and a phone. The WAN model generates incoming Ethernet traffic, the LAN model generates output Ethernet traffic, and the phone model generates in/out phone traffic. Specifically, we would like to simulate a 20 second phone call during simultaneous network traffic.

Each IO device uses the same initiator class, configured to generate different types of traffic. In addition, each IO device also instantiates a target port to support programmer's view, which allows bit accurate control and status register access. The target models are simple memories with storage capabilities and read/write timing delays. The router is an address decoder, and the arbiter uses a simple round-robin algorithm. Our CPU model simulates random traffic to memory and also loads device drivers for each IO device. The device drivers manage transmit and receive queues on the IO devices, transfer received packets to destination NICs, and manage buffer allocation.

The following figure shows how the target finally processes an initiator request using the **`tlm_transport_if`**<*`Req, Rsp`*> interface. First, a SystemC process (e.g., testbench, stimulus generator, etc.) calls the initiator to issue a memory write request. The initiator calls the **`transport`**() function via the TLM interface port, **`init_port`**. At this point, the request may travel through other components, including channels, routers, and arbiters, before finally arriving at the memory target.

The target "receives" the write request in the form of a function call to its **`transport`**() function and processes the request accordingly. Note that this interface, **`tlm_transport_if`**<*`Req, Rsp`*>, is a blocking interface. This means that the originating SystemC thread will block until it receives a response from the slave.

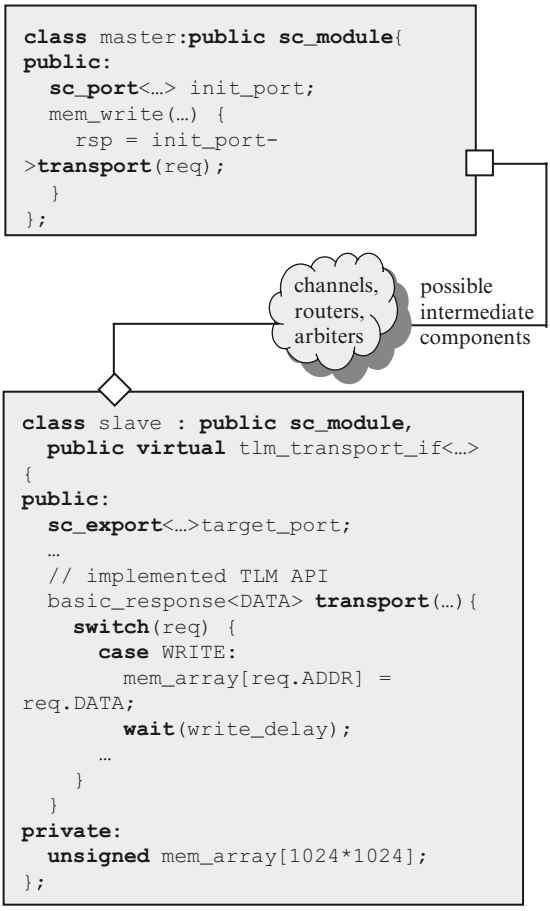


Fig. 16.15 Transaction sequence via **`tlm_transport_if`** interface

By using this system model, we are able to validate and analyze several characteristics about our design, including:

- Bus sizing
- Single cycle vs. block transfer vs. bus widths
- Block diagram—identify all required blocks and IO
- HW/SW partitioning
- Memory partitioning and performance—separating instruction and data memory.
- Memory access contention using the arbiter
- Memory transmit/receive processing queue size
- HW/SW interface

One advantage to designing a model at this level of abstraction is the simulation performance. We simulated our model for 20 seconds, which only required 8.43

seconds of CPU time; 42% of simulated time. Of course, your simulation performance is heavily dependent on your designs and your host computer. We used timed approximations for all read/write transactions, used an event-driven vs. polling where possible, and used burst transactions for data transactions.

To summarize, TLM modeling is great for system design feedback. Doing so requires a reasonable amount of effort, and the simulation performance lets you change your design and re-simulate very quickly. Some important thoughts to remember are to focus on what you are trying to achieve with your model, and leverage the existing IP where possible.

16.7 Summary

In this chapter, we presented the OSCI TLM 1.0 standard, which defines a set of TLM API interfaces. We also presented some of the main components for TLM designs, including channels, initiator, target, router, and arbiter. With this knowledge, you are ready to design a TLM system-level model.

Before you start your design you may want to review the TLM methodology chapter earlier in the book. While we have focused on the "how" of TLM-based design in this chapter, the TLM methodology discussion earlier in the book discusses the "why" of TLM design and how it fits with project flow.

It will become apparent, that while one design group may use this methodology for architectural exploration, another group may extend the design for another purpose, such as early software development. It is important to understand the usage model and create the system-level design to be flexible and to allow for different levels of abstraction.

Finally, keep in mind that the material in this chapter applies to the OSCI TLM 1.0 standard. The OSCI TLM Working Group is currently developing the TLM 2.0 standard. The updated standard is targeting improvements in the TLM APIs, adding generic data structures, and adding timing annotation capabilities. The API improvements and generic data structures will help greatly in model interoperability, and the timing annotation capabilities will allow faster simulations. We strongly encourage the reader to find out more on both the TLM 1.0 and planned features for TLM 2.0 on the OSCI SystemC site, www.systemc.org.

16.8 Exercises

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

Exercise 16.1: Using the TLM 1.0 example channels, create a simple design that uses both the **`sc_fifo`**<*T*> and the **`tlm_fifo`**<*T*>. Discuss how these differ from one another. In what situations might you prefer the TLM FIFO over the core FIFO?

Exercise 16.2: Write your own TLM FIFO derived from the **`tlm_fifo`**<*T*> that adds the following features:

- Add a monitor to efficiently count the number of times the FIFO becomes full or empty, and report the count at the end of the simulation.
- Add a monitor to determine the average FIFO depth.
- Add a functional coverage monitor to determine how many different values were placed in the FIFO and report the coverage at the end of simulation.
- Create a testbench to exercise your FIFO.

Exercise 16.3: Discuss the advantages and disadvantages of passing arguments by-value vs. by-reference. Explain data ownership and data lifetime. How might the boost.org shared pointer class help this situation?

Chapter 17
Odds & Ends

Performance, Gotchas, and Tidbits

This chapter wraps up with a few simple observations on using SystemC to its greatest advantage. The authors provide hints about ways to keep simulation performance high and provide observations about the modeling language in general. This chapter contains no exercises. We leave application to your individual creativity.

17.1 Determinants in Simulation Performance

We sometimes hear comments from folks such as, "We tried SystemC, but our simulations were slower than Verilog." Such comments betray a common misconception. SystemC is not a faster simulator. The OSCI reference version of the SystemC simulator has several opportunities for optimization, and there are EDA vendors hoping to capitalize on that situation. More importantly, simulation performance is not so much about the simulator as it is the way the system is modeled.

KEY POINT: SystemC simulation speed is linked directly to the use of higher levels of modeling using un-timed and transaction-level concepts.

For all simulators (e.g., SPICE, Verilog, VHDL, or SystemC), there are a fundamental set of tasks that must be performed: moving data, updating event queues, keeping track of time, etc. Any simulator simulating detailed pin-level activity and timing information will provide a certain level of performance. Almost all simulators for a given class of detail will perform within a factor of two or so.

No so long ago, cycle-based simulators were all the rage due to their advertised speed. Problems arose when designers discovered that these simulators didn't provide the same level of accuracy as their event-driven counterparts. Indeed, that was exactly the reason they ran faster!

That said, RTL simulates at RTL speeds. Certainly, there are simulators that do RTL better than others, but they still have the limitation of keeping track of all the same details. A good optimizer may improve performance by finding commonalities, but the improvement will be bounded.

D.C. Black et al., *SystemC: From the Ground Up*, Second Edition,
DOI 10.1007/978-0-387-69958-5_17, © Springer Science+Business Media, LLC 2010

GUIDELINE: To improve simulation performance, reduce details and model at higher levels of abstraction whenever possible.

It is possible to obtain dramatic speed improvements by keeping as much of the system as possible at very high levels of abstraction, and only using details where absolutely required. This approach has the effect of minimizing the number of simulation context switches, which will keep performance high. That said, even if you are at the right level of abstraction, it is necessary to limit the number of calls to `wait()`.

Part of the problem lies with understanding what a given simulation is supposed to accomplish. Ask yourself, "What question is this simulation model supposed to answer?" For example, early in the design process the architect may wish to know if a new algorithm even works. At this level, timing and pins are not really interesting. A simple executable that takes input data and produces output for analysis is all that is required. Timing should not be a part of this model. Sequential execution is probably sufficient.

Another set of questions might be, "Have all the parts been connected? Have we defined paths for all the information required to perform the system functions?" These questions may be answered by creating a module for every component and using a simple transaction-level model to interconnect the pieces. Cycle accuracy should not be a concern at this point in the design.

17.1.1 Saving Time and Clocks

How can you live without time or clocks[1]? This is really quite simple. For instance, suppose you need to model time to determine performance. Rather than coding a wait for N rising edges, it is much more efficient to simply delay by *N**`clock_period`.

Another common technique occurs when using handshakes. If you need to wait for a signal, then simply wait on the signal directly. The hardware may do sampling at clock edges, but that wastes time. If you really need to synchronize to the clock, then do both, as follows:

```
wait(acknowledge->posedge_event());
if (not clock->event()) wait(clock->posedge_event());
```

Fig. 17.1 Synchronized wait for a signal

Perhaps you need to transfer information from one port to another in the design. Even though you know the result will be delayed through a FIFO over multiple clocks, there is no need to create a FIFO. Just read it from the input, delay, and write it to the output.

[1] We ask this question from an electronic system design perspective, not from a philosophical perspective.

```
In->read(packet);
wait(50*clock_period);
Out->write(packet);
```

Fig. 17.2 Example of FIFO elimination

Events are also a powerful way of communicating information. If you don't really need to test the value of a signal but are only interested in the change, it is more efficient to use an event than an **`sc_signal<bool>`**. Earlier in the book, we illustrated some primitive channels to do just this (e.g., in the `heartbeat` example).

Does your clock really need to oscillate at 100 GHz? Perhaps it would suffice to use a higher level clock. Do you really need GHz, or are MHz or even KHz sufficient?

Another overlooked issue is using too much or too little resolution. For instance, resolution should probably allow distinguishing at least clock periods for the fastest modeled clock.

17.1.2 Moving Large Amounts of Data

So, the model is efficiently using time, but it still seems to be simulating too slowly. Perhaps you are attempting to move too much data. Do you really need to move the data or do you just need to record the fact that data was moved and that an appropriate amount of time has passed?

For example, perhaps you could model the movement of a chunk of data as follows instead of moving the actual data:

```
struct payload {
unsigned long byte_count;
unsigned value; // a single unique value
};
```

Fig. 17.3 **`struct`** for `payload`

Now, you will need to modify the read/write routines in the channels to do something like this:

```
void Bus<payload>::write(unsigned addr, payload data)
{
  wait(data.byte_count*t_BYTE_DELAY);
  // transfer the data
}
```

Fig. 17.4 Bus write with `payload`

Perhaps, you need to transfer the data, but how much data do you really need to test the problem at hand? For instance, if dealing with video graphics, would a small 64 x 48 pixel buffer suffice to test an algorithm, rather than a full 640 x 480 or larger frame?

Perhaps, you need to transfer a large block of data across the bus, but can you model it using smart pointers instead? In other words, manage the chunk of simulator memory with a pointer. We recommend you use a Boost.org smart pointer, or the equivalent, to avoid problems with memory leaks or corruption.

Thus, you might have:

```
struct payload {
  unsignedlong byte_count;
  smart_ptr<int> pValues;
  payload(unsignedlong bc)
  : byte_count(bc)
  {
    pValues = new int[bc];
  }
};
```

Fig. 17.5 Smart pointer with `payload`

Now, you are simply passing around a pointer and only manipulating the data when it really needs to be manipulated.

Do you really need to fully populate a memory, or would a sparse memory model suffice? The SystemC Verification library contains a very nice sparse memory model that is very easy to use.

17.1.3 Too Many Channels

Another interesting area for SystemC designers to watch is channels. Every channel interaction involves at least two calls (producer and consumer), two events, and possibly two copy operations. Hierarchical **`sc_port`** to **`sc_port`**connections are very efficient because they simply pass a pointer to the target channel at elaboration. You will also find **`sc_export`** to **`sc_export`** hierarchical connections are similarly efficient. Another efficient way of communicating is **`sc_port`** to **`sc_export`**, since it can be implemented as a simple function call. If you find yourself writing a process that merely copies one port to another, consider the possibility of re-architecting the connectivity.

17.1.4 Effects of Over Specification

Often designers tend to think in terms of the final implementation rather than the general problem being designed. This approach sometimes results in too much specification. For instance, a behavior may be specified as a finite state machine (FSM), when the real issue is simply a handshake or data transfer. Be careful when presented with myriads of detail to abstract the real needs of the design.

17.1.5 Keep it Native

Keeping data native has already been discussed under data types earlier in the book, but this topic bears repeating. Data types are an abused subject. Does the model at hand really need to specify 17 bits, or would a simple **`int`** suffice? Native C++ data types will simulate many times faster than their SystemC hardware-specific counterparts. Similarly, what do you gain using **`sc_logic`**? Is the unknown value relevant to the current level of modeling? Once again, the issue is to model only those items that will affect the results of the simulation.

17.1.6 C++ Compiler Optimizations

Depending on the stability of your model, you may want to consider looking at optimizing your use of the C++ compiler. Many times, default `make` scripts assume that the developer wants maximum debug visibility, and the compiler obliges with additional visibility that may affect simulation performance.

When looking for maximum performance, make sure that your SystemC library and your system design are compiled without a debug option. Additionally, some compilers have switches that perform additional run-time optimizations at the expense of increased compile time. If you plan to run extensive simulation with the same model, it may pay to wade through the documentation for your compiler.

Another example, ensuring that **`#ifndef`** is on the first line of a header file improves performance for some compilers.

17.1.7 C++ Compilers

Many folks begin their SystemC explorations on the native C++ compiler that comes with their system, usually either GNU g++ or Microsoft Visual C++. It should not be a surprise that commercial alternatives exist with even better optimization.

17.1.8 Better Libraries

The STL library that comes with most C++ compilers is not always the most efficient implementation. There are commercial implementations of the STL that

should have much higher performance. The same can be said for the SystemC libraries that come from OSCI. If you are concerned with performance, it is probably worth your time to investigate your options with commercial solutions if you can afford it.

17.1.9 Better and More Simulation Computers

At the current rate of improvement in cost-performance, be sure you are running on the latest technology. It's a shame to not be spending $300-$1,000 for a potential 2x-3x performance gain. Similarly, why not increase the number of compute engines and run two, three, ten, or even hundreds of simulations at the same time. Compute farms are very effective for many types of modeling problems.

17.2 Features of the SystemC Landscape

Because SystemC is a C++ class library rather than a truly independent language, SystemC has some aspects that seem to annoy its users (particularly experienced designers from an RTL background). This section simply notes these aspects. Keep in mind that part of the power of SystemC is the fact that it is C++, and therefore, it is extremely compatible with application software.

17.2.1 Things You Wish Would Just Go Away

For the novice, just getting a design to compile can be a challenge. This section lists some of the most common problems. All of them relate directly to C++.

Syntax errors in **#include** files often are reported as errors in the including implementation (i.e., .cpp file). The most common error is forgetting to put the trailing semicolon on a **SC_MODULE**, which is really a **class**.

The use of **semicolons** in C++ may seem odd at times. The **class** and **struct** require a closing semicolon.

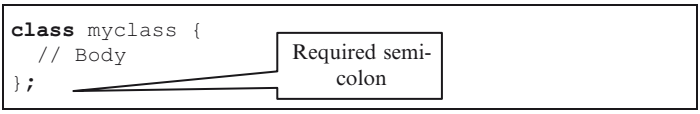


Fig. 17.6 C++ class requires semicolon

On the other hand, function definitions and code blocks do not require a semicolon.

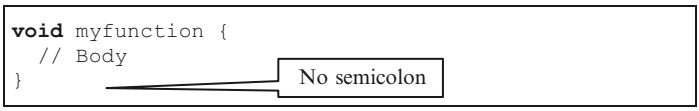


Fig. 17.7 C++ function does not use semicolon

Similarly, **SC_FORK**/SC_**JOIN**have the odd convention of using commas. This punctuation is used because they are really just fancy macros.

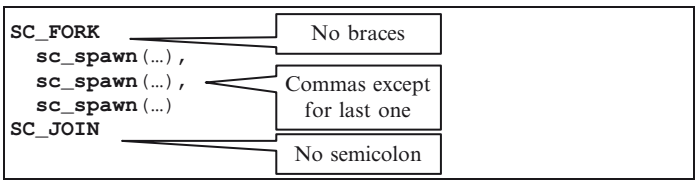


Fig. 17.8 C++ fork/join idiosyncrasy

SystemC relies heavily on templates. The templates have the annoying space between the greater than brackets.

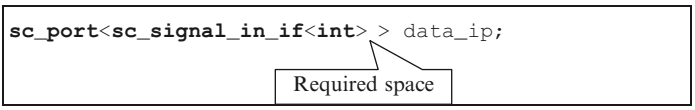


Fig. 17.9 C++ template idiosyncrasy

Inside the basic syntax of a module, **sensitive** and **dont_initialize** methods must be tied to the immediately preceding **SC_THREAD**, **SC_METHOD** or **SC_CTHREAD** registration. This tying is usually a lot easier to deal with if you indent the code slightly relative to the registration. For example:

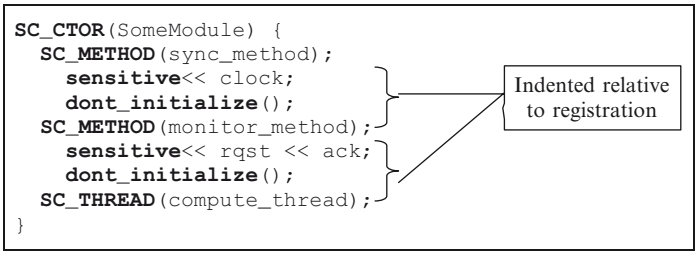


Fig. 17.10 Example of using indents to highlight registrations

dont_initialize brings up another issue common to **SC_METHOD**. Unless you specify otherwise, all processes are executed once at initialization despite the appearance of static sensitivity unless **dont_initialize** is used. For some, this behavior can be confusing at first. Try to remember that all simulation processes are run during initialization unless **dont_initialize** is applied.

17.2.2 Development Environment

What is the best way to address these idiosyncrasies besides just learning them? We highly recommend obtaining language-sensitive text editors with color highlighting, and we recommend obtaining lint tools designed specifically for SystemC. The authors' favorite text editor is vim in graphical mode, also known as gvim. You can obtain a copy of vim from www.vim.org for almost any platform.

Other users are quite successful using emacs (graphical of course) or nedit. All three of these editors have environments available for download that support SystemC. You can obtain these from our web site.

Another interesting environment is Eclipse.

There are a few C++ lint tools and at least one lint tool focused on SystemC that is commercially available[2]. Some EDA tools have built-in lint-like checkers. Your mileage will vary, and we highly recommend a careful evaluation before committing to any of these tools.

17.2.3 Conventions and Coding Style

Coding styles are a well-known issue, and lots of C++ rules and guidelines exist. Since SystemC is C++, this is a good starting point for SystemC coding guidelines.

Hardware designers have special issues to consider. Probably one of the best books written on this subject for hardware design is the *Reuse Methodology Manual for System-on-a-Chip Design*s by Michael Keating and Pierre Bricaud. Most of the concepts presented there have direct application to SystemC. Let's just touch on a few.

A name is a name, right? Wrong! Names of classes, variables, functions, and other matters are a critical part of making your code readable and understandable. If you have been observant, you will notice we've inserted various naming conventions specific to SystemC in the examples. For instance, processes always have a suffix of `_thread`, `_method` or `_cthread`. This convention is used because **wait**`()` results in a run-time error when used with **SC_METHOD**, and visa versa for **next_trigger**`()`.

[2] Actis Design www.actisdesign.com.

Similarly, we adopted a convention when addressing ports and probably you should do likewise for using anything **sc_signal**<*T*> or otherwise supporting the evaluate-update paradigm.

17.3 Next Steps

If you have read this far, you are probably considering adopting SystemC for an upcoming project. Or, perhaps you have already started, and you are looking for help moving forward. This section provides some ideas.

17.3.1 Guidelines for Adopting SystemC

In the fall of 2003, the authors presented a paper on the subject of language adoption, "How to Really Mess Up Your Project Using a New Language" at the Synopsys User's Group in Boston, MA. We included a number of key points, which we provide for your consideration.

1. Don't do it alone—Obtain management support.
2. Doing things the same way will produce the same results regardless of the language.
3. Look at the big picture, the product or system—Not the small tasks.
4. Don't skimp on training—Obtain good formal training.
5. Obtain mentoring.
6. Adopt the new paradigm to gain the advantages of a new language.
7. Specifications should use the appropriate level of abstraction for the new paradigm.
8. Put coding discipline in place quickly with coding guidelines, lint tools, and reviews.
9. Choose templates approved by seasoned experts in the new language.
10. Start automation and environment simply and cleanly.
11. Evaluate EDA tools for the big picture.
12. Insist on well-documented and supported tools in all areas including tools version and configuration.
13. Apply the technology to a pilot project that focuses on the big picture.

There are a number of companies supporting SystemC methodologies. A quick visit to the OSCI web site www.systemc.org can provide a starting point. Or, visit our web site, www.scftgu.com, for our view.

17.3.2 Resources for Learning More

For the readers who would like more information on SystemC extensions, we recommend the following resources for further study.

Table 17.1 SystemC resources

	Type	Details
1	Web site	Starting point for SystemC. Retrieved March 2004 from: http://www.systemc.org/. This site has several great papers and white papers as well as email forums for getting help or discussing SystemC.
2	Web site	The European SystemC Users Group web site. This site has additional quality papers and additional news and activities. Retrieved March 2004 from: http://www-ti.informatik.uni-tuebingen.de/~systemc/
3	Web site	The web site for the North American SystemC Users Group. Focused on SystemC activities in North America. Retrieved March 2004 from: http://www.nascug.org/
4	Web site	References for SystemPerl HDL tools. Retrieved March 2004 from: http://www.veripool.com/
5	Web site	The web site for sharing Open Source SystemC IP, tool and concepts. Retrieved July 2007 from http://www.greensocs.org/
6	Web site	The web site for the Latin America SystemC Users Group. Focused on SystemC activities in South America. Retrieved August 2007 from http://www.lascug.org/

For the readers still gasping for help with C++, here are some additional recommendations for further study.

Table 17.2 C++ resources

	Type	Details
1	Book	Koenig, A., Moo, B. *Accelerated C++*. Boston: Addison-Wesley, 2000. A highly recommended textbook for learning to speak C++ natively.
2	Book	Stroustrup, B. *The C++ Programming Language*. Florham Park, New Jersey:Addison_Wesley, 2000. Probably the best C++ reference and is written by the creator of C++.
3	Book	Loudon, K. *C++ Pocket Reference*. Sebastopol, California: O'Reilly & Associates, Inc., 2003. A convenient and reasonably organized quick reference. Good for those who are not yet C++ experts.
4	Book	Josuttis, N. *The C++ Standard Library*: A Tutorial and Reference. Indianapolis, Indiana: Addison-Wesley, 1999. A complete reference and good tutorial for the STL, a very useful library for modeling.
5	Web site	Stroustrup, B. Definitive reference for C++ by the author of C++. Retrieved March 2004 from:http://www.research.att.com/~bs/C++.html
6	Tool	van Heesch, D. Documentation system for C++ code. Retrieved March 2004 from: http://www.stack.nl/~dimitri/doxygen/index.html
7	Web site	References for C++ programming. Retrieved March 2004 from: http://www.cplusplus.com/
8	Book	Henricson, M., Nyquist. E. Industrial Strength C++. Upper Saddle River, New Jersey: Prentice Hall, 1996. (Online Version: http://www.elho.net/dev/doc/industrial-strength.pdf)
9	Web book	A free online book. Retrieved March 2004 from: http://www.mindview.net/Books/TICPP/ThinkingInCPP2e.html
10	Article	Hoff, T. C++ Coding Standard. Retrieved March 2004 from: http://oopweb.com/CPP/Documents/CodeStandard/VolumeFrames.html

(continued)

Table 17.2 (continued)

	Type	Details
11	Article	Baldwin, J. 1992. An Abbreviated C++ Code Inspection Checklist. Retrieved March 2004 from: http://www.chris-lott.org/resources/cstyle/Baldwin-inspect.pdf
12	Book	Pressman, R. *Software Engineering: A Practitioner's Approach*. McGraw-Hill, 2001. (Online Version: http://www.rspa.com/about/sepa.htm)
13	Web class	Free C++ class based on an inexpensive tool. Retrieved March 2004 from: http://www.codeWarriorU.com/
14	Web class	Free C++ class. Retrieved March 2004 from:http://www.free-ed.net/fr03/lfc/030203/120/
15	Book	Sutter,H & Alexandrescu, A. *C++ Coding Standards*, Addison-Wesley, 2004
16	Book	McConnell, S. Code Complete, 2nd Ed., Microsoft Press, 2004

We hope you'll be ready for our next book when we introduce topics such as the SystemC assertions, and we go deeper into a discussion of SystemC design methodologies and design styles. We also expect to provide updates as SystemC version 2.3, which is just now appearing on the horizon and will effect changes to the standard.

Appendix A
C++ Basics

A Quick Refresher

This appendix provides an extremely quick review of C++ with an eye towards those features used by the typical SystemC designer. We assume you already have a programming background that includes C/C++. If you do not have this background, you might consider this appendix as a guide to topics you need to master.

Here's what is covered:

Background of C++
Structure of a C program
Comments
Streams (I/O)
Streaming vs. printf
Basic C Statements
Expressions & operators
Conditional
Looping
Altering flow
Data Types
Built-in data types
User-defined data types
Constants
Declaration vs. definition
Functions
Pass by value & return
Pass by reference
Overloading
Constant arguments
Exceptions
Watching & catching exceptions
Throwing exceptions
Functions that throw
Defaults for arguments
Operators as functions
Classes
Member data & member functions
Constructors & destructors
Initialization
Inheritance
Access
Polymorphism
Constant members
Static members
Templates
Defining
Using
Names and Namespaces
Meaningful names
Ordinary scope
Defining namespace
Using names & namespaces
Anonymous namespace
Standard Library tidbits
Strings
File I/O
STL
References

A.1 Background of C++

C++ is a multi-paradigm programming language that owes much of its existence to BjarneStroustrup starting in 1980. It was originally designed to extend the C programming language to add features to enable easier object-oriented programming. In the end, it also added features that enabled modular programming, better data abstractions, and generic programming. C++ was eventually standardized in late 1998 as ISO/IEC 14882 (current version is 2003). C++ is not completely backward compatible with C, but it is close enough that probably 95% of C programs will compile quite easily as C++. For more information on the history of C++, please refer to the web page http://www.cplusplus.com/info/history.html.

A.2 Structure of a C Program

All C/C++ programs begin execution with a function known as **main**(). From there (Fig. A-1), data types are instantiated (created), statements are executed, and functions are called.

```
#include "headers"
// Declarations & definitions
int main(intargc,char* argv[]) {
  your_code_here
}
```

Fig. A-1. main.cpp

Normally, code is broken into separately compiled units consisting of two files: a header file, and an implementation file. Header files usually consist of pure declarations; whereas, implementation files contain the definitions of those declarations. A common file pair might look like (Fig. A-2 & A-3):

```
#ifndef ADVANCE_H
#define ADVANCE_H
int get_count(void);
void advance(void);
#endif
```

Fig. A-2. advance.h

```
#include "advance.h"
namespace { int count(0); }// Initialize count to 0
int get_count(void) {
return count;
}//end get_count
void advance(void) {
  ++count;
}//end advance()
```

Fig. A-3. advance.cpp

A.3 Comments

Comments (Fig. A-4) and white space should be used liberally in any programming language. White space helps guide the reader, which may be you several years down the line. Comments should not describe the syntax, but should focus on the nature of the algorithm, tricks employed to solve the problem, or some other non-intuitive aspect of the code.

```
// Comment to end of line - recommended style
/* Embedded comment - does NOT nest */
```

Fig. A-4. C/C+ comments

The "/* */" comment style is recommended for use only when debugging. The "//" comment style is preferred for general commenting because it is less likely to result in errors (e.g., when a code segment is temporarily commented out with "/* */").

A.4 Streams (I/O)

One of the most notable features to C users is the manner of handling I/O. In particular, C++ programmers use a feature known as streaming I/O rather than the familiar **`printf`**. The following is an example (Fig. A-5) of output followed by input:

```
#include<iostream>

using namespace std;

main() {
float f1, f2;
cout << "Enter 2 numbers separated by blanks: ";
cin >> f1 >> f2;
cout <<"You entered " << f1 << " " << f2 << endl;

return 0;
}
```

Fig. A-5. Example of streaming I/O

We illustrate this now to let us use it in subsequent discussion. Note that the **iostream** objects, **cout**, **cin**, and **endl**, are part of the **std** namespace. To use these objects, you must tell the compiler you are using the **std** namespace as indicated above. Otherwise, you must specify **std::** before each object in the standard library, i.e., **std::cout**.

A.4.1 Streaming vs. printf

Many C programmers wonder why they should use streaming I/O. One reason is that it is type checked unlike the **%s** and **%d** arguments of **printf**. The second reason relates to code reuse and ease of use when using complex types. This reason will become evident later when considerations of object definitions are discussed.

A.5 Basic C Statements

This section will briefly touch on standard C statements, which form a foundation for C++.

A.5.1 Expressions and Operators

Expressions normally take the form of assignment statements with arithmetic or Boolean computations taking place on the right-hand side (RHS). In C, it is not required that you store the result of an expression; however, C++ compilers will normally warn about this situation. Consider the following (Fig. A-6) common expressions:

```
a = b + (c = 7*j); // Notice assignment to c
error = (a < max) && (a > min);
++b;
c + 9; // results in a warning
```

Fig. A-6. Example of C++ expressions

Table A.1 shows a list of all the operators allowed in C++ in order of precedence. The last column indicates the order of associativity, which is either left to right (L2R) or right to left (R2L):

Table A.1. C++ operators

Prec	Operator	Description	Assoc
1	::	Scoping, global	R2L
	::	Scoping, class	L2R
2	()	Grouping	L2R
	[]	Array access	
	->	Member access from pointer	
	.	Member access[a]	
	++ --	Post-increment/decrement	
3	!	Logical negation	R2L
	~	Bitwise complement	
	++ --	Pre-increment/decrement	
	+ -	Unary plus/minus	
	*	Dereference	
	&	Address of	
	(Type)	Cast	
	sizeof	Return size in bytes[a]	
4	->*	Member dereference from pointer	L2R
	.*	Member dereference[a]	
5	* / %	Multiply, divide, modulus	L2R
6	+ -	Addition, Subtraction	L2R
7	<<	Bitwise shift left	L2R
	>>	Bitwise shift right	
8	<	Less than	L2R
	<=	Less than or equal	
	>	Greater than	
	>=	Greater than or equal	
9	== !=	Equality, inequality	L2R
10	&	Bitwise AND	L2R
11	^	Bitwise exclusive OR	L2R
12	|	Bitwise inclusive OR	L2R
13	&&	Logical AND (shortcut)	L2R
14	||	Logical OR (shortcut)	L2R
15	?:	Ternary conditional[a]	R2L
16	=	Assignment	R2L
	+= -=		
	*= /=		
	%= &=		
	^= | =		
	<<= >>=		
17	,	Sequential evaluation	L2R

[a]Cannot be overloaded

A few important notes on operators are useful:

1. Mixing more than one pre/post-increment/decrement operator may have undefined consequences. Consider (Fig. A-7):

```
a = i++ + i++; // legal syntax, undefined behavior
```

Fig. A-7. Abusing post-increment operators

2. The shortcut operators (**&&** and ||) can surprise you if the secondary expressions have side effects or depend on side effects of the primary expression. For instance, in the example below (Fig. A-8) if the right-hand side is unconditionally evaluated, the code would abort when the pointer is a null (0). The shortcut behavior avoids the abort.

```
ptr != 0 && ptr->next(); // avoids null pointer
```

Fig. A-8. Taking advantage of shortcut operators

3. Several operators have keyword alternatives that may be easier to read, especially when your text editor has keyword highlighting (e.g., vim, emacs, or nedit). Here (Table A-2) is a list with our recommendations:

Table A.2. Alternate names for operators

Useful		Distracting		Annoying	
&&	and	^	xor	&=	and_eq
||	or	&	bitand	|=	or_eq
!	not	|	bitor	!=	not_eq
~	compl			^=	xor_eq

4. Be careful not to abuse the ternary operator ?: because it can be confusing to debug if nested. Often **if-then-else** is better.
5. Most of the operators have a second name, not shown, that is constructed by preceding the symbol with keyword **operator**. For instance, **operator+** is the addition operator. This alternative name is for use with operator overloading discussed in Section A.7.6.
6. Overloading syntax of pre/post-increment/decrement is a bit odd in order to distinguish between pre and post. Look them up if needed.
7. The operators dot (**.**), scope (**::**), **?:**, and **sizeof** cannot be overloaded.

A.5.2 Conditional

There are two conditional statements (Fig. A-9): the **if** and the **switch**. It is strongly suggested that you use curly brackets ({}) around all *statements*.

```
if (expression) statement
else statement

switch (expression) {
case integral: statement
  …
default: statement
}
```

Fig. A-9. Conditional statement syntax

It is good practice to place a **break** statement after each **case**, since the behavior without a **break** is to drop through into the succeeding case. It is also good practice to always have a **default** case. Thus, a typical case statement might look as follows (Fig. A-10):

```
switch (c) {
case 'a':
cout<< "Aborting…" <<endl;
break;
case 'q':
case 'x':
cout<< "Quiting" <<endl;
break;
case 'c':
cout<< "Quiting" <<endl;
break;
default:
cout<< "Unknown command '" << c << "'" <<endl;
break;
}//endswitch
```

Fig. A-10. Switch statement

A.5.3 Looping

Loops (Fig. A-11) are the essence of most functional programming.

```
while (expression)statement
do statement while expression;
for (init_expr;test_expr2;next_expr3)statement
```

Fig. A-11. Looping statement syntax

It is common to define for loops as the following examples (Fig. A-12) demonstrate:

```
#include<vector>
for (int i(0); i!=10; ++i) {
code
}//endfor

std::vector<int> v;
typedef std::vector<int::iterator vint_iterator;
for(vint_iterator i(v.begin()); i!=v.end(); ++i) {
code; // iterator over elements of v
}//endfor
```

Fig. A-12. Typical for loops

Notice the local definition of the indexing variable. Also notice the code pattern for iterating over an STL container (e.g., **std::vector<>**).

A.5.4 Altering Flow

With exception of the **return** and **break** statements, the following statements (Fig. A-13) are used sparingly. The **return** statement is best used once at the end of a function. The **goto** statement is almost never used.

```
break; // exit a case, while, do or for loop
continue; // skip to the end of a loop
goto LABEL; // jump to a label with restrictions
LABEL:;
return [type]; // exit from a function
```

Fig. A-13. Flow altering statement syntax

A.6 Data Types

This section reviews built-in and user-defined data types as well as constants. Lastly, this section explains the difference between a declaration and a definition.

A.6.1 Built-In Data Types

C/C++ has several simple built-in data types as follows (Fig. A-14):

```
enum bool{false, true}// preferred over 0/non-0
int      i; // 32 bit signed integers
char     c; // single 'c' characters
float    f; // single precision floating point
double   d; // double precision floating point
long     l; // 4bytes; machine dependent;
short    s; // 2bytes; machine dependent;
unsigned u; // modifies int
```

Fig. A-14. Built-in data type definitions

A.6.2 User-Defined Data Types

User-defined data types are constructed from arrays, structures, or unions. Unions are rarely used. In addition, pointers and references may be specified as modifiers to any data type. In the following figures, the name T is used to denote a generic "type" and may be replaced with any predefined data type including other user-defined types.

One simple way to create the appearance of a user-defined type is using the **typedef** statement (Fig. A-15). Typedefs do not create a new type, but are simply an alias or shortcut to specifying a type. Here is the syntax to alias an **int** with T:

```
typedef int T;
T i; // i is really just an int
```

Fig. A-15. Built-in data type definitions

A.6.2.1 Pointers, Arrays, and References

Pointers and arrays come from C syntax; whereas, references are a new construct for C++.

Pointers underlie many complex types, but due to their nature they are extremely bug prone. When possible, avoid using pointers. It is preferred to use containers from the STL (later in this section).

References are more commonly used in functions and will be discussed in Section A.7.2.

Arrays are familiar to most programmers; however, due to the lack of bounds checking, most C++ programmers prefer to use STL vectors for this purpose (discussed later in this section). In any event, arrays are really pointers pointing to an area containing N copies of the base type. The notation arr[i] is equivalent to *(arr + i).

An important point for C++ is the use of the free store, which is managed by **new** and **delete**. Do **not** use **malloc** or **free** in C++ code. Here (Fig. A-16) is the syntax for **new** and **delete**:

```
// definitions used below
typedef int T;
 const int N = 5;
 T value(3);

// Simple pointer
T* my_ptr; // define pointer
my_ptr = new T; // allocate space
*my_ptr = value; // dereference/use pointer
delete my_ptr;

// Simple array
T  arr[N]; // Array of homogenous elements
arr[0] = value; // using the array
*(arr+1) = value;   // another way to use

// Create pointer to an array
T* my_arr;
my_arr = new T[N];
my_arr[3] = *my_ptr; // example of use
delete [] my_arr; //important syntax for array ptr

T & ref(N); // Reference to an object v of type T
```

Fig. A-16. Using **new** and **delete**

A.6.2.2 Structures

The following syntax (Fig. A-17) denotes declarations of new user types:

```
// Structures contain heterogeneous elements
struct Name {
  T1 element1;
  T2 element2;
  …
};

class Name { // Almost a struct with a twist
  T1 element1;
  T2 element2;
  …
};

union Name {
  T1 element1;
  T2 element2;
  …
};
```

Fig. A-17. Container declarations for user-defined data types

Each of the preceding types contains zero[1] or more elements. The user may then reference each element of the construct using the dot operator as illustrated in the following figure (Fig. A-18) for a **`struct:`**

```
using namespace std;
struct Race_Driver {
  string first_name, middle_name, last_name;
  unsigned age;
  int win_vs_loss;
  string prev_race;
  int prev_year;
};

Race_Driver Andy; // Instantiation of a race driver
Andy.first_name = "Andrew";
Andy.last_name = "Priaulx";
Andy.age = 2006-1976;
Andy.win_vs_loss = 1;
Andy.prev_race = "British Touring Car Championship";
Andy.prev_year = 2001;
```

Fig. A-18. Example using a struct

Classes will be discussed in Section A.8. Unions are a method of saving memory space. At any one point in time, only one of the union's elements is valid since they all share the same memory location. The size of a union is the size of the largest element type. Unions are rarely used.

[1] Usually one or more.

A.6.2.3 STL

It is worth mentioning that the STL has some alternative containers that may be preferable to the built-in types. The next example (Fig. A-19) shows a few. The syntax of templates is covered in Section A.9.

```
std::pair<T1,T2> p;     // 2-tuple (.first & .second)
std::vector<T>   v(N); // improved array type
std::list<T>     l;     // linked list
std::map<T1,T2>  m;     // associative array
std::set<T>      s;     // set of unique values
```

Fig. A-19. Common STL containers

We do not go further into the STL types here; however, you are strongly urged to learn more about them (there are many books on this subject) and use them whenever possible.

A.6.3 Constants

C++ provides the **const** construct (Fig. A-20) to denote constants. The **const** construct has a marked advantage over the traditional C **#define** approach because data types are checked during compilation and the error messages are easier to understand. It is also possible to use the **enum** construct for integral constants.

```
using namespace std;
int const CYLINDERS(10);
string const ERROR42("Earth does not exist");
enum { WHEELS=18 };
string * const mesg = &ERROR42; // constant pointer
```

Fig. A-20. Examples of constants

An important aspect of constants is that their values need to be initialized at the time they are constructed. More about initialization will be discussed with the topic of classes in Section A.8.

A.6.4 Declaration vs. Definition

It is important to recognize the difference between declaration and definition:

Declaration states the existence of an identifier and its characteristics.

Definition allocates memory space and defines functionality.

For global data, the **extern** directive serves to *declare* a variable. The absence of this keyword is a *definition* since space is allocated. For functions the distinction is easier to see. Consider the following (Fig. A-21) code snippet:

```
extern int A; // Declare existence of an integer A,
              //in global scope
int A; // Define storage for an integer named A
int F(); // Declare a function F with no parameters
int F() { return 5; } //Define function F's behavior

struct S; // Declare a struct P exists
S* as_p; // Define a pointer to structure object S
struct S { // Declare the contents of structure S
  float a;
  bool b;
};
as_p = new S; // Allocate storage for instance S
class T; // Declare a class T exists
class T { // Declare contents of class T
  int m_i; // - has an integer data member m_i
  public:
  void H();// - has a public member function H
};
void T::H() { // Define implementation of H
  cout<< "Hi" <<endl;
}
T x; // Define an object of type T
```

Fig. A-21. Declaration vs. definition

A.7 Functions

Functions are known as subroutines, procedures, or methods in other languages. Functions are an encapsulation of programming behavior that may be used to break down a problem. Functions have three syntaxes.

First (Fig. A-22), functions are declared to establish their argument syntax.

```
float add_time(float curr_hrs, int delta_secs);

void display(string message);
```

Fig. A-22. Examples of function declaration

Second, (Fig. A-23) functions are defined to establish their implementation code and behavior.

```
float add_time(float curr_hrs, int delta_secs) {
  return (curr_hrs + delta_secs/3600.0);
}

void display(string message) {
  cout << message <<endl;
  return; // optional
}
```

Fig. A-23. Examples of function definitions

Third (Fig. A-24), functions are called from other functions to initiate their behavior.

```
float total(0.0);
total = add_time(total,15);
display("Drivers, start your engines");
```

Fig. A-24. Examples of function calls

A.7.1 Pass By Value and Return

By default, arguments to functions are passed by value. This means they are copied into the arguments storage placed on the executing computer's stack. As demonstrated in the preceding example of add_time, a value may be returned using the **`return`** statement.

A.7.2 Pass by Reference

In addition to pass by value, C++ allows pass by reference. This feature reduces a common error in C that occurs when passing pointers. The purpose of references is twofold. First, references let us modify variables passed through the arguments of a function. For instance (Fig. A-25):

```
#include<iostream>
void advance(int & var, int max = 10) {
  if (var == max) var = 0;
  else            ++var;
}

int n(5); // n starts at 5
while (n != 4) {
  cout << "n is " << n << endl;
  advance(n);
}//endwhile
```

Fig. A-25. Example of reference usage

In the preceding (Fig. A-25), values printed out will be 5, 6, 7, 8, 9, 10, 0, 1, 2, 3.

A second use of pass by reference is to reduce copying (and hence, decrease computation time). If you pass a large structure by reference, the compiler doesn't have to copy the entire structure onto the stack.

A.7.3 Overloading

C++ has the useful ability to overload a function name and provide more than one function of the same name. Which function to use is determined by comparing the types of arguments and the number of arguments. The return type is **not** used to determine the signature of a function when overloading. Thus, the following (Fig. A-26) is a legal set of functions:

```
float add_time(float curr_hrs, int delta_secs);
void add_time(float & total_hrs, int delta_secs);
float add_time(float curr_hrs, float delta_hrs);
float add_time(float curr,int mins, int secs);
```

Fig. A-26. Example of overloaded function name

A.7.4 Constant Arguments

Using the **const** keyword in C++ indicates to the compiler that values will not be modified inside a function. If modification is attempted, a compile-time error will result. This usage is most commonly used with pass by reference. Consider the following (Fig. A-27):

```
#include <vector>
typedef std::vector<int>::const_iterator
                                        vint_iterator;
int average(std::vector<int> const & v){
  int sum(0);
  for (vint_iterator i=v.begin();i!=v.end();++i) {
  sum += *i;
}//endfor
  return sum/v.size();
}
```

Fig. A-27. Example of constant arguments usage

A.7.5 Defaults for Arguments

It is possible to specify default values for arguments as illustrated in the next example. Defaults may be specified for trailing arguments only. Furthermore, the default should be specified in one place only (typically in the declaration). Here (Fig. A-28) are a couple of examples:

```
void test(int data, bool random=false,
          bool debug=false);
typedef int packet;
void put(packet& p, int inject_errors=0);
```

Fig. A-28. Example of default arguments specification

Default arguments can create ambiguities that need to be considered. For instance (Fig. A-29), consider the following:

```
void test(int data, bool random=false,
          bool debug=false);
void test(int data); // Error: ambiguous
```

Fig. A-29. Example of ambiguity

The above is a problem because you can omit both the random and debug arguments. This omission causes the compiler to be confused over which test you mean to use.

A.7.6 Operators as Functions

C++ treats operators as functions and provides special names for all of the functions. This treatment allows operator overloading. Operator overloading is

really no different than function overloading. Consider the following (Fig. A-30) simple examples:

```
// Create a custom data type
enum color {black, red, magenta, yellow,
green, cyan, blue, white };

color operator+(color lhs, color rhs) {
  // Define what it means to add colors
  if (lhs == rhs)                    return lhs;
  else if (lhs == black)             returnrhs;
  else if (rhs == black)             return lhs;
  else if (lhs == white)             return white;
  else if (rhs == white)             return white;
  else if (lhs == red &&rhs == blue) return green;
  // etc…
}

// Modulus rotation through colors
color operator+(color lhs, intrhs) {
  return color((int(lhs) + rhs) % 8);
}
```

Fig. A-30. Operator overloading

A.8 Classes

For many programmers, the object-oriented (OO) aspect of C++ is the reason for using C++. Certainly OO is an important part of the language.

A.8.1 Member Data and Member Functions

The concept of an object is really quite simple. In C++, all data types are fundamentally objects. Objects have certain functions they can perform. For instance, an **int** may be queried (i.e., its value determined and displayed), set/modified (assigned to), and operated on with another **int** or even perhaps another data type. For user-defined types, we use a **struct** to describe an object type with a minor extension. Functions are allowed as members of a **struct** in C++.

We also introduce a new keyword, **class**, to document our intent when defining a class. The keyword has one minor difference from a **struct**, which relates to data encapsulation (hiding). This difference necessitates the introduction of a second keyword **public**, which allows class members to be visible from the outside. The next example (Fig. A-31) illustrates class declaration and definition.

One important aspect of a class is that it creates its own namespace. Thus, when member functions (methods) are defined, they must be prefixed with the class name. This prefix distinguishes member functions from ordinary functions and other classes.

```
// Declaration of a class
class Thermometer {
  int m_temp;
  string m_name;
public:
  void set_temp(intval);
  int get_temp();
  void set_name(string nm);
  int get_name();
};

// Definition of member functions
void Thermometer::set_temp(int val) {
  m_temp = val;
}
int Thermometer::get_temp() { returnm_temp; }
void Thermometer::set_name(string nm) {
  m_name = nm;
}
string Thermometer::get_name() { returnm_name; }

// Use of a class
#include<iostream>
int main(intargc, char *argv[]) {
  Thermometer dashboard;
  dashboard.set_temp(72);
  dashboard.set_name("inside");
  cout << dashboard.get_name() << "="
       << dashboard.get_temp() << endl;
}
```

Fig. A-31. Example of trivial class definition and usage

Notice that using the class follows the same syntax used for accessing member data in a **`struct`**.

This example (Fig. A-31) instantiates two simple object members, an integer, `m_temp`, and a string, m_name. We refer to the class as having a "has a" relationship with respect to the integer and string. Classes are typically built of many other classes this way. This usage is known as construction by composition.

A.8.2 Constructors and Destructors

Something assumed by most programmers is that when an object such as an integer is defined, the compiler allocates space for it, and ideally initializes it to an initial value (e.g., zero). This process of allocation and initialization is called construction.

For a non-trivial class (i.e., a class other than the built-in types) these steps may involve a bit of work.

C++ provides for a constructor method that is automatically called when construction occurs. The name of the constructor is the same as the name of the class. It is distinguished from other methods in that it has no return value.

Constructors may take zero or more arguments. If no constructor is defined by the programmer, then C++ provides a default constructor that takes no arguments. The default constructor simply allocates data member objects and calls their default constructors. Because C++ allows for function overloading, there may be more than one constructor defined in a class. Here (Fig. A-32) is an example:

```
// Declaration of a class
class Thermometer {
  int m_temp;
  string m_name;
public:
  // 4 Distinct Constructors
  Thermometer(); // Default constructor
  Thermometer(int val);
  Thermometer(string nm);
  Thermometer(string nm, int val);
  // Ordinary methods
  void set_temp(int val);
  int get_temp();
  void set_name(string nm);
  string get_name();
};

// Definition of methods
Thermometer::Thermometer() {m_name = "unknown"; }
Thermometer::Thermometer(int val) { m_temp = val; }
Thermometer::Thermometer(string nm) { m_name = nm; }
Thermometer::Thermometer(string nm, int val) {
  m_name = nm;
  m_temp = val;
}
void Thermometer::set_temp(int val) {
  m_temp = val;
}
int Thermometer::get_temp() { return m_temp; }
string Thermometer::get_name() { return m_name; }

// Use of a class
int main() {
  Thermometer i1, i2();
  Thermometer i3(15), i4("inside"),
              i5("inside", 72);
  Thermometer i6('B');
}
```

Fig. A-32. Example of a class with constructors

In the example, the default constructor is defined to initialize the initial name to "unknown". In the usage section, we illustrate six different instantiations of Thermometer class objects using the four different constructors. The first and second instances are identical in that they invoke the default constructor.

The last instance, i6 illustrates a problem. The character value **'B'** in single quotes is implicitly converted to an integer (value 66), and is probably not the intended result. To fix this situation, use the keyword **`explicit`** in the declaration as follows (Fig. A-33).

```
explicit Thermometer(int val);
```

Fig. A-33. Declaring a function to have explicit arguments

Now, the char situation becomes illegal. C++ enforces that the data type of the argument must be **`int`** explicitly, and C++ will not perform implicit conversions, which makes i6 illegal.

A.8.2.1 Initialization

It may seem that all issues with initialization are taken care of with constructors; however, there is one more syntactical device needed. Consider (Fig. A-34) a class member that comes from a class that has only a single constructor and that constructor requires an argument (i.e., there is no default constructor).

```
class Tire {
  unsigned m_size;
public:
  // Constructor declared & defined
  explicit Tire(unsigned size) {m_size = size;}
};
class Wheel {
  Tire tire_i;
  bool chrome;
public:
  Wheel(unsigned size);
};
Wheel::Wheel(unsigned size) { //Error
  // How to supply argument to tire_i?
}
```

Fig. A-34. A class with a single constructor instantiated in a second class

Because a constructor (Fig. A-34) is defined for Tire, the default constructor does not exist. This usage creates a problem because a constructor must be called

when the Wheel class is constructed. C++ solves this problem (Fig. A-35) by creating syntax for construction known as an initialization list as shown in the next example. The list is defined in the constructor and begins with a colon after the constructor signature. The list continues with comma-separated items.

```
Wheel::Wheel(unsigned size)
: tire_i(size), chrome(true)
{
  // other initialization
}
```

Fig. A-35. Class initialization list

In fact, most initialization can occur inside the initialization list. The order of initialization follows the order in which data members are declared—not the order of the initialization list.

A.8.3 Destructors

Suppose a class is created containing a pointer and during construction, the pointer is set to point at a new data object allocated on the heap with **new**. To avoid a memory leak, it will be necessary to delete the object when any instance of the class is destroyed.

This usage is an example of the need for a destructor method. A destructor is a method that is called whenever an object is destroyed. An object is destroyed when the object goes out of scope, an explicit **delete** is issued, or the program terminates. An object goes out of scope when the block of code in which it was defined terminates.

C++ defines a destructor method to have the same name as the name of the class prefixed with a tilde (~). A destructor has no arguments and there is only one per class. Here (Fig. A-36) is an example:

```
class Pickup {
public:
  ~Pickup(); // destructor declared
};
Pickup::~Pickup() { // destructor defined
  cout<< "Pickup destroyed" <<endl;
}
```

Fig. A-36. A destructor

A.8.4 Inheritance

One of the main features of object-oriented programming is the notion of code reuse through the mechanism of inheritance. Inheritance lets one define a class to inherit the functionality of a parent class (also known as a base class). The inheriting class is known as a child or derived class. Inheritance is established by specifying the parent class immediately after the child class name separated with a colon. Here (Fig. A-37) is a simple example:

```
class Tire {
  unsigned m_size;
public:
  Tire(unsigned size) { m_size = size; }
  unsigned size() {return   m_size; }
};
class Allweather
: public Tire // inherit from Tire class
{
  int traction;
  public:
  Allweather(int size);
  int friction();
};
```

Fig. A-37. A class with a single constructor instantiated in a second class

The child class `Allweather` inherits from the parent class `Tire`. Hence, `Allweather` has the methods of the parent class available. Because the inheritance specified public, these methods are available to users of the child class, `Allweather`. Thus, inheritance allows the child class to reuse the code already written for the parent.

The mechanism of inheritance establishes an "is a" relationship for the child. The `Allweather` class is a `Tire`. The converse is not true.

A.8.4.1 Adding Members

The class `Allweather` also extends the capabilities to include a traction data item and a `friction`() method. Thus, this class has extended capabilities.

A.8.4.2 Initialization of a Base Class

If the parent class requires a specific constructor to be called, call out the parent class with appropriate arguments (Fig. A-38) in the constructors' initialization list.

```
Allweather::Allweather(int size)
: Tire(size)
{
  // other initialization
}
```

Fig. A-38. Specifying a parent class constructor

A.8.4.3 Overriding Inherited Member Functions

A derived class may specify different behaviors for an inherited method. Furthermore, the behaviors of the parent class may be called by adding scope information to the name. Here (Fig. A-39) is an example:

```
class Tire {
unsigned m_size;
public:
Tire(unsigned size) { m_size = size; }
unsigned size() { return m_size; }
};
class Allweather
: public Tire
{
  int traction;
public:
  Allweather(int size):Tire(size){};
  int friction();
  unsigned size() {
    cout<< "overrides Tire's size()" <<endl;
    return m_size+1;
  }
};
```

Fig. A-39. A class with a single constructor instantiated in a second class

A.8.4.4 Multiple Inheritance

C++ allows for inheriting from more than one parent. Simply add additional parents as a comma-separated list in the inheritance specification. Although debated in some circles, multiple inheritance has proven quite useful in a number of applications including SystemC.

A.8.5 Public, Private and Protected Access

C++ supports data hiding. In Section A.8.1, we introduced the keyword **`public`**. There are two other related keywords, **`private`** and **`protected`**, related to this concept of access:

Public members are available for access by users of a class and internally.

Private members are only available to member functions of the class in which they are defined.

Protected members are available to both the class and derived (inheriting) classes. Thus, protected members are private with respect to users.

A.8.5.1 Friends

A class may have private or protected members that it wishes to make available to non-class member functions. It can do so by explicitly declaring a function to be a friend. Friend functions have complete access to anything inside the class. In other words, a friend is considered to have public access to all the members of a class that declares it a friend.

A.8.6 Polymorphism

Sometimes it is useful to create functions that operate on more than one class by means of a parent class. For instance, a `Vehicle` class might have common weight property (member data). It would be useful to have a function determine the aggregate weight of a variety of vehicles that are described with various derived classes. However, the **`weight`**() method shown in the following example (Fig. A-40) may have been overridden:

```
class Vehicle {
public:
  unsigned weight()
  { abort(); } // No valid implementation
};
class Truck : public Vehicle {
  unsigned m_weight;
public:
  unsigned weight() { return m_weight; }
};
class AirShip : public Vehicle {
  unsigned m_weight;
  bool m_inflated;
public:
  unsigned weight()
  { return (m_inflated ? 0 : m_weight); }
};
unsigned add_weights(Vehicle& v1, Vehicle v2) {
  return v1.weight() + v2.weight(); // Aborts!
}
```

Fig. A-40. The need for polymorphism

This preceding implementation does not implement polymorphism. For that we need an additional construct, the virtual designation.

A.8.6.1 Virtual

Adding the **virtual** qualifier to a method's declaration causes the compiler to add a small indirection table to the object structure for each member declared **virtual**. Each time the method is invoked, the compiler uses this table to determine where the method's code lives. Notice the designation must be added at the point where polymorphism is desired. For the example of figure A-41, we must designate the vehicle's **weight**() method to be **virtual**.

```
class Vehicle {
public:
virtual unsigned weight()
{ abort(); return 0; } // No valid implementation
                       needs a return value
};
class Truck : public Vehicle {
  unsigned m_weight;

public:
  unsigned weight() { return m_weight; }
};
class AirShip : public Vehicle {
  unsigned m_weight;
  bool m_inflated;
public:
  unsigned weight()
  { return (m_inflated ? 0 : m_weight); }
};
unsigned add_weights(Vehicle& v1, Vehicle v2) {
  return v1.weight() + v2.weight(); // Aborts!
}
```

Fig. A-41. Using polymorphism

Notice that the keyword **virtual** only needs to be added once to the topmost class.

A.8.6.2 Abstract and Interface Classes

Although adding the virtual designator to the preceding example enables polymorphism, it leaves an undesirable feature. It is possible to instantiate an object of the `Vehicle` class and call its **weight**() method. Sadly, this results in an abort. It would be desirable to prevent this situation from arising in the first place. For that reason, C++ has the concept of a pure virtual method. The syntax is modified by replacing the implementation with "**= 0**". Conceptually, this state of the method has no implementation.

Here (Fig. A-42) is the modified Vehicle class:

```
class Vehicle {
public:
  virtual unsigned weight() = 0; // Pure virtual
};
```

Fig. A-42. Pure virtual method makes an abstract class

With the addition of pure virtual methods to a class, it now becomes a compile-time error to attempt to instantiate an object of that class. The only way to use this class is to derive another class from it and provide an overriding implementation.

A.8.7 Constant Members

C++ constants must be given a value at the point of construction. For a class, this means (Fig. A-43) the construction must be specified in the initialization list.

```
class A {
  int const the_answer;
  A() // Constructor
  :the_answer(42) // Initialization list
  {} // Body of constructor
};
```

Fig. A-43. A class constant

A.8.8 Static Members

Member data and member functions of a class declared **static** are common to the entire class (Fig. A-44). A static data member that needs a non-default constructor must be constructed external to the class declaration. A static function member may call other static member functions only.

```
class A {
  static int m_count;
  static int count() {return m_count; }
  A() { m_count++; } // Constructor
  ~A() { m_count--; }// Destructor
};
int A::m_count(0); // initialize
```

Fig. A-44. Static class members

A.9 Templates

C++ supports the paradigm of generic programming through the use of the template construct. Templates apply to both functions and classes. The STL is a collection of classes that make heavy use of the template concept.

A.9.1 Defining Template Functions

Defining a template is best considered with a small example (Fig. A-45). Consider the problem of creating a destroy function that takes a pointer by reference, deletes it and sets the pointer to the null pointer value. Since C++ is heavily typed, we need to create a function for every pointer type. Here is how to do this with templates:

```
template<typename T>
void destroy(T*& p) { delete p; p = 0; }
```

Fig. A-45. Defining a function template

For every type, T, we can now have a destroy function.

Template parameters are limited to typenames (keywords **typename** or **class**) and integral types (e.g., **int**, **unsigned**, and enumerations).

It is possible (Fig. A-46) to have more than one template parameter and optionally specify default values for a templated class.

```
template<int max, int min, typename T>a
T limit(T val) {
   assert(min > max);
   if (val< min)       return min;
   else if (val> max) return max;
   else return val;
}
```

Fig. A-46. Defining a function template

A.9.2 Using Template Functions

Using function templates is much easier than defining them. Simply (Fig. A-47) specify the function name with the template parameters inside angle brackets.

```
string* msg_ptr = new string("Hello");
…
destroy<string>(msg_ptr);

cin>> v;
cout<< "Limit value " << limit<15,-3>(v) <<endl;
```

Fig. A-47. Using function templates; does not compile

A.9.3 Defining Template Classes

Class templates are very similar to function templates. Class templates just carry more complexity because they are larger. Consider (Fig. A-48) a FIFO template class that allows FIFOs of any data type:

```
template<typename T, int maxdepth=1>
class fifo {
 vector<T> m_fifo;
 public
 void push(T v);
 T pop();
 bool full()   { return m_fifo.size() >= maxdepth; }
 bool empty()  { return m_fifo.size() == 0; }
};
```

Fig. A-48. Defining a class template

A.9.4 Using Template Classes

Usage of a template class (Fig. A-49) is practically trivial.

```
fifo<double> readout_fifo;
fifo<string> message_fifo;

readout_fifo.push(2.71);
```

Fig. A-49. Using a class template

A.9.5 Template Considerations

There are many subtle aspects to templates that are well beyond the scope of this appendix. For example, many (99%) C++ compiler implementations restrict

templates from being compiled separately. One common gotcha happens when using a class template of a class template.

Most of the subtleties are related to defining the templates. Well-defined templates are easy to use. Entire books are devoted to discussing templates, and we advise consulting them if you intend to define your own templates.

A.10 Names and Namespaces

Names are used for many things including keywords, which are reserved, and user identifiers, which are used for variables, constants, and functions.

A.10.1 Meaningful Names

Please consider that using carefully chosen meaningful names is a very important part of any programming activity. Obtaining and using a coding standard is strongly recommended.

A.10.2 Ordinary Scope

Variables defined inside a block have a scope that exists from the point of declaration forward until the end of the block. They are constructed at the point of definition, and destroyed upon exit from the block. Unlike C, variables in C++ may be defined just in time for usage. Consider (Fig. A-50):

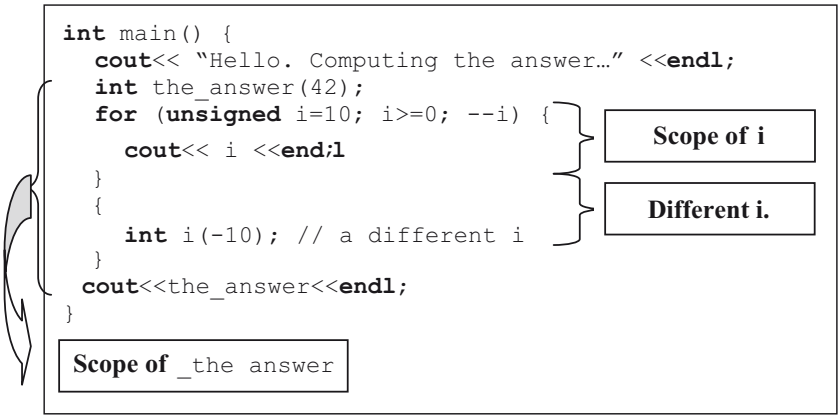


Fig. A-50. Ordinary scope

A.10.3 Defining Namespaces

Because there are many libraries with many identifiers, names can collide. To remedy this situation, C++ includes (Fig. A-51) the concept of an explicit namespace.

```
namespace your_name {
 your_code
}
```

Fig. A-51. Declaring a namespace

A namespace may be added to by simply reusing the same name. Namespaces may also be nested.

A.10.4 Using Names and Namespaces

To use a namespace, the **using** directive can be specified in one of two syntaxes (Fig. A-52):

```
using your_name::identifier; //for a single variable
using namespace your_name; // to include all names
```

Fig. A-52. Using namespaces

Some namespaces are predefined. For example, the standard C++ library contains several hundred identifiers, some of which are common words. Thus, the standard library is wrapped inside a namespace called **std** (Fig. A-53).

```
using namespace std; //should only use in .cpp files
```

Fig. A-53. Using namespace std

A.10.5 Anonymous Namespaces

Occasionally, you may need to define global objects that have a scope limited to the file in which they appear. For this situation, C++ introduces the notion of an anonymous **namespace** (Fig. A-54). Code within or following the **namespace** definitions may use the names specified.

```
void func1(void) {
   // Cannot use hidden or secret() here, because
   // they have not been defined yet
}

namespace {
  int hidden(42);
  int secret(int v) { return v+7; }
}

int func2(void) {
   // OK to use hidden and secret, since they've
   // been defined previously.
   return hidden * secret(3);
}
```

Fig. A-54. Using anonymous namespace

It would also be impossible in the preceding example to attempt to access hidden or secret in another file (e.g., via **extern** directive) since there is no way to define a reference to these names.

A.11 Exceptions

Like several other modern languages, C++ has the ability to manage exceptions. An exception is a condition that is usually considered out of the ordinary. It might represent an improper value (e.g., attempting to divide by zero). In SystemC, an exception might represent a hardware interrupt or reset situation. To handle exceptions, there are two components, which are discussed in the next sections.

A.11.1 Watching for and Catching Exceptions

The first component is the code that watches (Fig. A-55), catches, and handles the exception condition.

```
try {
  //Code to monitor for exceptions.
  //In other words, this is where the
  //exceptions will occur. It is possible,
  // they occur within calls to functions
  // several levels down.
}
catch (type1 the_exception) {
  // Handle an exception of type1
}
catch (type2 the_exception) { // As many as desired
  // Handle an exception of type1
}
catch (...) { // This is optional
  // All uncaught exceptions here
}
```

Fig. A-55. `try-catch` syntax

To be sensible, the preceding example must have at least one catch block. Notice the type parameter of the catch clause. This parameter is usually a class, and the object caught may contain additional information about the exception.

A.11.2 Throwing Exceptions

The second component to handle exceptions is the code (Fig. A-56) that throws the exception to the catcher. To communicate what the exception situation is, C++ requires that the thrown object be able to be used by the catch clause.

```
throw OBJECT;
throw; // Only used to re-throw from within catch
```

Fig. A-56. `throw` syntax

Given the preceding syntax, we can present a complete example (FIg. A-57).

```
class Error {
public:
  string message
  short  value;
  Error(string msg, short val)
  :message(msg)
  , value(val)
  {}
};

short div(short a, short b) {
short result = 0;
try {
  if (a > 150) {
     throw Error("Bad value: a=",a);
     } else if (b == 0 or b > 150) {
     throw Error("Bad value: b=",b);
     }
     result =a/b;
  }
catch (Errorwhat) {
  cout<<what.message<<what.value<<endl;
  }
catch (...) {
  cout<< "Something bad happened in div" <<endl;
  throw;
  }
return result;
}
```

Fig. A-57. Exception handling example

A.11.3 Functions that Throw

When designing a function, C++ lets you explicitly document exceptions (Fig. A-58) that your code might throw. This documentation is useful in a header file to let the user know what to expect.

```
int div(int a; short b) throw (Error);
```

Fig. A-58. Declaring exception capabilities

A.12 Standard Library Tidbits

Finally, we need to cover a few topics in the C++ Standard Library lightly, but with the hope you will go much deeper. The C++ Standard Library is separate from the language itself; however, no coder can claim to be a C++ programmer without some familiarity with this library. All members of the C++ Standard Library are part of the namespace **std**. To keep things manageable, the library is broken into smaller header files, which conventionally do not have a ".h" appended to their file name. We show the **#include** statements for these in the examples that follow.

A.12.1 Strings

The C++ Standard Library provides a string class that is far superior to the old C-style char* string concept. This class allows for safe and convenient appending, searching, and even replacement of substrings. Here (Fig. A-59) is a brief sample of things you can do:

```
#include<string>
using std::string;
string mesg("Hello SystemC!");
mesg += " I concatenate";
cout
  <<mesg<<endl
  << "length=" <<mesg.length() <<endl
  << "substr(6,6)=" <<mesg.substr(6,6) <<endl
  << "find(\"stem\")=" <<mesg.find("stem") <<endl
  << "mesg[12]=" <<mesg[12] <<endl
  ;
// Convert to C-style string for use with printf
printf("%s\n", mesg.c_str());
```

Fig. A-59. Examples of std::string

A.12.2 File I/O

We've already discussed streaming I/O; however, the C++ Standard Library provides much more in the way of classes that support this concept. For instance, you can naturally open, close, and use files with the **fstream** header. The **iomanip** header provides I/O manipulation routines. There are a lot of different types of formatting options.

The following example (Fig. A-60) shows some useful operations:

```
#include<fstream>
#include<iomanip>
#include<stdlib.h>
using namespace std;
…
// Examples of input
ifstream infile("my.txt"); //Declare & implicit open
if (!infile) { // Make sure no open errors
  cerr<< "Unable to read file my.txt!?" <<endl;
  exit(1);
}//endif
string first_line;
infile>>first_line;
cout<< "first_line='" <<first_line<< "'" <<endl;
double dave;
infile>>dave;
cout<< "dave=" <<setprecision(3) <<dave<<endl;
infile.close(); // explicit close
// Examples of output
{
  ofstream fout; // Declare - open deferred
  fout.open("save.txt"); // explicit open
  if (!fout) { // Make sure no open errors
    cerr<< "Unable to read file my.txt!?" <<endl
    exit(1);
  }//endif
  fout
  <<setw(5)      // width of output is five
  <<setfill("#") // filler character is asterisk
  <<first_line.length() // some data
  <<flush// force output buffer to write
  // notice lack of parens
  ;
} // Leaving scope destroys output variable,
  // which implicitly closes the file
```

Fig. A-60. Examples of `fstream` and `iomanip`

You are referred to the C++ library manual (or Google) to learn about more of the I/O options and manipulators.

Another example (Fig. A-61) is the string stream class that lets you treat **`std::string`** as an object for streaming I/O. The following example shows some useful string stream operations:

```
#include<sstream>
// First examine an output string stream
using namespace std;
ostringstream sout;
sout << "Use I/O to create strings" <<endl;
sout << hex << 1234 << endl;
sout <<setprecision(3) << 4.9 << endl;
// Extract the string
string mesg = sout.str();
// Now try an input string stream
mesg = "height 5.78";
istringstream sin;
sin.str(mesg);
int i;
sin>> i >> mesg;
cout << "Field:" << mesg << " Value:"<< i << endl;
```

Fig. A-61. Examples of `ostringstream` and `istringstream`

String streams support most of the operations used with standard I/O because they are in fact streams. You are referred to the documentation elsewhere for more details (e.g. try Google).

A.12.3 Standard Template Library

We couldn't leave the discussion of C++ without one last reminder that the Standard Template Library is an essential part of every C++ programmer's toolkit. You should become familiar with the basic containers **pair**<*T1,T2*>, **vector**<*T*>, **list**<*T*>, **deque**<*T*>, **map**<*T1,T2*>, and **set**<*T*>. You should learn to add, fetch, remove, and loop through these basic containers. They are really quite simple to learn, and they have a lot of uses.

It is worth noting that there are many implementations of the STL available. For best performance, you may wish to consider purchasing a commercial version.

A.13 Closing Thoughts

There is a lot more to C++. What is covered in this appendix includes the essentials needed to code effectively in SystemC.

A.14 References

Many books are written about C++, and each addresses a different audience. Some of our favorites in no particular order include the following:

The C++ ProgrammingLanguage – Special Edition, Bjarne Stroustrup

Accelerated C++, Andrew Koenig& Barbara Moo

C++ How to Program, Harvey & Paul Deitel

Thinking in C++, Bruce Eckel http://www.mindview.net/Books/TICPP/ThinkingInCPP2e.html

Exceptional C++, Herb Sutter

C++ Templates: TheComplete Guide, David Vandevoorde& Nicolai Josuttis

Index