SoC Designer
SystemC Linking Guide

Copyright © 2016 ARM Limited. All rights reserved.

Release Information

The following changes have been made to this document.

<table>
<thead>
<tr>
<th>Date</th>
<th>Issue</th>
<th>Confidentiality</th>
<th>Change</th>
</tr>
</thead>
<tbody>
<tr>
<td>November 2016</td>
<td>A</td>
<td>Non-Confidential</td>
<td>Rebrand update for 9.0.0</td>
</tr>
</tbody>
</table>

Non-Confidential Proprietary Notice

This document is protected by copyright and other related rights and the practice or implementation of the information contained in this document may be protected by one or more patents or pending patent applications. No part of this document may be reproduced in any form by any means without the express prior written permission of ARM Limited (“ARM”). No license, express or implied, by estoppel or otherwise to any intellectual property rights is granted by this document unless specifically stated.

Your access to the information in this document is conditional upon your acceptance that you will not use or permit others to use the information for the purposes of determining whether implementations infringe any patents.

THIS DOCUMENT IS PROVIDED “AS IS”. ARM PROVIDES NO REPRESENTATIONS AND NO WARRANTIES, EXPRESS, IMPLIED OR STATUTORY, INCLUDING, WITHOUT LIMITATION, THE IMPLIED WARRANTIES OF MERCHANTABILITY, SATISFACTORY QUALITY, NON-INFRINGEMENT OR FITNESS FOR A PARTICULAR PURPOSE WITH RESPECT TO THE DOCUMENT. For the avoidance of doubt, ARM makes no representation with respect to, and has undertaken no analysis to identify or understand the scope and content of, third party patents, copyrights, trade secrets, or other rights.

This document may include technical inaccuracies or typographical errors.

This document may be translated into other languages for convenience, and you agree that if there is any conflict between the English version of this document and any translation, the terms of the English version shall prevail.

TO THE EXTENT NOT PROHIBITED BY LAW, IN NO EVENT WILL ARM BE LIABLE FOR ANY DAMAGES, INCLUDING WITHOUT LIMITATION ANY DIRECT, INDIRECT, SPECIAL, INCIDENTAL, PUNITIVE, OR CONSEQUENTIAL DAMAGES, HOWEVER CAUSED AND REGARDLESS OF THE THEORY OF LIABILITY, ARISING OUT OF ANY USE OF THIS DOCUMENT, EVEN IF ARM HAS BEEN ADVISED OF THE POSSIBILITY OF SUCH DAMAGES.

This document consists solely of commercial items. You shall be responsible for ensuring that any use, duplication or disclosure of this document complies fully with any relevant export laws and regulations to assure that this document or any portion thereof is not exported, directly or indirectly, in violation of such export laws. Use of the word “partner” in reference to ARM’s customers is not intended to create or refer to any partnership relationship with any other company. ARM may make changes to this document at any time and without notice.

If any of the provisions contained in these terms conflict with any of the provisions of any signed written agreement specifically covering this document with ARM, then the signed written agreement prevails over and supersedes the conflicting provisions of these terms.
# Contents

SoC Designer SystemC Linking Guide

## Preface

### About this document

---

## Chapter 1 Introduction

1.1 Overview of importing ................................. 1-1
1.2 System requirements ................................... 1-3
1.3 Soc Designer SystemC scheduler .................... 1-3
1.4 Controlling message output ........................... 1-4

## Chapter 2 SystemC to SoC Designer Import Wizard

2.1 Introduction ............................................. 2-1
2.2 Requirements and Prerequisites ...................... 2-2
2.3 Using the SystemC Import Wizard ................... 2-4

## Chapter 3 Direct Import of SystemC Models

3.1 Overview of direct import ............................ 3-1
3.2 Organizing the source files for SoC Designer ...... 3-3
3.3 Using the `sc_mx_import_module` ..................... 3-5
3.4 Using SystemC ports ................................... 3-10
3.5 Simulation control ...................................... 3-26
3.6 fifo example ........................................... 3-29
3.7 dpipe example .......................................... 3-37
Chapter 4 Generating a Wrapper for SystemC Models
4.1 Generating CASI wrapper components with the Component Wizard .......... 4-2
4.2 Instantiating SystemC modules ................................................................ 4-3
4.3 Clocking generated components .................................................................. 4-5
4.4 Connecting imported components to SoC Designer components .............. 4-6
4.5 The scport_mxsignal example ................................................................. 4-8
4.6 TLM master to SoC Designer AHB slave ............................................. 4-12

Chapter 5 Modeling Guidelines for SystemC
5.1 Modeling for Speed ................................................................................... 6-1

Appendix A SystemC Implementation
A.1 Cycle Model-specific SystemC Implementation Differences .................. A-2
A.2 Built-in primitive channels ....................................................................... A-3
Preface

This preface introduces the *SoC Designer SystemC Linking Guide*. It contains the following sections:

- About this document

About this document

This book describes how to modify SystemC models for use in SoC Designer.

Intended audience

This book is written for anybody who mixes SystemC models with SoC Designer. It assumes you have experience in creating and using SystemC models.

Organization

This book is organized into the following chapters:

Chapter 1 *Introduction*

Read this chapter for an overview of importing SystemC models.
Chapter 2 SystemC to SoC Designer Import Wizard

Read this chapter to learn how to generate SoC Designer components directly from SystemC modules using the SystemC to SoC Designer Import Wizard.

Chapter 3 Direct Import of SystemC Models

Read this chapter for an overview of direct import of SystemC models.

Chapter 4 Generating a Wrapper for SystemC Models

Read this chapter for an overview of using a wrapper generated by the Component Wizard to import SystemC models.

Chapter 5 Modeling Guidelines for SystemC

Read this chapter to improve the simulation performance of SystemC models.

Appendix A SystemC Implementation

This appendix lists the SystemC objects and primitive channels that are implemented in SoC Designer.

Typographical conventions

The following typographical conventions are used in this book:

*italic* Highlights important notes, introduces special terminology, denotes internal cross-references, and citations.

*bold* Highlights interface elements, such as menu names. Denotes processor signal names. Also used for terms in descriptive lists, where appropriate.

*monospace* Denotes text that can be entered at the keyboard, such as commands, file and program names, and source code.

*monospace* *italic* Denotes arguments to commands and functions where the argument is to be replaced by a specific value.

< and > Enclose replaceable terms for assembler syntax where they appear in code or code fragments. For example:
Further reading

This section lists related publications.

**Cycle Model publications**

The following publications provide information relate directly to SoC Designer:

- *SoC Designer User Guide*
- *SoC Designer Tools API Reference Manual*
- *SoC Designer Installation Guide*
- *ESL API Developer's Guide*
- *SoC Designer SystemC Linking Guide*
- *MxScript Reference Manual*
- *SoC Designer CDP HDL Cosimulation Guide*
- *SoC Designer Standard Model Library Reference Manual*

**External publications**

The following publications provide reference information about ARM® or AMBA®-related architecture:

- *AMBA Specification*
- *AMBA AHB Transaction Level Modeling Specification*
- *AMBA AXI Transaction Level Modeling Specification*
- *ARM Architecture Reference Manual*

See [http://infocenter.arm.com/help/index.jsp](http://infocenter.arm.com/help/index.jsp) for access to ARM documentation.

The following publications provide additional information on simulation:

Chapter 1
Introduction

This chapter introduces the support of SystemC models in SoC Designer. It contains the following sections:

• Overview of importing
• System requirements on page 1-3
• Soc Designer SystemC scheduler on page 1-3
• Controlling message output on page 1-4

1.1 Overview of importing

SoC Designer is based on the SystemC language and enables the import and simulation of any IEEE 1666-2011 (Accellera)-compliant SystemC Transaction-Level Models (TLM).

SoC Designer is a system-level simulation tool based on the SystemC version 2.3.1 language.

New SoC Designer components can be created using the SoC Designer Component Wizard.
For legacy SystemC TLM model reuse, existing SystemC models can be imported into SoC Designer with little or no modifications to the original source code.

SystemC import is an important feature of SoC Designer and it:

- enables legacy SystemC TLM reuse
- enables legacy SystemC signal-level model reuse
- supports SystemC event-driven features

**Note**

Cycle-based SoC Designer models do not require the event-driven scheduler. The event-driven scheduler is only enabled when models utilizing event-driven SystemC features (for example, threads or methods with a sensitivity list) are present in the system.

The SystemC kernel is fully integrated into SoC Designer. All SystemC constructs and data types are supported and any SystemC module can be imported into SoC Designer and simulated and debugged just like any of the native SoC Designer cycle-based components.

There are three different methods for importing SystemC models into the SoC Designer environment:

**SystemC Import Wizard**

Use the SystemC to SoC Designer Import Wizard as the fastest and easiest import method. This graphical wizard prompts for the name of the top-level design file, or a single SystemC module file, and then automatically creates the SoC Designer component or components. The SystemC Import Wizard is described in Chapter 2 SystemC to SoC Designer Import Wizard.

**Change inheritance**

Change the module class inheritance so that the user module inherits from a special SoC Designer base class `sc_mx_import_module` (instead of `sc_module`) that provides the default implementations of additional methods required for a SoC Designer Model. This direct import mechanism is described in Chapter 3 Direct Import of SystemC Models.

**Use a wrapper**

Use a SoC Designer wrapper to instantiate the SystemC user modules as sub-components. The code for the SoC Designer wrapper component instantiates the SystemC modules within the component class and
interconnects the internal SystemC ports with the external SoC Designer ports. This wrapper based import mechanism is described in Chapter 4 Generating a Wrapper for SystemC Models.

SoC Designer includes SystemC import examples in the 
MAXSIM_HOME/Examples/SystemCImport directory. The TLM subdirectory contains import examples for OSCI TLM modules.

### 1.2 System requirements

Refer to the SoC Designer Installation Guide for supported platforms and compilers.

<table>
<thead>
<tr>
<th>Caution</th>
</tr>
</thead>
<tbody>
<tr>
<td>On Windows platforms, all SystemC components must be built with the /vmg compiler flag set to ensure proper virtual function search order. (See the C/C++ command line options in the project Property Pages.)</td>
</tr>
</tbody>
</table>

MSVC++ project files generated from the component wizard already contain the /vmg flag. If you create new files manually, you must add the flag before building the imported SystemC component on Windows. Linux platforms are not affected. Models built without the /vmg compiler option might generate error messages when loaded into SoC Designer.

### 1.3 Soc Designer SystemC scheduler

The SoC Designer SystemC scheduler kernel operates in one of the following two modes:

#### pure cycle-based scheduler

The pure cycle-based scheduler mode is used by default when SoC Designer recognizes that none of the components in a given design is using even-driven features such as threads or methods sensitized on an event. This cycle-based scheduler can be used for optimizing simulation performance (see Chapter 5 Modeling Guidelines for SystemC).

SystemC callbacks (end_of_elaboration for example) are disabled when the scheduler chooses to operate in the cycle-based mode.

#### hybrid (cycle-based + event-driven) scheduler

The hybrid scheduler mode (cycle-based scheduler and event-driven scheduler are merged into a single scheduler kernel) is used when SoC Designer detects a component utilizing any event-driven features.
You can force SoC Designer to use the hybrid scheduler. This can be useful if you import legacy SystemC code into SoC Designer and you require the functionality that was coded in the SystemC callbacks. To force SoC Designer into the hybrid scheduler mode, invoke the `sc_mx_set_need_event_driven()` function from the component’s `init()` function with the parameter set to `true`:

```cpp
sc_get_curr_simcontext()->sc_mx_set_need_event_driven(true);
```

The `sc_mx_set_need_event_driven()` function is a member of the `sc_simcontext` class:

```cpp
void sc_simcontext::sc_mx_set_need_event_driven(bool v)
```

### 1.4 Controlling message output

Legacy SystemC code typically uses `cout()` to print debug messages out onto the console window.

`cout()` is preferred over `printf()` since all SystemC build-in types define overloaded insertion operators. This eliminates the requirement for specific output formatting required by `printf()`.

The `message()` function in SoC Designer enables better output control onto Simulator output windows. Use the `mxcout` object and the `dumpMsg()` function to use overloaded insertion operators for SystemC types and determine when the message is output. `mxcout` is similar to `cout()`, except that it buffers the output until `dumpMsg()` is called:

```cpp
void dumpMsg( eslapi::CASIMessageType msgtype = eslapi::CASI_MSG_INFO )
```

Example 1-1 shows how to use `dumpMsg()`:

```
Example 1-1 Using dumpMsg()

```cpp
void my_systemc_module::my_method()
{
    mxcout << "in my_method at time " << sc_time_stamp() << endl;
    // print the above message as a SoC Designer WARNING message
    mxcout.dumpMsg( eslapi::CASI_MSG_WARNING );
}
```

See the SystemC Import examples for more examples of code that uses `mxcout`. 
Chapter 2
SystemC to SoC Designer Import Wizard

This chapter describes how to use the SystemC to SoC Designer Import Wizard to generate SoC Designer components from existing SystemC modules. It contains the following sections:

• Introduction
• Requirements and Prerequisites on page 2-3
• Using the SystemC Import Wizard on page 2-4
2.1 Introduction

The SystemC Import Wizard is a graphical tool that analyzes the original SystemC module definitions and then generates the corresponding SoC Designer components. The original SystemC module is not affected by this process.

In addition to making the SoC Designer component, the tool also enables you to create component parameters for the component so you can make some configuration changes to the component during simulation.

Figure 2-1 shows a sample system with three SystemC modules and how they are generated into SoC Designer components that can be brought into SoC Designer.
2.2 Requirements and Prerequisites

Your SystemC modules must be organized in a consistent manner for the SystemC to SoC Designer Import Wizard to work correctly. These requirements are listed below:

- The class definition must be “includable” either by placing the class in a header file, or by using a header guard (ifdef), such as that used by Denali memories.

- SoC Designer 9.0.0 or greater must be installed with all of the required tools as specified in the *SoC Designer Installation Guide* and *SoC Designer User Guide*.

2.2.1 Files Generated by the SystemC Import Wizard

The SystemC Import Wizard generates files under `<output directory>/ARM` directory (you can set `output directory` from the wizard). Under the `ARM` directory, the structure looks as follows

`<output directory>/ARM:

- Imported model A/:
  - CMakeLists.txt
  - Imported Model A_wrapper.[cpp|h|def]
  - Imported Model A.maxlib.conf

- Imported Model B/:
  - CMakeLists.txt
  - [Debug|Release]Build_linux.sh
  - [Debug|Release]Build_W2005.sh
  - maxlib.conf
  - impconf.build.xml

Each imported SystemC module gets a subdirectory under `<output directory>/ARM`. These directories hold the SoC Designer component wrapper files and the maxlib configuration file for that imported model. Top level `maxlib.conf` includes the `maxlib.conf` files of each of the imported models for convenience (just register the top level `maxlib.conf` to get all of the models registered with SoC Designer). *cmake* (a cross-platform build system) is used for building the models, and *CMakeLists.txt* files are configuration files used by *cmake*. All user settings are saved in `impconf.build.xml` file. This file may be re-loaded into the wizard to make it convenient to make configuration changes without having to re-enter all of the configuration information. There should be no need to manually edit any of the generated files.

Additional files may be generated; these are only used internally by the tool.
2.3 Using the SystemC Import Wizard

To Launch the wizard.

For Linux, use:

$MAXSIM_HOME/bin/sys2socd

For Windows, use:

Start-> All Programs-> ARM > SoC Designer-> SoC Designer SystemC Import Wizard

Or

%MAXSIM_HOME%\Bin\Release\SC2SoCDImportWizard.bat

Figure 2-2 Import Configuration page
### Table 2-1 Import Configuration Fields

<table>
<thead>
<tr>
<th>Field</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Output Directory</td>
<td>Directory where the wizard output files will be generated. The subdirectory <em>/ARM</em> is appended to the specified directory.</td>
</tr>
<tr>
<td>Import File(s)</td>
<td>File(s) that identify your SystemC modules. An example is a cpp source file that instantiates the modules you would like to import; or, you can supply the header files for the modules.</td>
</tr>
<tr>
<td>Include Paths</td>
<td>Provide the paths that contain the header files referenced from the Import File(s).</td>
</tr>
<tr>
<td>AMBA TLM2 Modules</td>
<td>Check this box if you are importing AMBA TLM2 models. This will automatically add the include and library paths for amba_socket.</td>
</tr>
<tr>
<td>Advanced Options</td>
<td>The options specified below are available when Advanced Options box is checked.</td>
</tr>
<tr>
<td>Definitions</td>
<td>Provide any definitions, or compile-time macros, that should be passed to the pre-processor as –D &lt;value&gt;.</td>
</tr>
</tbody>
</table>
| Templated Modules      | This is applicable when your Import File(s) contain modules that take template argument(s), and no instantiation of the templated module is found in the design (e.g., you only provided the header files for the template module). The pre-processor used by the wizard does not analyze template modules unless an explicit instantiation of that module is found. Therefore, it is required that you provide the template argument value in this field. The import wizard actually maps these template parameters to CASI component parameters. The value entered in this configuration step is used as the default parameter value. An example is a module, my_comp, that takes <unsigned int BUSWIDTH> as a template argument. To import this module the user needs to provide a default value for the template argument:
my_comp<32> |
| Load Import Config File| Once a component is imported, all of the import configuration settings are saved in the ARM/impconf.build.xml file. This file can be re-loaded into the wizard to re-generate the component with different configuration settings.                |
| Inheritance Search Depth| The wizard, by default, analyzes up to 2 levels of class inheritance levels when looking for SystemC modules, ports and TLM2.0 sockets. This search depth can be changed by using the dial provided. Note that longer depth will likely result in longer time for the wizard to analyze the design. |
Click Next once you have filled in the necessary information. The wizard starts analyzing the design, and the output is given on the next screen. Figure 2-3 on page 2-6 shows results once the analyzer goes through successfully.

If the wizard finds any errors during the analysis, it will display the error and open the source code where the error is detected from and highlight the line.
Figure 2-4 Viewing Errors

The above shows an error where an include file cannot be found. At this point, you can either go back and fix the error (add to Include Paths), or view other errors by clicking on the error messages in the bottom half of the screen.

Once the design has been successfully analyzed, hit Next to continue.

In the next step, the wizard displays all of the sc_modules that were found. Note that you can choose to leave any of the modules out by un-checking the box next to the module. Also, you can give a different name for the module by editing the component name as shown below.
### Figure 2-5 Renaming the component

#### Table 2-2 Component Configuration

<table>
<thead>
<tr>
<th>Field</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Source Files</td>
<td>Select the source files that need to be compiled for the component. Use Check All or Uncheck All to quickly select or de-select components to import.</td>
</tr>
<tr>
<td>Build Settings</td>
<td>Compiler and Linker flags. Note that SoC Designer specific flags are pre-filled.</td>
</tr>
<tr>
<td>Ports</td>
<td>Displays the ports that will be registered with SoCDesigner. When the “Clk?” option is checked, the model’s clock input port will not be registered as a port, but it will directly be hooked up to the SoC Designer system clock.</td>
</tr>
<tr>
<td>Constructor Args</td>
<td>This tab lists the constructor arguments for your component (if any). For convenience, the wizard will turn constructor arguments into SoC Designer component parameters.</td>
</tr>
</tbody>
</table>
Hit Next once you have finished configuring the component generation. This will start the build process in the next page.

### Table 2-2 Component Configuration (continued)

<table>
<thead>
<tr>
<th>Field</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Parameters</td>
<td>You can add custom parameters for the generated model. Filling in the parameter information in the wizard will generate a .cpp file that you can use to handle component parameters. The .cpp file implements callback methods which are called from the component whenever parameter values are set. In the generated file, look for “// Add code”. This is where you can add custom code that depends on the parameter value. For templated modules, the wizard will pre-fill the parameter configuration table with each entry mapping to the template arguments. In the “Allowed Values” column, the user may enter the allowed values, separating each with a comma.</td>
</tr>
<tr>
<td>Defines</td>
<td>You can add any custom definitions in this tab.</td>
</tr>
</tbody>
</table>
Click **Finish** when the build has finished successfully.

Note: If you make any changes and need to rebuild the component(s), you can use the scripts in the ARM/output directory (Debug/ReleaseBuild_Linux.sh or Debug/ReleaseBuild_W2005.bat) to build them again.
Chapter 3
Direct Import of SystemC Models

This chapter describes how to modify the source code for a SystemC object to import it into SoC Designer. It contains the following sections:

- Overview of direct import
- Organizing the source files for SoC Designer on page 3-3
- Using the sc_mx_import_module on page 3-6
- Using SystemC ports on page 3-12
- Simulation control on page 3-28
- fifo example on page 3-30
- dpipe example on page 3-39
3.1 Overview of direct import

To import SystemC models into SoC Designer:

1. Change the module class inheritance from `sc_module` to `sc_mx_import_module`.
   This requirement is specific to SystemC module import. The `sc_mx_import_module` class implements the required virtual methods of SoC Designer component base class. (The component base class inherits from `sc_module`, the SystemC module base class.)
   `sc_mx_import_module` establishes the base for building SoC Designer components.

2. Provide a C-style entry point into the module library.
   This requirement is common to all SoC Designer components. All models must be built into a DLL (or shared object) and must provide a common entry point. The entry point for any SoC Designer shared object is the `MxInit()` function. This function is declared as `extern "C"` to facilitate linking.

   **Example 3-1 Entry point definition**

   ```c
   extern "C" void MxInit() {
       new MyModelFactory();
   }
   ```
   The `MxInit()` function must create an instance of the component factory object for the components in the shared library. The factory then registers itself with SoC Designer.

3. Define a factory class to instantiate the module. The factory class must inherit from `CASIFactory`. The only functions that must be defined are:
   - the constructor, this is named `MyModelFactory()` for the class in Example 3-2
   - the `createInstance()` function

   **Example 3-2 Factory class**

   ```c
   class MyModelFactory : public CASIFactory
   {
       public:
       
       MyModelFactory() : CASIFactory("MyModel"){}
       CASIModuleIF *createInstance(CASIModuleIF *c, const string &id);
       {
       eslapi::setDoNotRegisterInSystemC(false);
       eslapi::sc_mx_import_delete_new_module_name();
       return new MyModel(c, id.c_str());
       }
   }
   ```
3 Direct Import of SystemC Models

——— Note ————

The Component Wizard does not create a factory class for direct import. The factory class is not required if you are using the models in a pure SystemC environment.

APIs are available for registering existing SystemC ports in SoC Designer. See the ESL API Developer’s Guide for more information on creating components.

Other virtual methods of a SoC Designer component class can be overloaded in the module to implement specific features of SoC Designer, but these are for enhancements and are not required for importing.

3.2 Organizing the source files for SoC Designer

Figure 3-1 on page 3-4 shows how the original dpipe example appears in SoC Designer Simulator after the files have been converted to libraries and a top-level .mxs project has been created:
Figure 3-1 dpipe example in SoC Designer

Note

Figure 3-1 also shows a Register window open. Displaying the content of registers requires that the dpipe component implements a CADI interface for the registers.

A conventional SystemC model typically has almost all of the code (except for the include files) in a single source file. For SoC Designer, however, there are typically multiple source files where each file provides a specific functionality.

An overview of source files for the SoC Designer version of the dpipe system is shown in Figure 3-2 on page 3-5:
Figure 3-2 Files used in the SoC Designer dpipe system
3.3 Using the sc_mx_import_module

All imported SystemC TLM modules must inherit from the sc_mx_import_module base class. Default behaviors of SoC Designer Model interfaces are implemented in this base class. All derived classes are therefore valid in the SoC Designer Model.

The first step in importing a SystemC model into SoC Designer is to change the module class inheritance to inherit from sc_mx_import_module instead of sc_module. This hierarchy is shown in Figure 3-3 on page 3-7.

3.3.1 sc_mx_import_module class implementation

The location of the sc_mx_import_module class in the hierarchy is shown in Figure 3-3.
Figure 3-3 Class hierarchy for SystemC import

The sc_mx_import_module class header is listed in Example 3-3 on page 3-8. This header file is located in MAXSIM_HOME/include/sc_mx_import_module.h. This file should be included from the header file in which the user module class is defined.
Example 3-3 sc_mx_import_module class header

class WEXP sc_mx_import_module : public CASIModule
{
public:
    // constructor / destructor
    // note change in second constructor argument to type sc_module_name
    sc_mx_import_module(CASIModuleIF* c, const ::sc_core::sc_module_name &s, string name);
    virtual ~sc_mx_import_module();

public:
    const char* kind() {return "sc_mx_import_module";}
    // overloaded CASIModule methods
    string getName();
    void setName(string& strName);
    void setParameter(const string &name, const string &value);
    string getProperty(eslapi::CASIPropertyType property);
    void init();
    void terminate();
    void reset(eslapi::CASIResetLevel level, const CASIFileMapIF* filelist);
    ::sc_core::sc_port_base* getSCPortInfo(string portName_);

protected:
    // methods for registering SystemC ports with SoC Designer
    void registerSCGenericMasterPort(::sc_core::sc_port_base* masterport_, string portName_);
    void registerSCGenericSlavePort(::sc_core::sc_interface* slaveport_, string portName_);
    void registerSCGenericSlavePort(::sc_core::sc_export_base* slaveport_, string portName_);

private:
    string sc_mx_import_module_name;
    vector<sc_mx_import_port*> sc_mx_import_ports;
}

Default implementations of CASIModule methods are implemented in this base class, so it is not required to add implementations to any of the methods declared above. See the SoC Designer User Guide for more information on the APIs.

ARM recommends, however, that you override some of the methods to take advantage of various SoC Designer model features such as:

- adding a configuration parameter using the defineParameter and setParameter methods (the parameter is visible in the SoC Designer Parameter window)
- enabling a simulation reset feature by overloading the reset() method with appropriate module reset code.
### Note

`sc_mx_import_module` is a class, and not a struct like `sc_module`. You must explicitly declare public access specifiers for members that are to be public.

### 3.3.2 Convenience macros in `sc_mx_import_module.h`

`sc_mx_import_module.h` provides the following convenience macros:

**SC_MX_IMPORT_MODULE**

`SC_MX_IMPORT_MODULE` can substitute for the `SC_MODULE` macro defined in the SystemC language.

```c
#define SC_MX_IMPORT_MODULE(systemcMod) class systemcMod : \
    public sc_mx_import_module
```

**SC_MX_IMPORT_CTOR**

`SC_MX_IMPORT_CTOR` can substitute for the `SCCTOR` macro defined in the SystemC language.

```c
#define SC_MX_IMPORT_CTOR(systemcMod) \
    systemcMod(CASIModuleIF *c, const sc_module_name &id) : \
    sc_mx_import_module(c, id, #systemcMod)
```

### Note

`SC_MX_IMPORT_CTOR` does not include `SC_HAS_PROCESS(module_name)` as in `SCCTOR` SystemC macro. Explicitly declare `SC_HAS_PROCESS(…)` for modules that contain processes.

**SC_MX_FACTORY**

`SC_MX_FACTORY` macro can be used to define the SoC Designer Model factory class and provide an entry point into the DLL from SoC Designer.

If the module uses templates, or the module constructor requires additional arguments, use the expanded form of the macros to define the template data types and pass in additional parameters for the constructor as shown in Example 3-4.
Example 3-4 SC_MX_FACTORY macro

```c
#define SC_MX_FACTORY(systemcMod) class systemcMod##Factory : \
public MxFacory \
{ \
public: \
    systemcMod##Factory() : CASIFactory ( #systemcMod ) {} \
    CASIModuleIF *createInstance(CASIModuleIF *c, \
    const string &id) \
    { \
        eslapi::setDoNotRegisterInSystemC( false ); \
        eslapi::sc_mx_module_name(); \n        return new systemcMod(c, id.c_str()); \n    } \n}; \
extern "C" void \
MxInit(void) \
{ \n    new systemcMod##Factory(); \n} \
extern "C" void \
MxInit_SCImport(void) \
{ \n} \
```

——— Note ————

If you use the expanded form of SC_MX_FACTORY, you must define the function MxInit_SCImport() explicitly.

All SystemC modules must have the functions MxInit_SCImport() and MxInit() defined.

MxInit_SCImport() is used internally by SoC Designer to identify the SystemC modules and open the object files for the module with RTLD_GLOBAL. If you fail to define the MxInit_SCImport() function for a SystemC component, the missing definition causes gcc dynamic cast problems on Linux platforms. These dynamic cast problems cause SystemC port binding errors.
3.3.3 Generating a makefile from the Component Wizard

After the required modifications have been made to the source code, the module must be built into a single DLL before it can be used in SoC Designer. The Component Wizard provides an option to automatically generate Makefile and VC++ project files for this purpose:

1. Select **Component Wizard** from the **Tools** menu to launch the Component Wizard.

2. Select the **Project Files for SystemC Import** radio button on the Component Wizard dialog as shown in Figure 3-4 on page 3-11.

![Figure 3-4 Component Wizard for SystemC project file generation](image)

3. Enter the desired model name and the path location for the Makefile.

4. The next screen shows a summary of the files to be generated and prompt for confirmation.

5. A **makefile** (for Linux platforms) and **.vcproj** and **.def** files (for Windows) is generated into the user specified directory.

| Caution |

The project files only contains the necessary compiler and linker options appropriate for a SoC Designer Model. They do not contain the source files to be compiled or any external libraries that can be linked. You must add such information to the generated makefile or vcproj file.
3.4 Using SystemC ports

This section describes how to use SystemC ports in SoC Designer.

3.4.1 Registering SystemC ports and interfaces

Any SystemC ports can be registered with SoC Designer to enable graphical representation of the ports and interfaces and to enable port connections using SoC Designer Canvas (see Figure 3-5 on page 3-13).

The following methods are defined in sc_mx_import_module class to register distinct SystemC port types:

```c
void registerSCGenericMasterPort(sc_port_base* masterport_, string portName_)
```

Use this method to register any sc_port. masterport_ is the pointer to the sc_port and portName_ specifies the name for this port (the name is displayed as the port name in Canvas).

All sc_port or SCGenericMasterPort instances are considered transaction initiators or masters. These ports are represented in Canvas with a semi-circle pointing outwards from the module.

```c
void registerSCGenericSlavePort(sc_interface* slaveport_, string portName_);
```

Use this method to register an sc_interface. Interfaces define the access methods initiated by masters and are considered slaves to sc_port or masters that make use of the interface.

SCGenericSlave ports are represented by a semi-circle that points inwards towards the channel (see Figure 3-5 on page 3-13).

```c
void registerSCGenericSlavePort(sc_export_base* slaveport_, string portName_);
```

Use this method for registering an sc_export port.

These ports are extensions of interfaces and are represented by a semi-circle that points inwards towards the channel.
The only valid connection routes are from a SCGenericMasterPort to a SCGenericSlavePort. Canvas prohibits any other type of SystemC port connections. A SCGenericMasterPort to SCGenericSlavePort connection might still be invalid if incompatible interfaces are used. Interface incompatibility checking is performed during port binding and triggers an error if incompatible connections are present in the user system.

These port registration calls must be made in the SoC Designer component constructor. However, it is possible that the port instantiation is not made until later. In such cases, it is valid to call registerSCGenericMaster/SlavePort() with the first argument set to NULL. This will register the port type so that only the port appears in the SoC Designer Canvas. When doing this, another registerSCGenericMaster/SlavePort() must be called in the init phase with the first argument set to valid data in order to complete the port registration.
3.4.2 Using the sc_prim_channel model library

A library of SystemC built-in primitive channels are provided in SoC Designer. For each of the built-in sc_prim_channel types, an equivalent model library is available in SoC Designer.

For primitive channels that use template data types such as sc_signal and sc_fifo, a selection of such channel components are built into SoC Designer primitive channel library to cover a wide range of commonly used data types. These built-in primitive channel components can be loaded in SoC Designer Canvas through the SoC Designer Preferences dialog (select Manage Model Library from the Tools menu):

1. In the SoC Designer Preferences window, click on the radio button labeled Include SystemC primitive channel components in the Model Library Repository (this is located in the Components group box near the bottom of the dialog).

2. Click OK and Save to load all of the built-in primitive channels and file them under the PrimChnl tab in the component window. See Figure 3-6.

![Figure 3-6 List of primitive channel components](image)

These SoC Designer primitive channels inherit from the base class sc_mx_import_prim_channel which by default sets the component type as SystemC-PrimChannel. A special property is set for the derived modules that gives them unique appearance in SoC Designer Canvas. A few of the built-in primitive channel components are shown in Figure 3-7.

![Figure 3-7 Primitive channel displayed in the Canvas Diagram window](image)
Note

The channels expose their interfaces through SCGenericSlavePorts.

sc_buffer, sc_fifo and sc_signal channels register two slave ports for user convenience. These typically connect multiple modules, but the two slave ports represent the same interface and can be used interchangeably. Multiple master ports can connect to a single slave port (for example, multiple masters accessing a sc_mutex).

The primitive channel components can be resized, renamed, and the ports re-positioned using Canvas editing tools just like any other SoC Designer components.

All available built-in primitive channel components are listed in Built-in primitive channels on page A-3.

Using the Component Wizard to create a custom primitive channel

Although a wide range of primitive channels with built-in data types are already available as built-in components of SoC Designer, there might be a requirement for you to create your own primitive channel with a different data type (using a user-defined data type for example). These custom primitive channels can be automatically generated using the SoC Designer Component Wizard.

To create a new primitive channel:

1. Select Component Wizard from the Canvas Tools menu.
   Enter the new component name, the path to put the generated files into, and select SystemC Primitive Channel as the component type as shown in Figure 3-8 on page 3-16.
2. Select the channel type and declare the data type for that channel as shown in Figure 3-9. The \textit{sc\_lv Width} field is enabled when the \textit{Channel Type} is \textit{“sc\_signal\_rv”}. If you are defining this type of channel, enter the width of the signal in this text box.

\begin{figure}[h]
\centering
\includegraphics[width=0.8\textwidth]{figure3-9.png}
\caption{Specify channel and data type}
\end{figure}

\textbf{Note}

Because the type can be any user-defined data type, data type validity checking is not done in the Wizard.

Click \textbf{Next} to accept the channel information and proceed to the next step.
3. Check the boxes if you require a CADI interface, registers, or memory for the channel.

![Figure 3-10 Add CADI registers and Memory regions](image)

Two registers are supplied by default. Select a register and click **Edit** to change its properties or click **Delete** to delete the register.

Click **Add** if you require additional registers for the component.

4. Click **Next>** to accept the CADI interfaces and proceed to the summary.

5. Figure 3-11 shows the summary. Click **Finish** to generate the sources for the new primitive channel component.

![Figure 3-11 Generate the component source files](image)
6. SoC Designer generates the source files for the new component. After the files are generated, the prompt in Figure 3-12 is displayed:

![Figure 3-12 Compile component](image)

--- Caution ---

If a user-defined data type (anything other than C/C++ or SystemC built-in types) is being used, **Compile Component and Add** must not be used.

You must select **Do Not Compile/Add Component** and add the definition of the data type into the generated files before building the component.

---

7. Click **OK** to go to the next step.

8. If you selected **Compile Component and Add**, you are prompted for the model library .conf file that the component is added to (see Figure 3-13). The newly generated channel will by default report **SystemC-PrimChannel** as the component type is placed into the **PrimChnl** group in the component window.

![Figure 3-13 Select repository for component](image)
If you selected **Do Not Compile/Add Component**, modify the created files (see Figure 3-14) and use the Manage Library section of the Preferences dialog to add the component type to the component window.

![Figure 3-14 Generated component](image)

9. Click **OK** to close the Component Wizard.

### 3.4.3 External SystemC ports

As with normal SoC Designer Models, you can create hierarchical systems with imported SystemC models. If you are exporting a SystemC port for connection into the upper hierarchy, select **Add External Port** from the **Insert** menu or click on the **Port** icon. Use the **SCGeneric** option in the External Port configuration window as shown in Figure 3-15 on page 3-20.
The *sc_export_multi* example supplied with SoC Designer shows a multi-hierarchy system that uses a SystemC external port. See Figure 3-16 on page 3-21.

![SCGeneric External Port dialog](image)
Select Open Sub-System… from the context menu to display the subsystem that contains an external port. See Figure 3-17.
3.4.4 Mixing SoC Designer and SystemC ports

SoC Designer can use the `scsig2sdsig` and `sdsig2scsig` components to enable communication between SystemC and SoC Designer components as described in:

- *SystemC signal output to SoC Designer signal slave* on page 3-22
- *SoC Designer signal master to SystemC signal input* on page 3-26

Connecting SystemC buses and SoC Designer buses is more complex and requires modifying the source code generated components as described in *TLM master to SoC Designer AHB slave* on page 4-13.

**SystemC signal output to SoC Designer signal slave**

The `scsig2sdsig` example uses a component that interfaces between SystemC signals and SoC Designer signals as shown in Figure 3-18 on page 3-23:

- `testdriver_sc2sd` is a SystemC component that outputs Boolean signals from the `scout` port
- `sig<bool>` is a primitive channel that connects the `scout` and `sc2sc` ports
- `scsig2sdsig` has a SystemC port (`scout`) and a standard SoC Designer master port (`sc2sd`)
- `mxstub` has a standard SoC Designer slave port (`p_in0`).

——— Note ————

Because the connection between `scsig2sdsig` and `mxs_out` is between SoC Designer master and slave ports, a monitor can be attached to the port connection.
The `scsig2sdsig.h` file contains the component class definition as shown in Example 3-5:

```
#include "sc_mx_import_module.h"
class scsig2sdsig : public sc_mx_import_module
{
public:
    SC_HAS_PROCESS(scsig2sdsig);
    scsig2sdsig(CASIModuleIF* c, const sc_module_name &s);
    virtual ~scsig2sdsig();
    void initSignalPort(CASISignalMasterIF* signalIf);
    void transferSignal();
private:
```

Figure 3-18 SystemC signal to SoC Designer signal component
The `testdriver_sc2sd.h` file has the definition for the testdriver component as shown in Example 3-6:

**Example 3-6 testdriver component**

```cpp
#include "sc_mx_import_module.h"
SC_MX_IMPORT_MODULE(testdriver_sc2sd) {
public:
    SC_HAS_PROCESS(testdriver_sc2sd);
    testdriver_sc2sd(CASIModuleIF* c, const sc_module_name &s);
    void init();
    void driveOut();
private:
    bool tempVal;
    sc_out<bool> scout;
    sc_in_clk m_clk;
};
```

The `scsig2sdsig.cpp` file includes code to register and initialize the SystemC port as shown in Example 3-7:

**Example 3-7 Registering the SystemC port**

```cpp
#include "scsig2sdsig.h"
#include <stdio.h>

scsig2sdsig::scsig2sdsig(CASIModuleIF* c, const sc_module_name &s) :
    sc_mx_import_module(c, s, "scsig2sdsig")
{
    sc2sd_SMaster = new sc_port<CASISignalIF>(this, "sc2sd");
    initSignalPort((CASISignalMasterIF*)sc2sd_SMaster);
    registerPort(sc2sd_SMaster, "sc2sd");
    registerSCGenericMasterPort(&sc2sc_SSlave, "sc2sc");

    SC_METHOD(transferSignal);
    sensitive << sc2sc_SSlave;
}
void scsig2sdsig::transferSignal()
```
The `testdriver_sc2sd.cpp` file contains code to write to the port as shown in Example 3-6 on page 3-24:

```
#include "testdriver_sc2sd.h"
testdriver_sc2sd::testdriver_sc2sd(CASIModuleIF* c, const sc_module_name &s) :
    sc_mx_import_module(c, s, "testdriver_sc2sd"), tempVal(false)
{  
    SC_THREAD(driveOut);
    sensitive << m_clk.pos();
    registerSCGenericMasterPort(&scout, "scout");
}
void testdriver_sc2sd::init()
{  
    m_clk(*(getMxSCClock()));
}

// drive out tempVal every other cycle
void testdriver_sc2sd::driveOut()
{  
    for(;;) {
        message(MX_MSG_INFOR, "Write data", 0);
        scout.write(tempVal);
        wait();
        tempVal = !tempVal;
        wait();
    }
}
SC_MX_FACTORY(testdriver_sc2sd)
```
Note

The two SystemC ports (\texttt{scout} in testdriver and \texttt{sc2sc} in \texttt{sdsig2scsig}) are output ports. Two SystemC output ports can function as a master/slave interface if a channel is connected between them.

Because the two ports use Boolean signals, the primitive channel used is \texttt{sig<bool>}.

SoC Designer signal master to SystemC signal input

The \texttt{sdsig2scsig} example uses a component that interfaces between SystemC signals and SoC Designer signals as shown in Figure 3-19 on page 3-27:

- \texttt{testdriver} is a SystemC component that receives integer signals from the \texttt{scout} port
- \texttt{sig<int>} is a primitive channel that connects the \texttt{scout} and \texttt{scin} ports
- \texttt{sdsig2scsig} has a SystemC port (\texttt{scout}) and a standard SoC Designer slave port (\texttt{sd2sc})
- \texttt{mxstub} has a standard SoC Designer master port (\texttt{p_out0})

Note

Because the connection between \texttt{sdsig2scsig} and \texttt{p_out0} is between SoC Designer master and slave ports, a monitor can be attached to the port connection.

The \texttt{sdsig2scsig.h} file contains the component class definition as shown in Example 3-9:

Example 3-9 sdsig2scsig.h file

```c++
#include "sc_mx_import_module.h"
class sdsig2scsig : public sc_mx_import_module
{
    friend class sd2sc_SS;
public:
    sd2sc_SS* sd2sc_SSlave;
    sdsig2scsig(CASIModuleIF* c, const sc_module_name& s);
    virtual ~sdsig2scsig();
    void init();
private:
    sc_out<int> sd2sc_SMaster;
};
```
3.4.5 Clocking SystemC clock ports

SystemC clock input ports are of type `sc_in<bool>` which are driven by `sc_clock`. You can use the `getMxSCClock()` API to get a handle to the SoC Designer system clock and use that to drive your clocks, or alternatively, use an external clock driver. An example of a SystemC clock driver is provided in `$MAXSIM_HOME/examples/SystemCImport/sc_clock_gen`.

Figure 3-19 SoC Designer signal to SystemC signal
3.5 Simulation control

This section describes functions and classes related to simulation control.

3.5.1 Simulation features

The requirements for SystemC direct import are:
1. Inherit from `sc_mx_import_module` instead of `sc_module`.
2. Define the factory class.
3. Provide an entry point into the module DLL.

This converts the module into a SoC Designer compliant model. To take full advantage of SoC Designer debug capabilities however, ARM recommends that you implement the CADI or CAPI interfaces on top of the imported module.

All of the built-in primitive channels have CADI registers to provide visibility of the data stored in the channel. You can use the CADI register window to observe the component state during simulation run-time. This typically simplifies the system by eliminating the requirement of adding static tracing hooks into the component or attaching a debugger to examine the internal data structures.

Because transaction debug features (such as transaction monitors and breakpoints) are not available for SCGeneric port connections, CADI can be used to expose debug transactions (such as the address, data and other control information associated with the transfer).

3.5.2 Clocking the SystemC modules

If you are integrating existing SystemC modules into the SoC Designer environment, ARM recommends using the `getMxSCClock()` function to attach the imported modules to the SoC Designer master clock.

The `getMxSCClock()` function

This function is provided by SoC Designer and returns an `sc_clock` that represents the SoC Designer master clock. By connecting your modules to this clock, the modules are clocked at the same rate as the SoC Designer master clock (specified by the SoC Designer system cycle period).

```c
sc_clock *getMxSCClock();
```

The system clock cycle period is equivalent to a SoC Designer simulation cycle and this period can be customized in Canvas from the System Properties dialog:

1. Select `System Properties` from the `Edit` menu.
2. Use the dialog (see Figure 3-20 on page 3-29), to set SoC Designer system clock period.

![System Properties dialog](image)

Figure 3-20 Edit System Properties dialog

3. You can optionally set the resolution and default time unit.

**The SoC Designer Clock Period**

The SystemC simulation is controlled by the SoC Designer interface. If you press **Run** in SoC Designer for example, the SystemC simulator starts simulating the SystemC modules and advances the simulation time with one SoC Designer clock period for every simulated cycle.

For instance, setting the SoC Designer cycle period to 10ns results in the simulation advancing by 10ns for every **step** command in SoC Designer.

——— Caution ————

When displaying the simulation performance, SoC Designer Simulator uses the SoC Designer Cycle Period for computing the number of cycles-per-second shown.

Depending on the ratio between your **sc_clock** period and the SoC Designer cycle period, you must adjust the speed displayed by SoC Designer to obtain the cycles-per-second speed in terms of your **sc_clock** cycles. The simulation performance of your SystemC modules are the same as the simulation performance displayed by the SoC Designer Simulator only when the **sc_clock** cycle period is the same as the SoC Designer **Cycle period**.
If you use the `sc_clock` returned from `getMxSCClock()`, the period of the `sc_clock` is the same as the SoC Designer system clock.

### The System Time Resolution

The **Time Resolution** represents the smallest amount of time that can be represented by all `sc_time` objects in the simulation. The default value is 1ps.

### The Default Time Unit

The **Default Time Unit** is the default value that are used for the time unit when an `sc_time` value is specified with no time unit declared. The default value is 1ns.

#### 3.5.3 Resetting the imported SystemC components

SystemC modules might not implement a reset behavior. To support the system reset functionality in SoC Designer, a reset behavior must be implemented by each of the imported SystemC modules. All imported components must implement the `sc_mx_import_method::reset()` method to reset their internal resources to the initial state.

#### 3.5.4 CADI functions

In addition to the CADIBase class, there are several functions that support debug access to component memories and registers from SoC Designer. For example:

- `CADIMemGetSpaces()`
- `CADIMemGetBlocks()`
- `CADIMemRead()`
- `CADIRegGetGroups()`
- `CADIRegGetMap()`
- `CADIRegWrite()`
- `CADIRegRead()`

#### 3.6 fifo example

This section describes the code that results from rewriting the original fifo example (see `simple_fifo.cpp` in the original subdirectory) to create the SoC Designer source files.
3.6.1 Files in the simple_fifo directory

The original fifo example used one C++ source file to hold all of the class definitions for the producer, fifo, and consumer. The SoC Designer implementation uses several different files and subdirectories to create the different components:

**producer files**

The `producer.cpp`, `producer.h`, and `producer.vcproj` files provide the classes for the producer component.

**fifo files**

The `fifo.cpp`, `fifo.h`, and `fifo.vcproj` files provide the classes for the basic functionality of the fifo component.

The `fifo_MxDI.cpp` and `fifo_MxDI.h` files contain code that provides a debug interface to the fifo component. The debug interface enables SoC Designer to display the contents of registers inside the fifo.

**consumer files**

The `consumer.cpp`, `consumer.h`, and `consumer.vcproj` files provide the classes for the consumer component.

**interface files**

The `common_if.h` and `simple_fifo_if.h` files define the interface that is supported by the components.

**project build files**

The `simple_fifo.sin` file is used to rebuild all of the components under Microsoft Visual C++ in a Windows environment. The `consumer.vcproj`, `producer.vcproj`, and `fifo.vcproj` files define the projects for the individual components.

Makefile is used in Linux environments.

See *Generating a makefile from the Component Wizard* on page 3-11 for details on producing the project files.

**SoC Designer files**

`Test.mxp` is an example SoC Designer Canvas project file. `Test.mxs` is the corresponding SoC Designer Simulator file.

After the component is built, the lib subdirectory contains the Debug and Release versions of the component library files as `.dll` (for Windows) or `.so` (for Linux).
The library configuration file contains the locations and descriptions of the fifo components. Use the Preferences dialog to add this .conf file to the model libraries used by SoC Designer.

### 3.6.2 Interface definition

Example 3-10 shows the source code for the interface that is used by the producer, fifo, and reader:

**Example 3-10 common_if.h**

```cpp
class write_if : virtual public sc_interface
{
    public:
    virtual void write(char) = 0;
    virtual void reset() = 0;
};
class read_if : virtual public sc_interface
{
    public:
    virtual void read(char &) = 0;
    virtual int num_available() = 0;
};
```

### 3.6.3 fifo component

Example 3-11 show how to convert an existing sc_module into a sc_mx_import_module.

**Example 3-11 SoC Designer fifo.h**

```cpp
#include "sc_mx_import_module.h"
#include "common_if.h"

// changes from original code:
// 1. class inheritance: from sc_channel -> sc_mx_import_module
// 2. constructor:               empty -> register ports (interfaces)
// 3. add MxFactory registry
// EXTRA

class fifo_MxDI;

class fifo : public sc_mx_import_module, public write_if, public read_if
{
    friend class fifo_MxDI;
```
The `fifo.cpp` file implements the constructor, initialization, and CADI functions for SoC Designer Module as shown in Example 3-12.

Example 3-12 fifo constructor and CADI

```cpp
public:
    fifo(CASIModuleIF *c, const sc_module_name& id);
    void write(char c);
    void read(char &c);
    void reset() { num_elements = first = 0; }
    void init();
    void terminate();
    int num_available() { return num_elements;}
    // The CADI interface.
    CADI* getCADI();

private:
    enum e { max = 10 };  
    char data[max]; 
    int num_elements, first; 
    sc_event write_event, read_event; 
    // Declare CADI Interface 
    CADI* cadi;
    char read_mxdi(int index);
};
```

```cpp
fifo::fifo(CASIModuleIF *c, const sc_module_name& id) :
    sc_mx_import_module(c, id, "fifo"), num_elements(0), first(0)
{
    cadi = NULL;
    // register interfaces (slave ports)
    registerSCGenericSlavePort( this, "read_if" );
    registerSCGenericSlavePort( this, "write_if" );
}

void fifo::init()
{
    cadi = new fifo_MxDI(this);
    sc_mx_import_module::init();
}

void fifo::terminate()
{
    sc_mx_import_module::terminate();
    if(cadi!=NULL) {
        delete cadi;
    }
    cadi=NULL;
}

CADI* fifo::getCADI()
```
3 Direct Import of SystemC Models

Add the \texttt{SC\_MX\_FACTORY} macro to \texttt{fifo.cpp} to define the \texttt{fifo} SoC Designer Module factory class and provide an entry point into the \texttt{fifo} Model DLL as shown in Example 3-13.

\begin{verbatim}
    SC_MX_FACTORY(fifo)
\end{verbatim}

\section*{fifo CADI}

The CADI interface accesses the \texttt{fifo} data structure as a memory block. The \texttt{fifo.cpp} file contains a \texttt{read\_mxdi()} function as shown in Example 3-14:

\begin{verbatim}
void fifo::read(char &c)
{
    if (num_elements == 0)
        wait(write_event);
    c = data[first];
    --num_elements;
    first = (first + 1) % max;
    read_event.notify();
}
char fifo::read\_mxdi(int index)
{
    return data[ index % max ];
}
\end{verbatim}

\textbf{Note}

Unlike the \texttt{read()} function, \texttt{read\_mxdi()} does not alter the \texttt{fifo} data or element counters.

The \texttt{CADIMemRead()} function in \texttt{fifo\_MxDI.cpp} uses \texttt{read\_mxdi()} as shown in Example 3-15 on page 3-35:
Example 3-15 CADIMemRead()

```c
ADIReturn_t fifo_MxDI::CADIMemRead(CADIAddrComplete_t startAddress,
    uint32_t unitsToRead, uint32_t unitSizeInBytes, uint8_t *data,
    uint32_t *actualNumOfUnitsRead, uint8_t doSideEffects )
{
    uint32_t i = 0;
    for ( i = 0; i < unitsToRead * unitSizeInBytes; )
    {
        uint32_t tmp = (uint32_t)target->read_mxdi((int)startAddress.location.addr);
        // TODO: Read the data from memory.
        if ( unitSizeInBytes == 1 )
        {
            data[i++] = (uint8_t)(tmp & 0xFF);
        }
        else if ( unitSizeInBytes == 2 )
        {
            data[i++] = (uint8_t)(tmp & 0xFF);
            data[i++] = (uint8_t)(( tmp >> 8 ) & 0xFF);
        }
        else if ( unitSizeInBytes == 4 )
        {
            data[i++] = (uint8_t)(tmp & 0xFF);
            data[i++] = (uint8_t)(( tmp >> 8 ) & 0xFF);
            data[i++] = (uint8_t)(( tmp >> 16 ) & 0xFF);
            data[i++] = (uint8_t)(( tmp >> 24 ) & 0xFF);
        }
    }
    *actualNumOfUnitsRead = i / unitSizeInBytes;
    return CADI_STATUS_OK;
}
```
3.6.4 producer component

Example 3-16 show how to convert an existing producer `sc_module` into a `sc_mx_import_module`.

**Example 3-16 SoC Designer simple_fifo producer**

```cpp
// from producer.h
class producer : public sc_mx_import_module
{
public:
    sc_port<write_if> out;
    SC_HAS_PROCESS(producer);
    SC_MX_IMPORT_CTOR(producer)
    {
        SC_THREAD(main);
        // register ports
        registerSCGenericMasterPort(&out, "out");
    }
    void main();
};
```

Use the `SC_MX_FACTORY` macro in a cpp source file to define the `producer` SoC Designer Module factory class and provide an entry point into the `producer` Model DLL as shown in Example 3-17.

**Example 3-17 producer factory**

```cpp
#include "producer.h"
SC_MX_FACTORY(producer)
```
3.6.5 consumer component

This section describes the code that implements the consumer component. Example 3-16 on page 3-36 show how to convert an existing consumer sc_module into a sc_mx_import_module.

Example 3-18 consumer.h

class consumer : public sc_mx_import_module
{
public:
    sc_port<read_if> in;
    void main();
    SC_HAS_PROCESS(consumer);
    SC_MX_IMPORT_CTOR(consumer)
    {
        SC_THREAD(main);
        // register ports
        registerSCGenericMasterPort(&in, "in");
    }
};

The consumer reads data from the port and uses mxcout to output status as shown in Example 3-19:

Example 3-19 consumer.cpp

void consumer::main()
{
    char c;
    while (true) {
        in->read(c);
        mxcout << c;
        mxcout.dumpMsg();
        if (in->num_available() == 1)
        {
            mxcout << "<1>";
            mxcout.dumpMsg();
        }
        if (in->num_available() == 9)
        {
            mxcout << "<9>";
            mxcout.dumpMsg();
        }
        wait(10, SC_NS);
}
3.6.6 Running the fifo simulation

Load the Test.mxs file (from the simple_fifo directory) into SoC Designer Simulator to and use the Step button to demonstrate the flow through the fifo as shown in Figure 3-21 on page 3-38:

Figure 3-21 fifo simulation
The fifo component implements CADI for the internal memory and the contents can be displayed in the Memory debug window.

Note

The fifo implements the interface defined in common_if.h and functions as a channel. The consumer and producer can be directly connected to the fifo without using primitive channel components.

3.7 dpipe example

This section describes the code that results from rewriting the original dpipe example (see main.cpp in the original subdirectory) to produce the SoC Designer source files.

3.7.1 Files in the dpipe example directory

The original dpipe source code has one file (main.cpp) that contains the pipe, reader, and writer. The conversion splits the various elements into several distinct files:

**dpipe component**

dpipe.cpp and dpipe.h implement the pipe component.
dpipe_MxDI.cpp and dpipe_MxDI.h implement the debug interface.

**reader component**

reader.cpp and reader.h implement the component that reads from the pipe interface.

**writer component**

writer.cpp and writer.h implement the component that writes to the pipe interface.

**project build files**

The dpipe.sin file is used to rebuild all of the components under Microsoft Visual C++ in a Windows environment. The writer.vcproj, reader.vcproj, and dpipe.vcproj files define the projects for the individual components.

Makefile is used in Linux environments.

See Generating a makefile from the Component Wizard on page 3-11 for details on producing the project files.
SoC Designer files

libwriter.dll, libreader.dll, and libpipe.dll are the component libraries that are used in the Windows version of SoC Designer Simulator.

Test.mxp and Test.mxs are SoC Designer project and simulation files that demonstrate how the different component are connected together.

3.7.2 dpipe source

This section describes the changes made to the dpipe component.

Changes to the class definition

Example 3-20 on page 3-40 shows the use of sc_mx_import_module and registerSCGenericSlavePort() in dpipe.h.

Example 3-20 dpipe.h for SoC Designer

```
template<class T, int N> class dpipe : public sc_mx_import_module
{
    typedef sc_export<sc_signal_inout_if<T>> in;
    typedef sc_export<sc_signal_in_if<T>> out;

    public:
        SC_HAS_PROCESS(dpipe);
        SC_MX_IMPORT_CTOR(dpipe)
        {
            m_in(m_pipe[0]);
            m_out(m_pipe[N-1]);
            SC_METHOD(rachet);
            sensitive << m_clk.pos();
            // register ports
            registerSCGenericSlavePort(&m_in, "m_in");
            registerSCGenericSlavePort(&m_out, "m_out");
        }
    void rachet();

    // re-implemented CASIModule methods
    void init();

    // The CADI interface
    CADI* getCADI();

    sc_in_clk m_clk; // Pipe synchronization
    in m_in // Input to delay pipe
```
out m_out // Output from delay pipe
     sc_signal<T> m_pipe[N]; // pipeline stages

};

The factory class in dpipe.cpp creates a new dpipe component using the templated constructor as shown in Example 3-21. The type for atom is defined in the common.h file.

Example 3-21 The dpipe factory

```cpp
class dpipeFactory : public CASIFactory
{
    public:
        dpipeFactory() : CASIFactory( "dpipe" ) {}
        CASIModuleIF *createInstance(CASIModuleIF *c, const string &id)
        {
            eslapi::setDoNotRegisterInSystemC( false );
            return new dpipe<atom, 4>(c, id.c_str());
        }
};

// entry point from SoC Designer
extern "C" void MxInit(void)
{
    new dpipeFactory();
}
extern "C" void MxInit_SCImport(void)
{
}
```

___ Note _______

The SC_MX_IMPORT macro is expanded to enable template data type specification when instantiating the dpipe module. Binding to the SoC Designer system clock is also done in the init() method. See also Clocking the SystemC modules on page 3-28.
CADI support

The `dpipe.cpp` file includes code to create and delete the CADI interface as shown in Example 3-22:

Example 3-22 CADI in `dpipe`

```cpp
template<class T, int N> void dpipe<T, N>::init()
{
    // Create CADI Interface
    cadi = new dpipe_CADI(this);
    // bind m_clk to SoC Designer system clock
    m_clk( *( getMxSCClock() ) );
}

template<class T, int N> void dpipe<T, N>::terminate()
{
    sc_mx_import_module::terminate();
    // Release the CADI Interface
    if( cadi!=NULL ) {
        delete cadi;
        cadi=NULL;
    }
}

template<class T, int N> CADI* dpipe<T, N>::getCADI()
{
    return cadi;
}
```

The only new property for the `dpipe` component is CADI support. Other requests for property values are forwarded to the base class as shown in Example 3-23:

Example 3-23 `getProperty()`

```cpp
template<class T, int N> string dpipe<T, N>::getProperty(
    eslapi::CASIPROPERTYTYPE property )
{
    switch ( property )
    {
    case CADI_PROP_MXDI_SUPPORT:
        return "yes";
    default:
        return sc_mx_import_module::getProperty( property );
    }
```
The files `dpipe_MxDI.h` define the CADI classes and functions for the dpipe component as shown in Example 3-24:

```
Example 3-24 dpipe_MxDI.h

template< typename, int > class dpipe;
class dpipe_MxDI : public CADIBase
{
public:
    dpipe_MxDI(dpipe<atom, 4>* c);
    virtual ~dpipe_MxDI();

public:
    // Register access functions
    CADIReturn_t    CADIRegGetGroups(uint32_t groupIndex, uint32_t desiredNumOfRegGroups,
                                       uint32_t* actualNumOfRegGroups,  CADIRegGroup_t* reg );
    CADIReturn_t    CADIRegGetMap(uint32_t groupID, uint32_t regIndex,  uint32_t registerSlots,
                                   uint32_t* registerCount,  CADIRegInfo_t* reg );
    CADIReturn_t    CADIRegWrite(uint32_t regCount, CADIReg_t* reg,  uint32_t* numRegsWritten,
                                 uint8_t doSideEffects );
    CADIReturn_t    CADIRegRead(uint32_t regCount, CADIReg_t* reg,  uint32_t* numRegsRead,
                               uint8_t doSideEffects );

private:
    dpipe<atom, 4>*    target;
    // Register related info
    CADIRegInfo_t*    regInfo;
    CADIRegGroup_t*   regGroup;
};
```

See the `dpipe_CADI.cpp` file and the CADI section of the *SoC Designer User Guide* for implementation details.
3.7.3 reader source files

Example 3-25 and Example 3-26 on page 3-44 show the converted code for the reader component.

Example 3-25 reader.h for SoC Designer

```cpp
#include "sc_mx_import_module.h"
#include "common.h"
class reader : public sc_mx_import_module
{
public:
    SC_HAS_PROCESS(reader);
    SC_MX_IMPORT_CTOR(reader)
    {
        SC_METHOD(extract)
        sensitive << m_clk.pos();
        dont_initialize();

        registerSCGenericMasterPort(&m_from_pipe, "m_from_pipe");
    }
protected:
    void extract();
public:
    void init();
public:
    sc_in_clk m_clk; // Module synchronization.
    sc_in<atom> m_from_pipe; // Output from delay pipe.
};
```

The reader.cpp file contains the implementation of extract() and init() for the reader component as shown in Example 3-26 on page 3-44:

Example 3-26 reader.cpp

```cpp
#include "reader.h"
void reader::extract()
{
    mxcout << sc_time_stamp().to_string() << ": " << m_from_pipe->read() << endl;
    mxcout.dumpMsg();
}

void reader::init()
{
    // bind m_clk to SoC Designer system clock
```
3.7.4 writer source files

Example 3-27 and Example 3-28 on page 3-46 show the converted code for the writer component.

Example 3-27 writer.h for SoC Designer

```c++
#include "sc_mx_import_module.h"
#include "common.h"
// Testbench writer of values to the pipe:
class writer : public sc_mx_import_module
{
public:
    SC_HAS_PROCESS( writer );
    SC_MX_IMPORT_CTOR(writer)
    {
        SC_METHOD(insert)
            sensitive << m_clk.pos();
            m_counter = 0;
            registerSCGenericMasterPort(&m_to_pipe, "m_to_pipe");
    }
    void insert();
public:
    void init();
    void reset( eslapi::CASIResetLevel level, const CASIFileMapIF *filelist );
sc_in_clk m_clk;       // Module synchronization.
atom m_counter;        // Write value.
sInOut<atom> m_to_pipe; // Input for delay pipe.
};
```

The writer.cpp file contains the implementation of insert() and init() and the factory macro for the writer component as shown in Example 3-28:
3 Direct Import of SystemC Models

Example 3-28 writer.cpp

```c++
void writer::insert()
{
    m_to_pipe->write( m_counter );
    m_counter++;
}
void writer::init()
{
    // bind m_clk to SoC Designer system clock
    m_clk( *( getMxSCClock() ) );
}
void writer::reset( eslapi::CASIResetLevel level, const CASIFileMapIF *filelist )
{
    mxcout << "resetting m_counter" << endl;
    mxcout.dumpMsg();
    m_counter = 0;
}
// writer factory class + entry point for SoC Designer
SC_MX_FACTORY(writer)
```

3.7.5 Running the dpipe simulation

Load the Test.mxs file (in the dpipe directory) into SoC Designer Simulator and click the Step button to demonstrate the flow through the delay pipe as shown in Figure 3-22 on page 3-47:

Note

The dpipe implements an interface for the in and out ports. (See the use of `sc_export<sc_signal_inout_if<T>>` and `sc_export<sc_signal_in_if<T>>` in the dpipe.h file). The interface type is defined in common.h. The writer and reader can be directly connected to the dpipe without using a primitive channel component.

——— Note ————
The dpipe component implements CADI for the internal registers and the register contents can be displayed in the Register debug window.
This chapter describes how to use the Component Wizard to generate a wrapper that simplifies the import of a SystemC model.

This chapter contains the following sections:

• Generating CASI wrapper components with the Component Wizard on page 4-2
• Instantiating SystemC modules on page 4-4
• Clocking generated components on page 4-5
• Connecting imported components to SoC Designer components on page 4-7
• The scport_mxsignal example on page 4-9
• TLM master to SoC Designer AHB slave on page 4-13

——— Note ————

Early versions of SoC Designer required a wrapper component for SystemC import. The wrapper module instantiated the module and effectively established a hierarchy of modules. This import mechanism is still fully supported in newer versions of SoC Designer and is described in this section.
4.1 Generating CASI wrapper components with the Component Wizard

To generate a SoC Designer CASI wrapper component using the Component Wizard in SoC Designer Canvas, select Component Wizard from the Tools menu. The dialog shown in Figure 4-1 is displayed.

![Component Wizard dialog](image)

Figure 4-1 Component Wizard dialog

Enter the settings for the component:

1. Enter the name of the component
2. Enter the directory location
3. Select the Standard SoC Designer Component radio button
4. Click Next.
5. The Component Wizard guides you through the generation of the files for the top-level SoC Designer component.
6. You must edit the generated files to add the functionality required for your component and then recompile the library models.
4.1.1 Files generated for SystemC import

The component wizard typically generates the following files:

- The top level project files:
  - makefile for Linux systems
  - component-name.vcproj and component-name.def for Microsoft Visual Studio .Net (Visual C++) systems

- A .cpp and .h file for the SoC Designer component:
  - component-name.cpp
  - component-name.h

- A .cpp and .h file for each slave port:
  - slave-port-name_TS.cpp for transaction slave ports
  - slave-port-name_TS.h for transaction slave ports
  - slave-port-name_SS.cpp for signal slave ports
  - slave-port-name_SS.h for signal slave ports.

- A .cpp and .h file CADI if selected:
  - component-name_CADI.cpp
  - component-name_CADI.h

The newly generated SoC Designer component does not contain any behavior. It provides a top-level CASI module that is a valid SoC Designer component and provides a container for the imported SystemC. This hierarchical construction of SystemC sub-modules is detailed in the following sections.

--- Note ---

All of the files listed above are not always generated. The files are generated only if they are required. This is determined in the Component Wizard by the selections during the component generation process.
4.2 Instantiating SystemC modules

The generated SoC Designer CASI wrapper component contains a module that acts as a hierarchical place-holder for the SystemC modules to be imported. The imported SystemC modules must be instantiated as sub-modules of the generated component:

1. Insert the existing SystemC files into the generated makefile or Microsoft Visual C++ project for the SoC Designer CASI wrapper component:
   - **Linux**: Insert the filenames of the existing SystemC files into the Makefile generated for the SoC Designer component.
   - **Windows**: Insert the existing SystemC files into the Microsoft Visual C++ (MSVC++) project generated for the SoC Designer component. For instance, for the component `systemc_component`, the MSVC++ project files are called `systemc_component.dsw` and `systemc_component.dsp`. Start the Microsoft Visual C++ Studio and open the `systemc_component.dsw` workspace file, then add the desired files to the project.

   ______ Note ______

   If the original SystemC files contained several components, you can create project files for each component and a separate project file that builds the top-level component and all of the contained components.

   ______

2. Remove any `sc_main` functions from your existing SystemC files. This `sc_main` function is no longer required. Instead, all the existing SystemC modules must be instantiated hierarchically as sub-modules of the generated SoC Designer SystemC module.

   ______ Note ______

   Do not call `sc_start()`, `sc_initialize()`, or `sc_cycle()` from the imported code. Simulation control is done automatically by SoC Designer.

   `sc_stop()` is still supported in SoC Designer and any SystemC module can call `sc_stop()` to halt the simulation. The simulation is stopped permanently if a `sc_stop()` is called and warning message is issued if the user tries to continue the simulation after the `sc_stop()` invocation.

   ______

3. Move the SystemC module declarations from your old `sc_main` into the generated component class (in `component_name.h`). These SystemC modules become sub-modules of the generated component.
4. Move any instantiation and connection code from the old `sc_main` function into the `init()` function of the generated SoC Designer CASI wrapper module (in `component_name.cpp`).

The interconnection between the external SoC Designer ports (if any are present) and the internal SystemC ports must be done in the `init()` function.

5. After editing and recompiling the source code, add the name and location of the library dll or so files to a library configuration file. You must add the new library details to an existing file or create a new library configuration file and add the details in that file.

6. If you created a new library configuration file, use the **Model Library** section of SoC Designer Preferences dialog to add the new library configuration file to this list of library files.

    ______ Note ______

    If you keep the newly created library configuration file and the .mxp files in the same directory, you can use the option **Use directory that contains the SoC Designer Project file**. This option simplifies working on different projects, but it is slower because the Component Window must be refreshed every time you change working directories.

### 4.3 Clocking generated components

The generated wrapper module can use the `getMxSCClock()` method to get a handle to the SoC Designer system clock and pass that clock down to the imported sub-modules. (See *Clocking the SystemC modules for high performance*).

You can however explicitly create a `sc_clock` signal in the generated wrapper component by declaring it in the class definition and connecting it to the existing component in the component `init()` function.

For instance, to run a component with a 10ns clock period, use the following clock:

```
sc_clock(10, SC_NS);
```

    ______ Note ______

The imported SystemC modules are clocked using explicit `sc_clock` objects in the SystemC code instead of directly using the SoC Designer cycle-based clocking. The generated SoC Designer wrapper component is not, therefore, required to show a clock port when it is displayed in the diagram window. This does not mean that the imported SystemC sub-modules are not clocked. It only means that the generated hierarchical component is not clocked using the SoC Designer cycle-based clock.
4.3.1 Clocking the SystemC modules for high performance

The ideal way of clocking imported SystemC modules for high simulation performance is to use the SoC Designer cycle-based clocking mechanism and avoid any event-driven features from SystemC (see The getMxSCClock() function on page 3-28).

If the imported SystemC modules are cycle-based and do not use any threads or explicit event notifications, you can use the SoC Designer cycle-based clocking mechanism. The instructions in this section assume that the imported SystemC modules contain only SC_METHOD and any triggering of these methods is done on the rising edge of the clock.

In order to use the SoC Designer cycle-based clocking mechanism:

1. Modify the generated SoC Designer SystemC component to make it clocked by adding communicate/update functions, registering the clock port, and registering the component with the scheduler (calling registerClockSlave()). See the SoC Designer User Guide for details on how to do this.

2. Remove the sc_clock object instantiation (and any ports and connections to it) from the original SystemC code.

3. Explicitly call the SC_METHOD code from the communicate() function of the generated hierarchical SoC Designer component:

   ```c
   void <my_SoC.Designer.component>::communicate()
   {
       <my_imported_module>->my_sc_method_function();
       ...
   }
   ```

4. Use a clock divider in Canvas to obtain the desired clocking rate of the imported modules.
4.4 Connecting imported components to SoC Designer components

Use the wrapper component signal and transaction ports to connect imported SystemC modules to other SoC Designer components.

In general, the ports and channels of legacy imported SystemC modules are based on user-defined SystemC interfaces. In order to connect these ports/channels to other SoC Designer components, it is necessary to translate the user-defined interfaces to the SoC Designer simulation interfaces.

1. For each port or channel of the imported modules that is required to connect to another SoC Designer component, create a corresponding channel or port in the generated wrapper component with the same interface.
   
   In Figure 4-2 on page 4-8 for example, for the port p1 of the imported SystemC module, create a corresponding channel in the generated SoC Designer SystemC component and name it p3.

   For channel p2 of the imported SystemC module, create a corresponding port in the generated SoC Designer component and name it p4.

   The channel p3 and port p4 use user_if and this is the same user-defined sc_interface that is used by the imported SystemC module.

2. In the init() function of the wrapper module, connect the imported SystemC module ports p1 and p2 to the corresponding p3 and p4 ports created in the previous step.

3. Perform the required protocol conversions to implement the translation from the SystemC user-type ports p3 and p4 to the CASI ports p1 and p2 of the generated component.
   
   For Figure 4-2 for example, add the translation code to the p3 channel code and to the p2 slave port code by translating the data structures between the two interfaces and calling the corresponding read/write functions in the ports p1 and p4 of the generated component. For more information on the CASI interfaces, see the SoC Designer User Guide.

4. Use the SoC Designer Canvas tools to connect the CASI ports to other SoC Designer components.
   
   Figure 4-2 shows the connection structure for translating user-defined SystemC ports from the imported modules to the generated ports in the generated wrapper module.
Figure 4-2 Port connection structure for an imported SystemC module
4.5 The scport_mxsignal example

The old subdirectory in the SystemCImport directory contains examples of using wrappers for importing SystemC components into SoC Designer.

The scport_mxsignal example uses a component with SoC Designer master and slave signals ports that are connected to an internal SystemC component.

4.5.1 top.h

The top.h file contains the original SystemC component as shown in Example 4-1:

```cpp
Example 4-1 SystemC component in top.h

class to_mx_smaster_if : public sc_interface
{
public:
    virtual void send_to_mx_smaster_port( int ) = 0;
};

class top : public sc_module
{
public:
    SC_CTOR( top ) {
        SC_METHOD( sig_received );
        sensitive << input_signal;
        dont_initialize();
        write_value = 0;
    }
    sc_in<int> input_signal;
    sc_port< to_mx_smaster_if > out;
    int write_value;
    void sig_received() {
        mxcout << "top::sig_received() - value = " << input_signal.read() << endl;
        mxcout.dumpMsg();
        // send value onto SoC Designer signal master port via to_mx_smaster_if
        out->send_to_mx_smaster_port( ++write_value );
    }
};

Note

The to_mx_smaster_if interface provides the communication between the SystemC port out and the SoC Designer port.
4.5.2 portmap.cpp

The `portmap()` constructor in the `portmap.cpp` file creates and registers the SoC Designer master and slave ports as shown in Example 4-2:

**Example 4-2 SystemC component in top.h**

```c
portmap::portmap(CASIModuleIF* c, const string &s) : CASIModule(c, s)
{
    initComplete = false;
    // SoC Designer signal slave port
    mx_sig_in = new sslave( this );
    registerPort( mx_sig_in, "in" );
    // SoC Designer signal master port
    mx_sig_out = new sc_port< CASISignalIF >( this, "out" );
    registerPort( mx_sig_out, "out" );
    cadi = NULL;
    // Place parameter definitions here:
    defineParameter( "Enable Debug Messages", "false", eslapi::CASI_PARAM_BOOL);
}
```

The `init()` function in the `portmap.cpp` file provides the interface between the SoC Designer component and the SystemC component as shown in Example 4-3:

**Example 4-3 SystemC component in top.h**

```c
void portmap::init()
{
    // instantiate systemc module
    top_module = new top( "top" );

    // connect sc_port<int> of top to
    // sc_signal<int> of SoC Designer wrapper class
    top_module->input_signal( input_signal );

    // output of top module will talk to mx_sig_out via to_mx_smaster_channel
    out = new to_mx_smaster_channel( "out", mx_sig_out );
    top_module->out( *out );

    // Create CADI Interface
    cadi = new portmap_MxDI(this);
    if (cadi == NULL)
    {
        message(eslapi::CASI_MSG_FATAL_ERROR,"init: Failed to allocate memory for CADI"
    }
```
// TODO: Allocate memory and initialize data here:
// …

// Call the base class after this has been initialized.
CASIModule::init();
// Set a flag that init stage has been completed
initComplete = true;
}

4.5.3 sslave.cpp

The `driveSignal()` function in the `sslave.cpp` file forwards the SoC Designer slave port signal to the SystemC port as shown in Example 4-4:

**Example 4-4 SystemC component in top.h**

```c
void sslave::driveSignal(uint32_t value, uint32_t* extValue)
{
    // forward input value to systemc module sc_port
    owner->input_signal.write( value );
}
```

4.5.4 SoC Designer Simulator with portmapping component

Figure 4-3 on page 4-12 shows a system that uses the port mapping component:
Figure 4-3 Test simulation with portmapping component

The monitors on the in and out SoC Designer ports show the values into and out of the component.
4.6 TLM master to SoC Designer AHB slave

The TLM2AHB example system with bus monitors is shown in Figure 4-4. The system consists of the following components:

- **AHB_Master** component that provides a wrapper for a TLM module
- **MxAHB** component that interfaces the bus master and slave ports
- **AHB_Slave** component that contains a TLM memory component

This system shown in Figure 4-4 does not display the internal SystemC components. The TLM components are intended as a shell to add additional SystemC functionality as shown in Figure 4-5 on page 4-14.
For more detail about the TLM code, see the example code in the TLM2AHB example directories:

**include**

The `basic_initiator_port.h`, `basic_protocol.h`, `basic_slave_base.h`, and `bus_types.h` files that implement the basic TLM protocol and the `basic_initiator_port` helper class.

`AHB_Transaction.h`, `persister.h`, and `MxPortCheck.h` are the include files for AMBA2.

**AHB_Slave**

`mem_slave.cpp` and `mem_slave.h` contain code for the SystemC TLM slave memory object.

The `AHB_Slave` files contain the code for the AHB_Slave component.

The `slave_TS` files contain the code for the AHB_Slave port.

**AHB_Master**

The `AHB_Master` files are generated from the Component Wizard.

`AHB_Transaction.h`, `persister.h`, and `MxPortCheck.h` are the include files for AMBA2.

The `master` files contain the SystemC TLM Master component.

The `slave` files provide the interface to the AHB bus.
Chapter 5
Modeling Guidelines for SystemC

This chapter contains modeling guidelines to improve performance of SystemC models in SoC Designer. It contains the following section:

• **Modeling for Speed**

  —**Caution**—

Optimum simulation speed is attained by:

• developing models that use CASI communication library natively (no protocol translation)

• using a cycle-based modeling paradigm instead of using even-driven features such as `SC_THREAD` or `SC_METHOD` with a sensitivity list.
5.1 Modeling for Speed

SoC Designer supports simulation of any SystemC-compliant models. Not all models, however, are optimized for performance and they might not run at a satisfactory simulation speed. Use the CASI communication library and the recommendations in this section to write efficient SystemC models.

SystemC offers a wide variety of constructs for many different applications, from high-level simulations all the way down to low-level hardware simulation. If simulation performance is very important, avoid constructs that introduce a large simulation overhead.

Threads require task switching and are therefore not recommended. Using `SC_METHOD` instead of `SC_THREAD` always results in higher simulation performance.

The `wait` statement of SystemC provides a fairly convenient way to pause a thread until a certain event occurs. However using `wait` implies task switching because the execution of the current function must be suspended until the event occurs. This is not required in a purely cycle-based environment. The same behavior can be expressed using a **Finite State Machine (FSM)** that has only one entry point and is accessed in every cycle. Transforming an `SC_THREAD` with `wait` statements into an `SC_METHOD` with an FSM is typically a trivial task.

Models that are only operating occasionally and do not require being continuously clocked can unregister themselves from the clock after an operation has completed. As soon as they are activated again, they can register themselves to be clocked until the operation is done.

For purely reactive models that forward a transaction from their slave port directly to the master port, it might be appropriate to not clock the models at all.
Appendix A
SystemC Implementation

This appendix documents the SystemC implementation in SoC Designer. It contains the following sections:

• *Cycle Model-specific SystemC Implementation Differences* on page A-2
• *Built-in primitive channels* on page A-3
A.1 Cycle Model-specific SystemC Implementation Differences

<table>
<thead>
<tr>
<th>Feature</th>
<th>Description</th>
<th>Support in SoC Designer</th>
</tr>
</thead>
<tbody>
<tr>
<td>Support for programs with own main() function</td>
<td>Support for user main() through SystemC sc_main_main() function.</td>
<td>Not supported. Components are loaded as DLLs controlled by SoC Designer, and should not have a main() function.</td>
</tr>
<tr>
<td>Getting copies of the start-up arguments</td>
<td>SystemC enables access to start-up arguments through sc_argv() function.</td>
<td>Not supported. Component parameters should be used instead. Parameter values can be passed down to SystemC modules from the SoC Designer wrapper.</td>
</tr>
<tr>
<td>Object code release tagging</td>
<td>Binary interface compatibility checking.</td>
<td>Customized vendor tag and version are encoded in the SoC Designer SystemC library. Link against this library if importing SystemC modules.</td>
</tr>
<tr>
<td>sc_cycle()</td>
<td>sc_cycle() enables running the simulation for a fixed number of cycles, but its use was deprecated in SystemC 2.2.</td>
<td>sc_cycle() and sc_start(0) are not supported in SoC Designer. Use the control buttons in SoC Designer Simulator for simulation control.</td>
</tr>
<tr>
<td>sc_start(0)</td>
<td>sc_start(0) performs delta cycles based on pending events and assignments</td>
<td>sc_start(0) is not supported in SoC Designer. Simulation control granularity is limited to one system clock cycle.</td>
</tr>
<tr>
<td>sc_stop() semantics change for sc_set_stop_mode()</td>
<td>sc_set_stop_mode() defines two modes of sc_stop(): SC_STOP_IMMEDIATELY, SC_STOP_FINISH_DELTA.</td>
<td>sc_stop() and sc_set_stop_mode() are supported in SoC Designer.</td>
</tr>
<tr>
<td>Calling sc_start() after sc_stop() is an error</td>
<td>After sc_stop() has been called sc_start() produces an error message</td>
<td>SoC Designer issues an error message if the user tries to continue execution after an sc_stop().</td>
</tr>
</tbody>
</table>
A.2 Built-in primitive channels

Table A-2 lists the primitive channels that are supplied with SoC Designer. These primitive channels are used to connect ports that support the corresponding SystemC interface.

Table A-2 Primitive channels

<table>
<thead>
<tr>
<th>Channel type</th>
<th>Data type</th>
<th>Component name</th>
</tr>
</thead>
<tbody>
<tr>
<td>sc_buffer</td>
<td>bool</td>
<td>sc_buffer&lt;bool&gt;</td>
</tr>
<tr>
<td></td>
<td>char</td>
<td>sc_buffer&lt;char&gt;</td>
</tr>
<tr>
<td></td>
<td>double</td>
<td>sc_buffer&lt;double&gt;</td>
</tr>
<tr>
<td></td>
<td>float</td>
<td>sc_buffer&lt;float&gt;</td>
</tr>
<tr>
<td></td>
<td>int</td>
<td>sc_buffer&lt;int&gt;</td>
</tr>
<tr>
<td></td>
<td>long</td>
<td>sc_buffer&lt;long&gt;</td>
</tr>
<tr>
<td>sc_clock</td>
<td>-</td>
<td>sc_clock</td>
</tr>
</tbody>
</table>
Table A-2 Primitive channels (continued)

<table>
<thead>
<tr>
<th>Channel type</th>
<th>Data type</th>
<th>Component name</th>
</tr>
</thead>
<tbody>
<tr>
<td>sc_fifo</td>
<td>bool</td>
<td>sc_fifo&lt;bool&gt;</td>
</tr>
<tr>
<td></td>
<td>char</td>
<td>sc_fifo&lt;char&gt;</td>
</tr>
<tr>
<td></td>
<td>double</td>
<td>sc_fifo&lt;double&gt;</td>
</tr>
<tr>
<td></td>
<td>float</td>
<td>sc_fifo&lt;float&gt;</td>
</tr>
<tr>
<td></td>
<td>int</td>
<td>sc_fifo&lt;int&gt;</td>
</tr>
<tr>
<td></td>
<td>long</td>
<td>sc_fifo&lt;long&gt;</td>
</tr>
<tr>
<td>sc_mutex</td>
<td>-</td>
<td>sc_mutex</td>
</tr>
<tr>
<td>sc_semaphore</td>
<td>-</td>
<td>sc_mutex</td>
</tr>
<tr>
<td>sc_signal</td>
<td>bool</td>
<td>sc_signal&lt;bool&gt;</td>
</tr>
<tr>
<td></td>
<td>char</td>
<td>sc_signal&lt;char&gt;</td>
</tr>
<tr>
<td></td>
<td>double</td>
<td>sc_signal&lt;double&gt;</td>
</tr>
<tr>
<td></td>
<td>float</td>
<td>sc_signal&lt;float&gt;</td>
</tr>
<tr>
<td></td>
<td>int</td>
<td>sc_signal&lt;int&gt;</td>
</tr>
<tr>
<td></td>
<td>long</td>
<td>sc_signal&lt;long&gt;</td>
</tr>
<tr>
<td></td>
<td>sc_logic</td>
<td>sc_signal&lt;sc_logic&gt;</td>
</tr>
<tr>
<td></td>
<td>long</td>
<td>sc_signal&lt;long&gt;</td>
</tr>
<tr>
<td></td>
<td>short</td>
<td>sc_signal&lt;short&gt;</td>
</tr>
<tr>
<td>sc_signal_resolved</td>
<td>-</td>
<td>sc_signal_resolved</td>
</tr>
</tbody>
</table>