SoC Designer

Version 9.2

SystemC Linking Guide

ARM

ARM DUI1041C

SoC Designer

SystemC Linking Guide

Release Information

Document History

Issue	Date	Confidentiality	Change
A	November 2016	Non-Confidential	Rebrand update for 9.0
B	24 February 2017	Non-Confidential	Update for 9.1
C	31 May 2017	Non-Confidential	Update for 9.2

Non-Confidential Proprietary Notice

ARM Limited. Company 02557590 registered in England.

110 Fulbourn Road, Cambridge, England CB1 9NJ.

LES-PRE-20349

Confidentiality Status

This document is Non-Confidential. The right to use, copy and disclose this document may be subject to license restrictions in accordance with the terms of the agreement entered into by ARM and the party that ARM delivered this document to.

Unrestricted Access is an ARM internal classification.

Product Status

The information in this document is Final, that is for a developed product.

Web Address

http://www.arm.com

Contents
SoC Designer SystemC Linking Guide

Preface

This preface introduces the *SoC Designer SystemC Linking Guide*.

It contains the following:

About this book

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

Product revision status

The r*m*p*n* identifier indicates the revision status of the product described in this book, for example, r*1*p*2*, where:

r*m* Identifies the major revision of the product, for example, r1.

p*n* Identifies the minor revision or modification status of the product, for example, p2.

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.

Using this book

This book is organized into the following chapters:

Chapter 1 Introduction
This chapter introduces the support of SystemC models in SoC Designer.

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.

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.

Chapter 4 Generating a Wrapper for SystemC Models
This chapter describes how to use the Component Wizard to generate a wrapper that simplifies the import of a SystemC model.

Chapter 5 Modeling Guidelines for SystemC
This chapter contains modeling guidelines to improve performance of SystemC models in SoC Designer.

Appendix A SystemC Implementation
This appendix documents the SystemC implementation in SoC Designer.

Glossary

The ARM Glossary is a list of terms used in ARM documentation, together with definitions for those terms. The ARM Glossary does not contain terms that are industry standard unless the ARM meaning differs from the generally accepted meaning.

See the *ARM Glossary* for more information.

Typographic conventions

italic
Introduces special terminology, denotes cross-references, and citations.

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

`monospace`
Denotes text that you can enter at the keyboard, such as commands, file and program names, and source code.

mono`space`
: Denotes a permitted abbreviation for a command or option. You can enter the underlined text instead of the full command or option name.

`monospace italic`
: Denotes arguments to monospace text where the argument is to be replaced by a specific value.

`monospace bold`
: Denotes language keywords when used outside example code.

`<and>`
: Encloses replaceable terms for assembler syntax where they appear in code or code fragments. For example:

```
MRC p15, 0, <Rd>, <CRn>, <CRm>, <Opcode_2>
```

SMALL CAPITALS
: Used in body text for a few terms that have specific technical meanings, that are defined in the *ARM Glossary*. For example, IMPLEMENTATION DEFINED, IMPLEMENTATION SPECIFIC, UNKNOWN, and UNPREDICTABLE.

Timing diagrams

The following figure explains the components used in timing diagrams. Variations, when they occur, have clear labels. You must not assume any timing information that is not explicit in the diagrams.

Shaded bus and signal areas are undefined, so the bus or signal can assume any value within the shaded area at that time. The actual level is unimportant and does not affect normal operation.

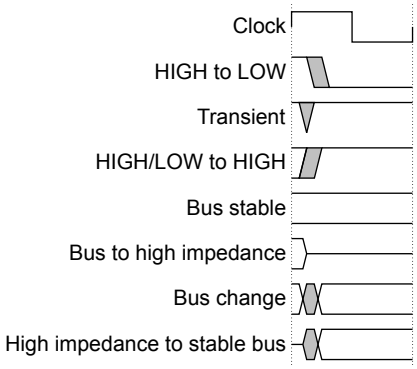

Figure 1 Key to timing diagram conventions

Signals

The signal conventions are:

Signal level
: The level of an asserted signal depends on whether the signal is active-HIGH or active-LOW. Asserted means:
- HIGH for active-HIGH signals.
- LOW for active-LOW signals.

Lowercase n
: At the start or end of a signal name denotes an active-LOW signal.

Additional reading

This book contains information that is specific to this product. See the following documents for other relevant information.

ARM publications

- *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*.

Other 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*.

The following publications provide additional information on simulation:

- *IEEE 1666 SystemC Language Reference Manual*, (IEEE Standards Association).
- *SPIRIT User Guide*, Revision 1.2, SPIRIT Consortium.

Feedback

Feedback on this product

If you have any comments or suggestions about this product, contact your supplier and give:

- The product name.
- The product revision or version.
- An explanation with as much information as you can provide. Include symptoms and diagnostic procedures if appropriate.

Feedback on content

If you have comments on content then send an e-mail to *errata@arm.com*. Give:

- The title *SoC Designer SystemC Linking Guide*.
- The number ARM DUI1041C.
- If applicable, the page number(s) to which your comments refer.
- A concise explanation of your comments.

ARM also welcomes general suggestions for additions and improvements.

Note

ARM tests the PDF only in Adobe Acrobat and Acrobat Reader, and cannot guarantee the quality of the represented document when used with any other PDF reader.

Chapter 1
Introduction

This chapter introduces the support of SystemC models in SoC Designer.

It contains the following sections:

1.1 Import overview for SystemC models

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.

You can create new SoC Designer components 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.

1.2 Import methods

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.

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.

Use a wrapper
: Use a SoC Designer wrapper to instantiate the SystemC user modules as subcomponents. 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.

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

Related references

Chapter 2 SystemC to SoC Designer Import Wizard on page 2-17.
Chapter 3 Direct Import of SystemC Models on page 3-26.
Chapter 4 Generating a Wrapper for SystemC Models on page 4-61.

1.3 System requirements

See the *SoC Designer Installation Guide* (ARM DUI 0953) for supported platforms and compilers.

——— **Caution** ———

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.)

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.

———————————

1.4 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.

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 `init()` function of the component with the parameter set to `true`:

```
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:

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

Related references

Chapter 5 Modeling Guidelines for SystemC on page 5-67.

1.5 Controlling message output

Legacy SystemC code typically uses `cout()` to print debug messages to 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:

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

1.5.1 Using dumpMsg()

The following example shows how to use `dumpMsg()`.

Using dumpMsg()

```
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:

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.

The following figure shows a sample system with three SystemC modules and how they are generated into SoC Designer components that can be brought into SoC Designer.

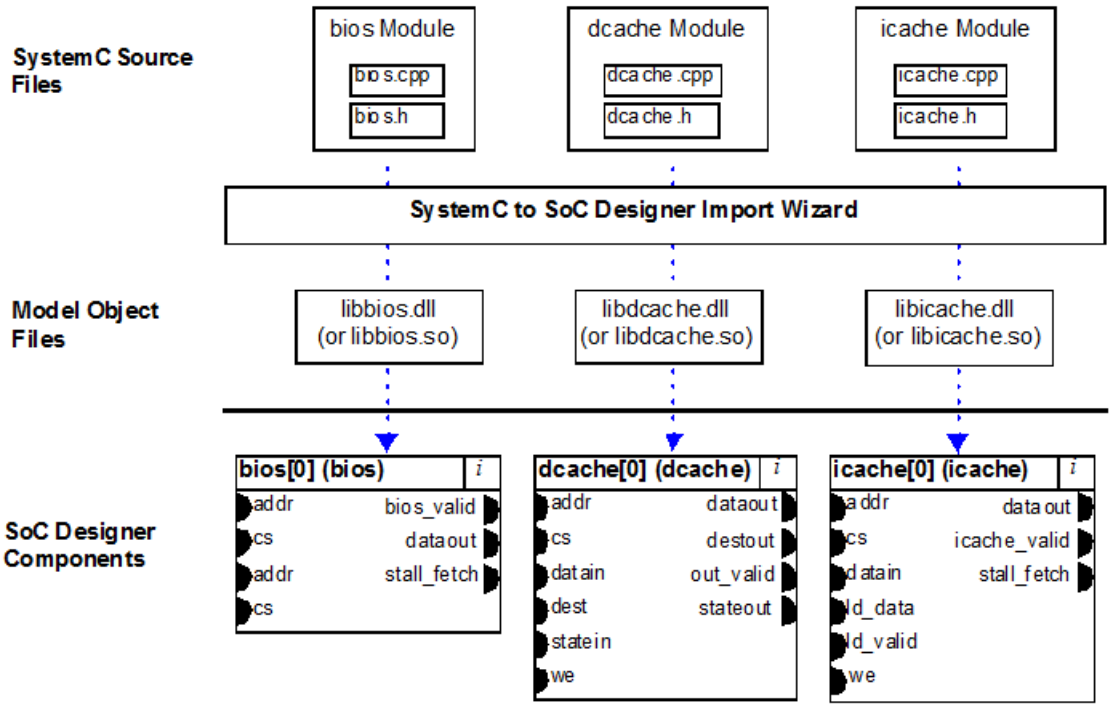

Figure 2-1 SystemC Import Wizard Sample File Generation

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:

- 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* (ARM DUI0953).

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. The top-level `maxlib.conf` file includes the `maxlib.conf` files of each of the imported models for convenience. Register the top-level `maxlib.conf` to register all of the models 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 can be reloaded into the wizard to enable convenient configuration changes without having to reenter 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

The following process describes how you can configure each window of the SystemC to SoC Designer Import Wizard.

1. Launch the wizard from Linux or from Windows. For Linux, use:

   ```
   $MAXSIM_HOME/bin/sys2socd
   ```

   For Windows, navigate to:

   Start > **All Programs** > **ARM** > **SoC Designer** > **SoC Designer SystemC Import Wizard**.

   For Windows, you can also use:

   ```
   %MAXSIM_HOME%\Bin\Release\SC2SoCDImportWizard.bat
   ```

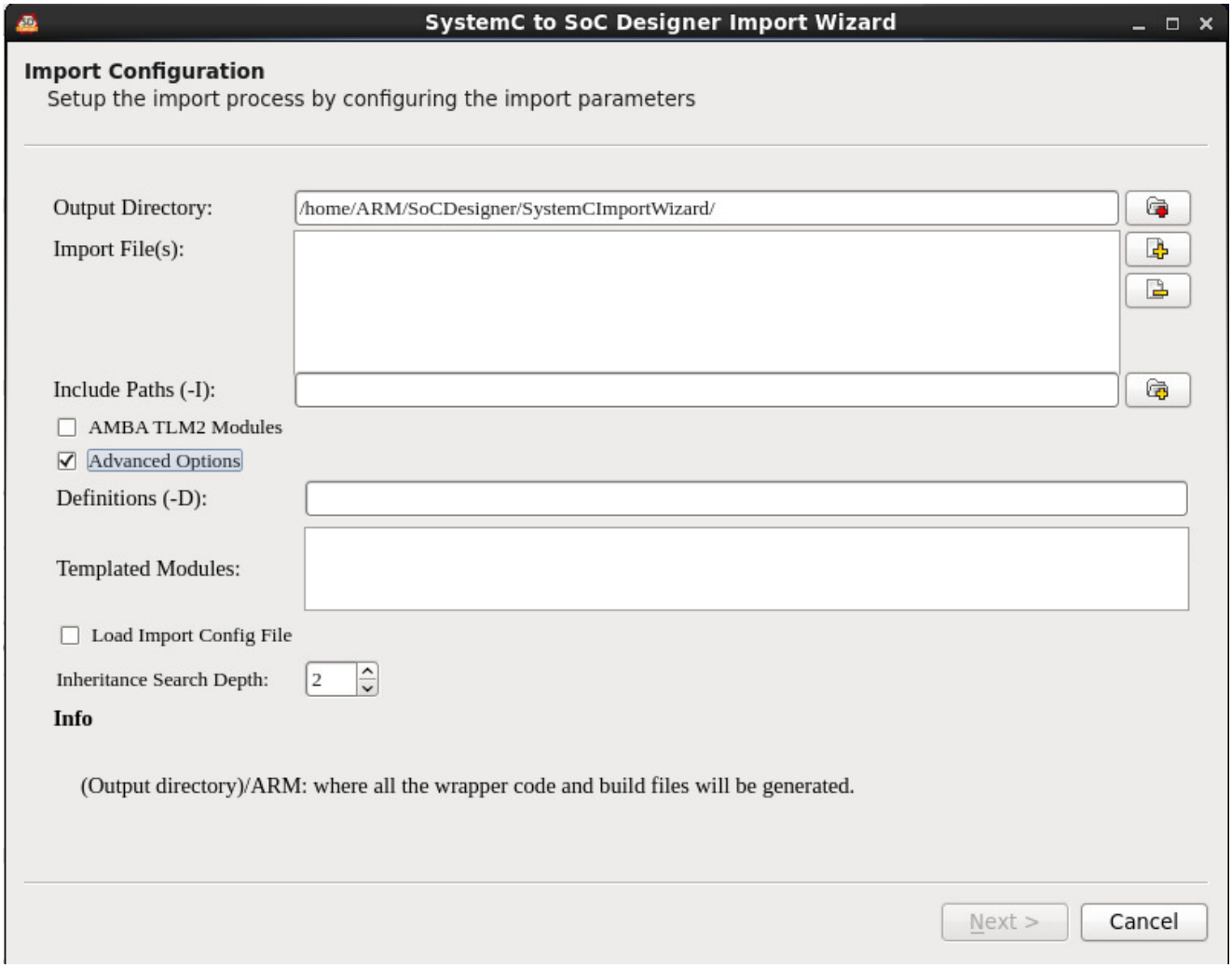

Figure 2-2 Import Configuration page

Table 2-1 Import Configuration Fields

Field	Description
Output Directory	Directory where the wizard output files are generated. The subdirectory `/ARM` is appended to the specified directory.
Import File(s)	Files 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.
Include Paths	Provide the paths that contain the header files referenced from the Import Files.
AMBA TLM2 Modules	Check this box if you are importing AMBA TLM2 models. This automatically adds the include and library paths for amba_socket.
Advanced Options	New options are available when Advanced Options box is checked.
Definitions	Provide any definitions, or compile-time macros, that should be passed to the pre-processor as `-D <value>`.
Templated Modules	This field is applicable when your Import Files contain modules that take template arguments, and no instantiation of the templated module is found in the design (for example, 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, you must provide the template argument value in this field. The Import Wizard 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, you must provide a default value for the template `argument:my_comp<32>`
Load Import Config File	When a component is imported, all of the import configuration settings are saved in the `ARM/impconf.build.xml` file. This file can be reloaded into the wizard to regenerate the component with different configuration settings.
Inheritance Search Depth	The wizard, by default, analyzes up to two levels of class inheritance when looking for SystemC modules, ports, and TLM2.0 sockets. Change the search depth by using the dial provided. ——— **Note** ——— Longer depth usually results in longer time for the wizard to analyze the design. ———

2. 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. The following figure shows results once the analyzer goes through successfully.

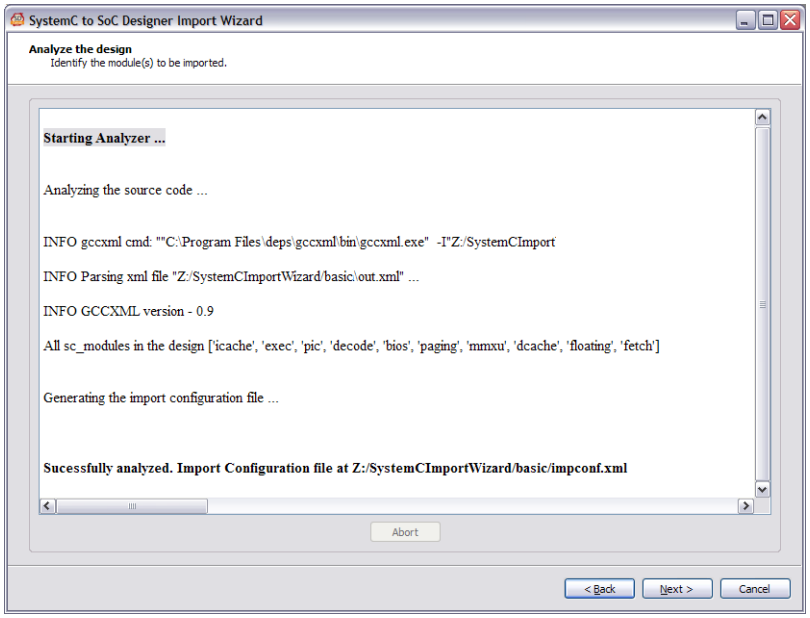

Figure 2-3 Analyzing the design

If the wizard finds any errors during the analysis, it displays the error and opens the source code where the error is detected and highlights the line.

In the following error, an include file cannot be found. You can either go back and fix the error (add to **Include Paths**), or view other errors by clicking the error messages in the bottom half of the screen.

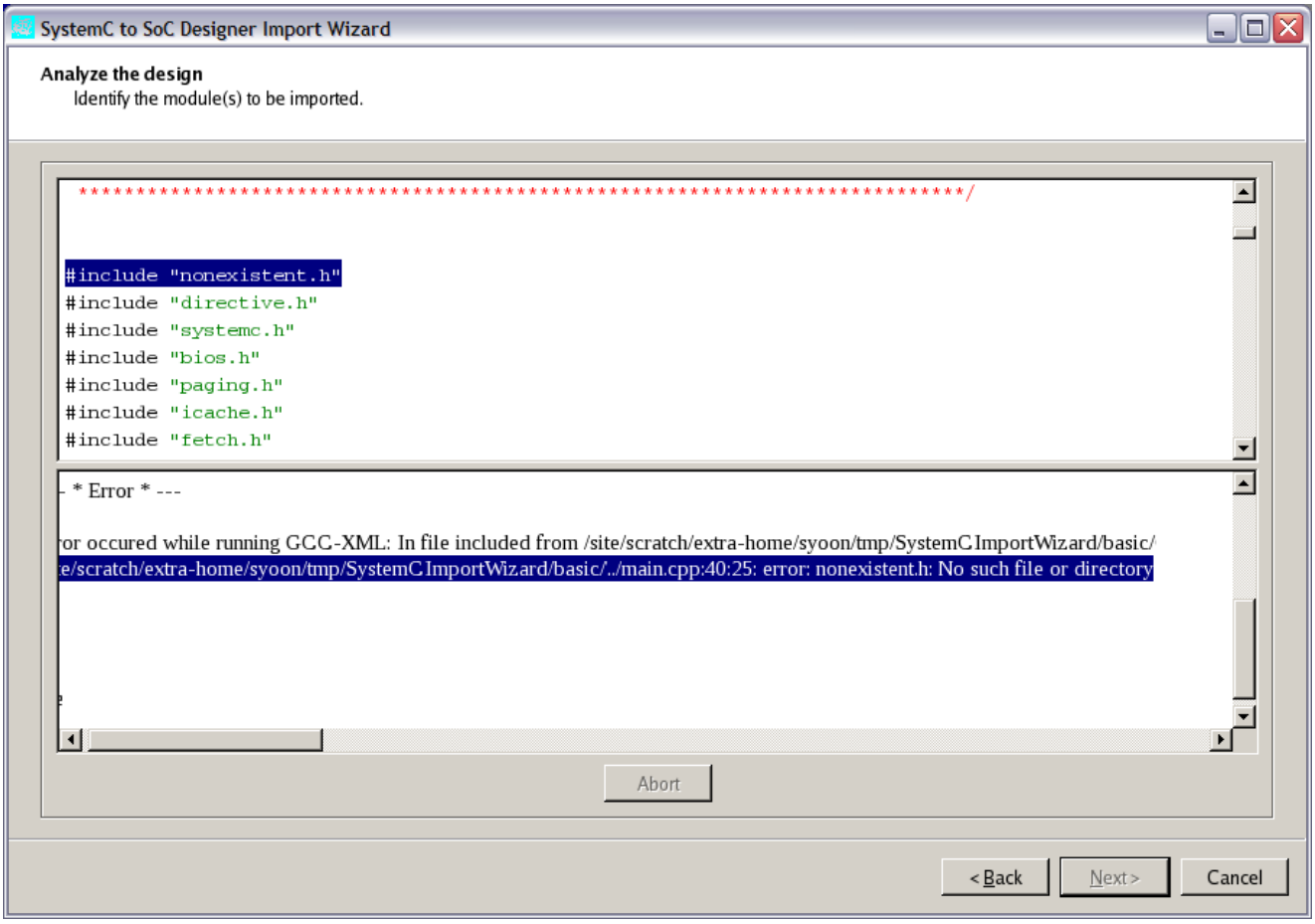

Figure 2-4 Viewing Errors

3. Once the design has been successfully analyzed, click **Next** to continue.
   In the next step, the wizard displays all of the `sc_modules` that were found.

   Note

   You can leave any of the modules out by unchecking the box next to the module. Also, you can give a different name for the module by editing the component name as shown in the following figure.

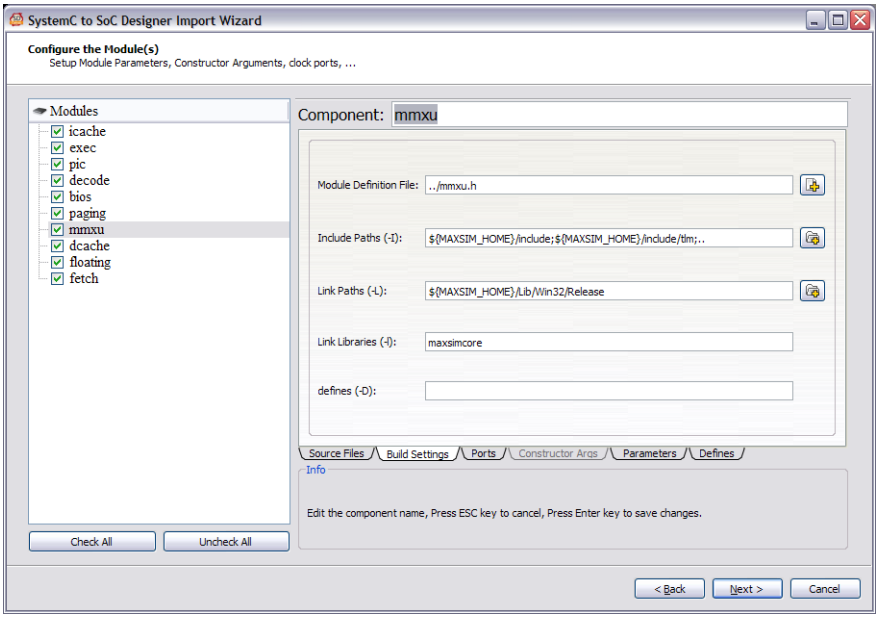

Figure 2-5 Renaming the component

The following table describes the purpose of each tab on the Configuration dialog.

Table 2-2 Component Configuration

Tab Name	Description
Source Files	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.
Build Settings	Compiler and Linker flags. **Note** SoC Designer-specific flags are pre-filled.
Ports	Displays the ports that are registered with SoC Designer. When the *Clk?* option is checked, the clock input port of the model is not registered as a port, but it is directly hooked up to the SoC Designer system clock.
Constructor Args	This tab lists the constructor arguments for your component (if any). For convenience, the wizard turns constructor arguments into SoC Designer component parameters.
Parameters	You can add custom parameters for the generated model. Filling in the parameter information in the wizard generates 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 pre-fills the parameter configuration table with each entry mapping to the template arguments. In the *Allowed Values* column, you may enter the allowed values, separating each with a comma.
Defines	You can add any custom definitions in this tab.

4. Click **Next** once you have finished configuring the component generation. This starts the build process.

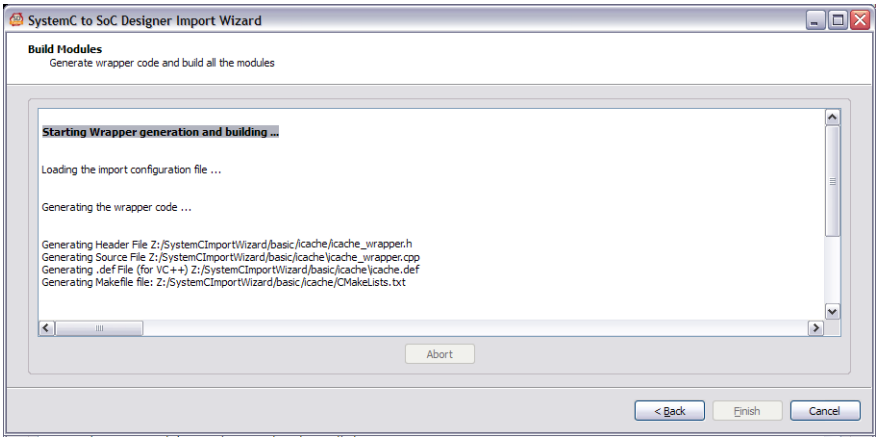

Figure 2-6 Build process

5. Click **Finish** when the build has finished successfully.

Note

If you make any changes and need to rebuild the components, 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:

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.

   Entry point definition example

   ```
   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 the following example.
   - The `createInstance()` function.

   Factory class example

   ```
   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());
   };
   ```

   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

The following section describes a dpipe example and the source files for the SoC Designer version of the dpipe system.

3.2.1 Original dpipe example

The following figure 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:

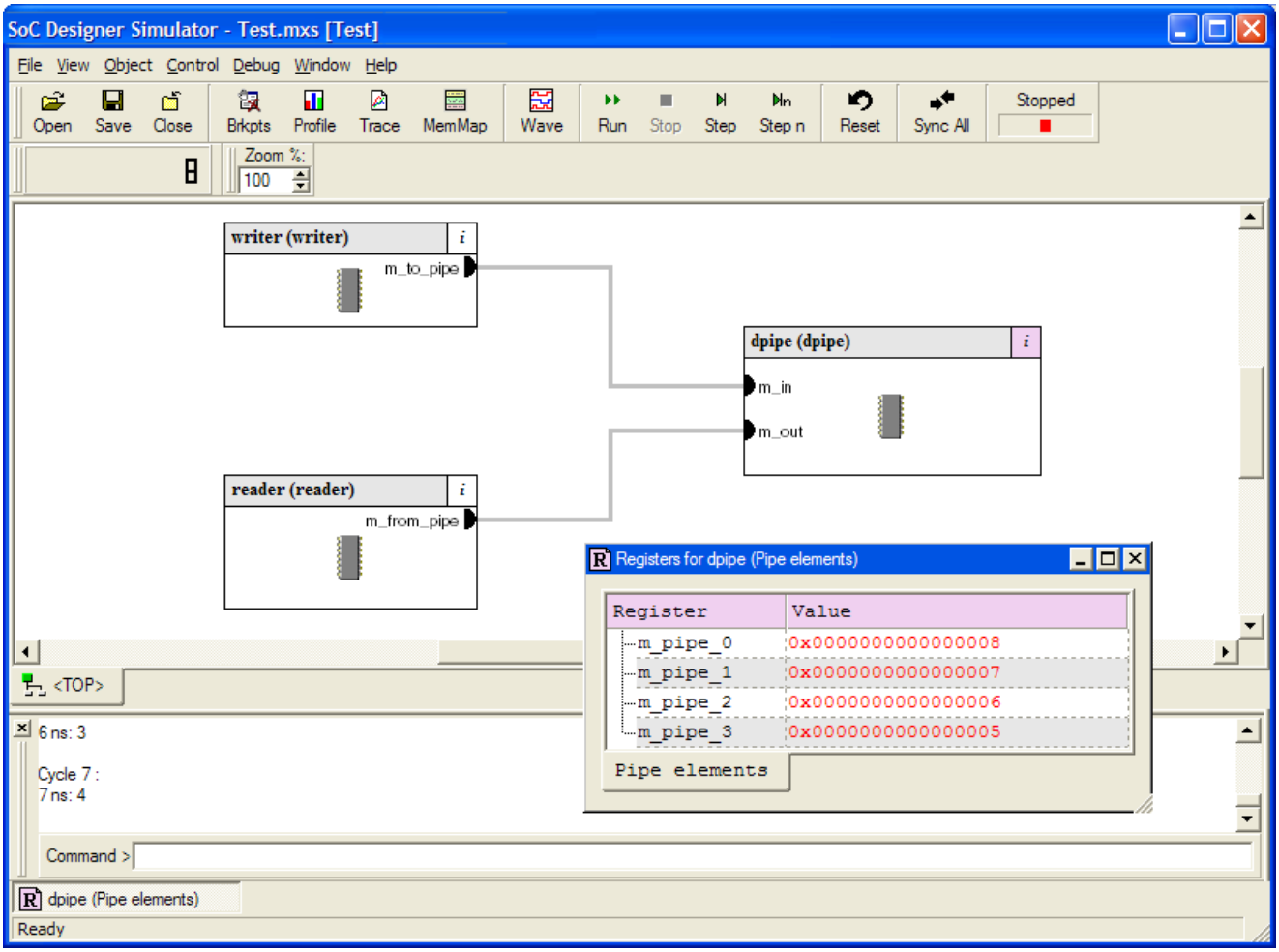

Figure 3-1 dpipe example in SoC Designer

Note

This figure also shows a Register window open. Displaying the content of registers requires that the `dpipe` component implements a CADI interface for the registers.

3.2.2 Source files of the dpipe system

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 the following figure:

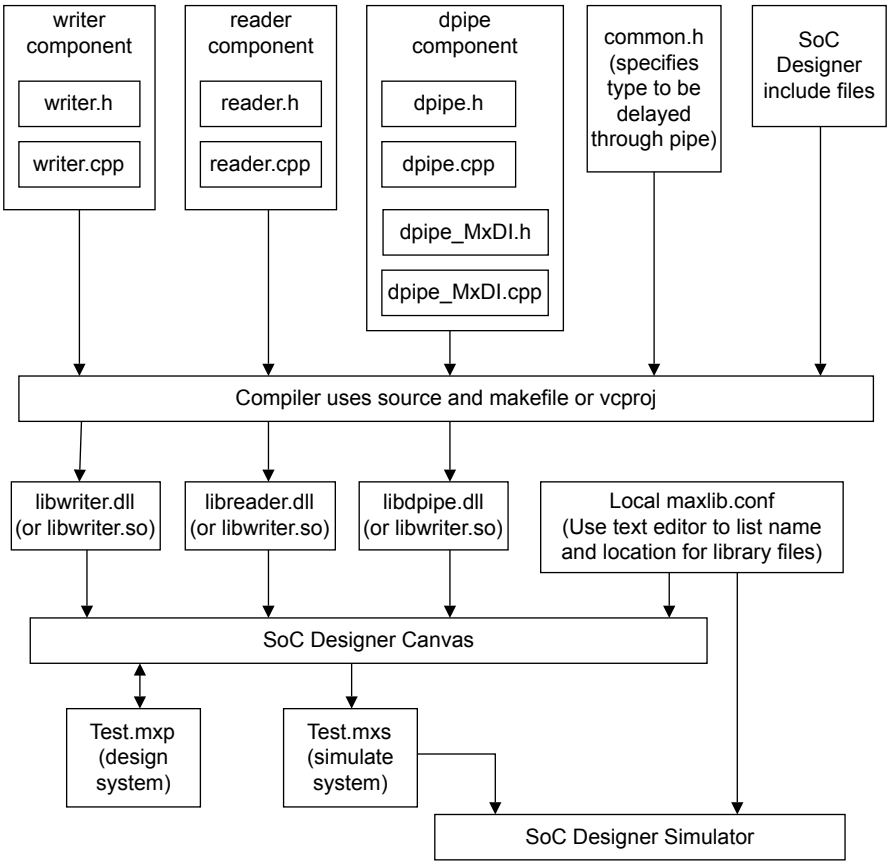

Figure 3-2 Files used in the SoC Designer dpipe system

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.

3.3.1 sc_mx_import_module class implementation

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 and the location of the `sc_mx_import_module` class in the hierarchy are shown in the following figure:

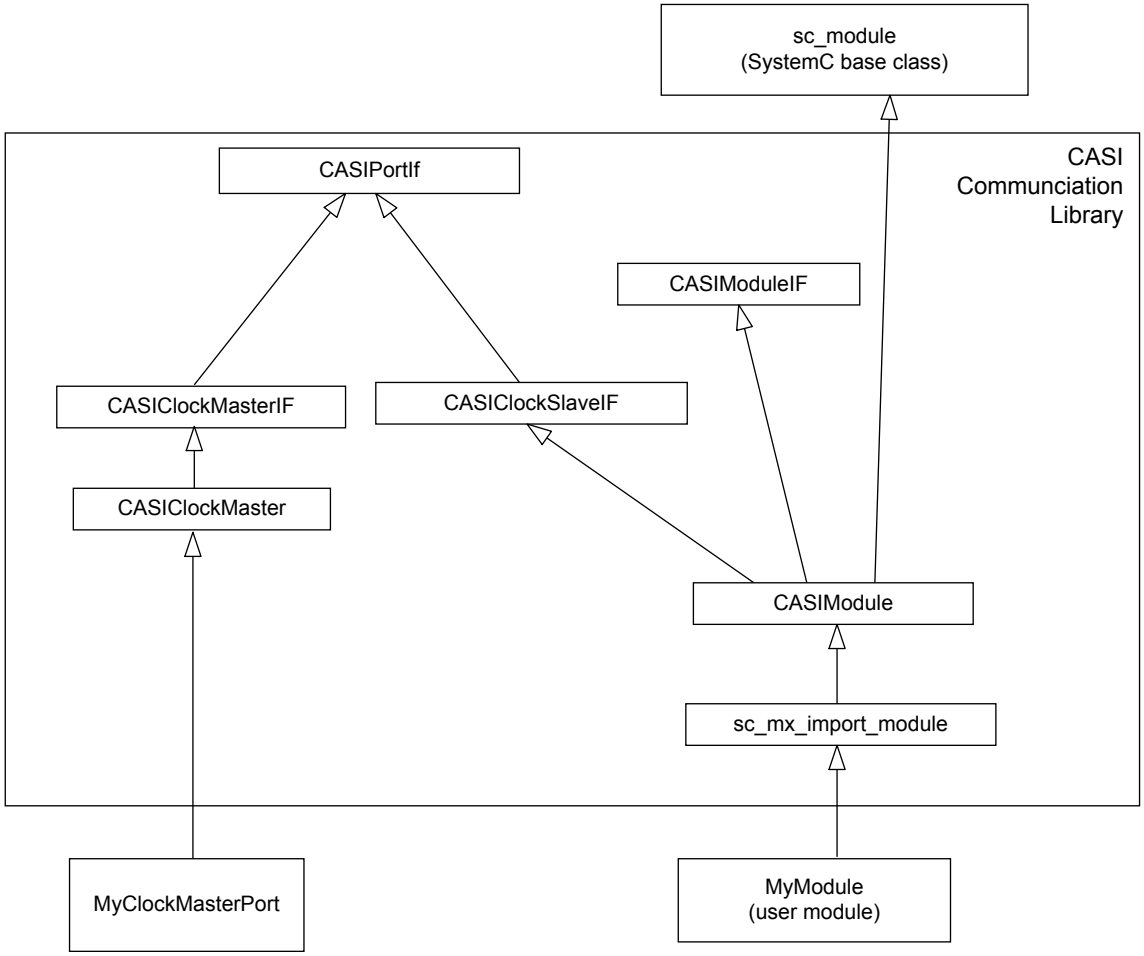

Figure 3-3 Class hierarchy for SystemC import

The `sc_mx_import_module` class header is listed in the following example. 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.

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* (ARM DUI0953) 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.

```
#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 SC_CTOR macro defined in the SystemC language.

```
#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 SC_CTOR 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 the following example.

SC_MX_FACTORY macro example

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

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:

Procedure

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 the following figure.

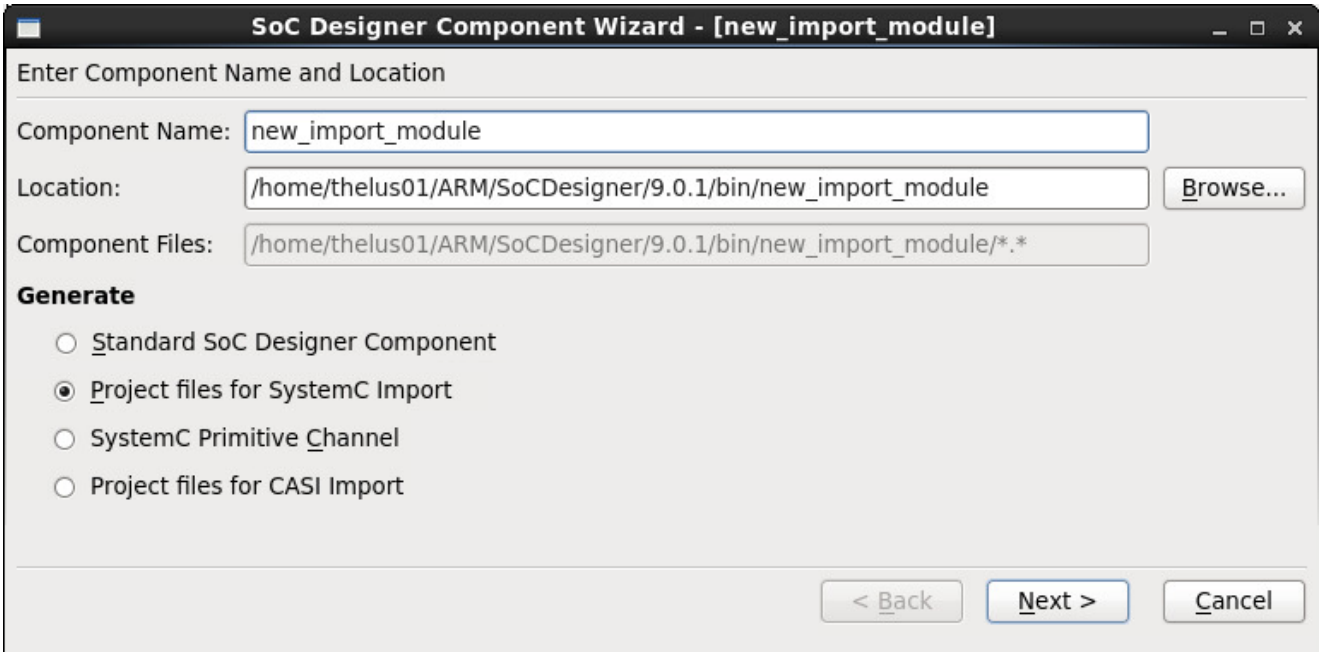

Figure 3-4 Component Wizard for SystemC project file generation

3. Enter the desired model name and the path location for the `Makefile`.
4. The next dialog 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 contain only 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.

This section contains the following subsections:

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.

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

`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.

`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 use the interface.

`SCGenericSlave` ports are represented by a semi-circle that points inwards towards the channel (see the following figure).

`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.

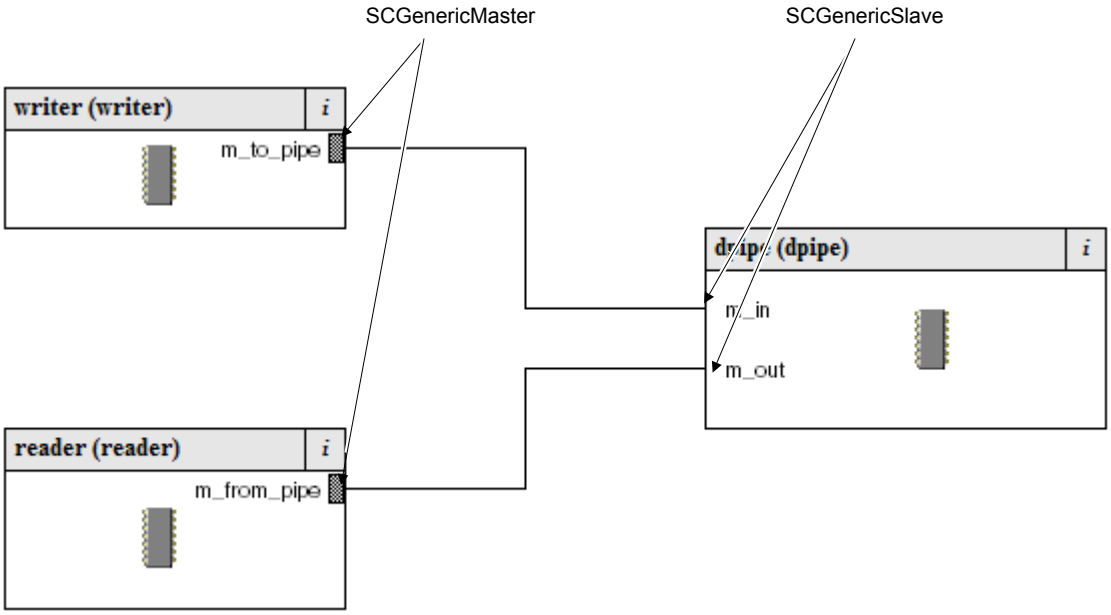

Figure 3-5 SCGeneric ports in the Canvas diagram window

The only valid connection routes are from an `SCGenericMasterPort` to a `SCGenericSlavePort`. SoC Designer 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 registers 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 to complete the port registration.

3.4.2 Using the sc_prim_channel model library

A library of SystemC built-in primitive channels is 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 the 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 if you select **Manage Model Library** from the **Tools** menu:

1. In the SoC Designer Preferences window, click 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 the following figure.

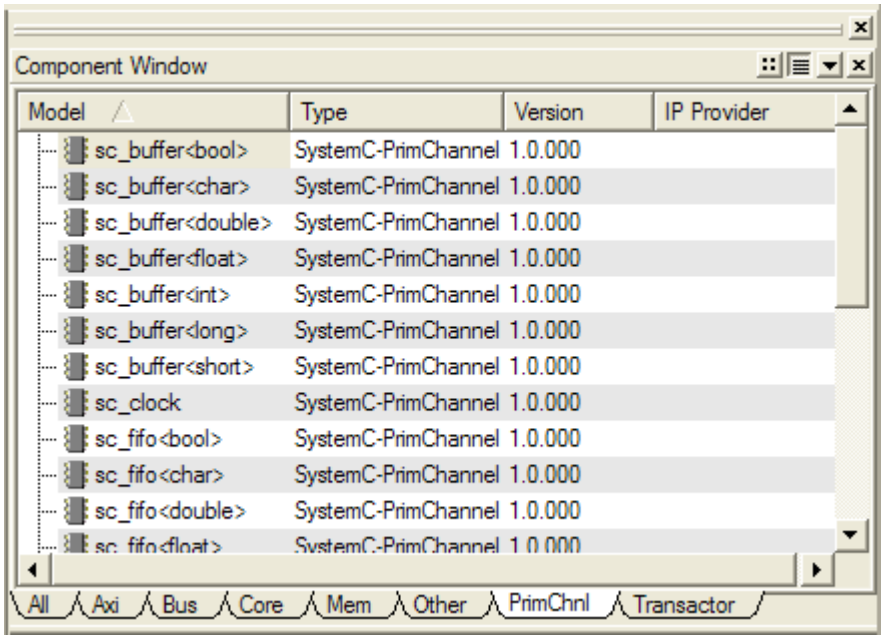

Figure 3-6 List of primitive channel components

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 a unique appearance in SoC Designer Canvas. A few of the built-in primitive channel components are shown in the following figure.

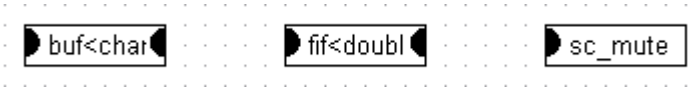

Figure 3-7 Primitive channel component displayed in the Diagram window

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 repositioned using SoC Designer Canvas editing tools just like any other SoC Designer components.

All available built-in primitive channel components are listed in *A.2 Built-in primitive channels* on page Appx-A-71.

Related references

A.2 Built-in primitive channels on page Appx-A-71.

3.4.3 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:

Procedure

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 the following figure.

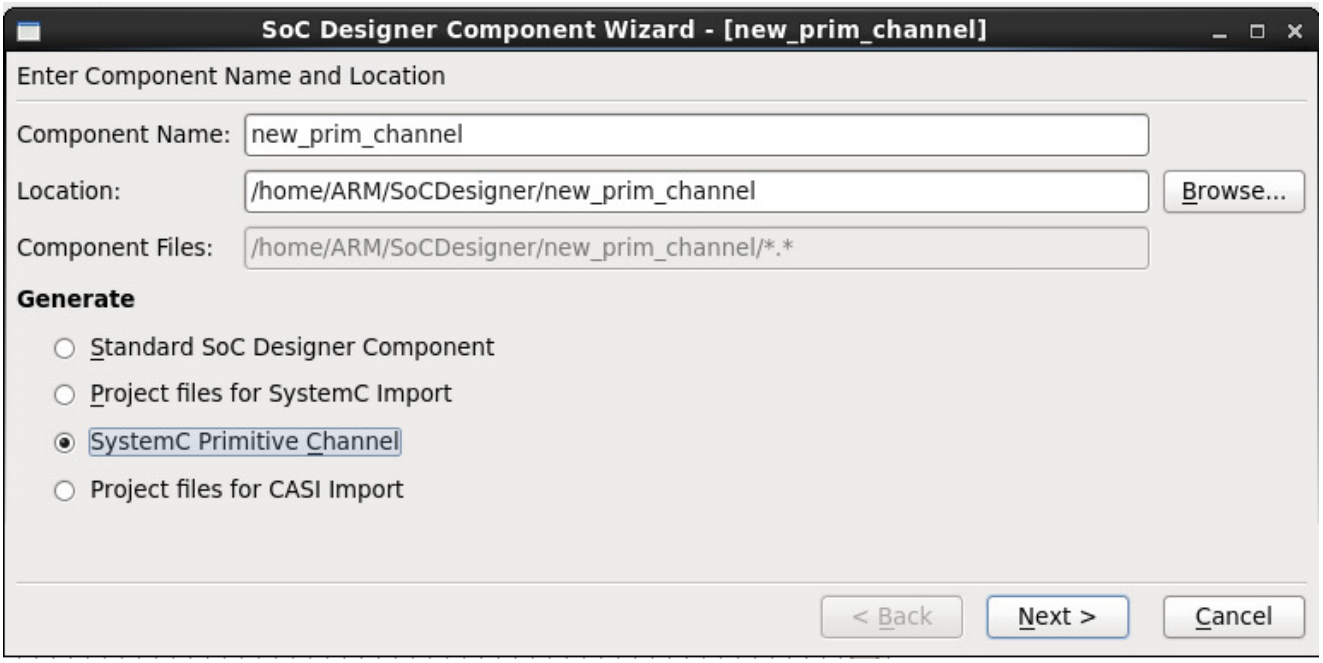

Figure 3-8 Component Wizard for primitive channel

2. Select the channel type and declare the data type for that channel as shown in the following figure. The **sc_lv Width** field is enabled when the **Channel Type** is “sc_signal_rv”. If you are defining this type of channel, enter the width of the signal in the **Data Type** field.

   ———— **Note** ————

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

   ————————————

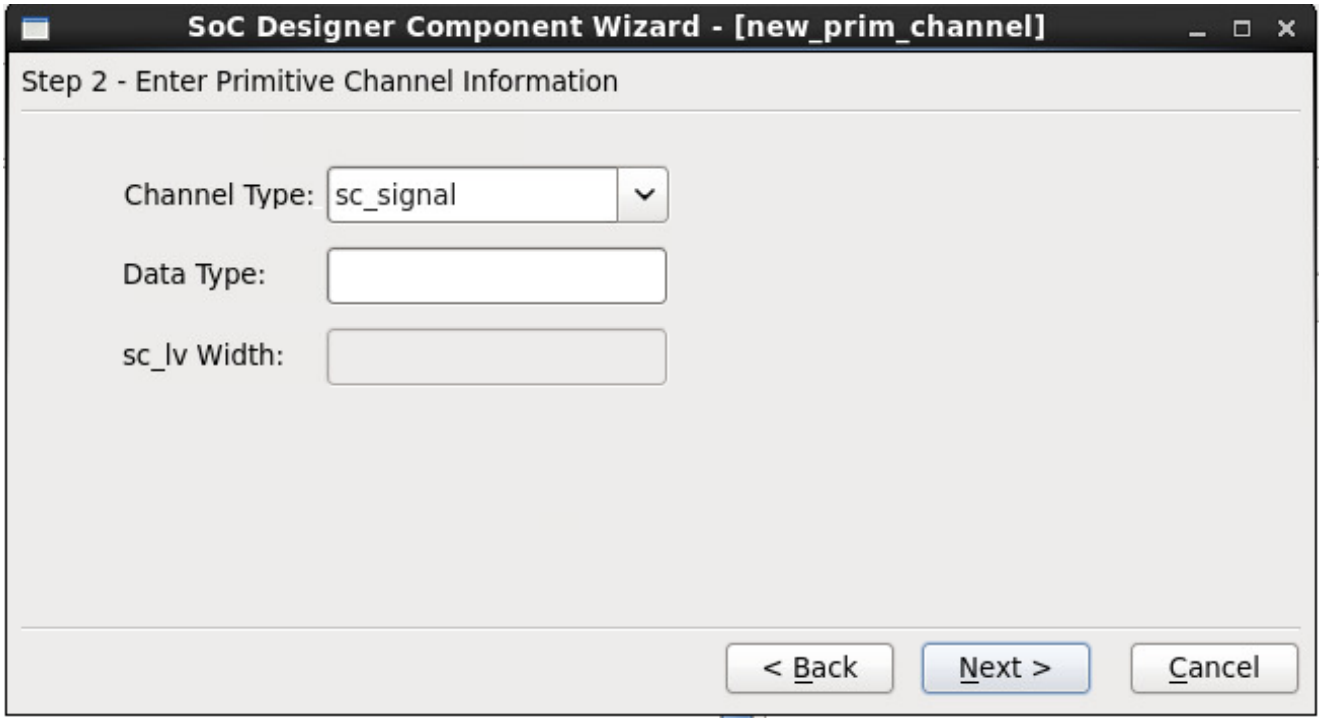

Figure 3-9 Specify channel and data type

Click **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.

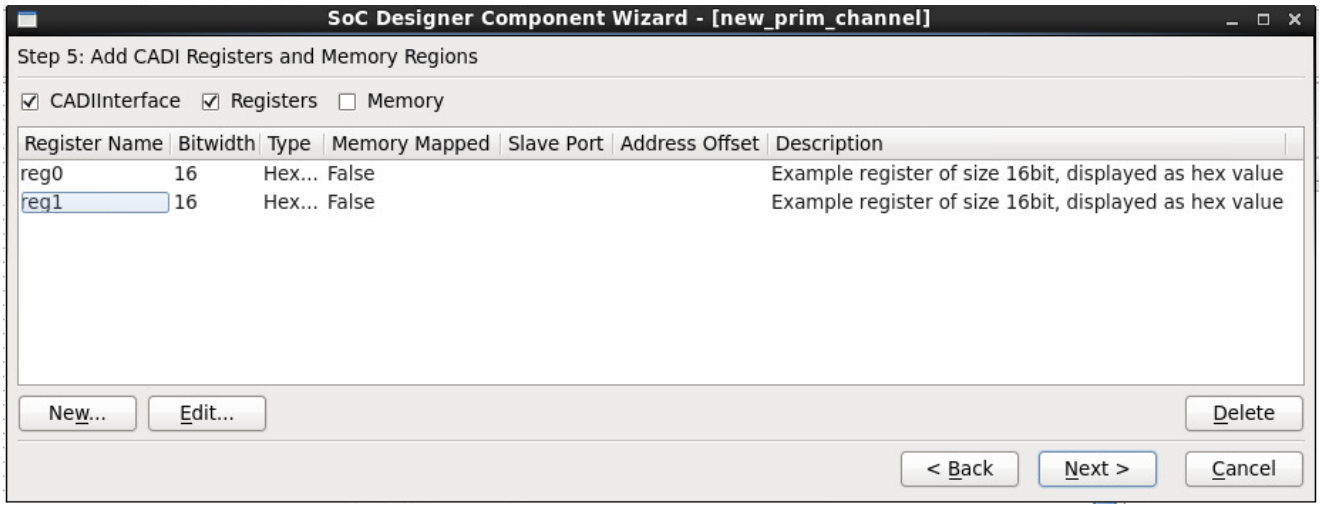

Figure 3-10 Add CADI registers and Memory regions

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

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

4. Click **Next** to accept the CADI interfaces and proceed to the summary.
5. On the Summary dialog, click **Finish** to generate the sources for the new primitive channel component.
6. SoC Designer generates the source files for the new component. After the files are generated, the following dialog prompts you for the action to take with the new source files:

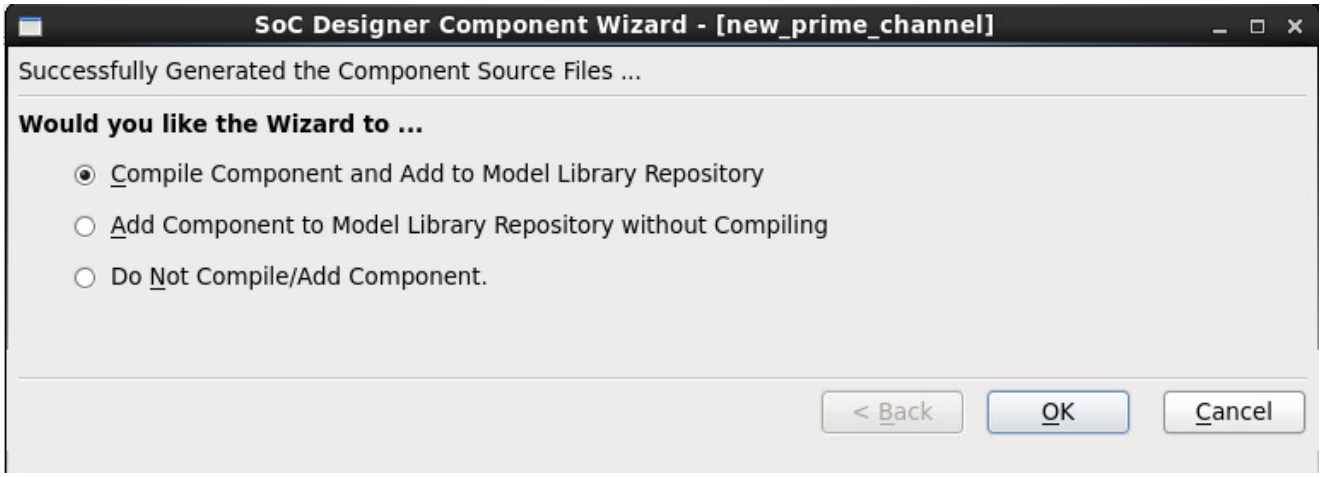

Figure 3-11 Compile component

Caution

If a user-defined data type (anything other than C/C++ or SystemC built-in types) is being used, do not specify **Compile Component and Add**. 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 the following figure). The newly-generated channel by default reports `SystemC-PrimChannel` as the component type that is placed into the **PrimChnl** group in the component window.

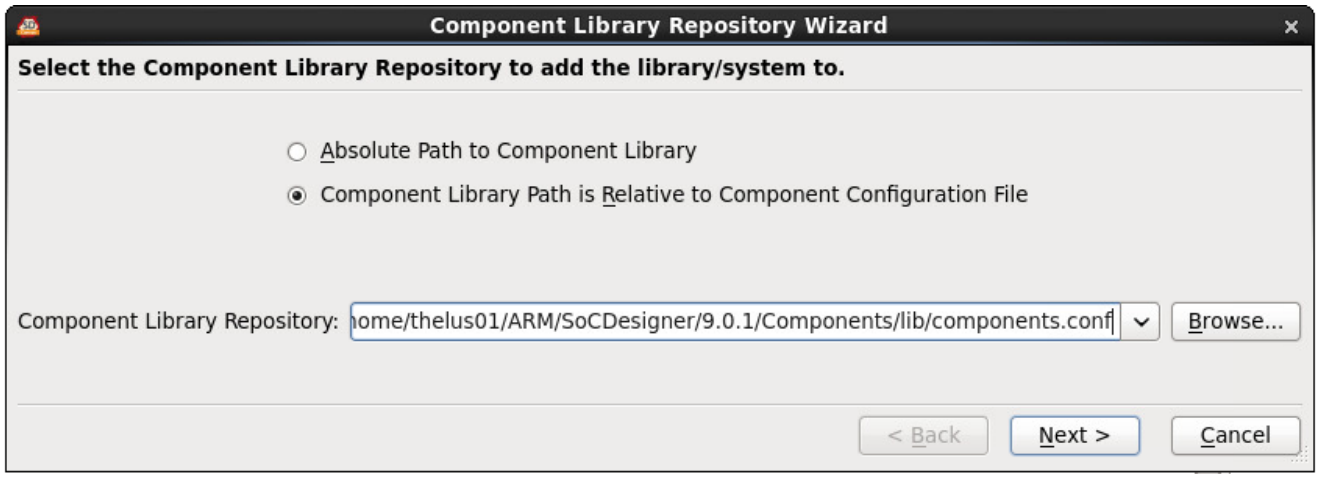

Figure 3-12 Select repository for component

If you selected **Do Not Compile/Add Component**, modify the created files (see the following figure) and use the Manage Library section of the Preferences dialog to add the component type to the component window.

9. Click **Next** to view a summary, then **Finish** to add the new component.

3.4.4 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 the **Port** icon. Use the **SCGeneric** option in the External Port configuration window as shown in the following figure.

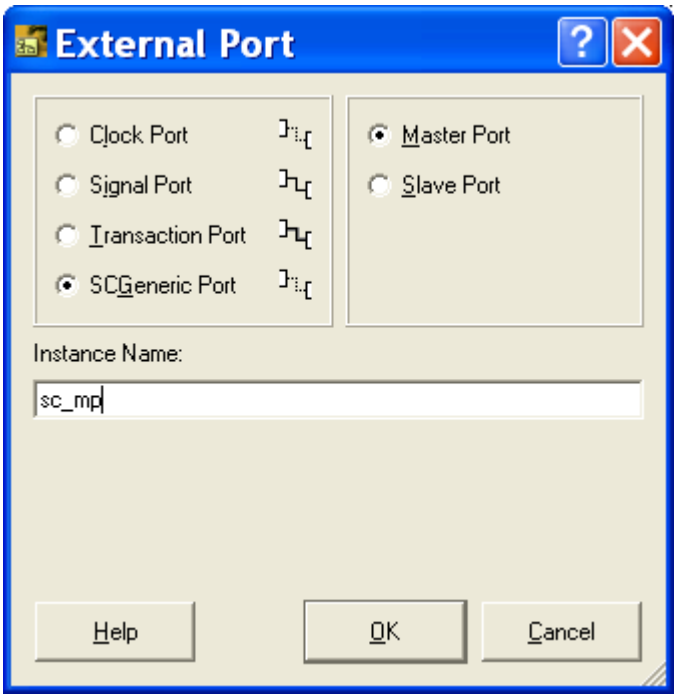

Figure 3-13 SCGeneric External Port dialog

The `sc_export_multi` example supplied with SoC Designer shows a multi-hierarchy system that uses a SystemC external port. See the following figure.

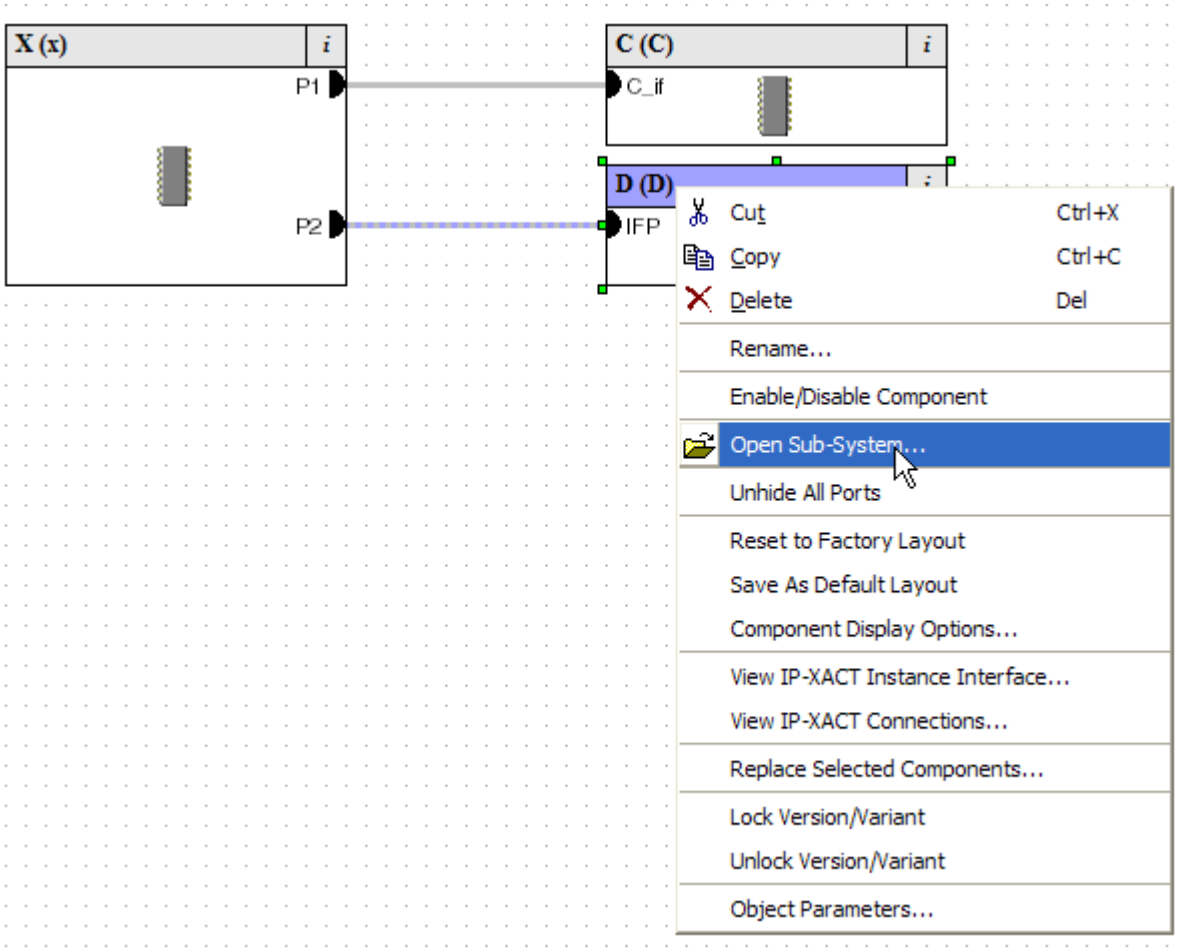

Figure 3-14 System containing a subsystem

Select **Open Sub-System…** from the context menu to display the subsystem that contains an external port. See the following figure.

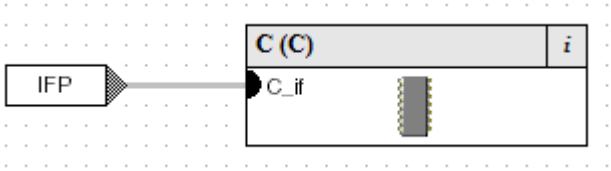

Figure 3-15 Subsystem with external port

3.4.5 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 the following sections:

- *SystemC signal output to SoC Designer signal slave* on page 3-41
- *SoC Designer signal master to SystemC signal input* on page 3-44
- *Clocking SystemC clock ports* on page 3-45

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 the following figure.

- `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.

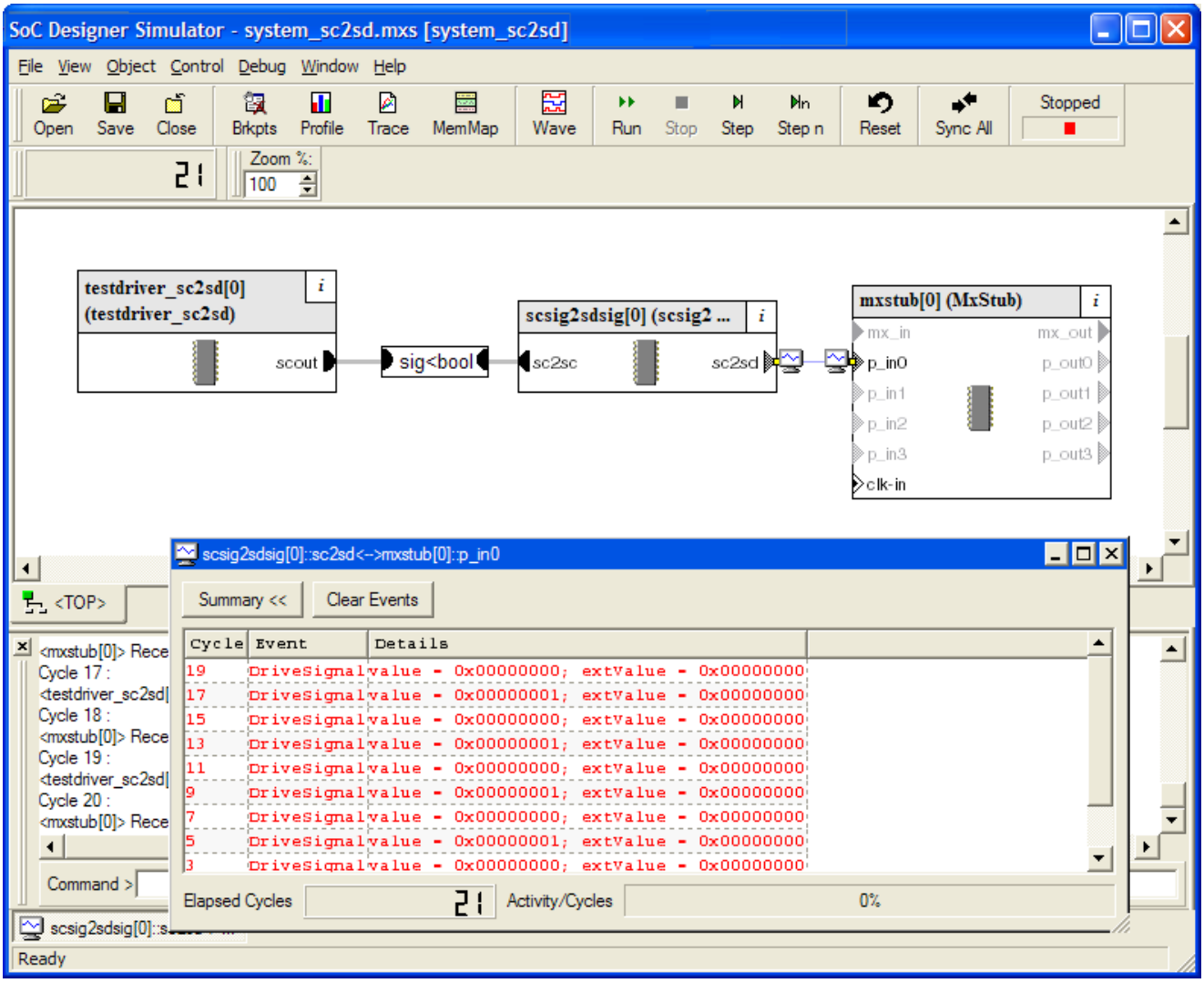

Figure 3-16 SystemC signal to SoC Designer signal component

`scsig2sdsig.h`

The `scsig2sdsig.h` file contains the component class definition as shown in the following example:

scsig2sdsig.h file

```
#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:
    sc_port<CASISignalIF> *sc2sd_SMaster;
    sc_in<bool> sc2sc_SSlave;
};
```

`testdriver_sc2sd.h`

The `testdriver_sc2sd.h` file has the definition for the testdriver component as shown in the following example:

testdriver component

```
#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;
};
```

`scsig2sdsig.cpp`

The `scsig2sdsig.cpp` file includes code to register and initialize the SystemC port as shown in the following example:

Registering the SystemC port

```
#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()
{
    sc2sd_SMaster->driveSignal( sc2sc_SSlave.read(), NULL );
}
void scsig2sdsig::initSignalPort(CASISignalMasterIF* signalIf)
{
    CASISignalProperties prop_sm1;
    memset( &prop_sm1, 0, sizeof(prop_sm1) );
    prop_sm1.isOptional = false;
    prop_sm1.bitwidth = 32;
    signalIf->setProperties( &prop_sm1 );
}
```

testdriver_sc2sd.cpp

The testdriver_sc2sd.cpp file contains code to write to the port as shown in testdriver component example:

Writing to the testdriver SystemC port

```
#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 (scout in testdriver and sc2sc in scsig2sdsig) 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 sig<bool>.

SoC Designer signal master to SystemC signal input

The sdsig2scsig example uses a component that interfaces between SystemC signals and SoC Designer signals as shown in the following figure:

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

Note

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

The sdsig2scsig.h file contains the component class definition as shown in the following example:

sdsig2scsig.h file

```
#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;
};
```

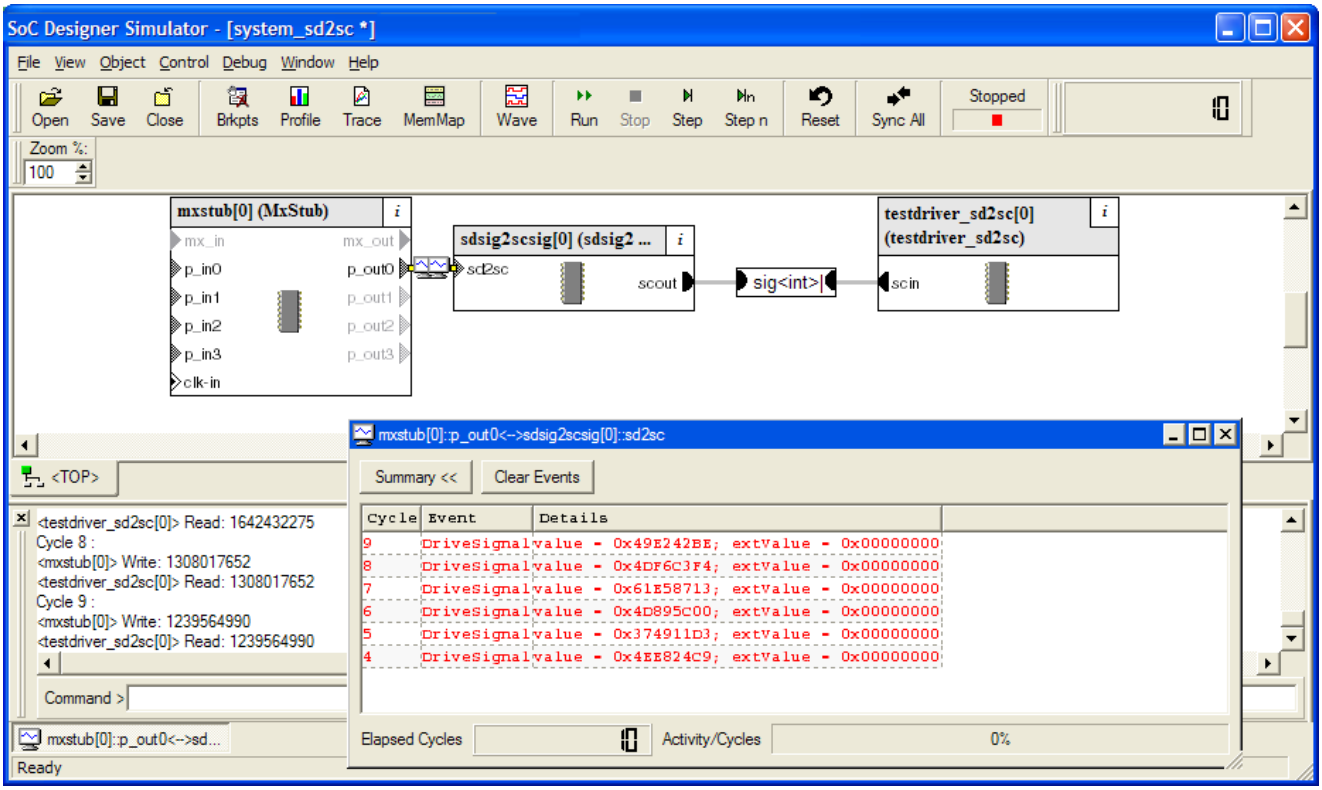

Figure 3-17 SoC Designer signal to SystemC signal

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`.

3.5 Simulation control

This section describes functions and classes related to simulation control.

This section contains the following subsections:

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 runtime. 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).

```
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:

Procedure

1. Select **System Properties** from the **Edit** menu.
2. Use the dialog (see the following figure), to set the SoC Designer system clock period.

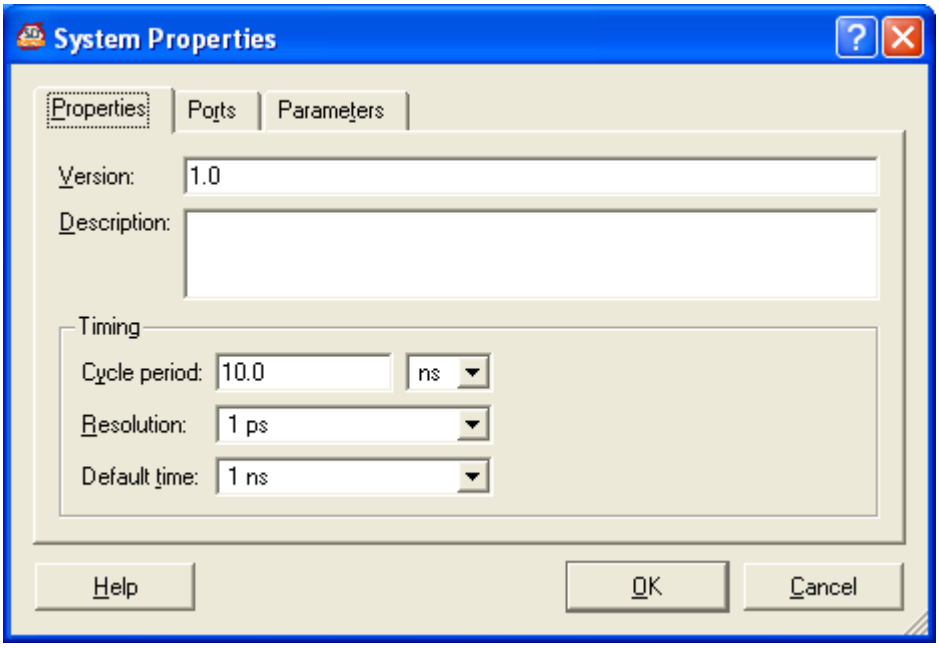

Figure 3-18 Edit System Properties dialog

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 is 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 is 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 `SystemCImport/original` subdirectory) to create the SoC Designer source files.

This section contains the following subsections:

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 *3.3.3 Generating a makefile from the Component Wizard* on page 3-32 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.

Related tasks

3.3.3 Generating a makefile from the Component Wizard on page 3-32.

3.6.2 Interface definition

The following example shows the source code for the interface that is used by the producer, fifo, and reader:

common_if.h

```
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

The following examples show how to convert an existing `sc_module` into a `sc_mx_import_module`

SoC Designer fifo.h

```
#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;
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);
};
```

The `fifo.cpp` file implements the constructor, initialization, and CADI functions for SoC Designer Module as shown in the following example.

fifo constructor and CADI

```
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()
{
    return cadi;
}
```

Add the `SC_MX_FACTORY` macro to `fifo.cpp` to define the `fifo` SoC Designer Module factory class and provide an entry point into the `fifo` Model DLL as shown in the following example.

fifo factory

```
SC_MX_FACTORY(fifo)
```

fifo CADI

The CADI interface accesses the fifo data structure as a memory block. The `fifo.cpp` file contains a `read_mxdi()` function as shown in the following example:

Reading the fifo data from CADI

```
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 ];
}
```

Note

Unlike the `read()` function, `read_mxdi()` does not alter the fifo data or element counters.

The `CADIMemRead()` function in `fifo_MxDI.cpp` uses read_mxdi() as shown in the following example:

CADIMemRead()

```
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

The following examples show how to convert an existing producer `sc_module` into a `sc_mx_import_module`.

SoC Designer simple_fifo producer

```
// 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 the following example.

producer factory

```
#include “producer.h”
SC_MX_FACTORY(producer)
```

Related references

3.6.5 consumer component on page 3-52.

3.6.5 consumer component

This section describes the code that implements the consumer component.

Note

See *3.6.4 producer component* on page 3-52 if you want to convert an existing `sc_module` into a `sc_mx_import_module`.

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 the following example:

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 );
    }
}
// consumer factory class + entry point provision
SC_MX_FACTORY(consumer)
```

Related references

3.6.4 producer component on page 3-52.

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 the following figure:

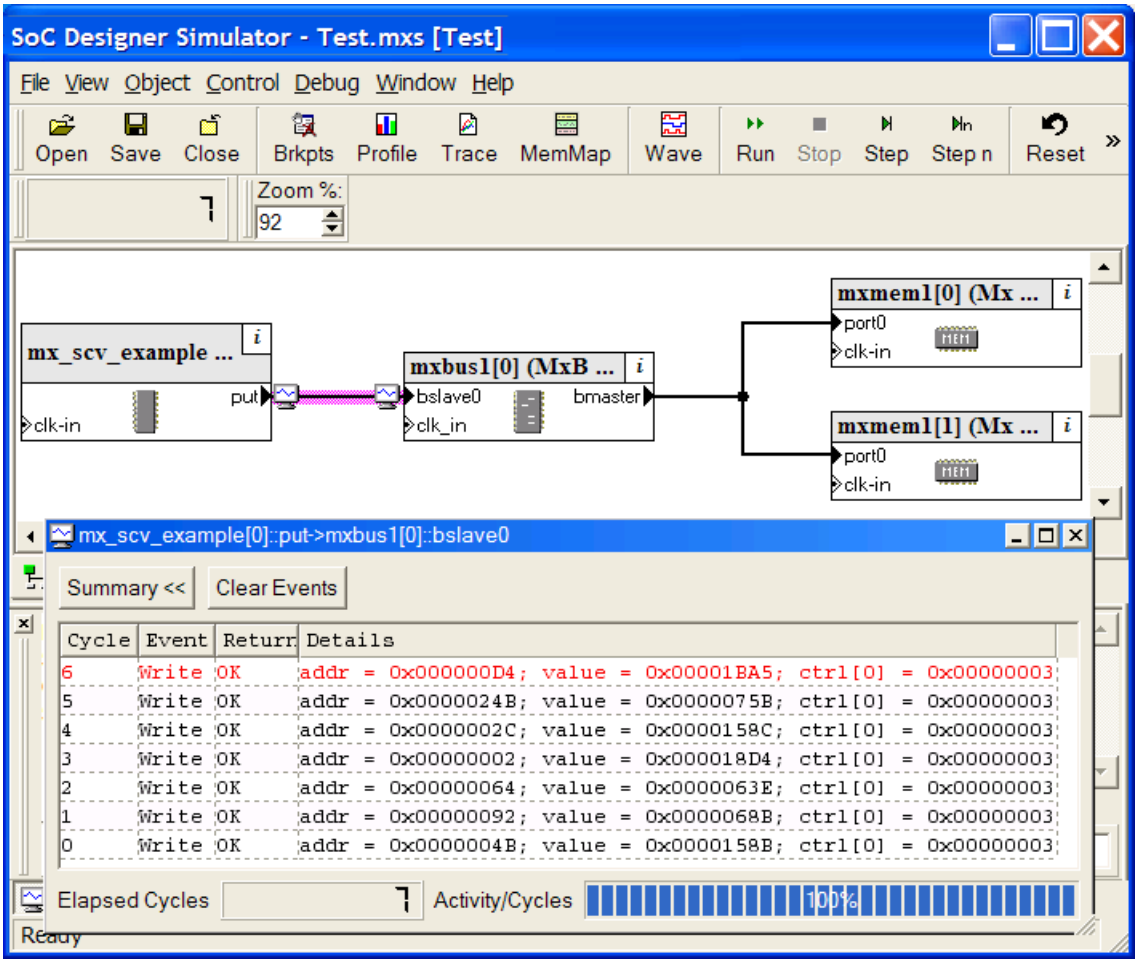

Figure 3-19 fifo simulation

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 `SystemCImport/dpipe/original` subdirectory) to produce the SoC Designer source files.

This section contains the following subsections:

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 *3.3.3 Generating a makefile from the Component Wizard* on page 3-32 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.

Related tasks

3.3.3 Generating a makefile from the Component Wizard on page 3-32.

3.7.2 dpipe source

This section describes the changes made to the dpipe component.

Changes to the class definition

The following example shows the use of `sc_mx_import_module` and `registerSCGenericSlavePort()` in `dpipe.h`.

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 the following example. The type for `atom` is defined in the `common.h` file.

The dpipe factory

```
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.

Related references

3.5.2 Clocking the SystemC modules on page 3-46.

CADI support

The `dpipe.cpp` file includes code to create and delete the CADI interface as shown in the following example:

CADI in dpipe

```
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 the following example:

getProperty()

```
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 );
    }
    return "";
}
```

The files `dpipe_MxDI.h` define the CADI classes and functions for the dpipe component as shown in the following example:

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* (ARM DUI0956) for implementation details.

3.7.3 reader source files

The following examples show the converted code for the reader component.

reader.h for SoC Designer

```
#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 the following example:

reader.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
    m_clk( *( getMxSCClock() ) );
}
// reader factory class + entry point for SoC Designer
SC_MX_FACTORY(reader)
```

3.7.4 writer source files

The following examples show the converted code for the writer component.

writer.h for SoC Designer

```
#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.
    sc_inout<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 the following example:

writer.cpp

```
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 **Step** to demonstrate the flow through the delay pipe as shown in the following figure:

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.

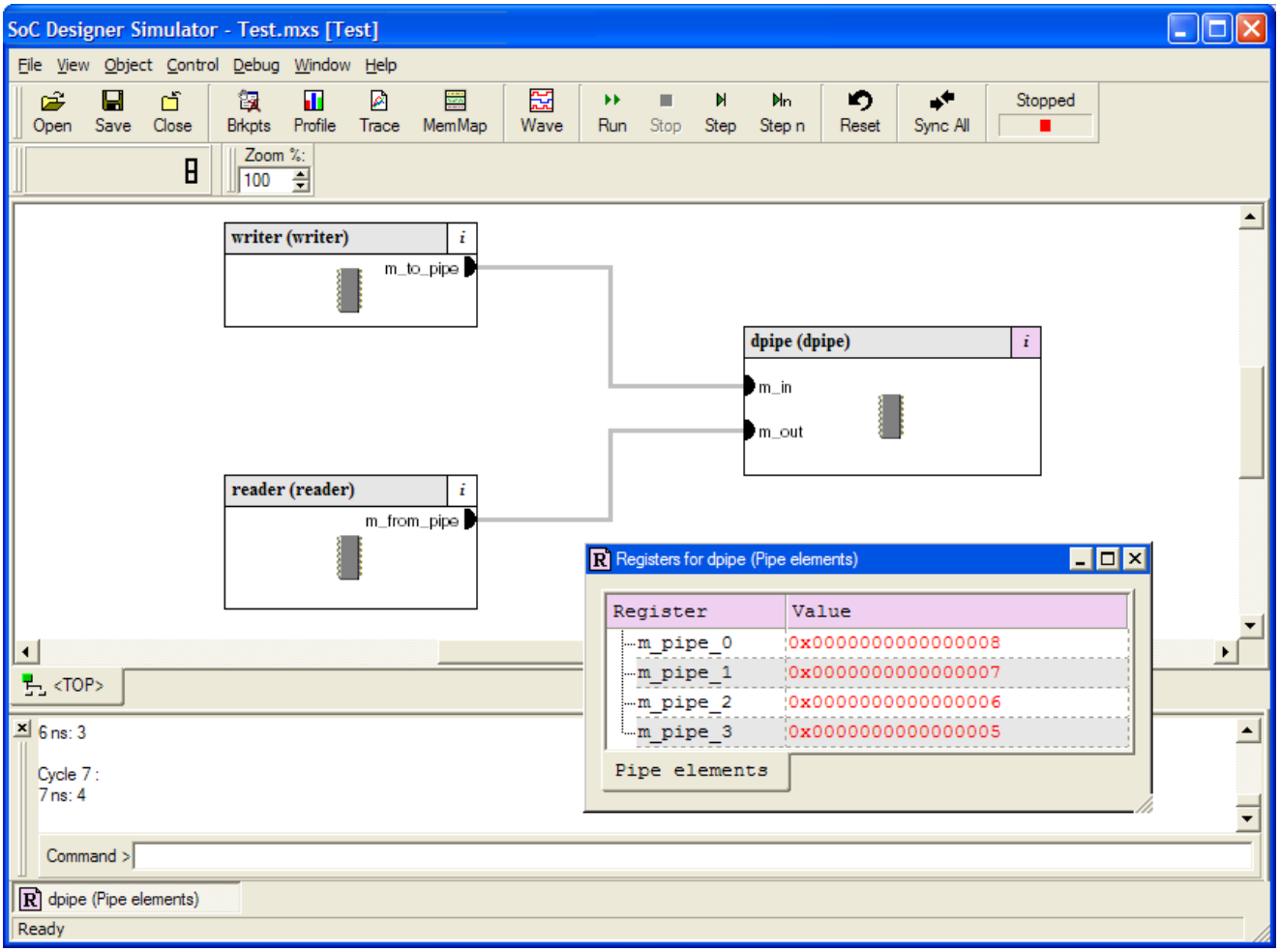

Figure 3-20 dpipe simulation

The dpipe component implements CADI for the internal registers and the register contents can be displayed in the Register debug window.

Chapter 4
Generating a Wrapper for SystemC Models

This chapter describes how to use the Component Wizard to generate a wrapper that simplifies the import of a SystemC model.

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.

It contains the following sections:

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:

Procedure

1. Select **Component Wizard** from the **Tools** menu.
   The dialog is displayed as the figure shows.

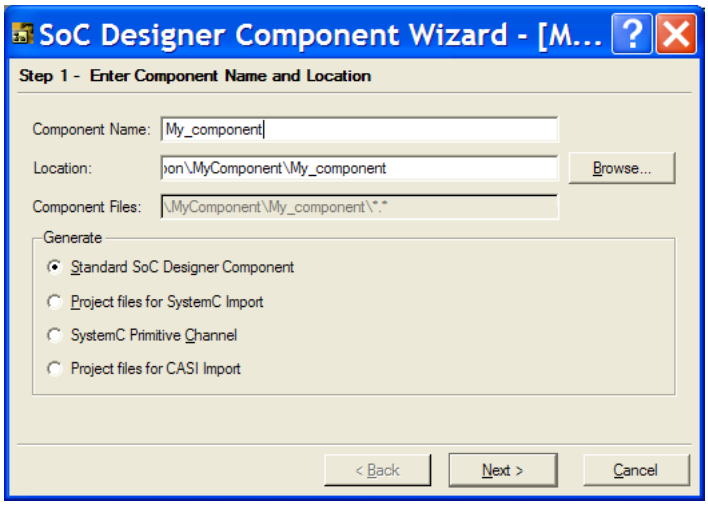

Figure 4-1 Component Wizard dialog

2. Enter the settings for the component.
3. Enter the name of the component.
4. Enter the directory location.
5. Select the **Standard SoC Designer Component** radio button.
6. Click **Next**.
7. The Component Wizard guides you through the generation of the files for the top-level SoC Designer component.
8. 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:

- 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 in this list 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 you try 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.

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.

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 the `SC_METHOD` and any triggering of these methods is done on the rising edge of the clock.

To use the SoC Designer cycle-based clocking mechanism:

1. Modify the generated SoC Designer SystemC component to make it clocked by adding `Communicate` and `Update` functions, registering the clock port, and registering the component with the scheduler (calling `registerClockSlave()`). See the *SoC Designer User Guide* (ARM DUI0956) 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:

```
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.

Related tasks

The getMxSCClock() function on page 3-46.

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. 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.

   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.

   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* (ARM DUI0956).
4. Use the SoC Designer Canvas tools to connect the CASI ports to other SoC Designer components.

   The following figure shows the connection structure for translating user-defined SystemC ports from the imported modules to the generated ports in the generated wrapper module.

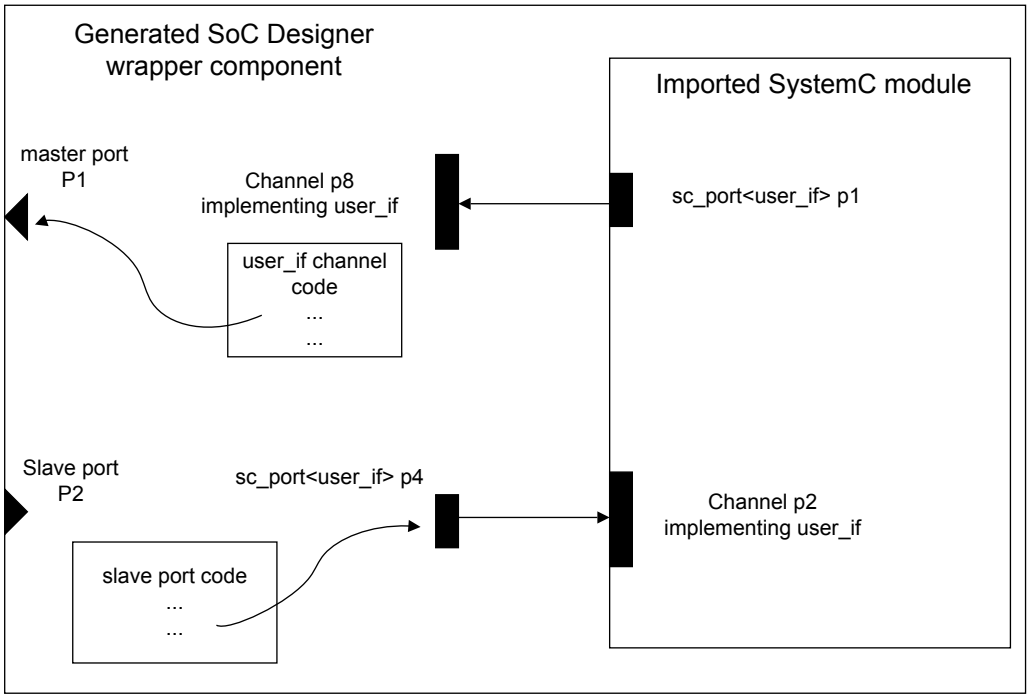

Figure 4-2 Port connection structure for an imported SystemC module

Chapter 5
Modeling Guidelines for SystemC

This chapter contains modeling guidelines to improve performance of SystemC models in SoC Designer.

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 event-driven features such as `SC_THREAD` or `SC_METHOD` with a sensitivity list.

It contains the following section:

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 to low-level hardware simulation. If simulation performance is 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 generally a straightforward task.

Models that are only operating occasionally and do not require continuous clocking can unregister themselves from the clock after an operation has completed. When 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:

A.1 Cycle Model-specific SystemC Implementation Differences

Table A-1 SystemC feature set

Feature	Description	Support in SoC Designer
Support for programs with own `main()` function	Support for user `main()` through SystemC `sc_main_main()` function.	Not supported. Components are loaded as `.dll`s controlled by SoC Designer, and should not have a `main()` function.
Getting copies of the start-up arguments	SystemC enables access to start-up arguments through `sc_argv()` function.	Not supported. Component parameters should be used instead. Parameter values can be passed to SystemC modules from the SoC Designer wrapper.
Object code release tagging	Binary interface compatibility checking.	Customized vendor tag and version are encoded in the SoC Designer SystemC library. Link against this library if importing SystemC modules.
`sc_cycle()`	`sc_cycle()` enables running the simulation for a fixed number of cycles, but its use was deprecated in SystemC v2.2.	`sc_cycle()` and `sc_start(0)` are not supported in SoC Designer. Use the control buttons in SoC Designer Simulator for simulation control.
`sc_start(0)`	`sc_start(0)` performs delta cycles based on pending events and assignments.	`sc_start(0)` is not supported in SoC Designer. Simulation control granularity is limited to one system clock cycle.
`sc_stop()` semantics change for `sc_set_stop_mode()`	`sc_set_stop_mode()` defines two modes of `sc_stop()`: `SC_STOP_IMMEDIATELY` `SC_STOP_FINISH_DELTA`.	`sc_stop()` and `sc_set_stop_mode()` are supported in SoC Designer.
Calling `sc_start()` after `sc_stop()` is an error	After `sc_stop()` has been called `sc_start()` produces an error message	SoC Designer issues an error message if you try to continue execution after an `sc_stop()`.
Calling `sc_stop()` after `sc_stop()` produces a warning.	After `sc_stop()` has been called, another call to `sc_stop()` produces a warning.	`sc_stop()` can only be called from a `sc_module` whose execution is controlled by SoC Designer Simulator. Consecutive `sc_stop()` are possible (the first `sc_stop()` halts all execution).
`sc_report()`	SystemC v2.1 introduced this new exception reporting API.	Supported. `sc_report()` errors are forwarded to SoC Designer `message()`.
`sc_vector` class.	Utility class allowing creation of vectors of SystemC objects.	Class is not included automatically by including `systemc` or `systemc.h`. User code must explicitly add `#include "sysc/utils/sc_vector.h"`, and provide a path for the Boost headers.

A.2 Built-in primitive channels

The following table 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

Channel type	Data type	Component name
sc_buffer	bool	sc_buffer<bool>
	char	sc_buffer<char>
	double	sc_buffer<double>
	float	sc_buffer<float>
	int	sc_buffer<int>
	long	sc_buffer<long>
sc_clock	-	sc_clock
sc_fifo	bool	sc_fifo<bool>
	char	sc_fifo<char>
	double	sc_fifo<double>
	float	sc_fifo<float>
	int	sc_fifo<int>
	long	sc_fifo<long>
sc_mutex	-	sc_mutex
sc_semaphore	-	sc_mutex
sc_signal	bool	sc_signal<bool>
	char	sc_signal<char>
	double	sc_signal<double>
	float	sc_signal<float>
	int	sc_signal<int>
	long	sc_signal<long>
	sc_logic	sc_signal<sc_logic>
	long	sc_signal<long>
	short	sc_signal<short>
sc_signal_resolved	-	sc_signal_resolved