SystemC Cycle Models

Version 10.0

User Guide

arm

101124_1000_00_en

SystemC Cycle Models

User Guide

Release Information

Document History

Issue	Date	Confidentiality	Change
0300-00	01 September 2017	Non-Confidential	Release 3.0.0
0301-00	23 February 2018	Non-Confidential	Release 3.1.0
1000-00	29 August 2018	Non-Confidential	Release 10.0.0

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
SystemC Cycle Models User Guide

Preface

This preface introduces the *SystemC Cycle Models User Guide*.

It contains the following:

About this book

This guide describes how to integrate Arm® SystemC Cycle Models into a SystemC design and simulation environment.

Using this book

This book is organized into the following chapters:

Chapter 1 Introduction

This section introduces Arm SystemC Cycle Models.

Chapter 2 Using SystemC Cycle Models

This section describes how to work with Arm SystemC Cycle Models, including connecting ports, working with the API, and incorporating models in your design.

Chapter 3 Working with SystemC CPAKs

This section introduces SystemC CPAKs, describes how to modify the CPAK test bench, and explains how to add a model to the CPAK.

Chapter 4 Debugging SystemC Cycle Models with DS-5

This section describes how to connect the Arm Development Studio 5 (DS-5) Debugger with Arm Cycle Models in SystemC CPAKs. This section applies to CADI-enabled models only.

Chapter 5 SystemC Export API function reference

This section describes the functions of the SystemC eXport (SCX) API that are supported by SystemC Cycle Models. Each description of a class or function includes the C++ declaration and the use constraints.

Appendix A Migrating from previous SystemC Cycle Model versions

This section contains instructions specific to upgrading from a previous verison of SystemC Cycle Models to version 10.0.

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.

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 *SystemC Cycle Models User Guide*.
- The number 101124_1000_00_en.
- 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.

Other information

- *Arm® Developer*.
- *Arm® Information Center*.
- *Arm® Technical Support Knowledge Articles*.
- *Technical Support*.
- *Arm® Glossary*.

Chapter 1
Introduction

This section introduces Arm SystemC Cycle Models.

Arm SystemC Cycle Models are compiled directly from RTL code. The SystemC model wrapper is provided in source form, which enables you to compile for any SystemC 2.3.1-compliant simulator. You can use SystemC Cycle Models within an Arm Performance Analysis Kit (CPAK) or integrate them directly into any IEEE 1666-compliant SystemC environment.

It contains the following sections:

1.1 Prerequisites to using SystemC Cycle Models

Review the following prerequisites to using Arm SystemC Cycle Models:

- Cycle Model SystemC Runtime. The Cycle Model SystemC Runtime installer includes:
  - Fast Models Runtime. This is required for functions related to setting parameters and debugging.
  - Cycle Model Studio Runtime. This is required for simulation and recompilation.

  See the *Cycle Model SystemC Runtime Installation Guide* (101146) for more information.
- You must have a SystemC environment configured. See the *Cycle Model SystemC Runtime Installation Guide* (101146) for more information.
- CPAKs may have additional prerequisites. If you are running an Arm CPAK, see *3.1 Introduction to Arm® CPAKs* on page 3-25.

Arm recommends familiarity with the Fast Models SystemC Export feature with Multiple Instantiation (MI) support. SystemC Cycle Models support a subset of the SystemC eXport (SCX) API functions (these are provided by Fast Models Exported Virtual Subsystems (EVSs)). See the *Fast Models User Guide* (100965) for more information.

Note

If you are updating from a previous version of Arm SystemC Cycle Models, see *A.1 Migrating from previous versions* on page Appx-A-64 for important migration tasks.

Related information

Arm IP Exchange

Fast Models User Guide

Cycle Model SystemC Runtime Installation Guide

1.2 Supported platforms, compilers, and simulators

This section describes the requirements for running SystemC Cycle Models.

This section contains the following subsections:

1.2.1 Supported platforms

Arm SystemC Cycle Models are supported on Red Hat Enterprise Linux version 6.6 (64-bit) and above.

1.2.2 Supported compilers

The SystemC Cycle Models have been tested on Linux with GCC 4.8.3 and GCC 6.4.0.

The SystemC Cycle Models include C++11 code, therefore the GCC you are using must support this.

1.2.3 Supported simulators

Arm SystemC Cycle Models can be compiled for any SystemC 2.3.1-compliant simulator.

1.3 Package contents

Each SystemC Cycle Model contains the following files:

Note

All features may not be available on all models.

***component*ResetModule.h**
Reset module used to drive the SystemC pin-level wrapper for the Reset sequence of the IP.

***component*.xmlAnswers**
Shows the configuration of the Cycle Model as built on Arm IP Exchange.

lib*component*.icm.so
RTL-based core of the Cycle Model. When you compile the system executable, this must be included.

lib*component*.h
Base function header exposed by the core Cycle Model. This is required to access functions in the core Cycle Model.

lib*component*.systemc.cpp and lib*component*.systemc.h
Pin-level SystemC wrapping header for the core Cycle Model. Compile this to generate a signal-level, linked SystemC model.

lib*component*_icm.h
Header file for `libcomponent.icm.so`, which is the RTL-based core of the Cycle Model.

loadTCMUtil/*
Set of files that support direct loading of the TCM.

Makefile
Compiles the pin-level model into the shared libraries included with the installation.

***component*_tarmac.h**
Cycle Model parameter definition to generate Tarmac traces.

***component*_params.cfg**
Cycle Model-specific parameter definitions.

***component*_pmu.h**
Cycle Model hardware profiling implementation to generate profiling events.

***component*.tlm.cpp and *component*.tlm.h**
TLM wrappers.

univent_tarmac.cpp
Tarmac interface implementation. Hook into the pin level Cycle Model to generate Tarmac traces.

univent_tarmac.h
Tarmac interface header which can be hooked into the pin level Cycle Model to generate Tarmac traces.

univentUtil/*
Contains model-specific Tarmac libraries which are needed to compile the `univent_tarmac.cpp` and `univent_tarmac.h` into the model.

Chapter 2
Using SystemC Cycle Models

This section describes how to work with Arm SystemC Cycle Models, including connecting ports, working with the API, and incorporating models in your design.

It contains the following sections:

2.1 Connecting model ports

All pins must be bound to a signal.

For a list of the pins on the SystemC Cycle Model, refer to the model header file `Lib*model*.Systemc.h`, or the `CM_IPXACT_*model*.xml` file.

Certain pins are tied and cannot be modified. For a list of tied pins, see *Tied pins*.

Refer to the SystemC documentation for information about native SystemC binding commands (`sc_in`, `sc_signal`, etc.).

This section contains the following subsection:

2.1.1 Available pins

When making changes to the model pins, be aware that certain pins are tied high or low, and can not be modified.

For a complete list of the pins on the SystemC Cycle Model, refer to the model header file `Lib*model*.Systemc.h`, or the `CM_IPXACT_*model*.xml` file. Certain pins in the list are tied and cannot be modified; contact Arm Support for more information.

2.2 Resetting the SystemC Cycle Model

A default reset sequence is provided in source form in *component*`ResetModule.h`.

If necessary, you can modify this file as needed to work with your system, then recompile the model after making your changes.

Ensure that the reset module is connected to the model.

Refer to the Technical Reference Manual for your IP for details about its reset sequence.

2.3 Setting model parameters

This section describes how to see a list of the parameters on the SystemC Cycle Model, and how to set them.

Initialization parameters

You can change initialization-time (Init) parameters either on the command line prior to simulation, or in the test bench (`system_test.cpp`) prior to the start of simulation (`sc_start`. Ensure that you recompile for the change to take effect. See *3.3 Working with Cycle Models and CPAKs* on page 3-28 for instructions.

Run-time parameters

For run-time parameters, change the parameter value on the command line and restart the simulation.

Available parameters

The following table describes the parameters supported by the model. This information is also available:

- In the *component*`_params.cfg` file included in your model installation.
- By entering `./system_test --list-params` in the `Systems` directory of the CPAK.

See the *Technical Reference Manual* for your IP for additional information about supported parameter values.

For CADI-enabled models, see *CADI RemoteConnection parameters* for additional parameters related to configuring CADI debug connections. These options do not appear in the *component*`_params.cfg` file.

2.4 Dumping waveforms

This section describes how to configure waveform dumping.

To enable and disable waveform dumping using parameter values within the system executable code, set the following parameters.

Note

Setting `WAVEFORM_TIMEUNIT` and `WAVEFORM_TYPE` is optional; set them only if you want to change the default settings. If you are changing them, call `WAVEFORMS_ENABLED` after setting `WAVEFORM_TIMEUNIT` and `WAVEFORM_TYPE`.

Table 2-1 Waveform parameters

Parameter	Available settings	Default setting
`WAVEFORM_TIMEUNIT`	Units defined by sc_time_unit(): `SC_FS`, `SC_PS`, `SC_NS`, `SC_US`, `SC_MS`, `SC_SEC`	`SC_PS`
`WAVEFORM_TYPE`	`FSDB`, `VCD`	`VCD`
`WAVEFORMS_ENABLED`	`true`, `false`	`false`

For example:

```
scx::scx_set_parameter("sc-module-name.WAVEFORM_TIMEUNIT",sc_core::SC_NS);
scx::scx_set_parameter("sc-module-name.WAVEFORMS_TYPE","FSDB");
scx::scx_set_parameter("sc-module-name.WAVEFORMS_ENABLED",true);
```

`sc-module-name` is the name given to the model when it is instantiated in the system executable.

2.5 Loading TCMs (Cortex-R52 SystemC Cycle Model only)

You can load the ATCMS, BTCMS, and CTCMS memories by setting parameter values within the system executable.

Set the following parameters:

Table 2-2 TCM parameters

Parameter	Available settings	Default setting
LOAD_ATCMS	true, false	false
LOAD_BTCMS	true, false	false
LOAD_CTCMS	true, false	false

To see the parameter names for the data files that are to be loaded, see the files *component*`_atcm.h`, *component*`_btcm.h`, and *component*`_ctcm.h`. You can change these files by setting the parameters in the same way.

2.6 Configuring cache and TCM sizes (Cortex-R52 SystemC Cycle Model only)

You can set the instruction cache, data cache, and TCM sizes by setting parameter values within the system executable.

For parameter names and possible values, refer to the *component*_params.h file. For example:

For example:

```
scx::scx_set_parameter("sc-module-name.ICACHE_SIZE_CPU0",7);
```

sc-module-name is the name given to the model when it is instantiated in the system executable.

2.7 Configuring PMU events

SystemC Cycle Model Performance Monitoring Unit (PMU) events are stored in C++ variables.

By default, calculations of PMU events are disabled in the SystemC Cycle Model. You can enable PMU events by setting a parameter value in the system executable code. Use the following parameters:

Table 2-3 PMU parameters

Parameter	Available settings	Default setting
PMU_ENABLED	true, false	false

For example:

```
scx::scx_set_parameter("sc-module-name.PMU_ENABLED",true);
```

sc-module-name is the name given to the model when it is instantiated in the system executable.

For information about C++ variable names for PMU events, refer to the file component_pmu.h located in the CPAK directory MODELS/component/gcc483/SystemC.

2.8 Configuring TARMAC trace

This section describes how to enable and disable TARMAC trace.

By default, TARMAC trace is disabled, and TARMAC buffers log file data. You can enable TARMAC tracing by setting parameter values in the system executable code, and specify the number of instructions after which to flush the log file.

Note

If you are setting `TARMAC_LOGFILE_NAME`, call `TARMAC_ENABLED` after setting `TARMAC_LOGFILE_NAME`.

Table 2-4 TARMAC trace parameters

Parameter	Description	Available settings	Default setting
`TARMAC_LOGFILE_NAME`	Sets TARMAC log file name. For multiple cores or clusters, use the @CPUID@ string in the name to specify where to place the CPU identification number.	*string*	`""`
`TARMAC_ENABLED`	Enables or disables TARMAC logging.	`true`, `false`	`false`
`TARMAC_FLUSH`	Flushes the Tarmac log file data after the specified number of instructions.	`integer`	`0`

For example, for a design with two cores and one cluster:

```
scx::scx_set_parameter("r8.TARMAC_LOGFILE_NAME","tarmac.r8.@CPUID@.log");
scx::scx_set_parameter("r8.TARMAC_LOGFILE_ENABLED",true);
```

This creates the files `tarmac.r8.0.log` and `tarmac.r8.1.log`.

2.9 Working with the SCX framework

Arm SystemC Cycle Models implement the SystemC Export (SCX) API provided by Fast Models Exported Virtual Subsystems (EVSs).

SCX API overview

You can configure the parameters and other settings for your SystemC model using either native SystemC signals or using the SCX API. The SCX API is fully described in the *Fast Models User Guide* (100965), section 7.6 (SystemC Export API).

Arm recommends not mixing parameter sets through the SCX framework and parameter sets through native SystemC signal writes, as this can produce unexpected results. For example, the following case describes what would happen in a case where both are used in succession in a system:

```
scx::scx_set_parameter("CortexR8.ACLKENST",1); //Statement 1
CortexR8.ACLKENST.write(0); //Statement 2
```

Due to intrinsic SystemC properties, the value ultimately assigned to ACLKENST depends on the previous value of the pin:

- If `ACLKENST` had an initial value of `0`, the `write(0)` is ignored because that was the previous value, and `ACLKENST` is assigned a value of `1`. Because of the SystemC property of `write`, if the previous value was `0`, `setParameter` takes precedence.
- If `ACLKENST` had a value of 1, then the `write` takes precedence and the value is set to `0`.

See *Chapter 5 SystemC Export API function reference* on page 5-50 for details about the functions supported by SystemC Cycle Models.

2.10 Using a SystemC model in your own design

This section describes how to integrate a SystemC model into IEEE-compliant SystemC environments, including required header files and libraries, SCX library instantiation, and other ordering requirements within the code.

Runtime-related include directories, libraries, and sources to build the SystemC Cycle Model system on Linux

To include the required directories, libraries, and sources, do the following:

- Include the common makefile at `$(CM_SYSC_HOME)/makefiles/cm_sysc_common.mak`.
- Add the `CM_SYSC_CXXFLAGS` flag to the compile line.
- Add the following flags to the link line:

  `CM_SYSC_LDFLAGS`

  `CM_SYSC_MODELS`

  `FM_LIB`

`CM_SYSC_HOME` differs based on whether you are using a CPAK system or a standalone Arm Cycle Model:

- For CPAKs: `CM_SYSC_HOME=`*`CPAK_RELEASE_PACKAGE`*`/MODELS/SystemCMisc/$`*`CM_PLATFORM`*
- For standalone Cycle Models, you must also download the SystemC Cycle Models Runtime package from Arm IP Exchange (see *1.1 Prerequisites to using SystemC Cycle Models* on page 1-10). In this case:

  ```
  CM_SYSC_HOME=PATH_TO_UNZIPPED_CM_SYSC_RUNTIME_RELEASE_PACKAGE
  ```

Model-related build wrapper sources and library requirements

To compile the SystemC model wrapper into your design, the following build wrapper sources are required:

- `MODELS/`*`component name`*`/gcc483/SystemC/*.cpp`

Library requirements are:

- `MODELS/`*`component_name`*`/gcc483/SystemC/lib/lib`*`component_name`*`.icm.so`

Libraries to run the virtual platform on Linux

The virtual platform might require specific libraries in addition to those required by the SystemC Cycle Model system:

- `MODELS/`*`component name`*`/gcc483/SystemC/lib/lib`*`model name`*`.icm.so`
- `MODELS/`*`component name`*`/gcc483/SystemC/lib/libicm_runtime.so`
- *`MODEL_DIR`*`/$(CM_PLATFORM)/SystemC/univentUtil/lib/`*`model_name`*`_tarmac_dpi.so` - For TARMAC Trace functionality, you must link against this `.so` file.
- `libprotobuf.so` - This open-source library might be required by *`model_name`*`_tarmac_dpi.so`.
- `libcarbon5.so` - This file may already be present as it is provided with the Cycle Model Studio runtime.

Adding a SystemC Cycle Model to your virtual platform

Adding the SystemC Cycle Model to your design requires modifications to the CPAK test bench. See *3.3 Working with Cycle Models and CPAKs* on page 3-28 for information about these modifications.

Additional include directories and libraries to build the SystemC Cycle Model on Linux

The SystemC Cycle Model system requires the following additional `include` directories and libraries:

Required `include` directories:

- `MODELS/`*`component name`*`/gcc`*`version`*`/SystemC`

Chapter 3
Working with SystemC CPAKs

This section introduces SystemC CPAKs, describes how to modify the CPAK test bench, and explains how to add a model to the CPAK.

It contains the following sections:

3.1 Introduction to Arm® CPAKs

Arm provides Performance Analysis Kits (CPAKs) that contain SystemC models, which demonstrate how a working system is constructed. You can use an Arm CPAK as a starting point to develop and fine-tune your custom system.

You can copy and migrate a SystemC Cycle Model that is part of a CPAK and build it into your own custom system. Alternatively, you can modify an Arm CPAK system by adding your own SystemC model classes to the Arm CPAK.

- If you want to include a different or additional SystemC Cycle Model in the CPAK, you need to modify the CPAK testbench and corresponding build system to include, instantiate, and connect the model. See *3.3 Working with Cycle Models and CPAKs* on page 3-28 for details.
- If you want to run a CPAK without altering the system, you need to configure the parameters that generate the performance data you are interested in and run the simulation. See *Chapter 2 Using SystemC Cycle Models* on page 2-13 for details.

  Visit Arm IP Exchange (*http://armipexchange.com/cpaks*) to download a SystemC CPAK.

Prerequisites

Arm CPAKs require the Cycle Model Studio runtime. Visit the Arm IP Exchange Support area (*http://armipexchange.com/websupport/support.aspx*) to download the Cycle Model Studio runtime.

3.2 Getting started with CPAKs

In this section, a Cortex-R8 SystemC CPAK is used to show the basics of working with Arm CPAKs.

CPAK components

The CPAK (R8_SysC) includes the core, and a bus, UART, and memory. Note that not all CPAKs follow this format. Additionally, this is an example of a TLM-based CPAK; CPAKs may be TLM-based or pin-based.

The CPAK test bench, `Systems/system_test.cpp`, defines the default CPAK configuration in `sc_main()`.

See the following diagram for a graphical representation of its main components, port bindings, and memory map.

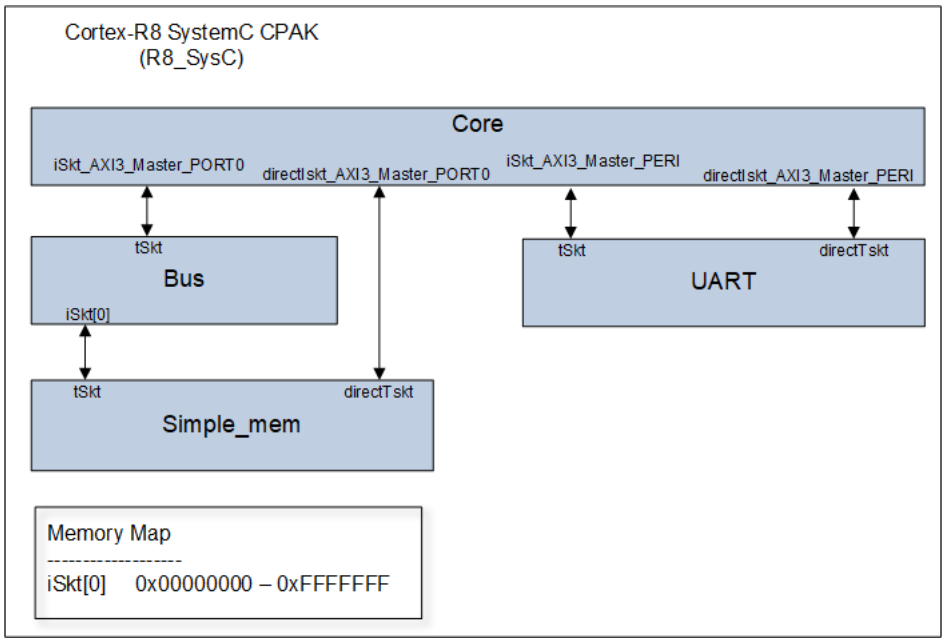

Figure 3-1 SystemC CPAK

CPAK directory structure

The following figure describes the general CPAK directory structure and summarizes its contents.

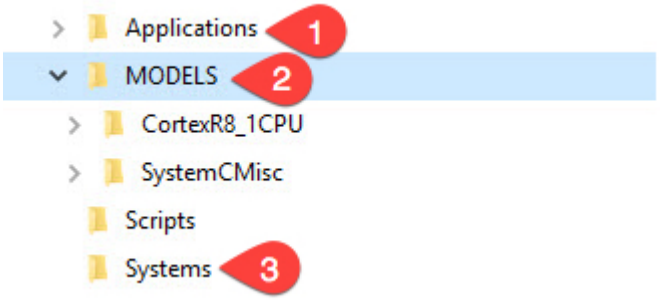

Figure 3-2 SystemC CPAK directory structure

1. The `Applications` directory contains source code, build files, and executable images for example applications that execute on the processor.
2. The `MODELS` directory contains model libraries and SystemC wrapper source code for all the models used to build a system.
3. The `System` directory contains top-level design files used to connect models. The `system_test.cpp` file contains the `sc_main` function.

Testing CPAK operation

After you download a CPAK from Arm IP Exchange (*http://armipexchange.com/cpaks*):

1. Untar the .tgz file:

```
tar xzvf R8-SysC-V10.0.0-CMS10.0.0.tgz
```

2. Open and review the `README.txt` file located at the top of the CPAK directory structure. CPAK readme files include instructions for setting up the environment, test applications that are available with the CPAK, requirements and dependencies, and other IP-specific instructions.
3. As described in the `README.txt` file, perform `make` and `makeclean` to create the `system_test` executable.
4. After you have verified that your environment meets the specified requirements, `cd` to the `Systems` directory and run the example test bench `system_test`. The test bench starts the simulation, and loads and runs the application specified by `system_test.cpp`.

   By default, `system_test.cpp` runs the `hello_world` application located in the `Applications` directory. Different CPAKs may run different applications by default:

```
$ ./system_test

SystemC 2.3.1-Accellera --- Jun 18 2015 09:17:13
Copyright (c) 1996-2014 by all Contributors,
ALL RIGHTS RESERVED

Loading: ../Applications/hello_world/armcc/elf/test.elf
[plover_tarmac] Detected capture module at 'PLOVERINTEGRATION.u_plover.u_noram0' (enabled)
[plover_tarmac] Sending stream to 'plover_tarmac_decode -f
tarmac.PLOVERINTEGRATION_u_plover_u_noram0.log'
[plover_tarmac] Detected capture module at 'PLOVERINTEGRATION.u_plover.u_noram2' (enabled)
[plover_tarmac] Sending stream to 'plover_tarmac_decode -f
tarmac.PLOVERINTEGRATION_u_plover_u_noram2.log'
[plover_tarmac] Detected capture module at 'PLOVERINTEGRATION.u_plover.u_noram3' (enabled)
[plover_tarmac] Sending stream to 'plover_tarmac_decode -f
tarmac.PLOVERINTEGRATION_u_plover_u_noram3.log'
[plover_tarmac] Detected capture module at
'PLOVERINTEGRATION.u_plover.u_noram1_wrapper.u_noram1' (enabled)
[plover_tarmac] Sending stream to 'plover_tarmac_decode -f
tarmac.PLOVERINTEGRATION_u_plover_u_noram1_wrapper_u_noram1.log'
Starting Simulation
<01><ff><01><ff><01><ff><01><ff><ff><01><01><ff>Hello World!
My name is Plover
I wish you a great day
** TEST PASSED OK **
<04>0x04 End Of Simulation message received by TrickBox
.
.
.
```

3.3 Working with Cycle Models and CPAKs

To make changes to the CPAK system, alter the CPAK test bench.

Each CPAK has its own test bench called `system_test.cpp`, which is located in the `Systems` directory of your CPAK. This test bench:

- Instantiates the models
- Defines the connections between models
- Initializes model parameters
- Provides simulation controls (start, stop, run, specific number of cycles, etc.)

After altering the test bench as described in the following sections, recompile the system. Models are recompiled automatically as part of the system-level recompile.

This section contains the following subsections:

- *3.3.1 Application loading* on page 3-28.
- *3.3.2 Modifying the test bench for pin-level models* on page 3-28.
- *3.3.3 Modifying the test bench for TLM models* on page 3-30.

3.3.1 Application loading

Pin-level CPAKs and TLM-based CPAKs handle application loading differently.

Pin-level CPAK application loading

For CPAKs implemented through pin-level connections, the application to be loaded is specified by the `.hex` file located in the `Systems` directory. The application is loaded into the CPAK upon initialization of the BP140 memory. See the `README.txt` file for more information on changing the application.

TLM-based CPAK application loading

For CPAKs implemented through TLM connections, the application is specified as a command line argument and parsed during the call of the SCX `scx_parse_and_configure()` function.

3.3.2 Modifying the test bench for pin-level models

This section describes the areas of the CPAK test bench you might want to change.

Note

See *3.3.1 Application loading* on page 3-28 for information about how pin-level CPAKs handle application loading.

Required includes

The test bench contains the SystemC wrapper files for any models in the system, the corresponding reset module file for each model (for pin-level models only), and `includes` required by the Fast Models runtime. Ensure you add these files for any new models added to your system. Here is an example of the `includes` section of a CPAK test bench:

```
// Include the systemc wrapper files for the models
#include "libCortexR8.systemc.h"            // CortexR8 CPU
#include "libNIC400.systemc.h"              // NIC400 Interconnect
#include "libBP140.systemc.h"               // BP140 Memory
#include "libBP140_TrickBox.systemc.h"      // BP140_TrickBox UART

// Include the reset modules for the above models (pin-level models only)
#include "CortexR8ResetModule.h"
#include "NIC400ResetModule.h"
#include "BP140ResetModule.h"
#include "BP140_TrickBoxResetModule.h"
#include "../perf_common.h"
```

```
#include <iostream>
#include <time.h>

// These includes are need by the SCX FastModel Runtime
#include <scx/scx.h>
#include <scx/scx_signal_sizer.h>
#include <scx_simcontroller.cpp>
#include <scx_scheduler_mapping.cpp>
#include <scx_report_handler.cpp>
```

Initialization of the simulation environment and model instantiation

The `sc_main()` section of the test bench must first initialize the SCX simulation environment. It must state clocking specifications, then instantiate all IP and reset models (for pin-level models only) for any models included in the system. For example:

```
int sc_main(int argc, char *argv[])
{
  // Debug initialization
  scx::scx_initialize("R8-SysC-Debug");

  // If you want to see messages about 'port not bound' change SC_DO_NOTHING to SC_DISPLAY.
  // If you want it to abort on a 'port not bound' error comment out the line below.
  sc_report_handler::set_actions(SC_ID_COMPLETE_BINDING_,SC_ERROR,SC_DO_NOTHING);

  // Clock Object
  sc_clock clk("clk", 1, SC_NS, 0.5);

// Instantiate IP
  CortexR8 core("cortexR8");
  NIC400 nic400("NIC400");
  BP140 bp140("BP140");
  BP140_TrickBox bp140_trickbox("BP140_TrickBox");
  ARM::Models::Cycle::ModelCortexR8::CortexR8ResetModule core_reset("core_reset");
  ARM::Models::Cycle::ModelNIC400::NIC400ResetModule nic400_reset("nic400_reset");
  ARM::Models::Cycle::ModelBP140::BP140ResetModule bp140_reset("bp140_reset");
  ARM::Models::Cycle::ModelBP140_TrickBox::BP140_TrickBoxResetModule
bp140_trickbox_reset("bp140_trickbox_reset");
```

Signal bindings

The test bench specifies all signal bindings, including those for reset modules.

- Declare bindings using an `sc_signal` call in the test bench file. The signal must be the same type and width as the two ports being connected. If the ports are the same type but different widths, use `scx_signal_sizer` instead of `sc_signal`. For example:

  ```
  scx::scx_signal_sizer<sc_uint<13>, sc_uint<16> >ARIDsignal1;
  core.ARIDM0.bind(ARIDsignal);
  nic400.arid_s_axi_64.bind(ARIDsignal);
  ```

- Signals must be bound to both ports. For example:

  ```
  sc_signal(bool) signal1;
  Inst1.port1.bind(signal1);
  sc_signal(bool) signal2;
  Inst2.port1.bind(signal2);
  ```

Clock bindings

All models must be bound to the system clock; for example:

```
// Bind all the models to the system (cpu) clock
core.CLKIN.bind(cpu_clk);
mem.ACLK.bind(cpu_clk);
bus.mainclk.bind(cpu_clk);
uart.ACLK.bind(cpu_clk);
core_reset.clk.bind(cpu_clk);
bus_reset.clk.bind(cpu_clk);
bp140_reset.clk.bind(cpu_clk);
bp140_trickbox_reset.clk.bind(cpu_clk);
```

Functions to generate performance data

The following example shows the use of SCX API functions to specify PMU and Tarmac output. See *Chapter 2 Using SystemC Cycle Models* on page 2-13 for more information:

```
// Enable PMU and TARMAC for the CortexR8
scx::scx_set_parameter("cortexr8_core.PMU_ENABLED",true);
scx::scx_set_parameter("cortexr8_core.TARMAC_ENABLED",true);
```

Parsing of command line arguments

The test bench calls the SCX `scx_parse_and_configure()` function to parse any command line arguments used by the SCX runtime:

```
scx::scx_parse_and_configure(argc, argv);
```

Simulation call

To simulate the system, the test bench calls `sc_start()`:

```
sc_start();
```

Specify all includes, initializations, bindings, functions, and command line options before `sc_start()`.

3.3.3 Modifying the test bench for TLM models

`sc_main` has the same basic flow for both pin-level and TLM models. One difference is that the TLM models are connected using TLM sockets rather than pins. This section describes the TLM-specific instructions.

Note

See the file `libcomponent.tlm.h` in your installation directory for socket names.

Note

See *3.3.1 Application loading* on page 3-28 for information about how TLM-based CPAKs handle application loading.

Required includes

The SystemC pin-level models are wrapped with TLM functionality. The test bench includes the SystemC wrapper files for any models included in the system, and `includes` required by the Fast Models runtime. Ensure you add these files for any new models added to your system. Here is an example of the `includes` section of a CPAK test bench:

```
// Include the systemc wrapper files for the models

#include "models/SimpleMem.h"
#include "models/SimpleFlash.h"
#include "models/SimpleFlashImp.h"
#include "models/RAMBlock.h"
#include "models/SimpleBus.h"
#include "models/BasicUART.h"
#include "models/ElfLoader.h"
#include "libCORTEXR8.tlm.h"                  // CPU
#include <tlm_utils/simple_initiator_socket.h>

#include <iostream>

// These includes are need by the SCX FastModel Runtime
#include <scx/scx.h>
#include <scx_simcontroller.cpp>
#include <scx_scheduler_mapping.cpp>
#include <scx_report_handler.cpp>
```

Initialization of the simulation environment and model instantiation

The `sc_main()` section of the test bench must first initialize the SCX simulation environment. It states clocking specifications, then instantiates any models included in the system. Ensure you instantiate any new models added to your system. For example:

```
int sc_main(int argc, char *argv[])
{
// Debug initialization
  scx::scx_initialize("R8-SysC");

  // If you want to see messages about 'port not bound' change SC_DO_NOTHING to SC_DISPLAY.
  // If you want it to abort on a 'port not bound' error comment out the line below.
  sc_report_handler::set_actions(SC_ID_COMPLETE_BINDING_,SC_ERROR,SC_DO_NOTHING);

  // Clock Object
  sc_clock cpu_clk("clk", 1, SC_NS, 0.5);

  ARM::Models::Cycle::ModelCortexR8::CortexR8Imp core("CortexR8");

  // Main Memory
  ARM::Models::RAMBlock ram_block;
  ARM::Models::SimpleMemConfig simple_mem_params;
  simple_mem_params.delay = 1;
  simple_mem_params.ram_block = &ram_block;
  simple_mem_params.busWidthBits = 64;

  ARM::Models::SimpleMem simple_mem("RAM", simple_mem_params);
  ARM::Models::BasicUART uart("UART", std::cout, "UART: ");

  // BUS & its Mappings
  ARM::Models::SimpleBus bus("Interconnect", 1, 1, 64);
  bus.addMap(0, 0, 0xFFFFFFFF); // Main Memory
```

Port bindings

The test bench specifies all TLM port bindings. Signal bindings required for the system are also specified in this area. Specify any additional TLM port bindings and signal bindings in this area:

```
// Core Main Memory Port Bindings
        core.iSkt_AXI3_Master_PORT0->bind(bus.tSkt);
        core.directIskt_AXI3_Master_PORT0.bind(simple_mem.directTskt);

// Core Low Latency Peripheral Port Bindings
        core.iSkt_AXI3_Master_PERI->bind(uart.tSkt);
        core.directIskt_AXI3_Master_PERI.bind(uart.directTskt);

// Bus iSkt[0] connected to Main Memory
        bus.iSkt[0]->bind(simple_mem.tSkt);

// Clock
        core.clk(cpu_clk);
        simple_mem.clock.bind(cpu_clk);
        uart.clock.bind(cpu_clk);
        bus.clock.bind(cpu_clk);
```

For information about signal bindings, clock bindings, performance data generation, command line arguments, and simulation calls, use the instructions in *3.3.2 Modifying the test bench for pin-level models* on page 3-28.

3.4 Building and running CPAK simulations

CPAKs include Makefiles for building its applications, models, and systems..

This section describes the available CPAK build, run, and simulation options. The instructions in this section are for Arm SystemC CPAK systems only.

CPAKs come with sample applications and application build scripts; these are located in the *CPAK_Name*/`Applications` directory.

The top-level CPAK `Makefile` is the command script for your build infrastructure. The `Makefile` includes the following available targets:

`make all`
Builds the system.

`make system`
Takes the CPAK test bench (`system_test.cpp`) and the library files, and generates the `system_test` binary. For more information about the test bench, see *3.3 Working with Cycle Models and CPAKs* on page 3-28.

`make app-setup [ APP=path ]`
where `path` is the path to the compiled application (`.elf`) file. This option sets up the particular application the system uses during simulation.

`make run [ APP=path ] [ RUNLOG=file_name ]`
where `path` is the path to the compiled application (`.elf`) file and `file_name` is the name of the file for log output. This option runs the simulation. If `APP` is not specified, it runs the default application or the one that was last set up using `app-setup`.

`make clean`
Removes all the binaries and object files.

`make help`
Prints all available targets.

Chapter 4
Debugging SystemC Cycle Models with DS-5

This section describes how to connect the Arm Development Studio 5 (DS-5) Debugger with Arm Cycle Models in SystemC CPAKs. This section applies to CADI-enabled models only.

It contains the following sections:

4.1 Prerequisites to debugging

Arm DS-5 Development Studio is required before you begin. The instructions in this chapter have been verified using DS-5 Version 5.28.1.

Note

The Windows version of DS-5 is not supported for SystemC Cycle Models. Only the Linux 64-bit version is supported.

- Download and install the Linux 64-bit version of DS-5 Development Studio (Ultimate Edition) from *https://developer.arm.com/products/software-development-tools/ds-5-development-studio/downloads*.

  In the DS-5 **Arm License Manager** dialog box (accessed from the **Help** menu), specify **Arm DS-5 Ultimate Edition** as the toolkit to use.
- See the *Arm® DS-5 Getting Started Guide* (100950) for DS-5 system requirements and installation instructions.

4.2 Models that support DS-5 connectivity

DS-5 connects only to CPU models that have debugger support.

The following SystemC Cycle Models support connection to DS-5:

- Cortex-R8 (single core and multi-core).
- Cortex-R52 (single core and multi-core). Debugger features on the Cortex-R52 model are BETA quality. This includes register view, memory view, and breakpoint/single step support. Debug memory views are only in a downstream direction, and only for the AXI master space. Debug memory views do not include internal TCM or Data Cache lookup.
- Cortex-A53 (single core and multi-core). Debugger features on the Cortex-A53 model are BETA quality. This includes register view, memory view, and breakpoint/single step support. Multicluster coherent memory views are not supported.

4.3 Supported debug features

This section describes DS-5 features that are supported on SystemC Cycle Models and debugging features that have been added to SystemC Cycle Models.

Note

CPUs are modeled as masters that issue debug access downstream to other components. Upstream debug access into CPU models through slave ports is not supported.

DS-5 features

SystemC Cycle Models support the following DS-5 functionality:

- Debugging of multi-core and multi-cluster configurations.You can specify whether you want to debug software running on multiple CPUs, or debug software on one CPU at a time. See the section *4.6 Multicore debugging* on page 4-48 for more information.
- Debugging of *Symmetric Multi Processing* (SMP) systems.

See the *Arm® DS-5 Debugger User Guide* (100953) for more information about debugging multi-core, multi-cluster, and SMP targets.

Support for memory and register views

The SystemC Cycle Model exposes memory spaces and a subset of the registers. However, their visibility varies depending on the debugger in use.

4.4 Restrictions and limitations

This section describes the restrictions and limitations for debugging SystemC Cycle Models.

Be aware of the following limitations related to debugging SystemC Cycle Models with DS-5:

- The Windows version of DS-5 is not supported for SystemC Cycle Models. Only the Linux 64-bit version is supported.
- Some multi-cluster systems may support cache coherency. Cycle Models in SystemC CPAKs do not currently show a coherent debug view of memory shared across clusters.
- System reset is not supported through the debugger interface.
- `sc_stop()` function calls are not supported during simulation, because they could result in termination of the debugger connection. A suggested workaround is to use an infinite loop at the end of the software being simulated.
- For certain cores, breakpoints may be missed during debug if they exist within short loops. See *4.6 Multicore debugging* on page 4-48 for workarounds.

4.5 Enabling DS-5 for use with SystemC Cycle Models

This section describes how to set up DS-5 to debug Cycle Models.

This section contains the following subsections:

4.5.1 Starting the simulation and selecting the model for debug

This section describes starting the simulation and selecting the SystemC model for debug.

After the DS-5 Configuration database has been created (see *Initial database configuration*):

1. Start the SystemC simulation with the CADI server enabled:

```
./system_test -S
```

2. Starting from a clean workspace, launch DS-5.
3. In DS-5, right-click on the Configuration Database you created during initial configuration, and select **New** > **Model Configuration**. The **New Model** dialog box appears.
4. In the **Select the database** panel, select the database in which to create the entry and click **Next**.

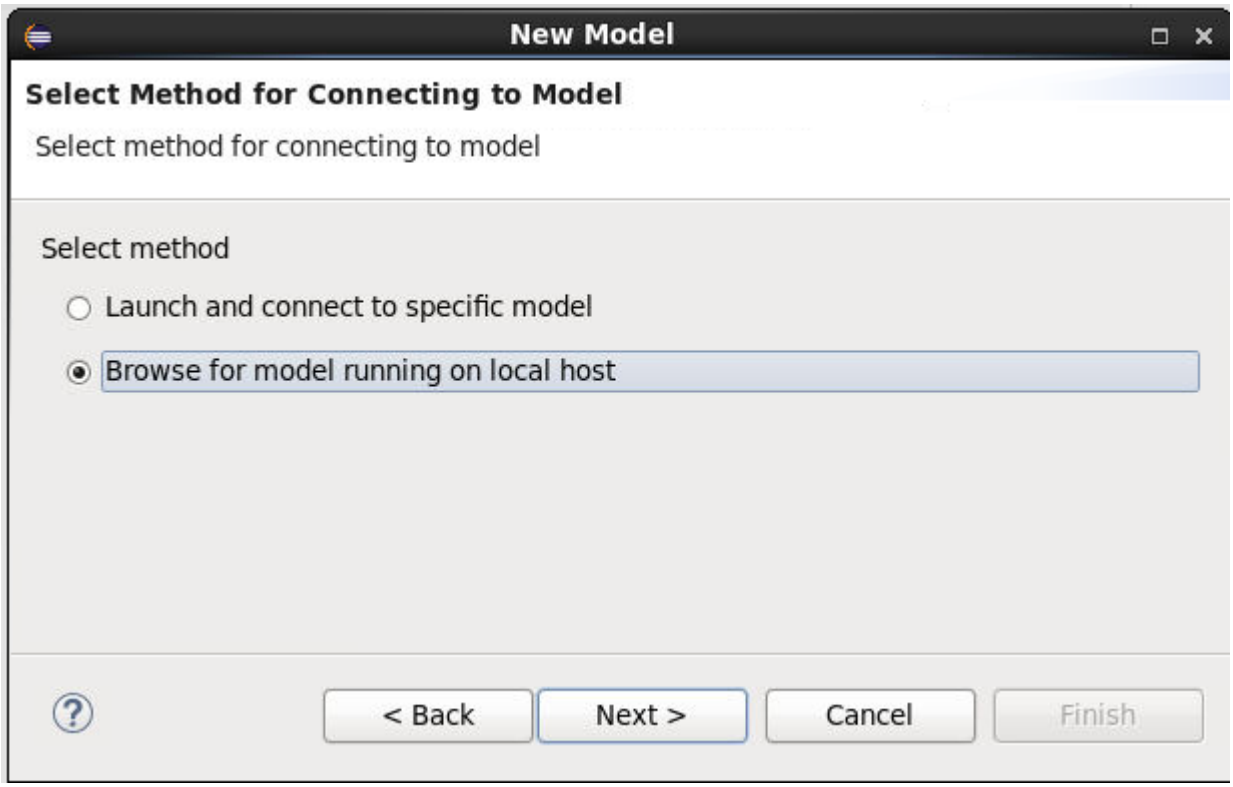

Figure 4-1 Selecting the method of connecting to a model

5. In the **Select Method for Connecting to Model**: dialog, select **Browse for model running on local host** and click **Next**.
6. In the **Browse for model running on local host** dialog, click **Browse**.

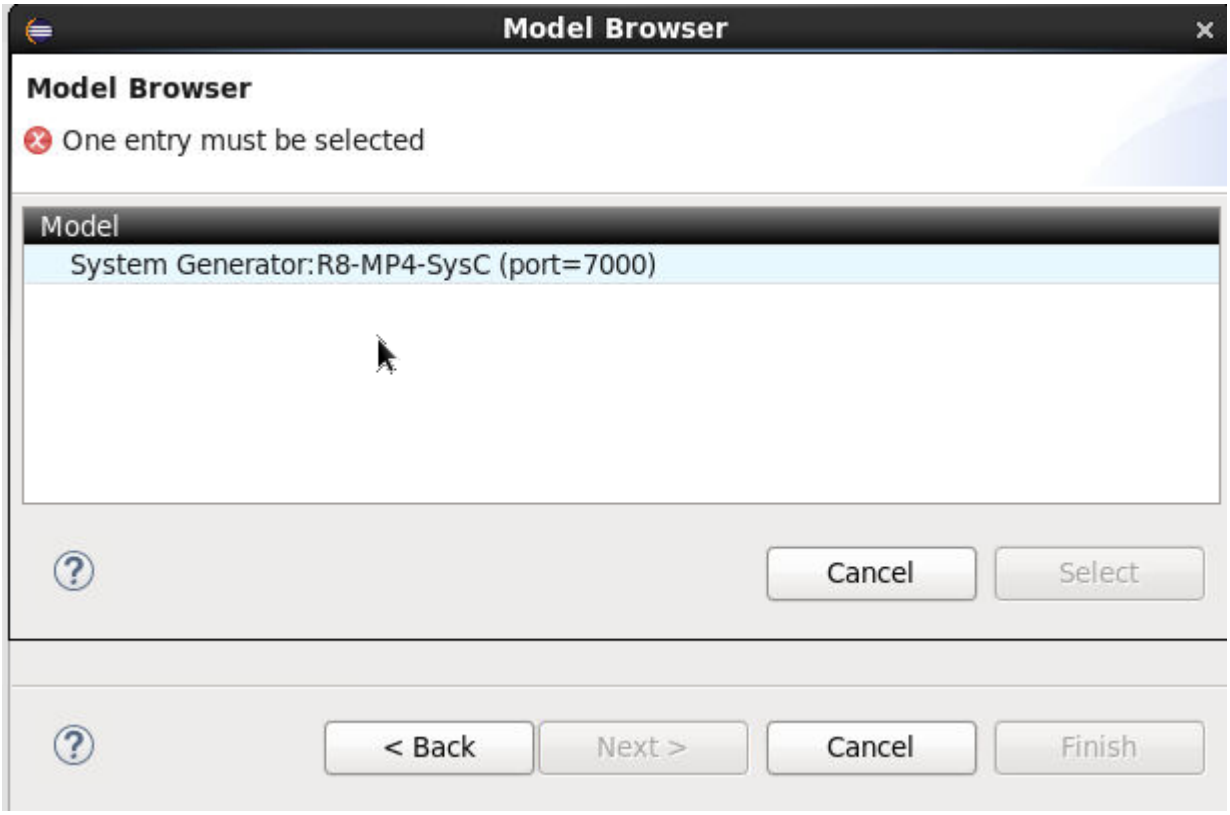

Figure 4-2 Selecting the method of connecting to a model

Result: DS-5 searches for the SystemC simulation sessions running on the host, and displays available simulations in the **Model Browser** dialog box.

7. Select the model for debug and click **Select**.
8. In the **New Model** dialog box, click **Finish**.

Result: DS-5 connects to the selected model and displays the cores available for debug.

4.5.2 Mapping memory spaces

This section describes memory space mapping for cores that support Secure, Non-Secure, or Hypervisor spaces.

Note

Memory space mapping applies only to the CPU types described below.

This section applies to:

- ARMv8-A cores, such as the Cortex-A53 CPU
- Other cores that have Secure, Non-Secure, or Hypervisor spaces

ARMv7 Cortex-R and Cortex-M CPU models do not support Secure, Non-Secure, or Hypervisor spaces. For this reason, there is no need to specify memory space mapping.

To set memory space properties:

1. In the DS-5 **Model Devices and Cluster Configuration** tab, select the CPU instance:

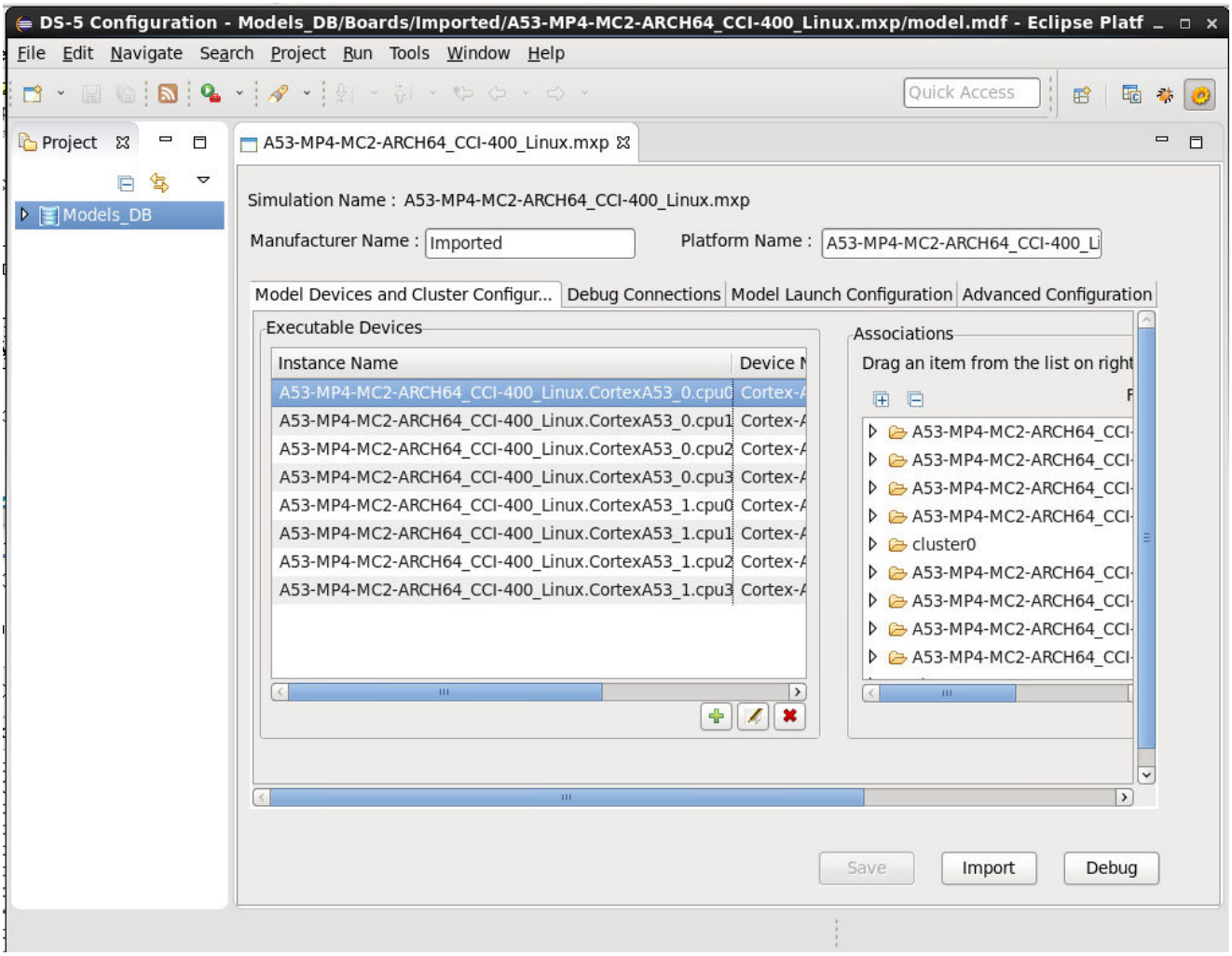

Figure 4-3 Selecting the CPU instance

2. Click **Edit selected row**:

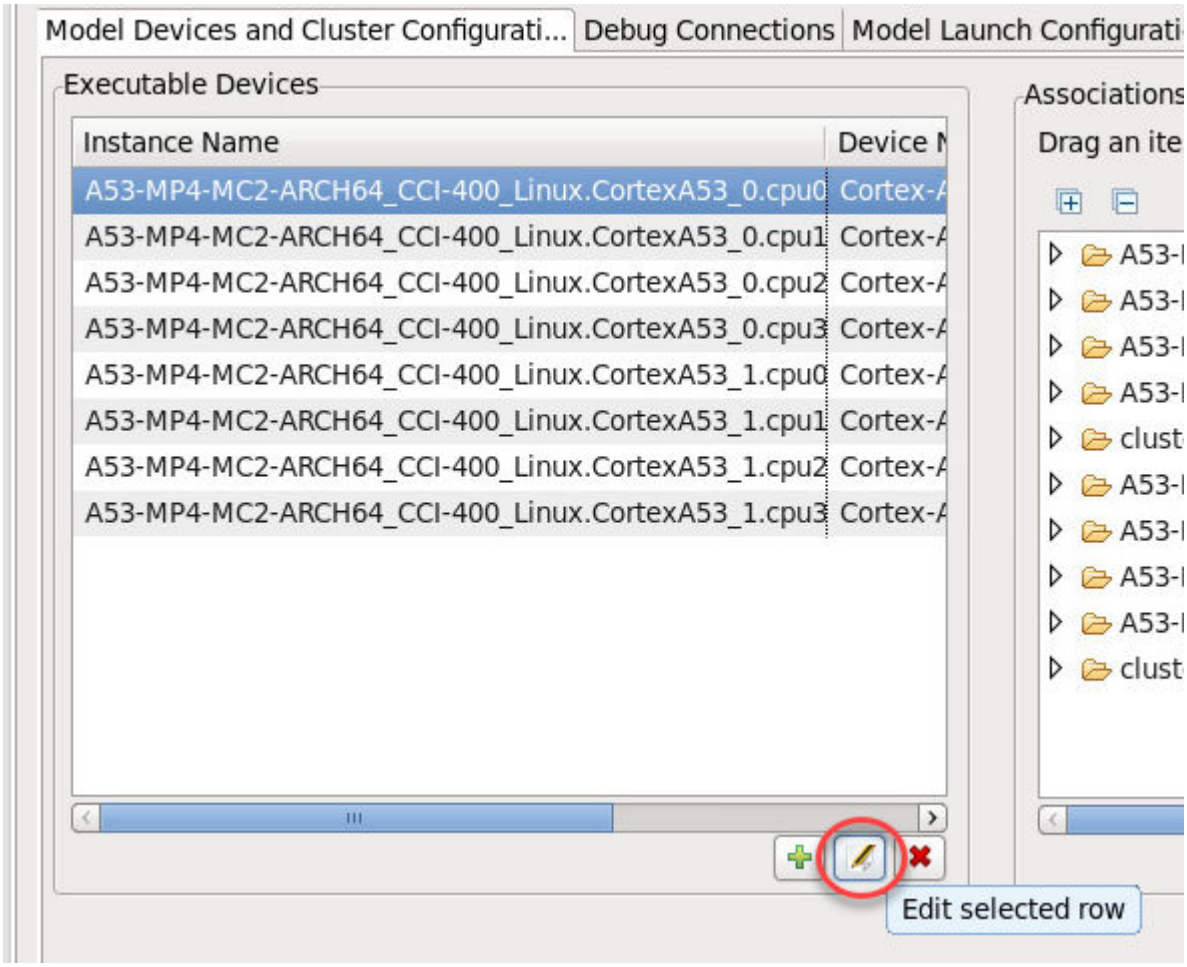

Figure 4-4 Selecting the CPU instance

3. In the **Edit instance** dialog box, set **Cluster** to `0`:

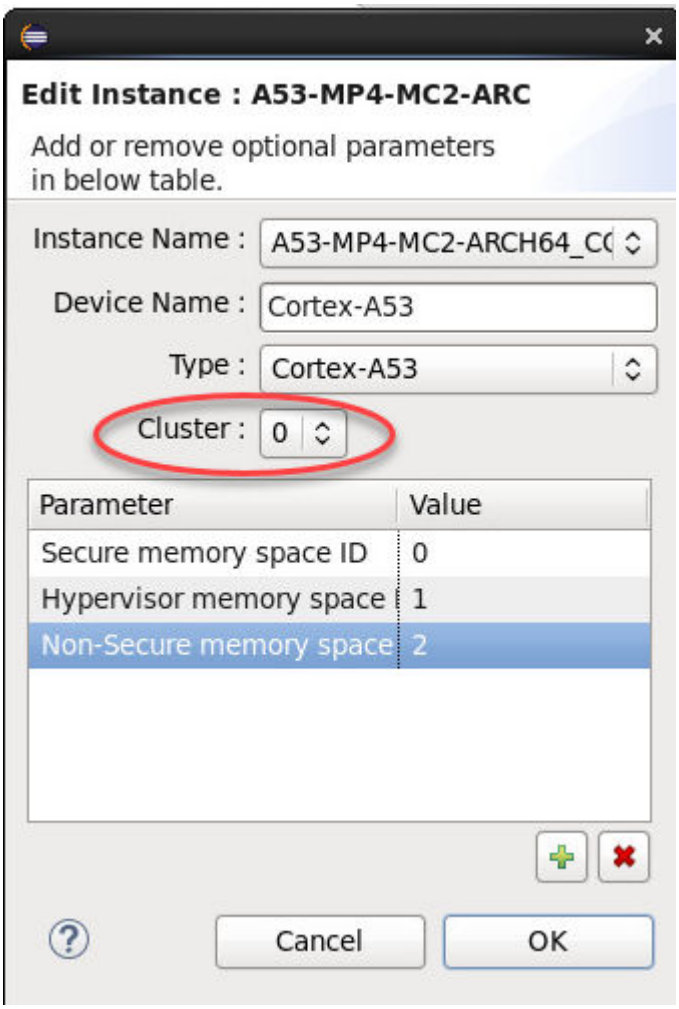

Figure 4-5 Selecting the CPU instance

4. Make sure that the parameter values are set as follows:

Table 4-1 Memory space settings

Parameter	Value
Secure memory space ID	`0`
Hypervisor memory space ID	`1`
Non-Secure memory space ID	`2`

4.5.3 Specifying the application to debug

This section describes how to launch a debug session.

1. In the DS-5 user interface, click **Debug**. This launches the **Debug Configurations** dialog box.
2. Under the **Connections** tab, check that the model you want is listed as a target. For example, in the following figure, `Cortex-A53_0` is the target:

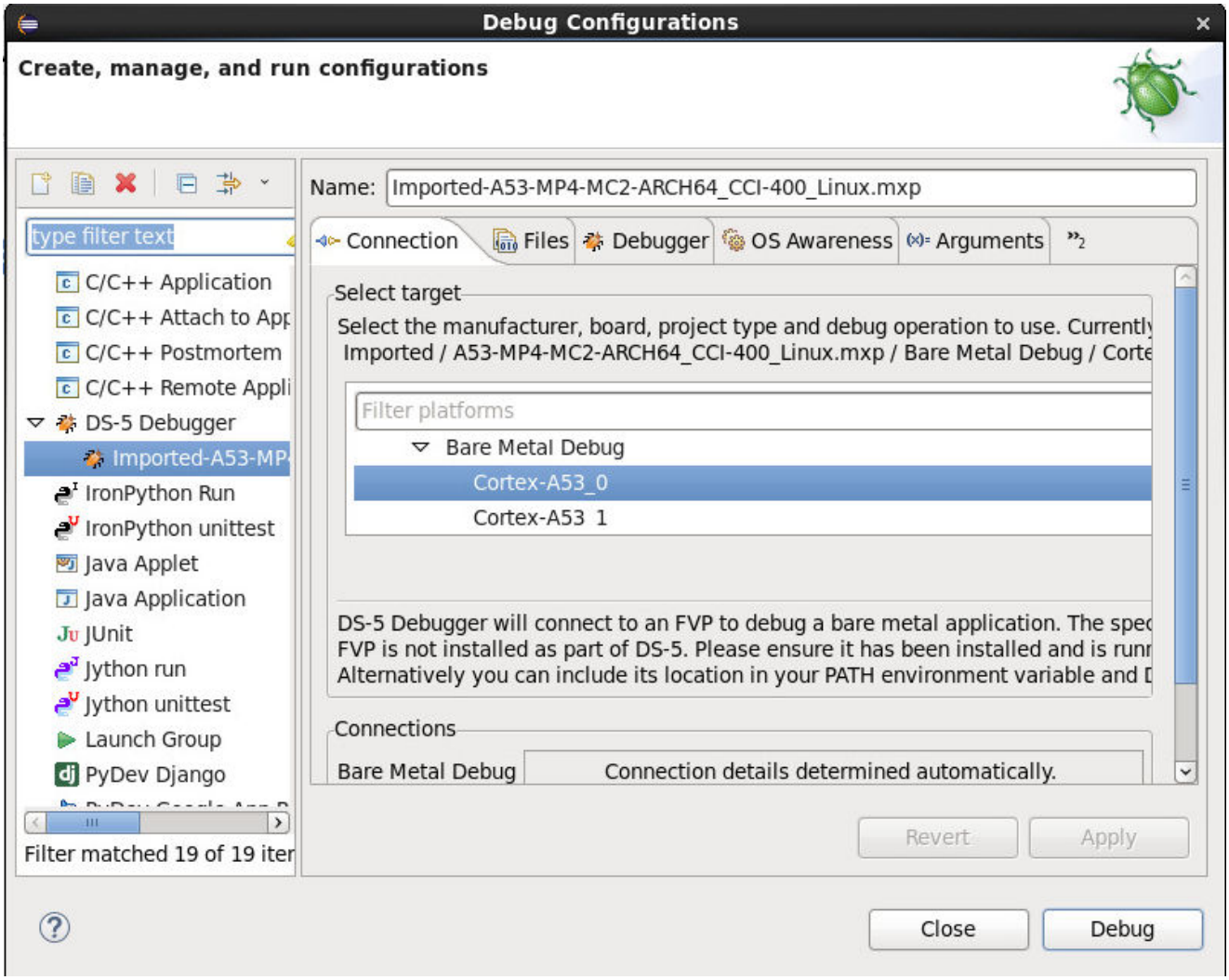

Figure 4-6 Selecting the target

3. Under the **Files** tab, browse to the application you want to run. For example, in the following figure, the application is `a53x4c2-v8-linux.axf`.

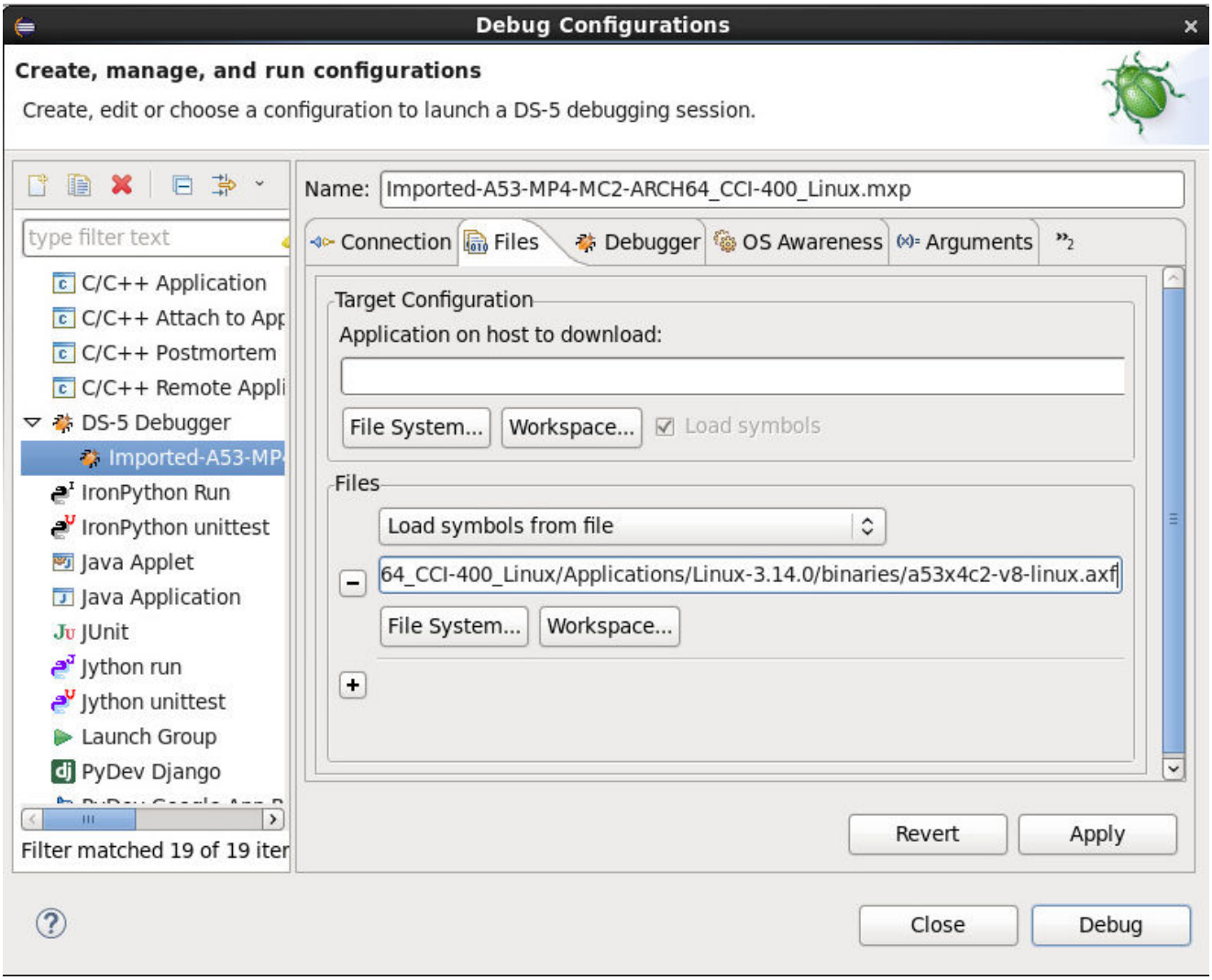

Figure 4-7 Selecting the application

4. Click **Apply**.
5. Under the **Debugger** tab, check that **Connect only** is selected in the **Run control** panel:

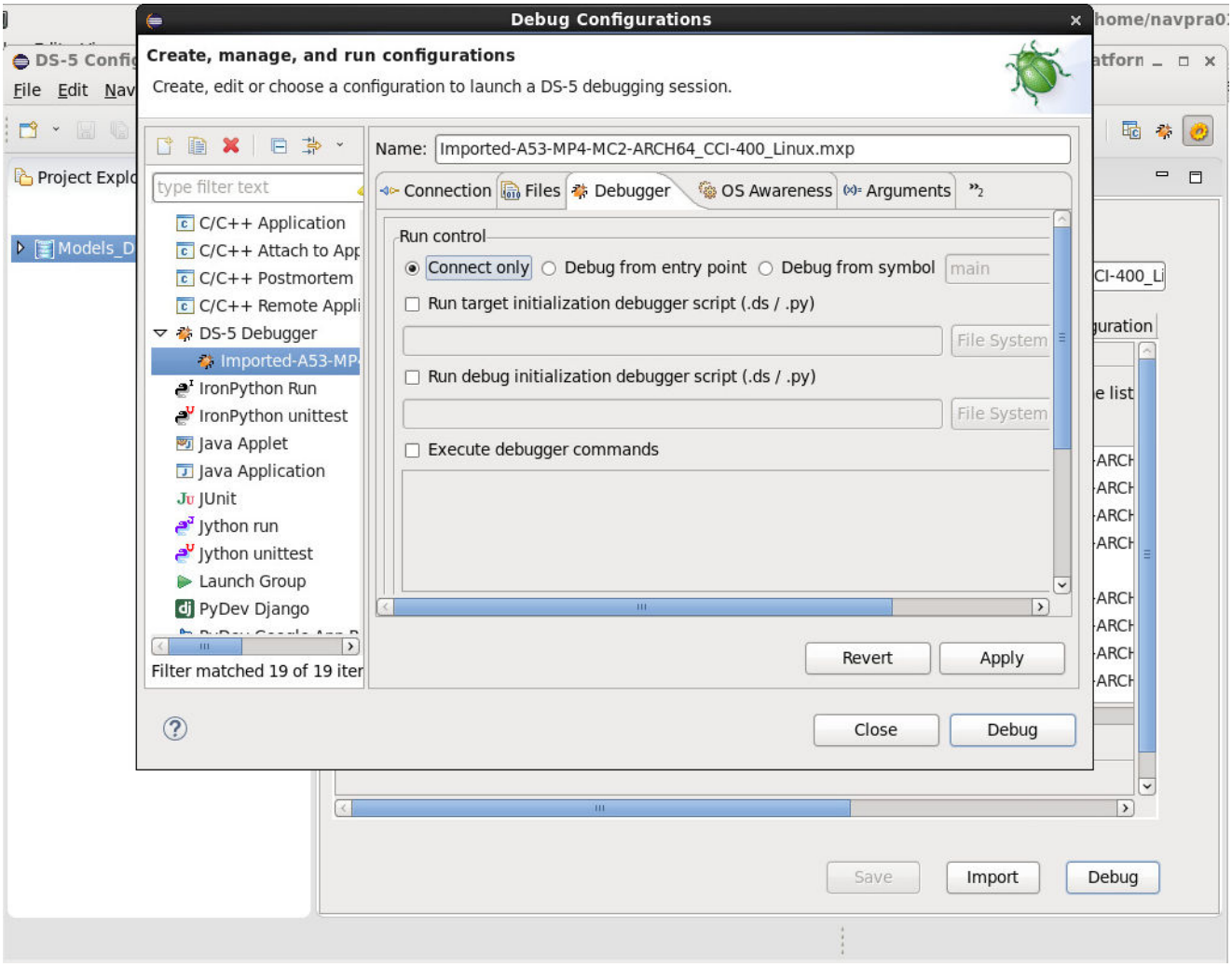

Figure 4-8 Selecting Connect only

6. Apply any changes and return to the **Connection** tab.
7. Select the core to debug:

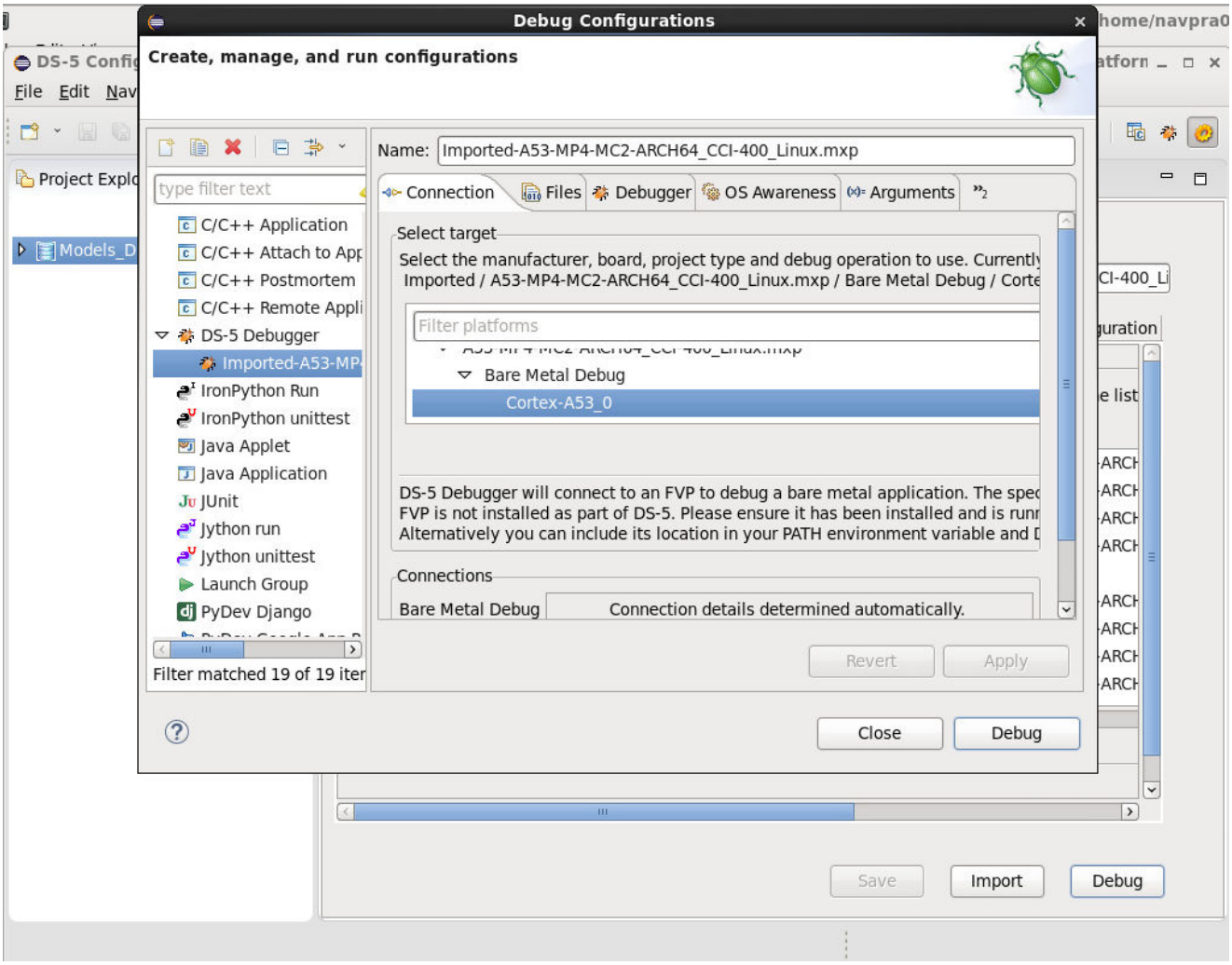

Figure 4-9 Selecting the core to debug

8. Click **Debug**. You are prompted to confirm the change to the debug perspective:

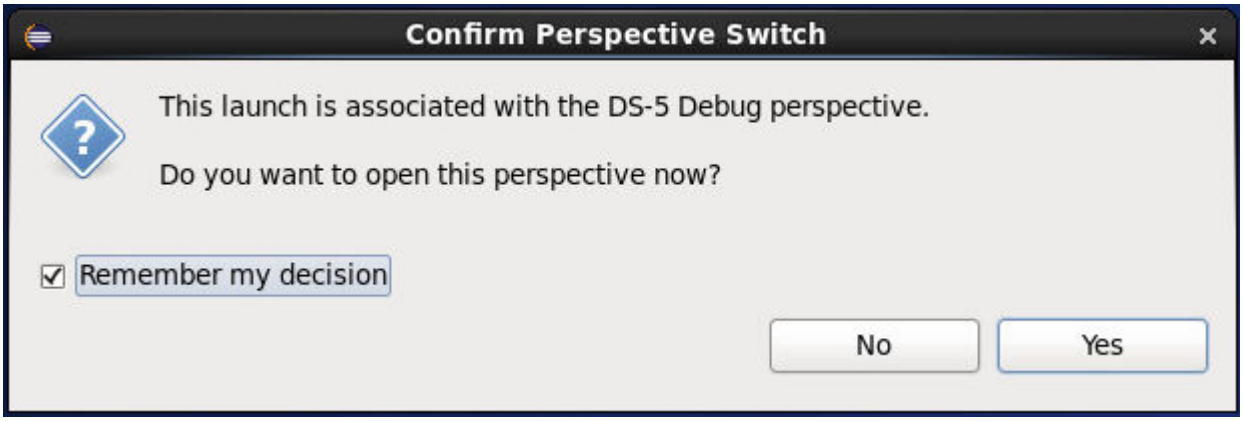

Figure 4-10 Confirming the perspective switch

9. Select **Remember my decision** to prevent subsequent prompts, and click **Yes**.

Result: The connected debug session launches in the DS-5 debug view:

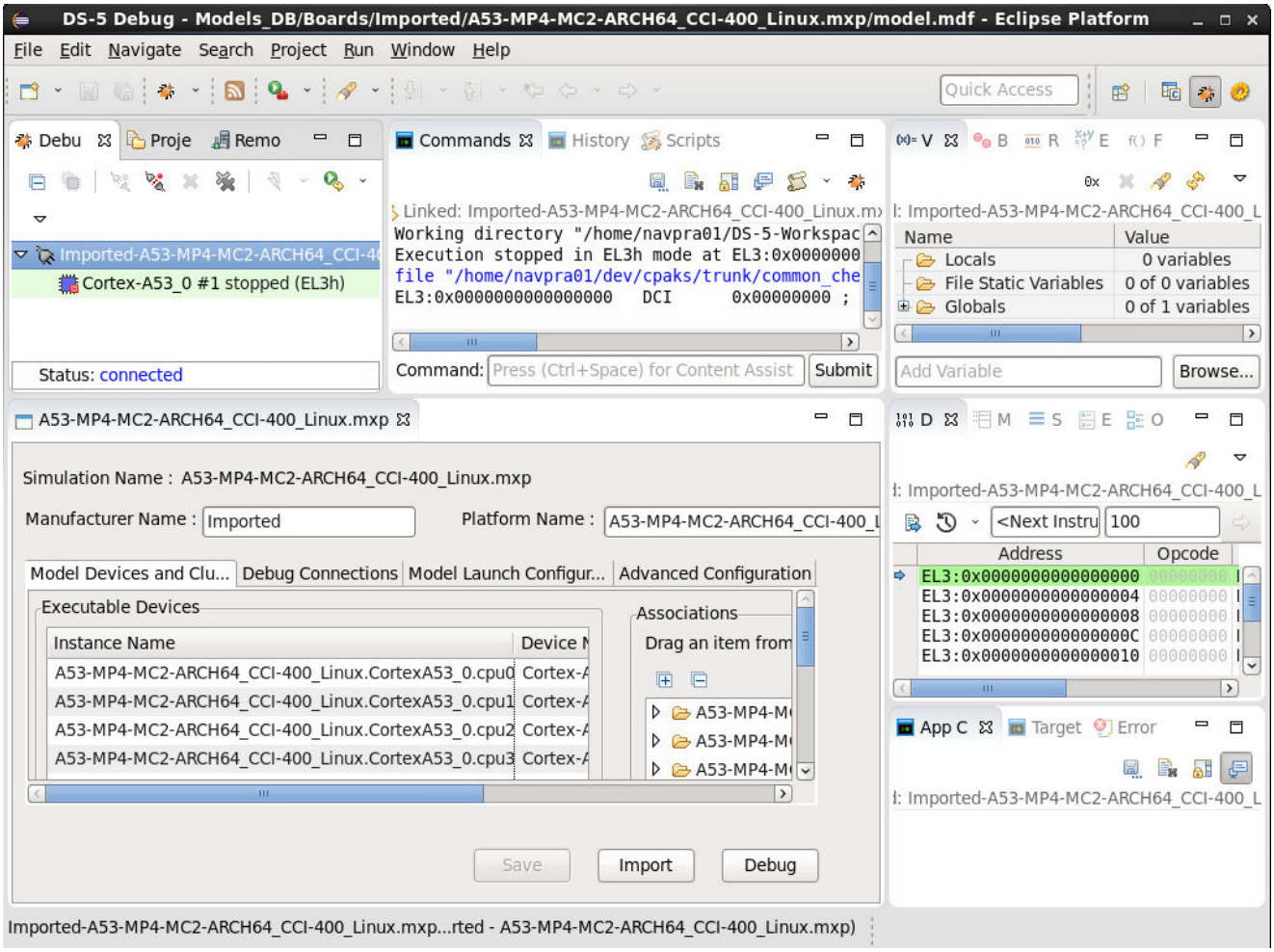

Figure 4-11 Connected debug session

4.6 Multicore debugging

This section contains information about debugging in SystemC Cycle Model multi-core and multi-cluster environments.

Multi-core, multi-cluster, and single-core debugging modes

For more information about debugging multi-core and multi-cluster targets, see the *Arm® DS-5 Debugger User Guide* (100953).

In SystemC Cycle Model multi-core and multi-cluster environments, you can specify whether to debug software running on multiple CPUs (this is the default), or whether to debug only on one CPU at a time.

To debug one CPU at a time, set the environment variable `CM_SCX_DEBUG_ONE` to `1` before running the simulation. When debugging a single CPU, only the CPU that hits a breakpoint has an accurate debug view. Impact on simulation performance in this mode is minimal, as only one CPU's pipeline is flushed.

To debug multiple CPUs, remove the environment variable `CM_SCX_DEBUG_ONE`.

Note

When debugging multiple CPUs, be aware that the impact on simulation performance is higher than when debugging one CPU at a time, because each of the core models performs additional debug logic to read data from internal pipelines. All CPUs attempt to accurately reflect the debug view, monitoring all CPU simulation stops, halts, single-steps, and breakpoints.

Timeouts and their effect on reliable debug views

This section describes how timeouts may interfere with reaching a debuggable point, and possible workarounds for timeouts. A *debuggable point* is a point in the simulation where the model's internal state can be accurately represented using architectural registers. Cycle Models must be at a valid debuggable point before they can provide a reliable debug view into registers and memory

If you issue a debugger halt, and one or more CPUs can not reach a debuggable point within the timeout interval, the simulation halt request times out, resulting in a warning similar to the following:

```
Warning: stop at a debug point failed: Simulation suspended before these target(s)
could reach debug point: r8_core.cpu1;r8_core.cpu3;
```

In these cases, the debug view of the affected CPU may show inaccurate values, and register or memory modifications are not allowed.

Scenarios that might cause a timeout include:

- Simulated software uses WFI (wait for interrupts) or WFE (wait for events), and after a single-step or breakpoint hit on a different CPU, the interrupts or events do not occur within the timeout window.
- Breakpoints within loops are not reached (see *4.4 Restrictions and limitations* on page 4-37). In these cases, lengthening the loop by adding `nop`s may allow the debugger to hit the breakpoint. For example:

```
end:
    nop
    nop
    nop
    nop
    B end
```

Workarounds to avoid timeouts and view the content of such cores include:

- avoid using WFI/WFE in the simulated software
- avoid tight loops such as:

```
self: branch self
```

- Changing the timeout setting (see *4.7 Changing the timeout setting* on page 4-49)

4.7 Changing the timeout setting

The timeout interval is counted by the simulation host. By default, the timeout interval is set to three seconds.

To change the timeout interval, set the environment variable `CM_SCX_STOP_TIMEOUT_SEC` before starting the simulation. For example, to set the timeout interval to five seconds using Linux bash shell:

```
export CM_SCX_STOP_TIMEOUT_SEC=5
```

The minimum interval allowed for this environment variable is one second.

Chapter 5
SystemC Export API function reference

This section describes the functions of the SystemC eXport (SCX) API that are supported by SystemC Cycle Models. Each description of a class or function includes the C++ declaration and the use constraints.

It contains the following sections:

5.1 scx::scx_initialize

This function initializes the simulation.

Initialize the simulation before constructing any exported subsystem.

```
void scx_initialize(const std::string &id,
                    scx_simcontrol_if *ctrl = scx_get_default_simcontrol());
```

`id`
: an identifier for this simulation.

`ctrl`
: a pointer to the simulation controller implementation. It defaults to the one provided with Arm models.

Note

Arm recommends specifying a unique identifier across all simulations running on the same host.

5.2 scx::scx_load_application

This function loads an application in the memory of an instance.

```
void scx_load_application(const std::string &instance,
                          const std::string &application);
```

`instance`
: the name of the instance to load into. The parameter `instance` must start with an EVS instance name, or with `"*"` to load the application into the instance on all EVSs in the platform. To load the same application on all cores of an SMP processor, specify `"*"` for the core instead of its index, in parameter `instance`.

`application`
: the application to load.

Note

The loading of the application happens at `start_of_simulation()` call-back, at the earliest.

5.3 scx::scx_set_parameter

This function sets the value of a parameter in components present in EVSs or in plug-ins.

- ```
  bool scx_set_parameter(const std::string &name, const std::string &value);
  ```
- ```
  template<class T>
  bool scx_set_parameter(const std::string &name, T value);
  ```

`name`
: the name of the parameter to change. The parameter `name` must start with an EVS instance name for setting a parameter on this EVS, or with `"*"` for setting a parameter on all EVSs in the platform, or with a plug-in prefix (defaults to `"TRACE"`) for setting a plug-in parameter.

`value`
: the value of the parameter.

This function returns `true` when the parameter exists, `false` otherwise.

Note

- Changes made to parameters within System Canvas take precedence over changes made with `scx_set_parameter()`.
- You can set parameters during the construction phase, and before the elaboration phase. Calls to `scx_set_parameter()` after the construction phase are ignored.
- You can change run-time parameters after the construction phase with the debug interface.
- Specify plug-ins before calling the platform parameter functions, so that the plug-ins load and their parameters are available. Any plug-in that is specified after the first call to any platform parameter function is ignored.

5.4 scx::scx_get_parameter

This function retrieves the value of a parameter from components present in EVSs or from plug-ins.

- `bool scx_get_parameter(const std::string &name, std::string &value);`
- `template<class T>`
  `bool scx_get_parameter(const std::string &name, T &value);`
- `bool scx_get_parameter(const std::string &name, int &value);`
- `bool scx_get_parameter(const std::string &name, unsigned int &value);`
- `bool scx_get_parameter(const std::string &name, long &value);`
- `bool scx_get_parameter(const std::string &name, unsigned long &value);`
- `bool scx_get_parameter(const std::string &name, long long &value);`
- `bool scx_get_parameter(const std::string &name, unsigned long long &value);`
- `std::string scx_get_parameter(const std::string &name);`

`name`
: the name of the parameter to retrieve. The parameter `name` must start with an EVS instance name for retrieving an EVS parameter or with a plug-in prefix (defaults to `"TRACE"`) for retrieving a plug-in parameter.

`value`
: a reference to the value of the parameter.

The `bool` forms of the function return `true` when the parameter exists, `false` otherwise. The `std::string` form returns the value of the parameter when it exists, empty string (`""`) otherwise.

Note

Specify plug-ins before calling the platform parameter functions, so that the plug-ins load and their parameters are available. Any plug-in that is specified after the first call to any platform parameter function is ignored.

5.5 scx::scx_get_parameter_list

This function retrieves a list of all parameters in all components present in all EVSs and from all plug-ins.

```
std::map<std::string, std::string> scx_get_parameter_list();
```

The parameter names start with an EVS instance name for EVS parameters or with a plug-in prefix (defaults to `"TRACE"`) for plug-in parameters.

Note

- Specify plug-ins before calling the platform parameter functions, so that the plug-ins load and their parameters are available. Any plug-in that is specified after the first call to any platform parameter function is ignored.
- If `scx_set_parameter()` is called after the simulation elaboration phase, the new value is not set in the model, although it is returned by `scx_get_parameter_list()`.

5.6 scx::scx_cpulimit

Sets the maximum number of CPU (User + System) seconds to run, excluding startup and shutdown.

```
void scx_cpulimit(double t);
```

`t`

the number of seconds to run. Defaults to unlimited.

5.7 scx::scx_timelimit

Sets the maximum number of seconds to run, excluding startup and shutdown.

```
void scx_timelimit(double t);
```

`t`

the number of seconds to run. Defaults to unlimited.

5.8 scx::scx_parse_and_configure

This function parses command-line options and configures the simulation accordingly.

```
void scx_parse_and_configure(int argc,
                char *argv[],
                const char *trailer = NULL,
                bool sig_handler = true);
```

`argc`
: the number of command-line options listed with `argv[]`.

`argv`
: command-line options.

`trailer`
: a string that follows the option list when printing help message (`--help` option).

`sig_handler`
: whether to enable signal handler function, `true` to enable (default), `false` to disable.

This function calls `std::exit(EXIT_SUCCESS)` to exit. It calls `std::exit(EXIT_FAILURE)` if there was an error in the parameter specification, or an invalid option was specified, or if the application or plug-in was not found.

Options

The application must pass the values of the options from function `sc_main()` as arguments to this function. The following options are supported:

--application, -a [INST=]*FILE*

This option specifies the application to load. The application to load must be the first argument on the command line.

Note

Use this option only for CPAKs with TLM models. For CPAKs with pin-level models, specifying `--application` has no effect and results in multiple warnings. The application for CPAKs with pin-level models is determined by the contents of the hex files in the CPAK `Systems` directory. See the CPAK `README.txt` file for more information.

`[INST=]`
: Specifies the core instance on which to load the application. This field is optional for Symmetric Multiprocessor (SMP) cores.

`FILE`
: Specifies the test case or application to be loaded.

The following example loads `test0.elf` on core 0, and `test1.elf` on core 1:

```
$ ./system_test -a CortexR8_core0=test0.elf -a CortexR8_core1=test1.elf -S -p
```

The following example for SMP cases loads `test.elf` on all cores:

```
$ ./system_test -a test.elf -S -p
```

--cadi-log, -L

This option logs all CADI calls to an XML log file. The simulation generates one XML log file per CPU and outputs them to the CPAK `Systems` directory with the filename `CADIlog-`*`model`*`_core.cpu`*`cpu-process_ID`*`.xml`. A cluster-level XML log file is also generated and output to this location with the filename `CADIlog-`*`model`*`_core-`*`process_ID`*`.xml`

For example:

```
$ ./system_test -L
```

--cadi-server, -S *FILE*

This option instructs a CADI server to wait for a debugger to connect and receive commands (such as `run`) before starting the simulation. If `-S` is not specified, the simulation starts immediately and connection to a CADI client or debugger is not allowed.

FILE
Specifies the test case or application to be loaded.

For example:

```
$ ./system_test test.elf -S
```

--config-file, -f *FILE*

This option loads model parameters from the specified configuration file.

FILE
Name of the configuration file.

For example:

```
$ ./system_test --config-file R8_config.cfg
```

--cpulimit

Maximum number of CPU (User + System) seconds to run, excluding startup and shutdown. Defaults to unlimited.

--help, -h

This option prints descriptions of available command line options.

Note

Arm Models support the full set of options that are printed when you enter `--help` or `-h`. Currently, Arm SystemC Cycle Models support a subset of these options. The options supported by this release of SystemC Cycle Models are described in this section.

For example:

```
$ ./system_test --help
```

--list-params, -l

This option prints a list of model parameters to standard output.

For example:

```
$ ./system_test -l
.
.
.
[plover_tarmac] Sending stream to 'plover_tarmac_decode -f
tarmac.PLOVERINTEGRATION_u_plover_u_noram1_wrapper_u_noram1.log'
Starting Sim
# Parameters:
# instance.parameter=value       #(type, mode) default = 'def value' : description :
[min..max]
#-------------------------------------------------------------------------------
REMOTE_CONNECTION.CADIServer.enable_remote_cadi=0     # (bool  , init-time) default =
'0'      : Allow connections from remote hosts
REMOTE_CONNECTION.CADIServer.listen_address=127.0.0.1  # (string, init-time) default =
'127.0.0.1' : Network address the server should listen on if enable_remote_cadi is set
("127.0.0.1" by default)
REMOTE_CONNECTION.CADIServer.port=31627              # (int   , init-time) default =
'0x7b8b' : TCP port the server should listen on if enable_remote_cadi is set (31627 by
default)
REMOTE_CONNECTION.CADIServer.range=0                 # (int   , init-time) default =
'0x0'    : If requested port is not avaliable, search for next avaliable port in range:
[port:port+range] (0 by default, only try specified port)
```

```
cortexr8_core.ACLKENSC=1                                # (int   , run-time ) default =
'0x1'    : ACLKENSC enable parameter
cortexr8_core.ACLKENST=1                                # (int   , run-time ) default =
'0x1'    : ACLKENST enable parameter
cortexr8_core.AFVALIDMD0=0                              # (int   , run-time ) default =
'0x0'    : Default value for AFVALIDMD0
cortexr8_core.AFVALIDMD1=0                              # (int   , run-time ) default =
'0x0'    : Default value for AFVALIDMD1
cortexr8_core.AFVALIDMD2=0                              # (int   , run-time ) default =
'0x0'    : Default value for AFVALIDMD2
cortexr8_core.AFVALIDMD3=0                              # (int   , run-time ) default =
'0x0'    : Default value for AFVALIDMD3
.
.
.
```

--list-regs

This option prints a list of model registers that are supported for viewing with a debugger. See also *Supported registers*. See the Technical Reference Manual for your IP for register descriptions.

For example:

```
$ ./system_test --list-regs
```

--quiet

Run quiet, suppress informational output.

--parameter, -C [INST.]*PARAMETER*=*VALUE*

This option sets the specified model parameter using the format `: -C INST.PARAM=VALUE`

`[INST=]`
Specifies the core instance. This field is optional for Symmetric Multiprocessor (SMP) cores.

PARAMETER
Specifies the parameter to set.

VALUE
Specifies the parameter value.

For example:

```
$ ./system_test -C cortexr8_core0.LOAD_DTCMS=true
```

--print-port-number, -p

This option causes the CADI server to print the TCP/IP port it is listening to.

For example:

```
$ ./system_test -S -p
.
.
.
Starting Sim
CADI server started listening to port 7001

Info: R8-MP4-SysC: CADI Debug Server started for ARM Models...
```

--stat

This option prints run statistics on simulation exit.

```
$ ./system_test -S --stat
```

After the simulation ends, statistics such as those shown in the following example are output:

```
--- R8-MP4-SysC statistics: -------------------------------------------------
Simulated time                          : 0.000000s
User time                               : 0.028996s
System time                             : 0.002999s
Wall time                               : 4.278761s
```

```
cortexr8_core.cpu0                      :   0.00 KIPS (           0 Inst)
cortexr8_core.cpu1                      :   0.00 KIPS (           0 Inst)
cortexr8_core.cpu2                      :   0.00 KIPS (           0 Inst)
cortexr8_core.cpu3                      :   0.00 KIPS (           0 Inst)
--------------------------------------------------------------------------------
```

--timelimit, -T

Maximum number of seconds to run, excluding startup and shutdown. Defaults to unlimited.

5.9 scx::scx_print_statistics

This function specifies whether to enable printing of simulation statistics at the end of the simulation.

```
void scx_print_statistics(bool print = true);
```

`print`

`true` to enable printing of simulation statistics, `false` otherwise.

Note

- You cannot enable printing of statistics once simulation starts.
- The statistics include LISA `reset()` behavior run time and application load time. A long simulation run compensates for this.

Appendix A
Migrating from previous SystemC Cycle Model versions

This section contains instructions specific to upgrading from a previous verison of SystemC Cycle Models to version 10.0.

It contains the following section:

A.1 Migrating from previous versions

Perform the actions in this section if you are upgrading to SystemC Cycle Models version 10.0 from a previous version.

Make the following additions to the `LD_LIBARY_PATH` environment variable:

- Add `MODELS/`*`component name`*`/gcc483/SystemC/lib`. This addition is required because `MODELS/`*`component name`*`/gcc483/SystemC/lib/lib`*`component`*`.a` has been replaced with `lib`*`modelname`*`.icm.so`.
- Add *`runtime install_path`*`/ARM/CycleModels/Runtime/cm_sysc/`*`version`*`/lib/Linux64_GCC-`*`x.y`*.

  Note

  In this filepath, *`version`* is the latest Cycle Model SystemC Runtime version (not the model version), and *`x.y`* is the GCC version.

  This addition is required because the installation package includes a new library, *`runtime install_path`*`/ARM/CycleModels/Runtime/cm_sysc/`*`version`*`/lib/Linux64_GCC-`*`x.y`*`/libicm_runtime.so`, which you are required to link.

SystemC Cycle Models version 10.0 includes TLM socket name changes; for example, `core.iSkt_M0` has changed to `core.iSkt_ACE5_Master_M0`. See the file `lib`*`component`*`.tlm.h` in your installation directory for socket names.