Cortex-A53 SystemC Cycle Model

Version 11.2

User Guide

101530_1102_01_en

Cortex-A53 SystemC Cycle Model

User Guide

Release Information

Document History

Issue	Date	Confidentiality	Change
1000-00	20 February 2019	Non-Confidential	First release
1100-00	31 May 2019	Non-Confidential	Release 11.0
1102-00	15 March 2020	Non-Confidential	Release 11.2
1102-01	01 May 2021	Non-Confidential	Documentation update

Non-Confidential Proprietary Notice

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

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

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

This document may include technical inaccuracies or typographical errors.

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

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

If any of the provisions contained in these terms conflict with any of the provisions of any click through or signed written agreement covering this document with Arm, then the click through or signed written agreement prevails over and supersedes the conflicting provisions of these terms. This document may be translated into other languages for convenience, and you agree that if there is any conflict between the English version of this document and any translation, the terms of the English version of the Agreement shall prevail.

The Arm corporate logo and words marked with ® or ™ are registered trademarks or trademarks of Arm Limited (or its subsidiaries) in the US and/or elsewhere. All rights reserved. Other brands and names mentioned in this document may be the trademarks of their respective owners. Please follow Arm's trademark usage guidelines at *https://www.arm.com/company/policies/trademarks*.

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

developer.arm.com

Progressive terminology commitment

Arm values inclusive communities. Arm recognizes that we and our industry have used terms that can be offensive. Arm strives to lead the industry and create change.

We believe that this document contains no offensive terms. If you find offensive terms in this document, please contact *terms@arm.com*.

Contents

Cortex-A53 SystemC Cycle Model User Guide

Chapter 5 SystemC Export API function reference

Preface

This preface introduces the *Cortex-A53 SystemC Cycle Model User Guide*.

It contains the following:
- *About this book* on page 7.

About this book

This guide describes how to integrate the Cortex®-A53 SystemC Cycle Model into a SystemC design and simulation environment.

Using this book

This book is organized into the following chapters:

Chapter 1 Cycle Model functionality and operating requirements
This section introduces the Arm Cortex®-A53 SystemC Cycle Model.

Chapter 2 Integrating models into your environment
This section describes using the Cycle Models Configuration Tool to extract required build options from Arm models, and how to specify custom build options.

Chapter 3 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 4 Debugging SystemC Cycle Models with Arm® Development Studio
This section describes how to connect the Arm Development Studio Debugger to Arm SystemC Cycle Models in a Cycle Model reference platform.

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.

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.

`monospace`
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 *Cortex-A53 SystemC Cycle Model User Guide*.
- The number 101530_1102_01_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® Documentation*.
- *Technical Support*.
- *Arm® Glossary*.

Chapter 1
Cycle Model functionality and operating requirements

This section introduces the Arm Cortex®-A53 SystemC Cycle Model.

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 IEEE 1666-compliant simulator. You can use SystemC Cycle Models within an Arm Cycle Model reference platform, or integrate them directly into any IEEE 1666-compliant SystemC environment.

It contains the following sections:

- *1.1 Functionality of the SystemC Cycle Model* on page 1-10.
- *1.2 Prerequisites to using SystemC Cycle Models* on page 1-11.
- *1.3 Supported platforms, compilers, and simulators* on page 1-12.

1.1 Functionality of the SystemC Cycle Model

The Arm Cortex-A53 SystemC Cycle Model simulates the Cortex-A53 MPCore processor.

Supported functionality

Most hardware features have been implemented. See the *Arm® Cortex®-A53 Technical Reference Manual* (100372) for more information. Functionality differences are described in the next section.

Unsupported hardware features

The following features of the Cortex-A53 hardware are not implemented in this release of the Cortex-A53 Cycle Model:

- SCU cache protection.
- CPU cache protection.
- The full set of registers. The functionality of all registers, however, does exist and can be accessed by software running on the virtual platform. See *4.3.1 Supported registers* on page 4-34 for information about getting a list of supported registers.

Additional features for Cycle Model usability

To enhance usability, the following features have been added to the Cycle Model, which do not exist in the Cortex-A53 hardware:

- Waveform dumping, including dumping of TCM memories. See *3.4 Dumping waveforms* on page 3-26.
- Support for viewing register values. See *4.3.1 Supported registers* on page 4-34 to learn about registers exposed on the Cycle Model.
- Support for debug view of internal and external memory contents. See *4.3.2 Supported memory views* on page 4-35 for information.
- Support for debug view of disassembly data. See your debugger documentation for information about accessing disassembly data.

1.2 Prerequisites to using SystemC Cycle Models

Review the prerequisites in this section for using Arm SystemC Cycle Models.

Details about the following prerequisites can be found in the *Cycle Model SystemC Runtime Installation Guide* (101146):

- Supported Cycle Model SystemC Runtime must be installed in your environment.
- Supported GCC version must be installed in your environment.
- Supported Cycle Model Studio Runtime is required for simulation and recompilation. This is installed as part of the SystemC Runtime.
- Configured SystemC environment.

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.

Prerequisites for Cycle Model reference platform environments

All models in a Cycle Model reference platform must be the same release (for example, all v10.x or all v11.x). Mixing different versions within a reference platform is not supported, and results in incorrect Cycle Model behavior, incorrect Tarmac results, or other issues.

Reference platforms may have additional prerequisites. See the *Cycle Model Reference Platform Getting Started Guide* (101497).

1.3 Supported platforms, compilers, and simulators

This section describes the requirements for running SystemC Cycle Models.

This section contains the following subsections:
- *1.3.1 Supported platforms* on page 1-12.
- *1.3.2 Supported compilers* on page 1-12.
- *1.3.3 Supported simulators* on page 1-12.

1.3.1 Supported platforms

Arm SystemC Cycle Models are supported to run on Red Hat Enterprise Linux versions 6.6 (64-bit) and 7 (64-bit).

1.3.2 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.

1.3.3 Supported simulators

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

Chapter 2
Integrating models into your environment

This section describes using the Cycle Models Configuration Tool to extract required build options from Arm models, and how to specify custom build options.

It contains the following sections:

- *2.1 Extracting build options using the Cycle Models Configuration Tool* on page 2-14.
- *2.2 Adding custom options to the Makefile* on page 2-20.

2.1 Extracting build options using the Cycle Models Configuration Tool

To integrate an Arm model into your build flow, use the Cycle Models Configuration Tool to extract its build options.

The Cycle Models Configuration Tool is a command-line utility included with the SystemC Cycle Model Runtime. It provides a standard interface to the Cycle Model SystemC Runtime and Model packages.

The Cycle Models Configuration Tool simplifies integration of models into your systems, build flow, or custom `Makefile` by extracting the required build and link options for all Arm Cycle Model components in the model or reference platform.

The Cycle Models Configuration Tool also flags incompatibilities between individual model requirements within a system. For example, if you add a new model to an existing system, the Cycle Models Configuration Tool determines the version of the SystemC Cycle Model Runtime that satisfies the version requirements of all of the models.

You can run the Cycle Models Configuration Tool at the command line or as part of the build flow.

Restrictions and limitations

The following restrictions and limitations apply to the Cycle Models Configuration Tool:

- For use on 64-bit Linux platforms only.
- Tested on GCC 4.8.3 and GCC 6.4.0.
- The Cycle Models Configuration Tool uses the directory it is run from as the default `searchpath`; use the `--searchpath` option to specify a different location to search.
- Backward compatibility is limited to Version 11.0 (and later) models. These models contain the data files required by the Cycle Models Configuration Tool.
- Models built on IP Exchange contain the data files required by the Cycle Models Configuration Tool. If you are working in a reference platform environment with models that were not built on IP Exchange, you must explicitly make the build options available to the `Makefile`. Your reference platform will not build successfully without the required build options for all components. See the *Cycle Model Reference Platform Getting Started Guide* for more information.

This section contains the following subsections:

- *2.1.1 Cycle Models Configuration Tool command syntax* on page 2-14.
- *2.1.2 Cycle Models Configuration Tool examples* on page 2-18.

2.1.1 Cycle Models Configuration Tool command syntax

Extracts compiler, link, and source data and dependencies for specified components.

Syntax

```
cm_config [-h] [--verbosity [{debug,error,info,warning}]] [--version]
[--list] [--list-req] [--use-tool USE-TOOL]
[--searchpath SEARCHPATH [SEARCHPATH ...]]
[--model MODEL [MODEL ...]] [--ignore IGNORE [IGNORE ...]]
[--compile [{defines,flags,includes}]] [--sources]
[--link [{dirs,dirs_rt,flags,libs}]]
[--model-type [{pin,tlm}]] [--use-env USE-ENV [USE-ENV ...]]
[--use-arm]
```

Arguments

--compile [{defines, flags, includes}]

Optional.
Outputs compile options for the specified component or components. By default, defines, flags, and includes are output. Optionally, you can specify one or more of the following options to ouput only the related data:

- `defines`
- `flags`
- `includes`

This example outputs define, flag, and link data:

```
$ cm_config --use-tool gcc:6.4.0 --searchpath ./ --model cms --compile
```

This example outputs define and flag data only:

```
$ cm_config --use-tool gcc:6.4.0 --searchpath ./ --model cms --compile defines --
compile flags
```

-h, --help

Optional.
Shows command help and exits.

Example:

```
$ cm_config --help
```

--ignore [{cms, cm_sysc, SystemC, model}]

Optional.
Directs the Cycle Models Configuration Tool to ignore the specified data when returning compiler, build, or link information. Use a space delimiter when specifying one or more of the following options:

- `cms` ignores data related to the Cycle Model Studio Runtime
- `cm_sysc` ignores data related to the SystemC Cycle Model Runtime
- `SystemC` ignores data related to the SystemC environment
- `component` ignores model- or component-related data. Use the --list argument for the exact component name.

Example:

```
$ cm_config --use-tool gcc:6.4.0 --searchpath MODELS --ignore cms cm_sysc SystemC --
model CortexA53
```

--link [{dirs, dirs_rt, flags, libs}]

Optional.
Outputs linker data for the specified component or components. Used without an option, returns directories, libraries, and flags. Optionally, specify one or more of the following options:

- `dirs`
- `dirs_rt` (returns the unformatted directories for dynamically loaded libraries)
- `flags`
- `libs`

This example returns directory, library, and flag data:

```
$ cm_config --use-tool gcc:6.4.0 --searchpath MODELS --model CortexA53 --link
```

This example returns flag and library data only:

```
$ cm_config --use-tool gcc:6.4.0 --searchpath MODELS --model CortexA53 --link flags
--link libs
```

--list

Optional.
Lists all available components. Optionally, use in combination with the `--searchpath` option to restrict to a particular directory.

Example:

```
$ cm_config --list
```

--list-req

Optional.
Lists all available components and the tools and components each one requires. Optionally, use in combination with the`--searchpath` option to restrict to a particular directory.

Example:

```
$ cm_config --list-req
```

--model MODEL [MODEL …]

Required unless the `--list` or `--list-req` option is used.
Specifies one or more components to retrieve information for. Optionally, specify a version with a comparison operator; for example: `"COMP_A>3.2.4"` or `"COMP_A > 3.2.4"`. Component names match the C++ class name defined at model build time. Versions must be only numbers and decimals. If greater or less than signs are used, the model name and version must be enclosed by quotations.

Example:

```
$ cm_config --use-tool gcc:6.4.0 --searchpath MyModelsAndRuntimeInstallPath --model
MyCPUModel MyInterconnectModel --
link
```

--model-type [{pin, tlm}]

Optional.
Models may be pin-based or TLM-based. By default, the Cycle Models Configuration Tool returns all data regardless of the model type. The --model-type argument returns only data related to the specified model type:

- `pin` returns pin-related data plus data common to both model types.
- `tlm` returns TLM-related data plus data common to both model types.

Example:

```
$ cm_config --use-tool gcc:6.4.0 --searchpath MODELS --model CortexA53 --model-type
tlm --link
```

--searchpath SEARCHPATH [SEARCHPATH ...]

Optional.
Specifies the directories to search for Models or Cycle Model SystemC Runtime components. When not specified, the Cycle Models Configuration Tool searches the directory in which the tool was run.

Example:

```
$ cm_config --use-tool gcc:6.4.0 --searchpath MODELS --model CortexA53 --link
```

--sources

Optional.
Outputs a list of source files.

Example:

```
$ cm_config --use-tool gcc:6.4.0 --searchpath MODELS --model CortexA53 --sources
```

--use-arm

Optional.
Extracts data only for Arm libraries and components. Recommended only when extracting data for custom flows.

Note

Use this option with care. Build failures may result if libraries other than Arm libraries are required to build an executable.

Example:

```
$ cm_config --use-tool gcc:6.4.0 --searchpath ./ --model CortexA53 --link libs --use-
arm
```

--use-env <COMPONENT>:<ENV> [<COMPONENT>:<env> ...]

Optional.
Formats data for one or more specified <component>:<env> pairs. For these components, the path data returned is relative to an environment variable that reflects the root of the component. Recommended for advanced users only.
Some examples of component pair options are:

- `cms:CARBON_HOME`
- `SystemC:SYSTEMC_HOME`
- `cm_sysc:CM_SYSC_HOME`
- `CortexM0Plus:MY_M0PLUS_HOME`

Example:

```
$ cm_config --use-tool gcc:6.4.0 --searchpath ./ --model cms --sources --use-env
cms:CARBON_HOME
```

--use-tool GCC:VERSION

Required.
Specifies which compiler and link options to return. Options are:

- `gcc:6.4.0`
- `gcc:4.8.3`

Example:

```
$ cm_config --use-tool gcc:6.4.0 --searchpath MODELS --model CortexA53 --sources
```

--verbosity VERBOSITY

Optional.
Specifies the verbosity of Cycle Models Configuration Tool execution feedback. Options are:

- `debug`
- `error` (default)
- `info`
- `warning`

Example:

```
$ cm_config --use-tool gcc:6.4.0 --searchpath MODELS --verbosity debug --model
CortexA53 --link
```

--version

Optional.
Returns the version of the Cycle Models Configuration Tool. Example:

```
$ cm_config --version
```

Related information

2.1.2 Cycle Models Configuration Tool examples on page 2-18

2.1.2 Cycle Models Configuration Tool examples

The examples in this section assume that the path for the Cycle Models Configuration Tool is part of the `PATH` environment variable (*install path*`/ARM/CycleModels/Runtime/cm_sysc/`*version*`/bin/`). Add the tool path to the `PATH` environment variable by sourcing one of the runtime setup scripts in `ARM/CycleModels/etc`.

Example use in a simple Makefile

Following is an example in which the compile and link steps are combined. There are two models: MyCPUModel and MyInterconnectModel. Both are in the directory `MyModelsAndRuntimeInstallPath`. The Cycle Models Configuration Tool is called once to create a list of source files, then a second time to retrieve all of the compile and link options.

```
# Tool name with baseline options.  Options that may change are specified here,
# such as compiler version, location of the Models, and the Model Names
CM_CONFIG:=cm_config --use-tool gcc:6.4.0 --searchpath MyModelsAndRuntimeInstallPath --
model MyCPUModel MyInterconnectModel
SRCS:=$(shell $(CM_CONFIG) --sources)

system: $(SRCS)
$(CXX) -o $@ $^ $(shell $(CM_CONFIG) --compile --link)
```

Example use in a complex Makefile

If your build flows separate includes, compiler flags, and linker options, use the arguments to the `--compile` option to return this data as shown:

```
CM_CONFIG:=cm_config --use-tool gcc:6.4.0 --searchpath MyModelsAndRuntimeInstallPath --
model MyCPUModel MyInterconnectModel
CINCS  := $(shell $(CM_CONFIG) --compile includes)
CFLAGS := $(shell $(CM_CONFIG) --compile flags)
LDOPTS := $(shell $(CM_CONFIG) --link)

SRCS := $(shell $(CM_CONFIG) --sources)
OBJS := $(patsubst %.cpp,%.o,$(SRCS))

system: system.o $(OBJS)
$(CXX) -o $@ $^ $(LDOPTS)

system.o: system.cpp
$(CXX) -c $(CFLAGS) $(CINCS) -o $@ $^

%.o: %.cpp
$(CXX) -c $(CFLAGS) $(CINCS) -o $@ $^
```

Example of retrieving source and link files for different model types

You may want to build a TLM or pin-level version of a SystemC Model.

The following example shows how to return the required file list and link options for a Cortex-A53 Cycle Model. The command specifies ignoring `cms`, `sysc`, and the protocol buffer directory, targeting only the model:

```
$ cm_config --use-tool gcc:6.4.0 --model CORTEXA53 --sources --link --model-type tlm --
ignore cms cm_sysc ProtocolBuffer_32-bit

/arm/models/A53_Model/gcc640/SystemC/univent_tarmac.cpp /arm/models/A53_Model/gcc640/SystemC/
libCORTEXA53.systemc.cpp /arm/models/A53_Model/gcc640/SystemC/CORTEXA53ResetImp.cpp /arm/
models/A53_Model/gcc640/SystemC/libCORTEXA53.tlm.cpp -L/arm/models/A53_Model/gcc640/SystemC -
L/arm/models/A53_Model/gcc640/SystemC/lib /arm/models/A53_Model/gcc640/SystemC/
univentUtil/lib/ca53_tarmac_dpi.so -lCORTEXA53.icm -licm_runtime
```

The following example shows how to return the required file list and link options for only the pin-level model:

```
$ cm_config --use-tool gcc:6.4.0 --model CORTEXA53 --sources --link --model-type pin --
ignore cms cm_sysc ProtocolBuffer_32-bit

/arm/models/A53_Model/gcc640/SystemC/univent_tarmac.cpp /arm/models/A53_Model/gcc640/SystemC/
libCORTEXA53.systemc.cpp -L/arm/models/A53_Model/gcc640/SystemC -L/arm/models/A53_Model/
```

```
gcc640/SystemC/lib /arm/models/A53_Model/gcc640/SystemC/univentUtil/lib/ca53_tarmac_dpi.so -
lCORTEXA53.icm -licm_runtime
```

Example of substituting environment variables for component roots

When extracting build data for integration in custom flows, you may need to substitute environment variables for component roots. In the following example, `CORTEXA53_HOME` is used as the model root. `SYSTEMC_HOME` is used for the SystemC root:

```
$ cm_config --use-tool gcc:6.4.0 --model CORTEXA53 --sources --link --model-type pin --
ignore cms cm_sysc ProtocolBuffer_32-bit  --use-env CORTEXA53:CORTEXA53_HOME
SystemC:SYSTEMC_HOME

${CORTEXA53_HOME}/gcc640/SystemC/univent_tarmac.cpp ${CORTEXA53_HOME}/gcc640/SystemC/
libCORTEXA53.systemc.cpp -L${CORTEXA53_HOME}/gcc640/SystemC -L${CORTEXA53_HOME}/gcc640/
SystemC/lib
${CORTEXA53_HOME}/gcc640/SystemC/univentUtil/lib/ca53_tarmac_dpi.so -lCORTEXA53.icm -
licm_runtime -L${SYSTEMC_HOME}/lib/Linux64_GCC-4.8 -lsystemc -lpthread
```

Example of extracting Arm® data

The following example shows using the `--use-arm` option to retrieve data owned or developed by Arm.

```
$ cm_config --use-tool gcc:6.4.0 --model CORTEXA53 --sources --link --model-type pin --
ignore cms cm_sysc ProtocolBuffer_32-bit --searchpath Runtime_v3.0 A53_Model --use-arm

/arm/models/A53_Model/gcc640/SystemC/univent_tarmac.cpp /arm/models/A53_Model/gcc640/SystemC/
libCORTEXA53.systemc.cpp -L/arm/models/A53_Model/gcc640/SystemC -L/arm/models/A53_Model/
gcc640/SystemC/lib /arm/models/A53_Model/gcc640/SystemC/univentUtil/lib/ca53_tarmac_dpi.so -
lCORTEXA53.icm -licm_runtime
```

2.2 Adding custom options to the Makefile

You may want to further customize your build, including using a different installation of SystemC than the one Arm includes in the runtime. In this case, you can use the information in this section to add build options into the `Makefile` without the need to edit it.

Arm Cycle Models support the flexibility to:

- Add arguments to the Cycle Models Configuration Tool command line. This is useful for adding searchpaths, models, or ignores.
- Specify build variables to add any extra sources and build options you may need, such as compile flags and defines, or link flags, directories, and libraries. The build variables also allow you to use your own version of SystemC.

Build variables

The following build variables exist in the model `Makefile`. In a Cycle Model reference platform environment, they are also present in the reference platform `Systems/Makefile`:

- `CM_CONFIG_ARGS` - Arguments added to the `cm_config` command line.
- `CXXFLAGS` - Compile flags, includes, and defines to be added into the build.
- `LDFLAGS` - Link flags, directories, and libraries to be added into the build.
- `RPATHS` - Runtime `rpaths` to be added into the build.
- `SRCS` - Sources to be added into the build.

The following build variable is present only in the model `Makefile`:

- `SRCS_TLM` - TLM sources to be added into the build.

Example 1: Specifying your own version of SystemC

The following example directs the Cycle Models Configuration Tool not to search for SystemC, and adds in build data for a custom SystemC installation, assuming `SYSTEMC_INC` and `SYSTEMC_LIB` are set to the `includes` and `library` directories:

```
$ make all CM_CONFIG_ARGS=’--ignore SystemC’ CXXFLAGS=’-I$SYSTEMC_INC’ LDFLAGS=’-L
$SYSTEMC_LIB -lsystemc’ RPATHS=’-Wl,-rpath,$SYSTEMC_LIB’
```

Example 2: Providing another runtime path

The following example provides a different runtime path than the default, allowing the Cycle Models Configuration Tool to pick the latest compatible runtime components:

```
$ make all CM_CONFIG_ARGS=`--searchpath path_to_alternative_runtime`
```

Example 3: Adding different debug or optimization parameters

The following example shows specifying alternate debug outputs and optimization parameters:

```
$ make all CXXFLAGS=`-g`
$ make all CXXFLAGS=`-ggdb`
$ make all CXXFLAGS=`-O3`
```

Chapter 3
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:

3.1 Connecting model ports

All pins must be bound to a signal.

For a list of the pins on the Cortex-A53 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 *3.1.1 Tied pins* on page 3-22.

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

This section contains the following subsections:

- *3.1.1 Tied pins* on page 3-22.
- *3.1.2 Port binding* on page 3-23.

3.1.1 Tied 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 Cortex-A53 SystemC Cycle Model, refer to the model header file `Lib`*`model`*`.Systemc.h`, or the `CM_IPXACT_`*`model`*`.xml` file. The following pins, which are listed in the header file and XML file, are tied and cannot be modified:

- CIHSBYPASS (low)
- CISBYPASS (low)
- CTICHIN (low)
- CTICHOUTACK (low)
- CTIIRQACK (low)
- DBGEN (high)
- DFTRAMHOLD (low)
- DFTMCPHOLD (low)
- DFTRSTDISABLE (low)
- DFTSE (low)
- nMBISTRESET (high)
- MBISTREQ (low)
- NIDEN (high)
- SPIDEN (high)
- SPNIDEN (high)

In AXI3 mode, the following ACE-specific signals are tied:

- SYSBARDISABLE (high)
- ACSNOOPM (low)
- ACADDRM (low)
- ACPROTM (low)
- ACVALIDM (low)
- ACREADYM (low)
- ARBARM (low)
- ARDOMAINM (low)
- ARSNOOPM (low)
- CRVALIDM (low)
- CRREADYM (low)
- CRRESPM (low)
- CDVALIDM (low)
- CDREADYM (low)
- CDDATAM (low)
- CDLASTM (low)

- RACKM (low)
- WACKM (low)

In ACE mode, the following AXI3-specific signal is tied:

- WIDM (low)

3.1.2 Port binding

This section summarizes how port binding and tying are implemented in Cycle Models, and how you can make changes.

By default, all signal ports of the model are bound to their corresponding internal sc_signal. This ensures that every signal port is bound, as required by SystemC, and prevents you from having to bind all ports even if they are not being used. These bindings are defined in the `CortexA53ResetImp.cpp` file located in the directory `gcc`*`version`*`/SystemC/` for the model.

The following example shows a portion of the port binding section of a `ResetImp.cpp` file:

```
// bind all the non-TLM ports to their corresponding signals
void CortexA53Imp::bind_nontlm_ports_to_signals()
{
#ifndef CM_SYSC_DONT_BIND_NONTLM_PORTS
    CLKEN.bind(CLKENsignal);
    FCLKEN.bind(FCLKENsignal);
    HCLKEN.bind(HCLKENsignal);
    CLK1EN.bind(CLK1ENsignal);
    .
    .
    .
```

If you need to tie these signals to a specific value or bind them to an external sc_signal, then the internal binding needs to be removed.

Procedure

To change the default port binding in the context of an Arm Cycle Model reference platform:

1. In the `CortexA53ResetImp.cpp` file, inside the `bind_nontlm_ports_to_signals()` function, remove the binding by commenting out the line for the port.
2. In the reference platform `Systems/system_test.cpp` testbench file, the unbound port can be either:
   - Driven directly from `sc_main`; e.g., `sc_signal.write(1)`.
   - Bound to another sc_modules port; e.g., *`another_sc_module.port`*`.bind(`*`cycle_model.port`*`)`. You can create a new sc_module in a separate `.cpp` file, or include it in the existing `Systems/system_test.cpp` file.
3. Recompile the reference platform using `Systems/Makefile`. This recompiles the Cycle Model and the platform.

3.2 Resetting the SystemC Cycle Model

A default reset sequence is provided in source form in the model directory `gcc`*`version`*`/SystemC/`.

If necessary, you can modify this file as needed to work with your system:

- For pin-level models, the file is `CortexA53ResetModule.h`
- For TLM models, the file is `CortexA53ResetImp.cpp`

After modifications, recompile the model. For pin-level models, ensure that the reset module is connected to the model (this step is not necessary for TLM models).

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

3.3 Setting model parameters

This section describes how to see a list of the parameters on the Cortex-A53 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.

Run-time parameters

For run-time parameters, change the parameter value on the command line using `-C INST.PARAM=VALUE` or `--parameter INST.PARAM=VALUE`.

The following example restarts the simulation, specifying the hello_world application with waveform dumping enabled:

```
$./system_test -a ../Applications/hello_world/armcc/elf/test.elf -C
CortexA53.WAVEFORMS_ENABLED=true
```

Available parameters

To list the parameters supported by the model:

- In a Cycle Model reference platform environment, enter `./system_test --list-params` in the `Systems` directory.
- View the file `CortexA53_params.cfg` located in the directory `MODELS/CortexA53_`*`x`*`CPU/gcc`*`version`*`/SystemC`.

Related information

- `CortexA53_params.cfg` located in the directory `MODELS/CortexA53_`*`x`*`CPU/gcc`*`version`*`/SystemC`. This file contains a complete list of the parameters supported by the model, and their supported values.
- *4.5 CADI RemoteConnection parameters* on page 4-41 describes parameters related to configuring CADI debug connections. Debug parameters do not appear in the `CortexA53_params.cfg` file.
- The Technical Reference Manual for your IP includes additional information about supported parameter values.

3.4 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`.

By default, waveform files are sent to the reference platform `Systems` directory with the default filename `arm_cm_`*`CPU`*`.fsdb` or `arm_cm_`*`CPU`*`.vcd`.

Table 3-1 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.

Following is an example of setting waveform values on the command line:

```
./system_test -a ../Applications/hello_world/armcc/elf/test.elf -C
CortexA53.WAVEFORM_TYPE=FSDB -C CortexA53.WAVEFORMS_ENABLED=true
```

Related information

- *3.3 Setting model parameters* on page 3-25

3.5 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 3-2 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 reference platform directory `MODELS/component/`*`gccversion`*`/SystemC`.

This section contains the following subsection:
- *3.5.1 Supported hardware profiling events* on page 3-27.

3.5.1 Supported hardware profiling events

The Cortex-A53 SystemC Cycle Model supports PMU visibility.

Table 3-3 Instruction stream

Event name	Description
0x08_INST_RETIRED	Instructions architecturally executed
0x09_EXC_TAKEN	Exception taken
0x0A_EXC_RETURN	Exception return architecturally executed
0x0B_CID_WRITE	Counts the number of instructions architecturally executed writing into the ContextID Register
0x06_LD_RETIRED	Counts the number of instructions architecturally executed writing into the ContextID Register
0x07_ST_RETIRED	Data write architecturally executed

Table 3-4 Pipeline stream

Event name	Description
0x0C_SW_PC_CHANGE	Software change of the PC
0x0D_BR_IMMED	Immediate branch architecturally executed
0x0E_BR_IMMED_RETIRED	Immediate branch architecturally executed
0x0E_BR_RETURN_RETIRED	Procedure return (other than exception returns) architecturally executed
0xOF_UNALIGNED_LDST_RETIRED	Unaligned load-store

Table 3-4 Pipeline stream (continued)

Event name	Description
0x10_BR_MIS_PRED	Mispredicted or not predicted branch speculatively executed
0x12_BR_PRED	Predictable branch speculatively executed
0x7A_BR_INDIRECT_SPEC	Predictable branch speculatively executed - indirect branch.

Table 3-5 I-Cache stream

Event name	Description
0x01_I_CACHE_REFILL	Level 1 instruction cache refill
0x02_L1I_TLB_REFILL	Level 1 instruction TLB refill
0x14_I_CACHE	Level 1 instruction cache access

Table 3-6 D-Cache stream

Event name	Description
0x03_D_CACHE_REFILL	Level 1 data cache refill
0x04_L1D_CACHE_ACCESS	Level 1 data cache access
0x05_L1D_TLB_REFILL	Level 1 data TLB refill
0x15_L1D_CACHE_WB	Level 1 data cache write-back

Table 3-7 Cycle stream

Event name	Description
0x11_CPU_CYCLES	Cycle
0x86_EXC_IRQ	IRQ exception taken
0x86_EXC_FIQ	FIQ Exception Taken

Table 3-8 Memory stream

Event name	Description
0x13_MEM_ACCESS	Data memory access
0x1A_MEM_ERROR	Local Data memory error
0xC0_EXT_MEM_REQ	External Memory Request
0xC1_NC_EXT_MEM_REQ	Non cacheable external memory request
0xC2_Linefill_Prefetch	Linefill because of prefetch
0xC3_Throttle	Instruction Cache Throttle occurred
0xC4_Entering_Read_Alloc	Entering read allocation mode
0xC5_Read_Alloc_Mode	Read Allocate Mode

Table 3-9 L2-Cache stream

Event name	Description
0x16_L2D_CACHE	Level 2 data cache access
0x17_L2D_CACHE_REFILL	Level 2 data cache refill
0x18_L2D_CACHE_WB	Level 2 data cache write-back

Table 3-10 Bus stream

Event name	Description
0x19_BUS_ACCESS	Bus access
0x1D_BUS_CYCLES	Bus cycles
0x60_BUS_ACCESS_LD	Bus access - Read
0x60_BUS_ACCESS_ST	Bus access - Write

Table 3-11 Microarchitecture stream

Event name	Description
0x1E_CHAIN	Odd performance counter chain mode
0xC6_PREDECODE_ERR	Predecode error
0xC7_DATA_WR_STALL	Data Write operation that stalls the pipeline because of the store buffer being full
0xC8_SCU_SNOOPED	SCU Snooped data from another CPU for this CPU
0xC9_COND_BR	Conditional branch executed
0xCA_INDIRECT_BR_MIS	Indirect branch mis-predicted
0xCB_INDIRECT_BR_MIS_MISC	Indirect branch mis-predicted because of address mis-compare
0xCC_COND_BR_MIS	Conditional branch mis-predicted
0xE0_attr_evnt_e0 — 0xE8_attr_evnt_e8	attr_evnt_e0 — attr_evnt_e8

3.6 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 5-44 for details about the functions supported by SystemC Cycle Models.

Chapter 4

Debugging SystemC Cycle Models with Arm® Development Studio

This section describes how to connect the Arm Development Studio Debugger to Arm SystemC Cycle Models in a Cycle Model reference platform.

It contains the following sections:

4.1 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 Arm Development Studio:

- The Windows version of Arm Development Studio 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 reference platforms 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 4-42 for workarounds.

4.2 Prerequisites to debugging

Arm Development Studio is required before you begin. The instructions in this chapter have been verified using Arm Development Studio Version 2018.0.

Linux version of Development Studio

Note

The Windows version of Arm Development Studio is not supported for SystemC Cycle Models. Only the Linux 64-bit version is supported.

Download and install the Linux 64-bit version of Arm Development Studio from *https://developer.arm.com/tools-and-software/embedded/arm-development-studio/downloads*.

Specify Active Product

Licensed version of Arm Development Studio Gold Edition. Open the Arm License Manager to confirm.

Related information

- See the *Arm® Development Studio Getting Started Guide* (101469) for system requirements, installation instructions, and licensing information.

4.3 Supported debug features

This section describes Arm Development Studio 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.

Arm® Development Studio features

SystemC Cycle Models support the following Arm Development Studio 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 4-42 for more information.
- Debugging of *Symmetric Multi Processing* (SMP) systems.

See the *Arm® Development Studio User Guide* (101470) for more information about debugging multi-core, multi-cluster, and SMP targets.

Support for memory and register views

The SystemC Cycle Model supports visibility into memory spaces and a subset of the registers. See:

- *4.3.1 Supported registers* on page 4-34 for information about supported registers.
- *4.3.2 Supported memory views* on page 4-35 for information about supported memory views.

Note

Registers and memory spaces are exposed on the model. However, their visibility varies depending on the debugger in use.

This section contains the following subsections:

- *4.3.1 Supported registers* on page 4-34.
- *4.3.2 Supported memory views* on page 4-35.

4.3.1 Supported registers

This section describes how to access the register views supported by the Cortex-A53 SystemC Cycle Model.

Note

Registers are exposed on the model. However, their visibility varies depending on the debugger in use. Memory mapped registers are not viewable using Arm Development Studio.

While the processor model is running, it does not present a coherent programmer's view state (a *debuggable point*); instructions in the pipeline may be in different execution states.

Note

Values in the registers are not guaranteed to be accurate unless the model is at a debuggable point.

To view a list of the registers that are viewable using a debugger, enter:

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

For a description of these registers, see the Technical Reference Manual for your IP.

4.3.2 Supported memory views

This section describes the memory views exposed by the Cycle Model.

Note

Memory spaces are exposed on the model. However, their visibility varies depending on the debugger in use.

Table 4-1 Memory views

Name	Description
Physical Memory (secure)	Main memory space view including TCMs, data cache, and downstream ports.
Secure Monitor	CPU virtual memory.
Guest	Uses the N_TTBR0, N_TTBR1, and N_TTBCR registers to translate from VA to PA. This space is active when (SCR.NS == 1) and (CPSR.M != HYP) and (CPSR.M !=MON).

4.4 Enabling Development Studio for use with SystemC Cycle Models

This section describes how to set up Arm Development Studio to debug Cycle Models.

Note

The examples in this section apply to all Arm CPU models that support debugging. The process of enabling Arm Development Studio is the same for all Arm CPU models.

This section contains the following subsections:

- *4.4.1 Connect Development Studio to the model* on page 4-36.
- *4.4.2 Mapping memory spaces* on page 4-40.

4.4.1 Connect Development Studio to the model

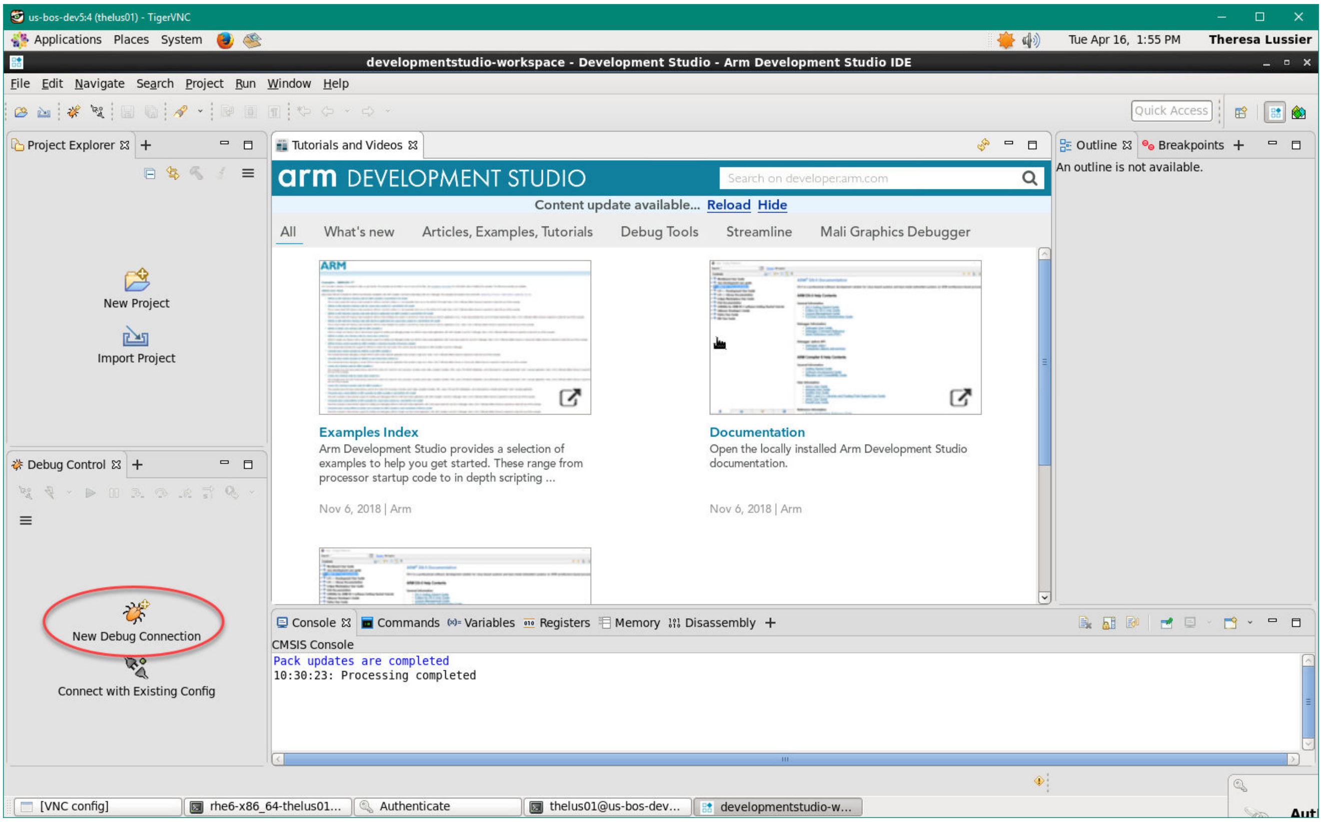
Start the simulation and select the SystemC model for debug.

Procedure

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

```
./system_test -S
```

2. Launch Arm Development Studio.
3. Click **New Debug Connection** to launch the debug connection wizard:



Content update available... Reload Hide

Examples Index
Arm Development Studio provides a selection of examples to help you get started. These range from processor startup code to in depth scripting ...
Nov 6, 2018 | Arm

Documentation
Open the locally installed Arm Development Studio documentation.
Nov 6, 2018 | Arm

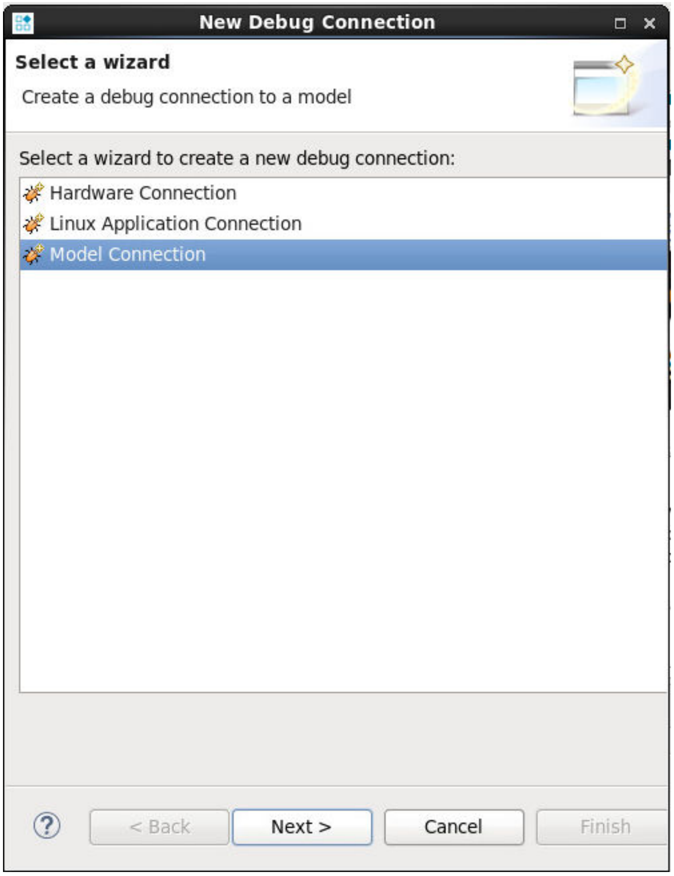
An outline is not available.

CMSIS Console

```
Pack updates are completed
10:30:23: Processing completed
```

Figure 4-1 Click New Debug Connection

4. In the New Debug Connection wizard, select **Model Connection** and click **Next**:



New Debug Connection

Select a wizard

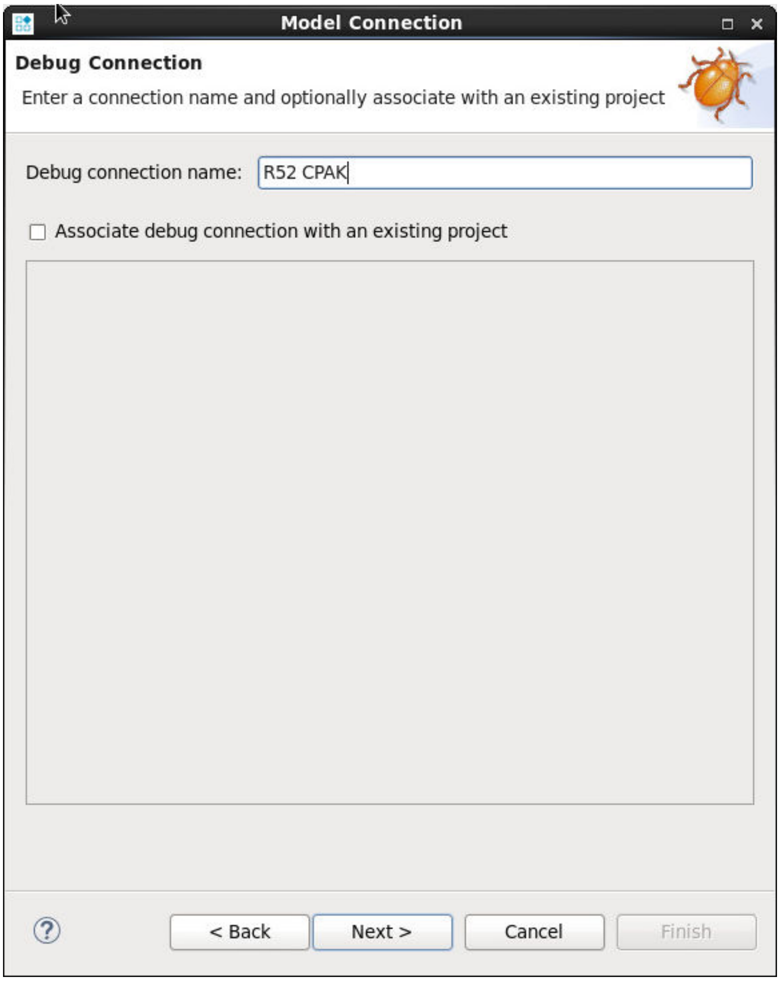
Create a debug connection to a model

Select a wizard to create a new debug connection:

- Hardware Connection
- Linux Application Connection
- Model Connection

Figure 4-2 Select model connection ast the debug connection type

5. In the Debug Connection dialog box, enter a name in the **Debug connection name** field and click **Next**:



Debug connection name: R52 CPAK

☐ Associate debug connection with an existing project

Figure 4-3 Name the debug connection

6. In the Target Selection dialog box, click **Add a new model**:

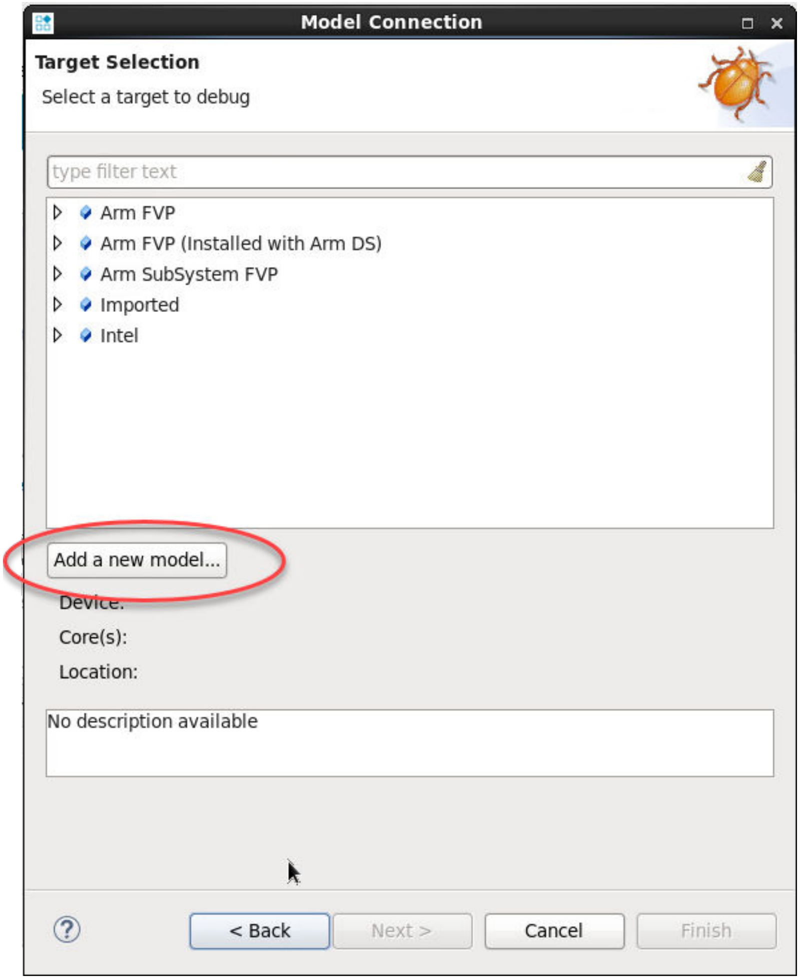


Figure 4-4 Add a new model

7. In the **Select Method for Connecting to Model**: dialog box, select **Browse for model running on local host** and click **Next**.
8. Click **Browse**.
9. In the **Model Running on Local Host** dialog box, click **Browse**. Development Studio searches for SystemC simulation sessions running on the host, and displays them in the **Model Browser** dialog box:

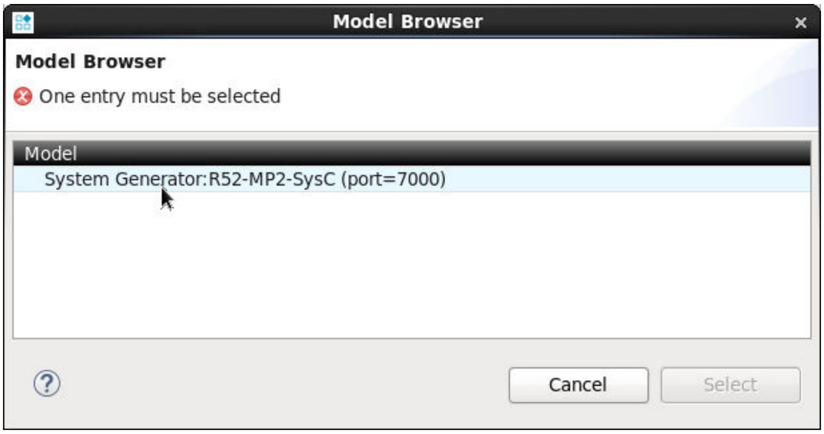
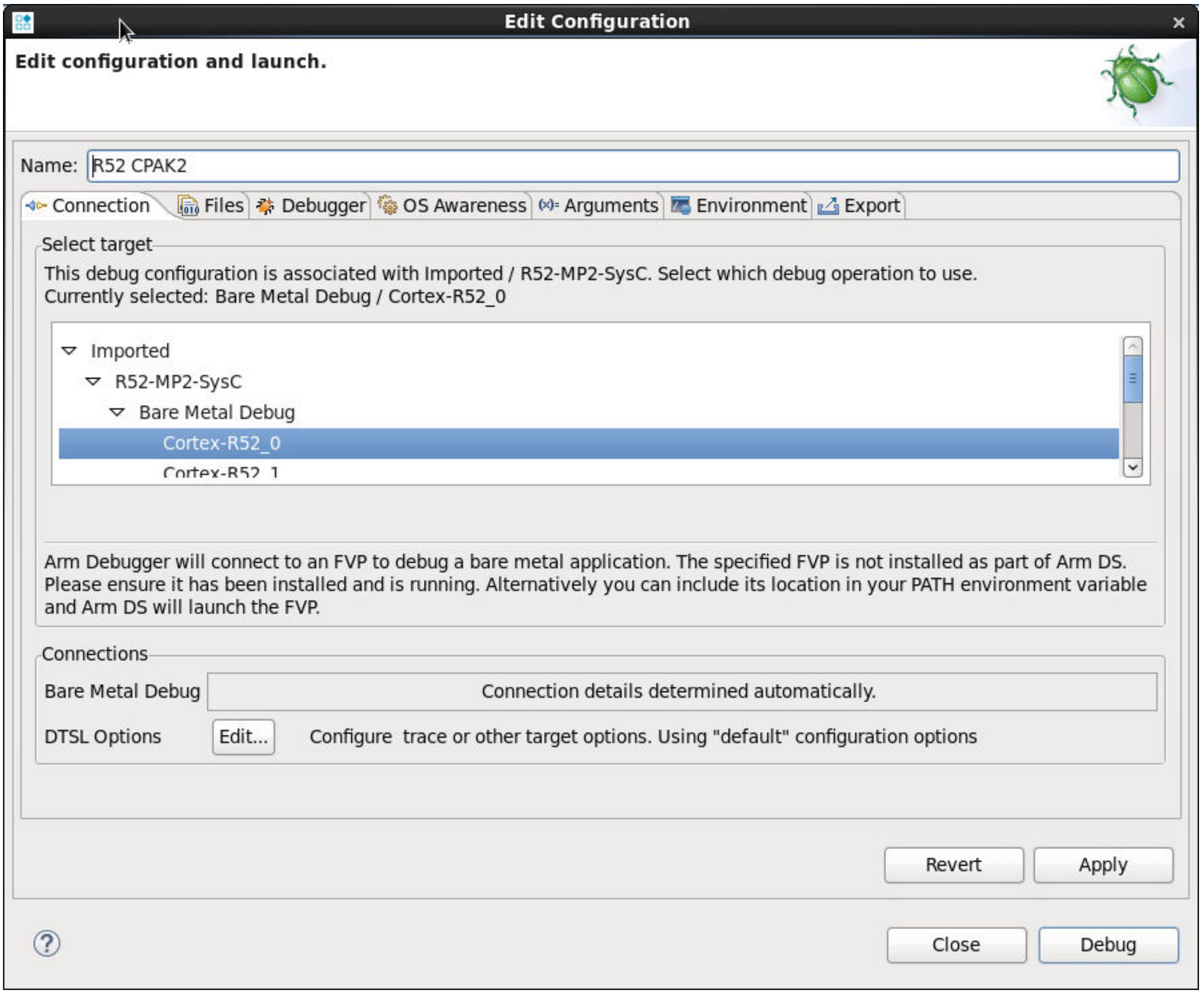


Figure 4-5 Model Browser

10. Select the model for debug and click **Select**.
11. Click **Finish**.

Result

Arm Development Studio connects to the model and displays the cores available for debug:



Edit configuration and launch.

Name: R52 CPAK2

Select target

This debug configuration is associated with Imported / R52-MP2-SysC. Select which debug operation to use.
Currently selected: Bare Metal Debug / Cortex-R52_0

- Imported
  - R52-MP2-SysC
    - Bare Metal Debug
      - Cortex-R52_0
      - Cortex-R52_1

Arm Debugger will connect to an FVP to debug a bare metal application. The specified FVP is not installed as part of Arm DS. Please ensure it has been installed and is running. Alternatively you can include its location in your PATH environment variable and Arm DS will launch the FVP.

Connections

Bare Metal Debug: Connection details determined automatically.

DTSL Options: Configure trace or other target options. Using "default" configuration options

Figure 4-6 Core available for debug

Related information

- *Arm® Development Studio Getting Started Guide* (101469)
- *Arm® Development Studio User Guide* (101470)

4.4.2 Mapping memory spaces

The Cortex-A53 MPCore processor supports Secure, Non-Secure, and Hypervisor memory spaces. Map these spaces for the debugger as described in this section.

1. Access the **Model Devices and Cluster Configuration** tab of the Arm Development Studio GUI.
2. Click **Edit selected row**. The **Edit Instance** dialog appears:
3. Set **Cluster** to `0` and manually set or verify the following parameter values:

Table 4-2 Memory space settings

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

4.5 CADI RemoteConnection parameters

This section describes the parameters for CADI connections.

Each parameter is prefixed with `REMOTE_CONNECTION.CADIServer`; for example:

```
REMOTE_CONNECTION.CADIServer.range
```

Note

The default value restricts connections to be from the localhost only. To enable remote connections, specify an IP address to listen to, or specify `0.0.0.0` to listen to all adapters.

Table 4-3 CADIIPCRemoteConnection parameters

Name	Type	Default value	Allowed values	Runtime	Description
`enable_remote_cadi`	`bool`	`false`	`true`, `false`	`false`	Allow connections from remote hosts.
`listen_address`	`string`	`"127.0.0.1"`	`""`	`false`	If `enable_remote_cadi` is set, this parameter is the network address the server listens on.
`port`	`int`	`0x7b8b`	`0x1` - `0xffff`	`false`	If `enable_remote_cadi` is set, this parameter is the TCP port the server listens on.
`range`	`int`	`0x0`	`0x0` - `0x64`	`false`	If the requested port is not available, search for the next available port in the range [port:port+range]. Only try the specified port.

See *5.8 scx::scx_parse_and_configure* on page 5-52 for information about CADI command-line options used with `scx::scx_parse_and_configure()`.

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® Development Studio User Guide* (101470).

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 on the console from which the simulation was run:

```
Warning: stop at a debug point failed: Simulation suspended before these target(s)
could reach debug point:ca53_core.cpu1;ca53_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 *4.1 Restrictions and limitations* on page 4-32). In these cases, lengthening the loop by adding `nops` 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
  ```

- Change the timeout setting (see *4.7 Changing the timeout setting* on page 4-43)

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 Cycle Model reference platforms with TLM models. For reference platforms with pin-level models, specifying `--application` has no effect and results in multiple warnings. The application for reference platforms with pin-level models is determined by the contents of the hex files in the reference platforms `Systems` directory. See the reference platforms `README` 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 CortexA53_core0=test0.elf -a CortexA53_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 reference system `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 ca53_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
.
.
.
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 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.