SystemC Cycle Models

Version 11.0

User Guide

arm

101124_1100_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
1000-01	30 September 2018	Non-Confidential	Release 10.0.1
1100-00	31 May 2019	Non-Confidential	Release 11.0

Non-Confidential Proprietary Notice

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 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 with Arm Cycle Models in SystemC CPAKs.

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 *SystemC Cycle Models User Guide*.
- The number 101124_1100_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 prerequisites in this section for using Arm SystemC Cycle Models.

See the *Cycle Model SystemC Runtime Installation Guide* (101146) for information about supported Runtime and GCC versions.

- Cycle Model SystemC Runtime. The Cycle Model SystemC Runtime installer includes 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. See the *Arm® SystemC Cycle Models CPAK Getting Started Guide* (101497).

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.

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).

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 files described in this section.

Note

All features may not be available on all models.

In a CPAK, these files are located in the root directory *CPAK*/MODELS/component/*gccversion*/SystemC.

For models downloaded from Arm IP Exchange (*http://armipexchange.com*), these files are located in the root directory *gccversion*/SystemC.

CM_busdefs.tar
Cycle Model IPXACT bus definition bundle.

CM_IPXACT_*component*.xml
Cycle Model IPXACT description.

cm_sysc_utils.h
SystemC utilities header file.

***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.

Makefile
Compiles the Cycle Model into the shared libraries included with the installation.

***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. Present only in TLM-based models.

TCM-related files

Models that support TCMs may have additional header files related to TCM loading and waveform dumping, if supported.

Tarmac Trace-related files

If the Cycle Model supports Tarmac Trace, the following files may be present:

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

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
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

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 CPAK package.

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.
- Backward compatibility is limited to Version 11.0 (and later) models. These models contain the data files required by the Cycle Models Configuration Tool.
- The Cycle Models Configuration Tool uses the directory it was run from as the default searchpath; use the `--searchpath` option to specify a different location to search.

This section contains the following subsections:

- *2.1.1 Cycle Models Configuration Tool command syntax* on page 2-15.
- *2.1.2 Cycle Models Configuration Tool examples* on page 2-19.

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 CortexR52
```

--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 CortexR52 --link
```

This example returns flag and library data only:

```
$ cm_config --use-tool gcc:6.4.0 --searchpath MODELS --model CortexR52 --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 CortexR52 --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 CortexR52 --link
```

--sources

Optional.
Outputs a list of source files.

Example:

```
$ cm_config --use-tool gcc:6.4.0 --searchpath MODELS --model CortexR52 --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 CortexR52 --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 CortexR52 --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 CortexR52 --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 2-19

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-R52 model in a CPAK environment:

```
$ cm_config --use-tool gcc:6.4.0 --searchpath MODELS --model CortexR52 --sources --link --
model-type tlm --ignore cms cm_sysc SystemC

CPAK_PATH/MODELS/CortexR52_2CPU/gcc640/SystemC/univent_tarmac.cpp
CPAK_PATH/MODELS/CortexR52_2CPU/gcc640/SystemC/libCortexR52.systemc.cpp
CPAK_PATH/MODELS/CortexR52_2CPU/gcc640/SystemC/CortexR52ResetImp.cpp
CPAK_PATH/MODELS/CortexR52_2CPU/gcc640/SystemC/libCortexR52.tlm.cpp
-L CPAK_PATH/MODELS/CortexR52_2CPU/gcc640/SystemC
-L CPAK_PATH/MODELS/CortexR52_2CPU/gcc640/SystemC/lib
CPAK_PATH/MODELS/CortexR52_2CPU/gcc640/SystemC/univentUtil/lib/kite_tarmac_dpi.so -
lCortexR52.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 --searchpath MODELS --model CortexR52 --sources --link --
model-type pin --ignore cms cm_sysc SystemC

CPAK_PATH/MODELS/CortexR52_2CPU/gcc640/SystemC/univent_tarmac.cpp
CPAK_PATH/MODELS/CortexR52_2CPU/gcc640/SystemC/libCortexR52.systemc.cpp
```

```
-L CPAK_PATH/MODELS/CortexR52_2CPU/gcc640/SystemC
-L CPAK_PATH/MODELS/CortexR52_2CPU/gcc640/SystemC/lib
CPAK_PATH/MODELS/CortexR52_2CPU/gcc640/SystemC/univentUtil/lib/kite_tarmac_dpi.so
-lCortexR52.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 einvironment variables for component roots. In the following example, CARBON_HOME is used as the Cycle Model Studio root:

```
$ cm_config --use-tool gcc:6.4.0 --searchpath ./ --model cms --sources --link --compile --
use-env cms:CARBON_HOME
-I${CARBON_HOME}/include -L${CARBON_HOME}/Linux64/lib/gcc/shared -lcarbon5 -lpthread -ldl
```

In the following example, the CORTEXR52_HOME, CARBON_HOME, CM_SYSC_HOME, and SYSTEMC_HOME environment variables are used as roots of their respective components:

```
$ cm_config --use-tool gcc:6.4.0 --searchpath ./ --model CortexR52 --link --use-env
cms:CARBON_HOME CortexR52:CORTEXR52_HOME SystemC:SYSTEMC_HOME cm_sysc:CM_SYSC_HOME
-L${CORTEXR52_HOME}/gcc640/SystemC -L${CORTEXR52_HOME}/gcc640/SystemC/lib ${CORTEXR52_HOME}/
gcc640/SystemC/univentUtil/lib/kite_tarmac_dpi.so -lCortexR52.icm -licm_runtime -L$
{CARBON_HOME}/Linux64/lib/gcc/shared -lcarbon5 -lpthread -ldl -L${SYSTEMC_HOME}/lib/
Linux64_GCC-6.4 -lsystemc -L${CM_SYSC_HOME}/lib/Linux64_GCC-6.4 -larmtlm ${CM_SYSC_HOME}/
FMRuntime/FastModelsPortfolio/lib/Linux64_GCC-6.4/libfmruntime.a ${CM_SYSC_HOME}/FMRuntime/
FastModelsPortfolio/lib/Linux64_GCC-6.4/libIrisSupport.a
```

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 --searchpath ./ --model CortexR52 --link libs --use-arm

CPAK_PATH/MODELS/CortexR52_2CPU/gcc640/SystemC/univentUtil/lib/kite_tarmac_dpi.so -
lCortexR52.icm -licm_runtime -larmtlm
CPAK_PATH/ARM/CycleModels/Runtime/cm_sysc/mainline/FMRuntime/FastModelsPortfolio/lib/
Linux64_GCC-6.4/libfmruntime.a
CPAK_PATH/ARM/CycleModels/Runtime/cm_sysc/mainline/FMRuntime/FastModelsPortfolio/lib/
Linux64_GCC-6.4/libIrisSupport.a -lcarbon5
```

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 CPAK environment, they are also present in the CPAK `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 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; contact Arm Support for more information.

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

This section contains the following subsection:

- *3.1.1 Available pins* on page 3-23.

3.1.1 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.

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 `model ResetModule.h`
- For TLM models, the file is `model ResetImp.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 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 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 *4.6 CADI RemoteConnection parameters* on page 4-41 for additional parameters related to configuring CADI debug connections. These options do not appear in the *`component`*`_params.cfg` file.

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 CPAK `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 model .WAVEFORM_TYPE=FSDB
-C model .WAVEFORMS_ENABLED=true
```

Related information

- *Setting model parameters*

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 CPAK directory MODELS/component/*gccversion*/SystemC.

3.6 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 3-3 TARMAC trace parameters

Parameter	Description	Available settings	Default setting
`TARMAC_LOGFILE_NAME`	Sets TARMAC log file name. This parameter should not be set in a multi-cluster environment; use the alternate instructions below.	*`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`

Enabling TARMAC trace in multicore environments

In multicore environments, use the @CPUID@ designation to name the TARMAC files. For example, for a design with two cores and one cluster:

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

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

Enabling TARMAC trace in multicluster environments

For multiple clusters, use the default TARMAC log file name (`model.`*`aff2.aff1.cpuid`*`.log`) rather than setting a different name with the TARMAC_LOGFILE_NAME parameter. Set the affinity values using the model parameters CLUSTERIDAFF1 and CLUSTERIDAFF2.

3.7 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 with Arm Cycle Models in SystemC CPAKs.

This section applies to CADI-enabled models only.

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 CPAKs do not currently show a coherent debug view of memory shared across clusters.
- Reset of system and CPU are 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 *4.7 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 Models that support Arm® Development Studio connectivity

Arm Development Studio connects only to CPU models that have debugger support.

The following SystemC Cycle Models support connection to Arm Development Studio:

- 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.
- Cortex-R5 (single core and multi-core). Debugger features on the Cortex-R5 model are BETA quality. This includes register view, memory view, and breakpoint/single step support.

4.4 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 *4.7 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 exposes memory spaces and a subset of the registers. However, their visibility varies depending on the debugger in use.

4.5 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 may use CPUs other than the . The process of enabling Arm Development Studio is the same for all Arm CPU models.

This section contains the following subsections:

4.5.1 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:

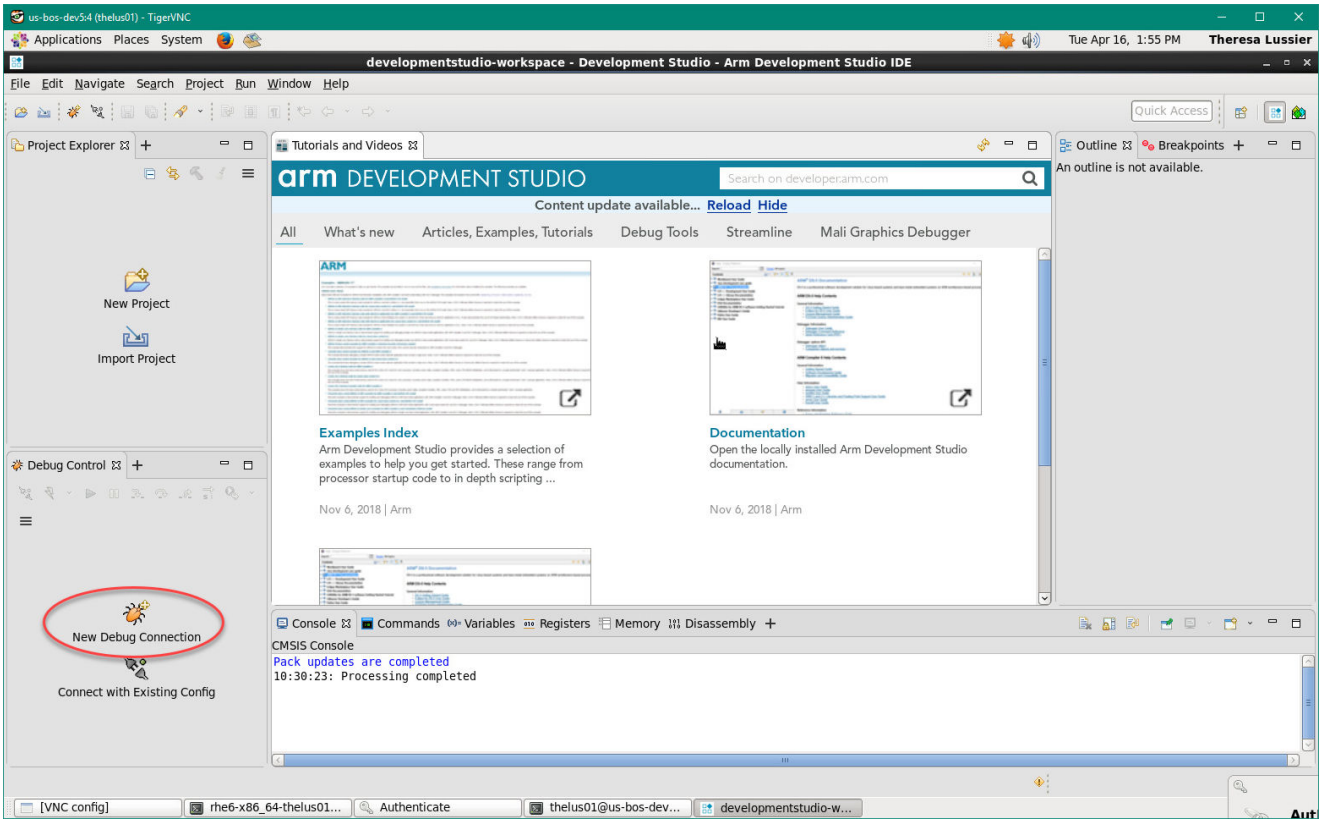

Figure 4-1 Click New Debug Connection

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

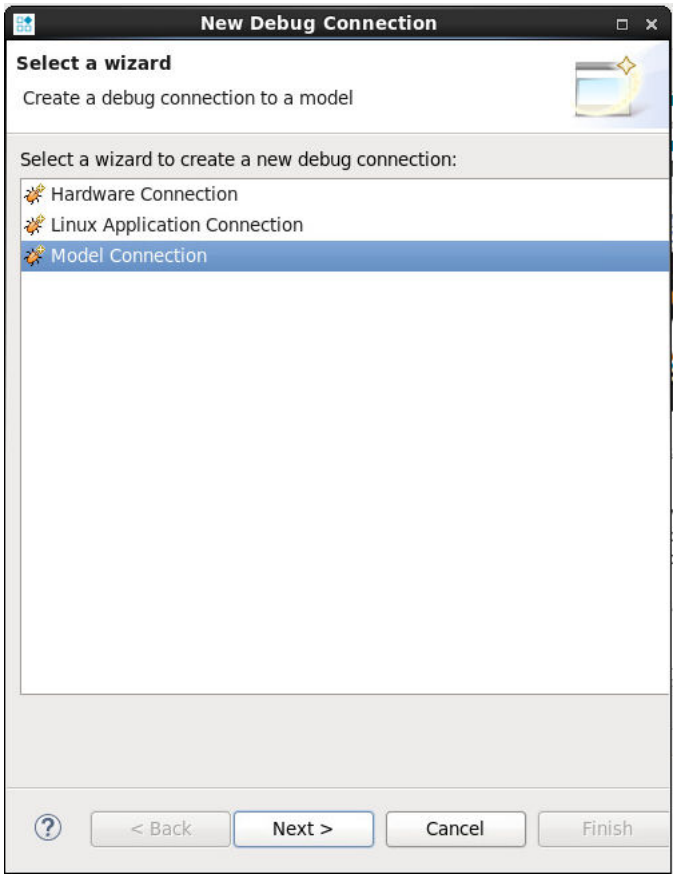

Figure 4-2 Select model connection ast the debug connection type

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

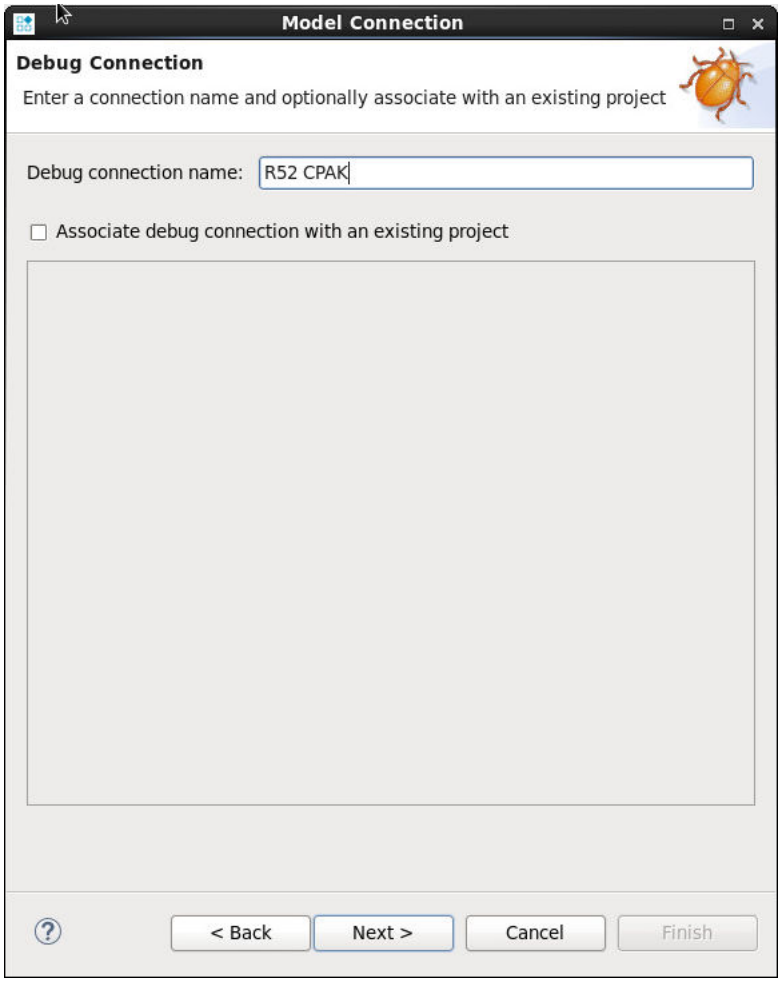

Figure 4-3 Name the debug connection

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

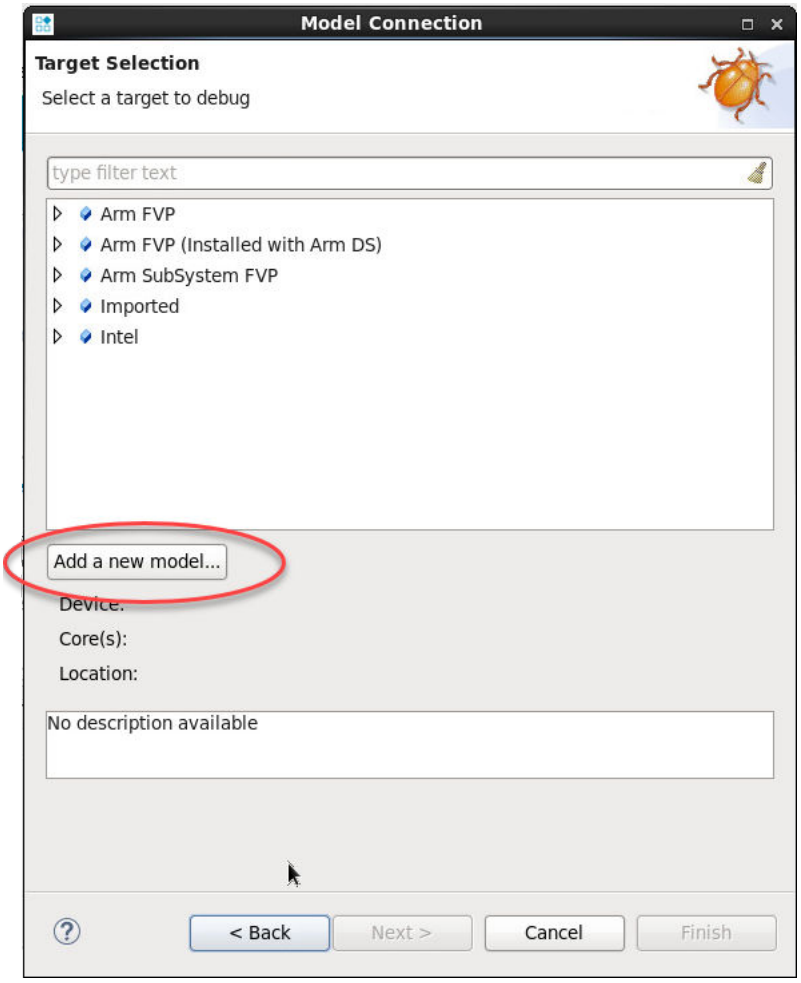

Figure 4-4 Add a new model

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:

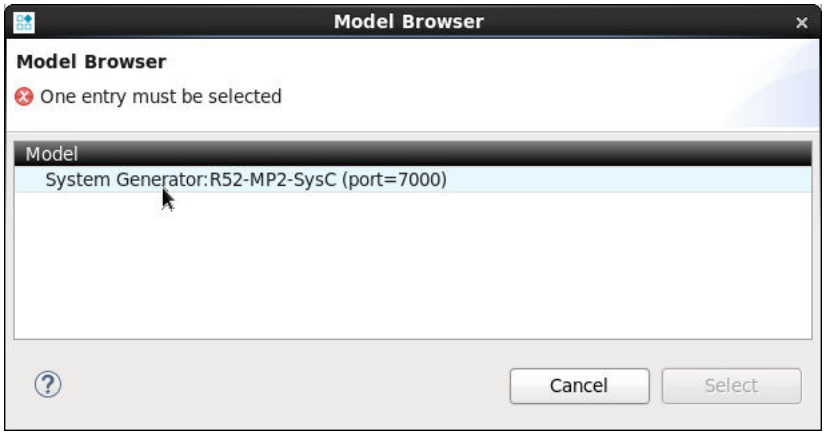

Figure 4-5 Model Browser

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:

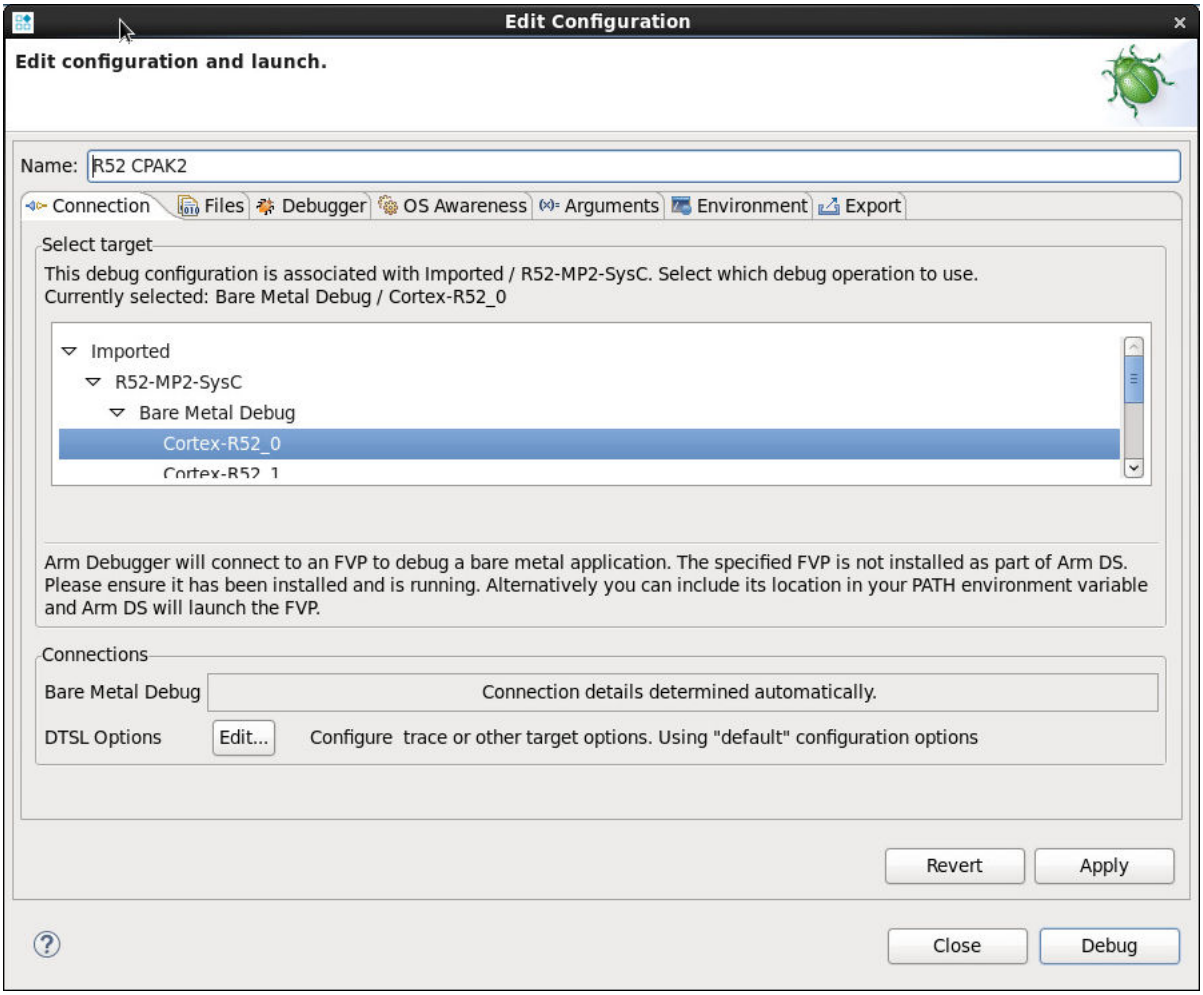

Figure 4-6 Core available for debug

Related information

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

4.5.2 Mapping memory spaces

For cores that support Secure, Non-Secure, or Hypervisor spaces, map these spaces for the debugger as described in this section.

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 map memory spaces for the debugger:

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

Table 4-1 Memory space settings

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

4.6 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 4-2 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()`.

4.7 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:model_core.cpu1;model_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 4-31). 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
```

- Change the timeout setting (see *4.8 Changing the timeout setting* on page 4-43)

4.8 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 model _core0=test0.elf -a model _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 model_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 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.