Cortex-M7 SystemC Cycle Model

Version 11.0

User Guide
Cortex-M7 SystemC Cycle Model
User Guide

Copyright © 2018, 2019 Arm Limited or its affiliates. All rights reserved.

Release Information

<table>
<thead>
<tr>
<th>Issue</th>
<th>Date</th>
<th>Confidentiality</th>
<th>Change</th>
</tr>
</thead>
<tbody>
<tr>
<td>1001-00</td>
<td>26 October 2018</td>
<td>Non-Confidential</td>
<td>First release</td>
</tr>
<tr>
<td>1100-00</td>
<td>31 May 2019</td>
<td>Non-Confidential</td>
<td>Release 11.0</td>
</tr>
</tbody>
</table>

Non-Confidential Proprietary Notice

This document is protected by copyright and other related rights and the practice or implementation of the information contained in this document may be protected by one or more patents or pending patent applications. No part of this document may be reproduced in any form by any means without the express prior written permission of Arm. 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 http://www.arm.com/company/policies/trademarks.

Copyright © 2018, 2019 Arm Limited (or its affiliates). All rights reserved.


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

Cortex-M7 SystemC Cycle Model User Guide

Preface

About this book ................................................................. ................................. 7

Chapter 1

Introduction

1.1 Functionality of the SystemC Cycle Model ....................................................... 1-10
1.2 Prerequisites to using SystemC Cycle Models .................................................. 1-11
1.3 Supported platforms, compilers, and simulators ............................................. 1-12
1.4 Package contents ............................................................................................. 1-13

Chapter 2

Integrating models into your environment

2.1 Extracting build options using the Cycle Models Configuration Tool ............... 2-16
2.2 Adding custom options to the Makefile .......................................................... 2-22

Chapter 3

Using SystemC Cycle Models

3.1 Connecting model ports .................................................................................. 3-24
3.2 Resetting the SystemC Cycle Model ............................................................... 3-26
3.3 Setting model parameters ............................................................................... 3-27
3.4 Dumping waveforms ...................................................................................... 3-28
3.5 Configuring cache sizes .................................................................................. 3-29
3.6 Configuring PMU events ................................................................................ 3-30
3.7 Configuring TARMAC trace .......................................................................... 3-31
3.8 Working with the SCX framework .................................................................. 3-32
Chapter 4  
Debugging SystemC Cycle Models with Arm® Development Studio

4.1 Restrictions and limitations ................................................................. 4-34
4.2 Prerequisites to debugging ............................................................... 4-35
4.3 Supported debug features ................................................................. 4-36
4.4 Enabling Development Studio for use with SystemC Cycle Models .......... 4-38
4.5 CADI RemoteConnection parameters ............................................. 4-43
4.6 Changing the timeout setting ........................................................... 4-44

Chapter 5  
SystemC Export API function reference

5.1 scx::scx_initialize ........................................................................... 5-46
5.2 scx::scx_load_application ............................................................. 5-47
5.3 scx::scx_set_parameter ................................................................. 5-48
5.4 scx::scx_get_parameter ................................................................. 5-49
5.5 scx::scx_get_parameter_list ........................................................... 5-50
5.6 scx::scx_cpulimit ........................................................................... 5-51
5.7 scx::scx_timelimit ........................................................................... 5-52
5.8 scx::scx_parse_and_configure ....................................................... 5-53
5.9 scx::scx_print_statistics ................................................................. 5-57
This preface introduces the Cortex-M7 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®-M7 SystemC Cycle Model into a SystemC design and simulation environment.

Using this book

This book is organized into the following chapters:

Chapter 1 Introduction
This section introduces the Arm Cortex®-M7 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 CPAK system.

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

\[ \text{MRC p15, 0, <Rd>, <CRn>, <CRm>, <Opcode_2>} \]

**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 number 101488_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 the Arm Cortex®-M7 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 Performance Analysis Kit (CPAK) 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.
• 1.4 Package contents on page 1-13.
1.1 **Functionality of the SystemC Cycle Model**

The Arm Cortex-M7 SystemC Cycle Model simulates the Cortex-M7 MPCore processor.

**Supported functionality**

This section summarizes the functionality of the Cycle Model compared to that of the hardware and describes the performance and accuracy of the Cycle Model. The Cycle Model supports:

- Cortex-M7 Integer Core.
- NVIC – Nested Vectored Interrupt Controller.
- WIC – Wakeup Interrupt Controller Interface Support (support is for the interface only).
- AXI4 Master Memory Interface.
- APB v3.0 interface for accessing the external Private Peripheral Bus.
- FPB – Flash Patch and Debug.
- DWT – Debug Watchpoint and Trace.
- MPU – Memory Protection Unit (optional).
- Floating Point Unit (optional) — Single and/or double precision.
- Level 1 memory system includes a three stage prefetch unit (PU) including a static branch predictor and call-return stack. The LSU is also three-stage and includes a store queue for efficient access to the Tightly Coupled Memory (TCM) ports.
- ETM supported to enable software profiling.
- DCache and ICache sizes configurable at runtime using model parameters.

**Unsupported hardware features**

The following features of the Cortex-M7 hardware are not implemented in this release of the Cortex-M7 Cycle Model:

- SW/JTAG-DP
- ITM
- Semihosting

**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-M7 hardware:

- Waveform dumping, including dumping of TCM memories. See [3.4 Dumping waveforms](#) on page 3-28.
- Support for viewing register values. See [4.3.1 Supported registers](#) on page 4-36 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 4-36 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.

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.
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 on Red Hat Enterprise Linux version 6.6 (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.
1.4 Package contents

Each SystemC Cycle Model contains the files described in this section.

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.

- **componentResetModule.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.

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

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

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

- **libcomponent_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 on page 2-16.
- 2.2 Adding custom options to the Makefile on page 2-22.
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 2-16.
• 2.1.2 Cycle Models Configuration Tool examples on page 2-20.

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
output 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 --sources

--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 2-20
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)
OBJJS := $(patsubst %.cpp,%.o,$(SRCS))

system: system.o $(OBJJS)
$(CXX) -o $@ $(shell $(CM_CONFIG) --link)
```

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/libCortexR52ResetImp.cpp
CPAK_PATH/MODELS/CortexR52_2CPU/gcc640/SystemC/libCortexR52.tlm.cpp
-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
```
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, CARBON_HOME is used as the Cycle Model Studio root:

```bash
$ 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:

```bash
$ 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.

```bash
$ cm_config --use-tool gcc:6.4.0 --searchpath ./ --model CortexR52 --link libs --use-arm
```

```bash
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:

```bash
$ make all CM_CONFIG_ARGS='--ignore SystemC' CXXFLAGS='-$SYSTEMC_INC' LDFLAGS='-$SYSTEMC_LIB -lsystemc' RPATHS='-$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:

```bash
$ 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:

```bash
$ 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 on page 3-24.
• 3.2 Resetting the SystemC Cycle Model on page 3-26.
• 3.3 Setting model parameters on page 3-27.
• 3.4 Dumping waveforms on page 3-28.
• 3.5 Configuring cache sizes on page 3-29.
• 3.6 Configuring PMU events on page 3-30.
• 3.7 Configuring TARMAC trace on page 3-31.
• 3.8 Working with the SCX framework on page 3-32.
3.1 Connecting model ports

All pins must be bound to a signal.

For a list of the pins on the Cortex-M7 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 3-24.

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 3-24.
- 3.1.2 Port binding on page 3-25.

3.1.1 Tied pins

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

For a complete list of the pins on the Cortex-M7 SystemC Cycle Model, refer to the model header file `lib/model/systemc.h`, or the `CM_IPXACT_model.xml` file. This list includes all pins, including those listed below, which are tied and cannot be modified:

- AFVALID (low)
- AHBSPRI (low)
- ATDATA (low)
- ATID (low)
- ATREADY (low)
- CTLPPBLOCK (low)
- D0TCMMBISTOUT (low)
- D1TCMMBISTOUT (low)
- DBGEN (high)
- DBGRESTART (low)
- DCCMINP (low)
- DCCMINP2 (low)
- DFTRAMHOLD (low)
- DFTRSTDDISABLE (low)
- DFTSE (low)
- ECOREVNUM (low)
- EDBGRQ (low)
- EXRESPP (low)
- IADBGPROT (low)
- ITCMMBISTOUT (low)
- MBISTADDR (low)
- MBISTARRAY (low)
- MBISTBE (low)
- MBISTCFG (low)
- MBISTINDATA (low)
- MBISTREADEN (low)
- MBISTREQ (low)
- MBISTWRITEEN (low)
- NIDEN (low)
- nMBISTRESET (high)
- SLEEPHOLDREQn (high)
- TSCLKCHANGE (low)
- TSVVALUEB (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 `CortexM7ResetImp.cpp` file located in the directory `gccversion/SystemC/` for the model.

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

```c++
// bind all the non-TLM ports to their corresponding signals
void CortexM7Imp::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 CPAK:

1. In the `CortexM7ResetImp.cpp` file, inside the `bind_nontlm_ports_to_signals()` function, remove the binding by commenting out the line for the port.
2. In the CPAK `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 CPAK 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 gccversion/SystemC/.

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

- For pin-level models, the file is CortexM7ResetModule.h
- For TLM models, the file is CortexM7ResetImp.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-M7 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 CortexM7.WAVEFORMS_ENABLED=true
```

Available parameters

To list the parameters supported by the model:
- In a CPAK environment, enter ./system_test --list-params in the Systems directory.
- View the file CortexM7_params.cfg located in the directory MODELS/CortexM7_xCPU/gccversion/SystemC.

Related information
- CortexM7_params.cfg located in the directory MODELS/CortexM7_xCPU/gccversion/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 4-43 describes parameters related to configuring CADI debug connections. Debug parameters do not appear in the CortexM7_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 CPAK Systems directory with the default filename `arm_cm_CPU.fsdb` or `arm_cm_CPU.vcd`.

### Table 3-1  Waveform parameters

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Available settings</th>
<th>Default setting</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>WAVEFORM_TIMEUNIT</code></td>
<td>Units defined by <code>sc_time_unit()</code>: <code>SC_FS</code>, <code>SC_PS</code>, <code>SC_NS</code>, <code>SC_US</code>, <code>SC_MS</code>, <code>SC_SEC</code></td>
<td><code>SC_PS</code></td>
</tr>
<tr>
<td><code>WAVEFORM_TYPE</code></td>
<td><code>FSDB</code>, <code>VCD</code></td>
<td><code>VCD</code></td>
</tr>
<tr>
<td><code>WAVEFORMS_ENABLED</code></td>
<td><code>true</code>, <code>false</code></td>
<td><code>false</code></td>
</tr>
</tbody>
</table>

For example:

```cpp
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 CortexM7.WAVEFORM_TYPE=FSDB -C CortexM7.WAVEFORMS_ENABLED=true
```

**Related information**

- 3.3 Setting model parameters on page 3-27
3.5 Configuring cache sizes

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

Table 3-2 ICache and DCache parameters

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Available settings</th>
<th>Default setting</th>
</tr>
</thead>
<tbody>
<tr>
<td>DCACHE_SIZE</td>
<td>0 - 4KB</td>
<td>0x3</td>
</tr>
<tr>
<td></td>
<td>1 - 8KB</td>
<td></td>
</tr>
<tr>
<td></td>
<td>3 - 16KB</td>
<td></td>
</tr>
<tr>
<td></td>
<td>7 - 32KB</td>
<td></td>
</tr>
<tr>
<td></td>
<td>15 - 64KB</td>
<td></td>
</tr>
<tr>
<td>ICACHE_SIZE</td>
<td>0 - 4KB</td>
<td>0x3</td>
</tr>
<tr>
<td></td>
<td>1 - 8KB</td>
<td></td>
</tr>
<tr>
<td></td>
<td>3 - 16KB</td>
<td></td>
</tr>
<tr>
<td></td>
<td>7 - 32KB</td>
<td></td>
</tr>
<tr>
<td></td>
<td>15 - 64KB</td>
<td></td>
</tr>
</tbody>
</table>

For example:

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

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

Following is an example of setting cache size on the command line:

```
./system_test -a ../Applications/hello_world/armcc/elf/test.elf -C CortexM7.ICACHE_SIZE=7
```

Related information

- 3.3 Setting model parameters on page 3-27
3.6 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 3-3  PMU parameters

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Available settings</th>
<th>Default setting</th>
</tr>
</thead>
<tbody>
<tr>
<td>PMU_ENABLED</td>
<td>true,false</td>
<td>false</td>
</tr>
</tbody>
</table>

For example:

```cpp
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.

This section contains the following subsection:

• **3.6.1 Supported hardware profiling events on page 3-30.**

3.6.1 Supported hardware profiling events

The Cortex-M7 SystemC Cycle Model supports PMU visibility.

Table 3-4  Core Events stream

<table>
<thead>
<tr>
<th>Event name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>CPI</td>
<td>Instruction Cycle Count</td>
</tr>
<tr>
<td>Exception</td>
<td>Exception Overhead Counter</td>
</tr>
<tr>
<td>Sleep</td>
<td>Sleep Overhead Counter</td>
</tr>
<tr>
<td>LSU</td>
<td>Load-Store Counter</td>
</tr>
<tr>
<td>IT Fold</td>
<td>Folded Instruction Counter</td>
</tr>
</tbody>
</table>
### 3.7 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 3-5  TARMAC trace parameters

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
<th>Available settings</th>
<th>Default setting</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>TARMAC_LOGFILE_NAME</code></td>
<td>Sets TARMAC log file name. This parameter should not be set in a multi-cluster environment; use the alternate instructions below.</td>
<td>string</td>
<td>&quot;&quot;</td>
</tr>
<tr>
<td><code>TARMAC_ENABLED</code></td>
<td>Enables or disables TARMAC logging.</td>
<td>true, false</td>
<td>false</td>
</tr>
<tr>
<td><code>TARMAC_FLUSH</code></td>
<td>Flushes the TARMAC log file data after the specified number of instructions.</td>
<td>integer</td>
<td>0</td>
</tr>
</tbody>
</table>

**Enabling TARMAC trace in multicore environments**

In multicore environments, use the `@CPUID@` designation to name the TARMAC files. For example, for a Cortex-M7 design with two cores and one cluster:

```plaintext
scx::scx_set_parameter("m7.TARMAC_LOGFILE_NAME","tarmac.m7.@CPUID@.log");
scx::scx_set_parameter("m7.TARMAC_LOGFILE_ENABLED",true);
```

This creates the files `tarmac.m7.0.log` and `tarmac.m7.1.log`.

**Enabling TARMAC trace in multicluster environments**

For multiple clusters, use the default TARMAC log file name (`m7.aff2.aff1.cpuid.1.log`) rather than setting a different name with the `TARMAC_LOGFILE_NAME` parameter. Set the affinity values using the model parameters `CLUSTERIDAFF1` and `CLUSTERIDAFF2`. 
3.8 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:

```cpp
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 5-45 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 CPAK system.

It contains the following sections:

• 4.1 Restrictions and limitations on page 4-34.
• 4.2 Prerequisites to debugging on page 4-35.
• 4.3 Supported debug features on page 4-36.
• 4.4 Enabling Development Studio for use with SystemC Cycle Models on page 4-38.
• 4.5 CADI RemoteConnection parameters on page 4-43.
• 4.6 Changing the timeout setting on page 4-44.
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 the SystemC Cycle Model with Arm Development Studio:

- Debugger features on the Cortex-M7 SystemC Cycle Model are BETA quality. This includes register view, memory view, and breakpoint/single step support.
- The Windows version of Arm Development Studio is not supported. Only the Linux 64-bit version is supported.
- 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.
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.

---


**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 debug connection to Arm Development Studio. See the Arm® Development Studio User Guide (101470) for more information.

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 4-36 for information about supported registers.
• 4.3.2 Supported memory views on page 4-36 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 4-36.
• 4.3.2 Supported memory views on page 4-36.

4.3.1 Supported registers

This section describes how to access the register views supported by the Cortex-M7 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.

——— Note ————

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

<table>
<thead>
<tr>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>memory</td>
<td>Unified memory view including data cache, and downstream master AXI ports.</td>
</tr>
<tr>
<td>axi_m</td>
<td>Downstream master AXI port memory view.</td>
</tr>
</tbody>
</table>
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 may use CPUs other than the Cortex-M7. The process of enabling Arm Development Studio is the same for all Arm CPU models.

This section contains the following subsection:
• 4.4.1 Connect Development Studio to the model on page 4-38.

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:

   ![New Debug Connection](image)

   Figure 4-1 Click New Debug Connection

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

---

**Figure 4-2** Select model connection as the debug connection type
6. In the Target Selection dialog box, click **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:

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:

Related information

- Arm® Development Studio Getting Started Guide (101469)
- Arm® Development Studio User Guide (101470)
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>
<thead>
<tr>
<th>Name</th>
<th>Type</th>
<th>Default value</th>
<th>Allowed values</th>
<th>Runtime</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>enable_remote_cadi</td>
<td>bool</td>
<td>false</td>
<td>true, false</td>
<td>false</td>
<td>Allow connections from remote hosts.</td>
</tr>
<tr>
<td>listen_address</td>
<td>string</td>
<td>&quot;127.0.0.1&quot;</td>
<td>&quot;&quot;</td>
<td>false</td>
<td>If <code>enable_remote_cadi</code> is set, this parameter is the network address the server listens on.</td>
</tr>
<tr>
<td>port</td>
<td>int</td>
<td>0x7b8b</td>
<td>0x1-0xffff</td>
<td>false</td>
<td>If <code>enable_remote_cadi</code> is set, this parameter is the TCP port the server listens on.</td>
</tr>
<tr>
<td>range</td>
<td>int</td>
<td>0x0</td>
<td>0x0 - 0x64</td>
<td>false</td>
<td>If the requested port is not available, search for the next available port in the range [port:port+range]. Only try the specified port.</td>
</tr>
</tbody>
</table>

See [5.8 scx::scx_parse_and_configure on page 5-53](#) for information about CADI command-line options used with `scx::scx_parse_and_configure()`.
4.6 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:

```bash
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` on page 5-46.
- 5.2 `scx::scx_load_application` on page 5-47.
- 5.3 `scx::scx_set_parameter` on page 5-48.
- 5.4 `scx::scx_get_parameter` on page 5-49.
- 5.5 `scx::scx_get_parameter_list` on page 5-50.
- 5.6 `scx::scx_cpulimit` on page 5-51.
- 5.7 `scx::scx_timelimit` on page 5-52.
- 5.8 `scx::scx_parse_and_configure` on page 5-53.
- 5.9 `scx::scx_print_statistics` on page 5-57.
5.1 \texttt{scx::scx\_initialize}

This function initializes the simulation.

Initialize the simulation before constructing any exported subsystem.

\begin{verbatim}
void scx\_initialize(const std::string &id,
                    scx\_simcontrol\_if *ctrl = scx\_get\_default\_simcontrol());
\end{verbatim}

\begin{itemize}
\item \texttt{id}  an identifier for this simulation.
\item \texttt{ctrl}  a pointer to the simulation controller implementation. It defaults to the one provided with Arm models.
\end{itemize}

\textbf{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.

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

```cpp
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.

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

t  
  the number of seconds to run. Defaults to unlimited.
5.7 \texttt{scx::scx\_timelimit}

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

\begin{verbatim}
void scx_timelimit(double t);
\end{verbatim}

\begin{itemize}
  \item \texttt{t} \quad the number of seconds to run. Defaults to unlimited.
\end{itemize}
### 5.8 scx::scx_parse_and_configure

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

```cpp
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 CortexM7_core0=test0.elf -a CortexM7_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 m7_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 :
#---------
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 available, search for next available port in range: [port:port+range] (0 by default, only try specified port)
--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
```

Copyright © 2018, 2019 Arm Limited or its affiliates. All rights reserved.
Non-Confidential
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.

```c
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.