Cortex-R82 SystemC Cycle Model

Version 11.5

User Guide

102386_1105_00_en

Cortex-R82 SystemC Cycle Model

User Guide

Release Information

Document History

Issue	Date	Confidentiality	Change
1104-00	12 February 2021	Non-Confidential	First release
1104-01	01 May 2021	Non-Confidential	Documentation update
1105-00	25 June 2021	Non-Confidential	Release 11.5

Arm Limited. Company 02557590 registered in England.

110 Fulbourn Road, Cambridge, England CB1 9NJ.

(LES-PRE-20349)

Confidentiality Status

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

Unrestricted Access is an Arm internal classification.

Product Status

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

Web Address

developer.arm.com

Progressive terminology commitment

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

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

Contents
Cortex-R82 SystemC Cycle Model User Guide

Preface

This preface introduces the *Cortex-R82 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®-R82 SystemC Cycle Model into a SystemC design and simulation environment.

Using this book

This book is organized into the following chapters:

Chapter 1 Cycle Model functionality and operating requirements
This section introduces the Arm Cortex®-R82 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 SystemC Export API function reference
This section describes the functions of the SystemC eXport (SCX) API that are supported by SystemC Cycle Models. Each description of a class or function includes the C++ declaration and the use constraints.

Glossary

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

See the *Arm Glossary* for more information.

Typographic conventions

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

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

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

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

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

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

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

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

SMALL CAPITALS

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

Feedback

Feedback on this product

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

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

Feedback on content

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

- The title *Cortex-R82 SystemC Cycle Model User Guide*.
- The number 102386_1105_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® Documentation*.
- *Technical Support*.
- *Arm® Glossary*.

Chapter 1
Cycle Model functionality and operating requirements

This section introduces the Arm Cortex®-R82 SystemC Cycle Model.

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

It contains the following sections:

1.1 Functionality of the SystemC Cycle Model

The Arm Cortex-R82 SystemC Cycle Model simulates the Cortex-R82 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:

- Configurations of up to eight CPUs.
- Configurable number of interrupts (0 to 480 in increments of 32).
- AXI master port.
- RAM protection.
- Power Policy Unit (PPU).
- Embedded Logic Analyzer (ELA600).
- Access to Tightly Couple Memory (TCM) via slave port.
- Variable ICache and DCache sizes.
- Internal ITCM and DTCM.
- Variable ITCM and DTCM sizes.
- Accelerator Coherency Port (ACP) and ACP bridge with configurable ID size.
- 0, 16, or 32 Memory Protection Unit (MPU) regions.
- Floating Point Unit (FPU).
- Fast Peripheral Port (FPP).
- Addition of one latency cycle to ITCM data read.
- Clock gating.

Unsupported hardware features

The following features of the Cortex-R82 hardware are not implemented in this release of the Cortex-R82 Cycle Model:

- Memory Built-In Self Test (MBIST) interface.
- Memory Reconstruction Port (MRP).
- Use of Synopsys DesignWare library blocks rather than the Arm equivalents.
- Configurable size for Branch Target Address Cache (BTAC). The default is 512.
- Configurable size of the PREDictor (PRED) RAM. The default is 4096.
- Support for additional signals to control power (required for UPF).
- Visibility of PMU events.

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-R82 hardware:

- Waveform dumping. See *3.4 Dumping waveforms* on page 3-27.
- Tarmac trace support. See *3.6 Configuring Tarmac trace* on page 3-29.

1.2 Prerequisites to using SystemC Cycle Models

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

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

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

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

Prerequisites for Cycle Model reference platform environments

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

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

1.3 Supported platforms, compilers, and simulators

This section describes the requirements for running SystemC Cycle Models.

This section contains the following subsections:

1.3.1 Supported platforms

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

1.3.2 Supported compilers

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

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

1.3.3 Supported simulators

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

1.4 Package contents

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

In a Cycle Model reference platform, these files are located in the root directory `reference_platform/MODELS/component/gccversion/SystemC`.

For models downloaded from Arm IP Exchange (*https://ipx.arm.com/*), these files and directories are located in the root directory `gccversion/SystemC`.

Note

Package content and filenames may differ slightly from model to model.

`.data/`
: Contains the XML data file for the model. The XML data file is readable by the Cycle Model Configuration Tool (`cm_config`), and provides required build and link data when you run `make`.

`icm/`
: Contains header files for Cycle Model APIs.

`lib/`
: Contains `libcomponent.icm.so`, the RTL-based core of the Cycle Model. When you compile the system executable, this must be included.

`univentUtil/` (Models that support Tarmac only)
: Contains files required for Tarmac tracing.

`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.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.

`component_icm.h`
: Header file for `libcomponent.icm.so`, located in `lib/`.

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

`component_params.cfg`
: Cycle Model-specific parameter definitions.

`component_pmu.h` (Models that support PMU only)
: Cycle Model hardware profiling implementation to generate profiling events.

***component*_tarmac.h (Models that support Tarmac trace only)**
Cycle Model Tarmac trace implementation.

***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.

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 reference platform.

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

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

Restrictions and limitations

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

This section contains the following subsections:
- *2.1.1 Cycle Models Configuration Tool command syntax* on page 2-16.
- *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 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 CortexR82
```

--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 CortexR82 --link
```

This example returns flag and library data only:

```
$ cm_config --use-tool gcc:6.4.0 --searchpath MODELS --model CortexR82 --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 CortexR82 --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 CortexR82 --link
```

--sources

Optional.
Outputs a list of source files.

Example:

```
$ cm_config --use-tool gcc:6.4.0 --searchpath MODELS --model CortexR82 --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 CortexR82 --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 CortexR82 --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 CortexR82 --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 2-20

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)
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-R82 Cycle Model. The command specifies ignoring `cms` and `sysc`, targeting only the model:

```
$ cm_config --use-tool gcc:6.4.0 --model CORTEXR82 --sources --link --model-type tlm --
ignore cms cm_sysc

/arm/models/R82_Model_gcc6/gcc640/SystemC/libCORTEXR82.systemc.cpp /arm/models/
R82_Model_gcc6/gcc640/SystemC/CORTEXR82ResetImp.cpp
/arm/models/R82_Model_gcc6/gcc640/SystemC/libCORTEXR82.tlm.cpp -L/arm/models/R82_Model_gcc6/
gcc640/SystemC -L/arm/models/R82_Model_gcc6/gcc640/SystemC/lib
/arm/models/R82_Model_gcc6/gcc640/SystemC/univentUtil/lib/kinfu_tarmac_carbon.so -
lCORTEXR82.icm -licm_runtime
-L/arm/models/R8-SysC-V11.0.0-CMS11.0.0-MK2019.05.31-SOCD9.6.0/ARM/ThirdPartyIP/Accellera/
SystemC/2.3.1/lib/Linux64_GCC-6.4 -lsystemc -lpthread
```

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

```
$ cm_config --use-tool gcc:6.4.0 --model CORTEXR82 --sources --link --model-type pin --
ignore cms cm_sysc

/arm/models/R82_Model_gcc6/gcc640/SystemC/libCORTEXR82.systemc.cpp -L/arm/models/
```

```
R82_Model_gcc6/gcc640/SystemC -L/arm/models/R82_Model_gcc6/gcc640/SystemC/lib
/arm/models/R82_Model_gcc6/gcc640/SystemC/univentUtil/lib/kinfu_tarmac_carbon.so -
lCORTEXR82.icm -licm_runtime
-L/arm/models/R8-SysC-V11.0.0-CMS11.0.0-MK2019.05.31-SOCD9.6.0/ARM/ThirdPartyIP/Accellera/
SystemC/2.3.1/lib/Linux64_GCC-6.4 -lsystemc -lpthread
```

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, `CORTEXR82_HOME` is used as the model root. `SYSTEMC_HOME` is used for the SystemC root:

```
$ cm_config --use-tool gcc:6.4.0 --model CORTEXR82 --sources --link --model-type pin --
ignore cms cm_sysc --use-env CORTEXR82:CORTEXR82_HOME SystemC:SYSTEMC_HOME
${CORTEXR82_HOME}/gcc640/SystemC/libCORTEXR82.systemc.cpp -L${CORTEXR82_HOME}/gcc640/SystemC
-L${CORTEXR82_HOME}/gcc640/SystemC/lib
${CORTEXR82_HOME}/gcc640/SystemC/univentUtil/lib/kinfu_tarmac_carbon.so -lCORTEXR82.icm -
licm_runtime -L${SYSTEMC_HOME}/lib/Linux64_GCC-6.4 -lsystemc -lpthread
```

Example of extracting Arm® data

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

```
$ cm_config --use-tool gcc:6.4.0 --model CORTEXR82 --sources --link --model-type pin --
ignore cms cm_sysc --use-arm

/arm/models/R82_Model_gcc6/gcc640/SystemC/libCORTEXR82.systemc.cpp -L/arm/models/
R82_Model_gcc6/gcc640/SystemC -L/arm/models/R82_Model_gcc6/gcc640/SystemC/lib
/arm/models/R82_Model_gcc6/gcc640/SystemC/univentUtil/lib/kinfu_tarmac_carbon.so -
lCORTEXR82.icm -licm_runtime
```

2.2 Adding custom options to the Makefile

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

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

Build variables

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

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

The following build variable is present only in the model `Makefile`:
- `SRCS_TLM` - TLM sources to be added into the build.

Example 1: Specifying your own version of SystemC

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

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

Example 2: Providing another runtime path

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

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

Example 3: Adding different debug or optimization parameters

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

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

Chapter 3
Using SystemC Cycle Models

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

It contains the following sections:

3.1 Connecting model ports

All pins must be bound to a signal.

For a list of the pins on the Cortex-R82 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 3-24.

3.1.1 Tied pins

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

The Cortex-R82 SystemC Cycle Model has no tied pins.

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 `CortexR82ResetImp.cpp` file located in the directory `gcc`*`version`*`/SystemC/` for the model.

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

```
// bind all the non-TLM ports to their corresponding signals
void CortexR82Imp::bind_nontlm_ports_to_signals()
{
#ifndef CM_SYSC_DONT_BIND_NONTLM_PORTS
    CLKEN.bind(CLKENsignal);
    FCLKEN.bind(FCLKENsignal);
    HCLKEN.bind(HCLKENsignal);
    CLK1EN.bind(CLK1ENsignal);
    .
    .
    .
```

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

Procedure

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

1. In the `CortexR82ResetImp.cpp` file, inside the `bind_nontlm_ports_to_signals()` function, remove the binding by commenting out the line for the port.
2. In the reference platform `Systems/system_test.cpp` testbench file, the unbound port can be either:
   - Driven directly from `sc_main`; e.g., `sc_signal.write(1)`.
   - Bound to another sc_modules port; e.g., *`another_sc_module.port`*`.bind(`*`cycle_model.port`*`)`. You can create a new sc_module in a separate `.cpp` file, or include it in the existing `Systems/system_test.cpp` file.
3. Recompile the reference platform using `Systems/Makefile`. This recompiles the Cycle Model and the platform.

3.2 Resetting the SystemC Cycle Model

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

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

- For pin-level models, the file is `CortexR82ResetModule.h`
- For TLM models, the file is `CortexR82ResetImp.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-R82 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
CortexR82.WAVEFORMS_ENABLED=true
```

Available parameters

To list the parameters supported by the model:

- In a Cycle Model reference platform environment, enter `./system_test --list-params` in the `Systems` directory.
- View the file `CortexR82_params.cfg` located in the directory `MODELS/CortexR82_xCPU/gccversion/SystemC`.

Related information

- `CortexR82_params.cfg` located in the directory `MODELS/CortexR82_xCPU/gccversion/SystemC`. This file contains a complete list of the parameters supported by the model, and their supported values.
- *CADI RemoteConnection parameters* describes parameters related to configuring CADI debug connections. Debug parameters do not appear in the `CortexR82_params.cfg` file.
- The Technical Reference Manual for your IP includes additional information about supported parameter values.

3.4 Dumping waveforms

This section describes how to configure waveform dumping.

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

Note

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

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

Table 3-1 Waveform parameters

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

For example:

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

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

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

```
./system_test -a ../Applications/hello_world/armcc/elf/test.elf -C
CortexR82.WAVEFORM_TYPE=FSDB -C CortexR82.WAVEFORMS_ENABLED=true
```

Related information

- *3.3 Setting model parameters* on page 3-26

3.5 Loading DTCM and ITCM

You can load the DTCMS and ITCMS memories by enabling the related parameters within the system executable.

Data files

Note

By default, the system loads data files with predefined names. Before enabling TCM memories, create the required `.dat` files and name them accordingly.

The data files are named as follows:

- `CortexR82_cpux_{D|I}TCM.dat` (for example, `CortexR82_cpu0_ITCM.dat`

If you do not want to use the predefined `.dat` file names, you can create `.dat` files with different names, and load them on the command line when you run the simulation. Call the upper-case parameter defined in the associated `.h` file (for example, `CPU0_ITCM_DAT_FILE`).

For more information, review the following files, located in `MODELS/CortexR82_xCPU/gcc`*`version`*`/SystemC`:

- `CortexR82_itcm.h`
- `CortexR82_dtcm.h`

These files contain the default parameter names for the data files to be loaded.

Related parameters

Set the following parameters:

Note

The names of the data files to be loaded are defined by the data file parameters (`CPUx_*_DAT_FILE`). If you want to customize the default values for the `CPUx_*_DAT_FILE` parameters, make these changes before setting `LOAD_DTCMS` and `LOAD_ITCMS`.

Table 3-2 TCM parameters

Parameter	Available settings	Default setting
`LOAD_DTCMS`	`true`, `false`	`false`
`LOAD_ITCMS`	`true`, `false`	`false`

See the files `CortexR82_dtcm.h` and `CortexR82_itcm.h` for more information.

Example of enabling TCM load on the command line

Following is an example of enabling DTCM memory load on the command line:

```
./system_test -a ../Applications/hello_world/armcc/elf/test.elf -C CortexR82.LOAD_DTCMS=true
```

Related information

- *3.3 Setting model parameters* on page 3-26

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 Cortex-R82 design with two cores and one cluster:

```
scx::scx_set_parameter("cr82.TARMAC_LOGFILE_NAME","tarmac.cr82.@CPUID@.log");
                scx::scx_set_parameter("cr82.TARMAC_LOGFILE_ENABLED",true);
```

This creates the files `tarmac.cr82.0.log` and `tarmac.cr82.1.log`.

Enabling Tarmac trace in multicluster environments

For multiple clusters, use the default Tarmac log file name (`cr82.`*`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 4 SystemC Export API function reference* on page 4-31 for details about the functions supported by SystemC Cycle Models.

Chapter 4
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:

4.1 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.

4.2 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.

4.3 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.

4.4 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.
  ```

4.5 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()`.

4.6 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.

4.7 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.

4.8 scx::scx_parse_and_configure

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

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

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

`argv`
: command-line options.

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

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

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

Options

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

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

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

Note

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

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

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

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

```
$ ./system_test -a CortexR82_core0=test0.elf -a CortexR82_core1=test1.elf -S -p
```

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

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

--cadi-log, -L

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

For example:

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

--cadi-server, -S *FILE*

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

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

For example:

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

--config-file, -f *FILE*

This option loads model parameters from the specified configuration file.

`FILE`
: Name of the configuration file.

For example:

```
$ ./system_test --config-file cr82_config.cfg
```

--cpulimit

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

--help, -h

This option prints descriptions of available command line options.

Note

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

For example:

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

--list-params, -l

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

For example:

```
$ ./system_test -l
.
.
.
Starting Sim
# Parameters:
# instance.parameter=value       #(type, mode) default = 'def value' : description :
[min..max]
#------------------------------------------------------------------------------
REMOTE_CONNECTION.CADIServer.enable_remote_cadi=0     # (bool  , init-time) default =
'0'      : Allow connections from remote hosts
REMOTE_CONNECTION.CADIServer.listen_address=127.0.0.1  # (string, init-time) default =
'127.0.0.1' : Network address the server should listen on if enable_remote_cadi is set
("127.0.0.1" by default)
REMOTE_CONNECTION.CADIServer.port=31627               # (int   , init-time) default =
'0x7b8b' : TCP port the server should listen on if enable_remote_cadi is set (31627 by
default)
REMOTE_CONNECTION.CADIServer.range=0                  # (int   , init-time) default =
'0x0'    : If requested port is not avaliable, search for next avaliable port in range:
[port:port+range] (0 by default, only try specified port)
cortexr8_core.ACLKENSC=1                              # (int   , run-time ) default =
'0x1'    : ACLKENSC enable parameter
```

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

--list-regs

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

For example:

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

--quiet

Run quiet, suppress informational output.

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

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

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

`PARAMETER`
Specifies the parameter to set.

`VALUE`
Specifies the parameter value.

For example:

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

--print-port-number, -p

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

For example:

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

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

--stat

This option prints run statistics on simulation exit.

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

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

```
--- R8-MP4-SysC statistics: ---------------------------------------------------
Simulated time                          : 0.000000s
User time                               : 0.028996s
System time                             : 0.002999s
Wall time                               : 4.278761s
cortexr8_core.cpu0                      :   0.00 KIPS (           0 Inst)
cortexr8_core.cpu1                      :   0.00 KIPS (           0 Inst)
```

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

--timelimit, -T

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

4.9 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.