Document History

<table>
<thead>
<tr>
<th>Issue</th>
<th>Date</th>
<th>Confidentiality</th>
<th>Change</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>31 May 2014</td>
<td>Non-Confidential</td>
<td>New document for Fast Models v9.0, from DUI0370R for v8.3.</td>
</tr>
<tr>
<td>B</td>
<td>30 November 2014</td>
<td>Non-Confidential</td>
<td>Update for v9.1.</td>
</tr>
<tr>
<td>C</td>
<td>28 February 2015</td>
<td>Non-Confidential</td>
<td>Update for v9.2.</td>
</tr>
<tr>
<td>D</td>
<td>31 May 2015</td>
<td>Non-Confidential</td>
<td>Update for v9.3.</td>
</tr>
<tr>
<td>E</td>
<td>31 August 2015</td>
<td>Non-Confidential</td>
<td>Update for v9.4.</td>
</tr>
<tr>
<td>F</td>
<td>30 November 2015</td>
<td>Non-Confidential</td>
<td>Update for v9.5.</td>
</tr>
<tr>
<td>G</td>
<td>29 February 2016</td>
<td>Non-Confidential</td>
<td>Update for v9.6.</td>
</tr>
<tr>
<td>H</td>
<td>31 May 2016</td>
<td>Non-Confidential</td>
<td>Update for v10.0.</td>
</tr>
<tr>
<td>I</td>
<td>31 August 2016</td>
<td>Non-Confidential</td>
<td>Update for v10.1.</td>
</tr>
<tr>
<td>J</td>
<td>11 November 2016</td>
<td>Non-Confidential</td>
<td>Update for v10.2.</td>
</tr>
<tr>
<td>K</td>
<td>17 February 2017</td>
<td>Non-Confidential</td>
<td>Update for v10.3.</td>
</tr>
<tr>
<td>1100-00</td>
<td>31 May 2017</td>
<td>Non-Confidential</td>
<td>Update for v11.0. Document numbering scheme has changed.</td>
</tr>
<tr>
<td>1101-00</td>
<td>31 August 2017</td>
<td>Non-Confidential</td>
<td>Update for v11.1.</td>
</tr>
<tr>
<td>1102-00</td>
<td>17 November 2017</td>
<td>Non-Confidential</td>
<td>Update for v11.2.</td>
</tr>
<tr>
<td>1103-00</td>
<td>23 February 2018</td>
<td>Non-Confidential</td>
<td>Update for v11.3.</td>
</tr>
<tr>
<td>1104-00</td>
<td>22 June 2018</td>
<td>Non-Confidential</td>
<td>Update for v11.4.</td>
</tr>
<tr>
<td>11.4.2-00</td>
<td>17 August 2018</td>
<td>Non-Confidential</td>
<td>Update for v11.4.2.</td>
</tr>
<tr>
<td>1105-00</td>
<td>23 November 2018</td>
<td>Non-Confidential</td>
<td>Update for v11.5.</td>
</tr>
<tr>
<td>1106-00</td>
<td>26 February 2019</td>
<td>Non-Confidential</td>
<td>Update for v11.6.</td>
</tr>
<tr>
<td>1107-00</td>
<td>17 May 2019</td>
<td>Non-Confidential</td>
<td>Update for v11.7.</td>
</tr>
<tr>
<td>1108-00</td>
<td>05 September 2019</td>
<td>Non-Confidential</td>
<td>Update for v11.8.</td>
</tr>
<tr>
<td>11.8.1-00</td>
<td>03 October 2019</td>
<td>Non-Confidential</td>
<td>Update for v11.8.1.</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 © 2014–2019 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

www.arm.com
Contents
Fast Models User Guide

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

Chapter 1  Introduction to Fast Models
1.1 What is Fast Models? .......................................................... 1-10
1.2 What does Fast Models consist of? .................................. 1-12
1.3 Fast Models glossary ...................................................... 1-15
1.4 Fast Models design ....................................................... 1-19
1.5 Data collection in Fast Models ...................................... 1-26

Chapter 2  Installing Fast Models
2.1 Requirements for Fast Models .......................................... 2-28
2.2 Installing Fast Models .................................................... 2-30
2.3 Installing the TPIP package .......................................... 2-31
2.4 Dependencies for Red Hat Enterprise Linux ................... 2-32

Chapter 3  System Canvas Tutorial
3.1 About this tutorial .......................................................... 3-35
3.2 Starting System Canvas .................................................. 3-36
3.3 Creating a new project .................................................... 3-37
3.4 Add and configure components ...................................... 3-39
3.5 Connecting components ............................................... 3-43
3.6 View project properties and settings ............................... 3-44
3.7 Changing the address mapping ...................................... 3-47
3.8 Building the system ................................................................. 3-49
3.9 Building SystemC ISIM targets ............................................... 3-50
3.10 Debugging with Model Debugger ......................................... 3-51

**Chapter 4**

**Debugging LISA+ components**

4.1 Building a debuggable model ................................................ 4-55
4.2 Debugging with GDB at the source level ................................ 4-56
4.3 Debugging with Microsoft Visual Studio at the source level ....... 4-57
4.4 Attaching Microsoft Visual Studio to a simulator process .......... 4-58

**Chapter 5**

**System Canvas Reference**

5.1 Launching System Canvas .......................................................... 5-60
5.2 System Canvas GUI ................................................................. 5-61
5.3 System Canvas dialogs ............................................................. 5-75

**Chapter 6**

**System Generator Reference**

6.1 About System Generator .......................................................... 6-109
6.2 SimGen command-line options .................................................. 6-110
6.3 Decreasing compilation time with num-comps-file and num-build-cpus .. 6-113
6.4 Setting command-line options from System Canvas ..................... 6-114

**Chapter 7**

**SystemC Export with Multiple Instantiation**

7.1 About SystemC Export with Multiple Instantiation .................. 7-116
7.2 Building a SystemC subsystem with System Canvas .................. 7-117
7.3 Adding a SystemC subsystem to a SystemC system ................. 7-122
7.4 Auto-bridging ................................................................. 7-123
7.5 SystemC Export generated ports ................................................. 7-124
7.6 SystemC Export API ............................................................. 7-128
7.7 Scheduler API ................................................................. 7-146
7.8 SystemC Export limitations ...................................................... 7-163

**Chapter 8**

**Generic Graphics Accelerator**

8.1 Introduction ................................................................. 8-165
8.2 Using the Generic Graphics Accelerator ................................... 8-169
8.3 The configuration file ........................................................... 8-187
8.4 Feedback on the Generic Graphics Accelerator ......................... 8-188

**Chapter 9**

**Timing Annotation**

9.1 Timing annotation ............................................................... 9-190
9.2 Enabling and disabling timing annotation ................................ 9-191
9.3 CPI files ............................................................................. 9-192
9.4 CPI file syntax ....................................................................... 9-193
9.5 BNF specification for CPI files ................................................ 9-198
9.6 Instruction and data prefetching .............................................. 9-200
9.7 Configuring cache and TLB latency ......................................... 9-202
9.8 Timing annotation tutorial ...................................................... 9-203

**Appendix A**

**SystemC Export generated ports**

A.1 About SystemC Export generated ports .................. Appx-A-219
Preface

This preface introduces the *Fast Models User Guide*.

It contains the following:

- *About this book on page 7.*
About this book

This document describes how to get started with Fast Models, and describes some key tools and features.

Using this book

This book is organized into the following chapters:

**Chapter 1 Introduction to Fast Models**
This chapter provides a general introduction to Fast Models.

**Chapter 2 Installing Fast Models**
This chapter describes the system requirements for Fast Models and how to install and uninstall Fast Models.

**Chapter 3 System Canvas Tutorial**
This chapter describes using System Canvas to build a system model.

**Chapter 4 Debugging LISA+ components**
This chapter describes how to use GDB on Linux or Microsoft Visual Studio on Microsoft Windows to debug the LISA source code of models.

**Chapter 5 System Canvas Reference**
This chapter describes the windows, menus, dialogs, and controls in System Canvas.

**Chapter 6 System Generator Reference**
This chapter describes System Generator (SimGen).

**Chapter 7 SystemC Export with Multiple Instantiation**
This chapter describes the Fast Models SystemC Export feature with Multiple Instantiation (MI) support.

**Chapter 8 Generic Graphics Accelerator**
This chapter describes the Generic Graphics Accelerator (GGA) feature in Fast Models.

**Chapter 9 Timing Annotation**
This chapter describes timing annotation. You can use this set of Fast Models features to estimate the time spent on various operations, for example instruction execution and prefetch. It allows the model to be used for basic benchmarking.

**Appendix A SystemC Export generated ports**
This appendix describes Fast Models SystemC Export generated ports.

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 Fast Models User Guide.
• The number 100965_1181_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 to Fast Models

This chapter provides a general introduction to Fast Models.

It contains the following sections:

• 1.1 What is Fast Models? on page 1-10.
• 1.2 What does Fast Models consist of? on page 1-12.
• 1.3 Fast Models glossary on page 1-15.
• 1.4 Fast Models design on page 1-19.
• 1.5 Data collection in Fast Models on page 1-26.
1.1 What is Fast Models?

The Fast Models product comprises a library of Programmer's View (PV) models and tools that enable partners to build, execute, and debug virtual platforms. Virtual platforms enable the development and validation of software without the need for target silicon. The same virtual platform can be used to represent the processor or processor subsystem in SoC validation.

Fast Models are delivered in two ways:

• As a portfolio of Arm IP models and tools to let you generate a custom model of your exact system.
• As standalone models of complete Arm platforms that run out-of-the-box to let you test your code on a generic system quickly.

The benefits of using Fast Models include:

Develop code without hardware

Fast Models provides early access to Arm IP, well ahead of silicon being available. Virtual platforms are suitable for OS bring-up and for driver, firmware, and application development. They provide an early development platform for new Arm technology and accelerate time-to-market.

High performance

Fast Models uses Code Translation (CT) processor models, which translate Arm instructions into the instruction set of the host dynamically, and cache translated blocks of code. This and other optimization techniques, for instance temporal decoupling and Direct Memory Interface (DMI), produce fast simulation speeds for generated platforms, between 20-200 MIPS on a typical workstation, enabling an OS to boot in tens of seconds.

Customize to model your exact system

Fast Models provides a portfolio of models that are flexible and can easily be customized using parameters to test different configurations. Using the System Canvas tool you can model your own IP and integrate it with existing model components.

You can also export components and subsystems from the Fast Models portfolio to SystemC for use in a SystemC environment. Such an exported component is called an Exported Virtual Subsystem (EVS). EVSs are compliant with SystemC TLM 2.0 specifications to provide compatibility with Accellera SystemC and a range of commercial simulation solutions.

Run standalone or debug using development tools

Generated platform models are equipped with Component Architecture Debug Interface (CADI). This allows them to be used standalone or with development tools such as Arm Development Studio or Arm Keil® MDK, as well as providing an API for third party tool developers.

Test architecture compliance

Fast Models provides Architecture Envelope Models (AEMs) for Armv8-A, Armv8-R, and Armv8-M. These are specialist architectural models that are used by Arm and by Arm architecture licensees to validate that implementations are compliant with the architecture definition.

Trace and debug interfaces

Fast Models provides the Model Trace Interface (MTI) and CADI for trace and debug. These APIs enable you to write plug-ins to trace and debug software running on models. Fast Models also provides some pre-built MTI plug-ins, for example Tarmac Trace, that you can use to output trace information.
Build once, run anywhere

Since the same binary runs on the model, the target development hardware, and the final product, you only need to build it using the Arm toolchain.

Host platform compatibility

Fast Models can be used on both Linux and Microsoft Windows hosts.

Related references

5.2 System Canvas GUI on page 5-61

Related information


About Model Debugger
1.2 What does Fast Models consist of?

The Fast Models package contains the tools and model components that are needed to model a system. The tools and the portfolio of models are installed under separate directories, `FastModelsTools_n.n` and `FastModelsPortfolio_n.n` respectively, where `n.n` is the Fast Models version number.

Arm also supplies a wide range of pre-built Fixed Virtual Platforms (FVPs), including a free of charge FVP called the Foundation Platform, separately from the Fast Models package.

This section contains the following subsections:
- 1.2.1 Fast Models tools on page 1-12.
- 1.2.2 Fast Models portfolio on page 1-13.
- 1.2.3 Other Fast Models products on page 1-14.

1.2.1 Fast Models tools

Fast Models tools is a set of tools that enable you to create custom models.

It consists of the following:

**System Canvas**

A GUI design tool for developing new model components written in LISA+ and for building and launching system models. It displays the model as either LISA+ source code, or graphically, in a block diagram editor:

![System Canvas](image)

**System Generator**

A backend tool that handles system generation. System Generator can either be invoked from the System Canvas GUI, or by using the `simgen` command-line utility. System models that are created using System Generator can be used with other Arm development tools, for example Arm Development Studio or Model Debugger, or can be exported to SystemC for integration with proprietary models.
Model Debugger
A GUI debugger that connects to a model through the CADI interface. It enables you to debug code running on the model.

Model Shell
A command-line tool for launching simulations. It can also run a CADI debug server to enable debuggers to connect to the model. Some models, for example Fixed Virtual Platforms (FVPs), are supplied as executables. These models are launched standalone rather than from Model Shell, and have the same command-line options as Model Shell.

Note
Arm deprecates Model Shell from Fast Models version 11.2.

1.2.2 Fast Models portfolio
Fast Models portfolio consists of a variety of components, including the following:

- A collection of models and protocols, provided as LISA+ components. You can use them to model a system using the Fast Models tools. Ports and protocols are used for communication between components. Some models are of Arm IP, while others are not. Examples of Arm IP models include:
  - Processors, including models of all Arm Cortex® processors and architectural models for Armv8, called AEMs.
  - Models of Arm media IP such as GPUs, video processors, and display processors.
  - Peripherals, for instance Arm CoreLink™ interconnect, interrupt controllers, and memory management units.

Some models are abstract components that do not model specific Arm IP, but are required by the software modeling environment. For example:
- PVBus components to model bus communication between components.
- Emulated I/O components to allow communication between the simulation and the host, such as a terminal, a visualization window, and an ethernet bridge.

- Platform model examples supplied as project files, so must first be built using System Generator. Examples are provided for both standalone simulation and for SystemC export, and include:
— Arm Versatile™ Express development boards for Armv7-A and Armv7-R processors.
— Armv8-A and Armv8-R Base Platform.
— MPS2 development boards for Armv6-M, Armv7-M, and Armv8-M processors.

- Accellera SystemC and TLM header files and libraries, which are required to build FVPs and the platform model examples.
- Model Trace Interface (MTI) plug-ins. MTI is the interface used by Fast Models to emit trace events during execution of a program, for example branches, exceptions, and cache hits and misses. Fast Models provides some pre-built MTI plug-ins that you can load into a model to capture trace data, without having to write your own plug-ins. For example:
  — TarmacTrace can trace all processor activity or a subset of it, for instance only branch instructions or memory accesses.
  — GenericTrace allows you to trace any of the MTI trace sources that the models can produce.
  — Fastline generates traces in .apc format that you can import into Arm Streamline for analysis.

Some trace plug-ins are provided in source form as programming examples. They can also be compiled and used.
- Some ELF images that you can run on models for evaluation purposes.
- Networking setup scripts to bridge network traffic from the simulation to the host machine’s network.

### 1.2.3 Other Fast Models products

The following Fast Models products are available separately from the main package:

**Fixed Virtual Platforms (FVPs)**

FVPs are models of Arm platforms, including processors, memory and peripherals. They are supplied as standalone executables for Linux and Windows. They are not customizable, although you can configure some aspects of their behavior through command-line parameters. FVPs can be based on Arm Versatile Express development boards, called VE FVPs, on the Armv8-A Base Platform, called Base FVPs, or on Arm MPS2 and Arm MPS2+ platforms, for Cortex-M series processors. They are available with a wide range of Armv7 and Armv8 processors and support the CADI and MTI interfaces, so can be used for debugging and for trace output. For more information, see [Arm Developer website](#).

Arm provides validated Linux and Android deliverables for the Armv8-A AEM Base Platform FVP and for the Foundation Platform. These are available on the Arm Community website at *[Arm Development Platforms](#)*. To get started with Linux on Armv8-A FVPs, see *[Armv8-A FVPs](#)* also on Arm Community.

**Foundation Platform**

A simple FVP that includes an Armv8-A AEM processor model, that is suitable for running bare-metal applications and for booting Linux. It is available for Linux hosts only and can be downloaded free of charge from the [Arm® Self-Service Portal](#), registration and login are required.

**System Guideline FVPs**

FVPs that include documentation to guide SoC design and a reference software stack that is validated on the FVP. They were formerly known as Reference Data FVPs. Details can be requested from your Arm account team.

**Third party IP**

A package that contains third party add-ons for Fast Models. These include some additional ELF images, including Dhrystone.
1.3 Fast Models glossary

This glossary defines some of the Arm-specific technical terms and acronyms that are used in the Fast Models documentation.

AMBA-PV

A set of classes and interfaces that model AMBA® buses. They are implemented as an extension to the TLM v2.0 standard.

See AMBA-PV extensions.

Architecture Envelope Model (AEM)

A model of the Arm architecture that aims to expose software bugs by modeling the extremes of behavior that the Arm architecture allows.

Auto-bridging

A Fast Models feature that SimGen uses to automatically convert between LISA+ protocols and their SystemC equivalents. It helps to automate the generation of SystemC wrappers for LISA+ subsystem models.

See 7.4 Auto-bridging on page 7-123.

Base Platform

An example platform that is provided as part of Fast Models. It is capable of booting Linux and Android. Variations of this platform are available for various core types, and with additional system IP. They are often used together with Linux images that Linaro provides.

See Base Platform.

Component Architecture Debug Interface (CADI)

A C++ interface that is used by debuggers to control a model. CADI enables convenient and accurate debugging of Fast Models and Cycle Models.

See About the Component Architecture Debug Interface.

Code Translation (CT)

A technique that processor models use to enable fast execution of code. CT models translate code dynamically and cache translated code sequences to achieve fast simulation speeds.

Cycle Models

Cycle-accurate software models of Arm IP, for example processors or peripherals. They are cycle-accurate and functionally accurate, so are usable for benchmarking. Cycle Models is a separate product from Fast Models, but they can be used alongside each other, in particular by using the Cycle Models Swap-and-Play feature.

Direct Memory Interface (DMI)

A TLM 2.0 interface that provides direct access to memory. It accelerates memory transactions, which improves model performance.

Exported Virtual Subsystem (EVS)

A SystemC module that is generated by using the SystemC Export feature to export a Fast Models subsystem.

See 7.1 About SystemC Export with Multiple Instantiation on page 7-116.
Fast Models

High performance software models of components of Arm SoCs, for example processors or peripherals. Components might have subcomponents to form a hierarchy, and might be connected together to form a platform model. Fast Models are functionally accurate, but not cycle-accurate.

Fixed Virtual Platform (FVP)

A pre-built platform model that enables applications and operating systems to be written and debugged without the need for real hardware. FVPs are also referred to as Fixed Virtual Prototypes. They were formerly known as RTSMs.

See About FVPs.

Foundation Model

See Foundation Platform.

Foundation Platform

A freely available, easy-to-use FVP for application developers that models the Armv8-A architecture. It can be downloaded from the Arm® Self-Service Portal, registration and login are required. Foundation Platform was formerly known as Foundation Model.

IMP DEF

Used in register descriptions in the Fast Models Reference Manual to indicate behavior that the architecture does not define. Short for Implementation Defined.

Integrated Simulator (ISIM)

An executable model binary that can run standalone, without the need for Model Shell or Model Debugger. ISIMs are generated by simgen by statically linking a model with the SystemC framework. ISIMs simplify model debugging and profiling.

See 3.9 Building SystemC ISIM targets on page 3-50.

Iris

An interface for debug and trace of model behavior.

See Iris Developer Guide

Language for Instruction Set Architectures (LISA, LISA+)

LISA is a language that describes instruction set architectures. LISA+ is an extended form of LISA that supports peripheral modeling. LISA+ is used for creating and connecting model components. The Fast Models documentation does not always distinguish between the two terms, and sometimes uses LISA to mean both.


Microcontroller Prototyping System (MPS2)

Arm Versatile Express V2M-MPS2 and V2M-MPS2+ are motherboards that enable software prototyping and development for Cortex-M processors. The MPS2 FVP models a subset of the functionality of this hardware.

See MPS2 - about.

Model Debugger

A Fast Models debugger that enables you to execute, connect to, and debug any CADI-compliant model. You can run Model Debugger using a GUI or from the command line.

See About Model Debugger.
Model Shell
See About Model Shell.

Model Trace Interface (MTI)
A trace interface that is used by Fast Models to expose real-time information from the model.

Platform Model
A model of a development platform, for example an FVP.

Programmers' View (PV) Model
A high performance, functionally accurate model of a hardware platform. It can be used for booting an operating system and executing software, but not to provide hardware-accurate timing information.
See Timing Annotation.

PVBus
An abstract, programmers view model of the communication between components. Bus masters generate transactions over the PVBus and bus slaves fulfill them.
See PVBus components.

Quantum
A set of instructions that the processor issues at the same point in simulation time. The processor then waits until other components in the system have executed the instructions for the same time slice, before executing the next quantum.

Real-Time System Model (RTSM)
An obsolete term for Fixed Virtual Platform (FVP).

SimGen
An alternative term for System Generator.

Synchronous CADI (SCADI)
An interface that provides a subset of CADI functions to synchronously read and write registers and memory. You can only call SCADI functions from the model thread itself, rather than from a debugger thread. SCADI is typically used from within MTI or by peripheral components to access the model state and to perform run control.
See About SCADI.

syncLevel
Each processor model has a syncLevel with four possible values. It determines when a synchronous watchpoint or an external peripheral breakpoint can stop the model, and the accuracy of the model state when it is stopped.
See syncLevel definitions.

System Canvas
An application that enables you to manage and build model systems using components. It has a block diagram editor for adding and connecting model components and setting parameters.
See 5.2 System Canvas GUI on page 5-61.
SystemC Virtual Platform (SVP)

Similar to an FVP, except that the interconnections between components are defined in SystemC rather than in LISA.

System Generator

A utility that uses a project file containing configuration information to generate a platform model. You can run System Generator from the command line, by invoking the simgen executable, or from the System Canvas GUI.

See 6.1 About System Generator on page 6-109.

System Model

An alternative term for Platform Model.

Tarmac

A textual trace of the program that is executing on the model. Fast Models provides a Tarmac plugin to produce this trace format.

See Tarmac Trace.

Timing Annotation

A set of Fast Models features that allow timing configuration for various operations, for instance instruction execution and branch prediction. This allows the model to be used for basic benchmarking.

See 9.1 Timing annotation on page 9-190.

Versatile Express (VE)

A family of Arm hardware development boards. The term is abbreviated to VE when used in model names. For example, FVP_VE_Cortex-A5x1 is an FVP model of the Versatile Express hardware platform, containing a single Cortex-A5 processor.

Related information

Arm Glossary
1.4 Fast Models design

This section describes the design of Fast Models systems.

This section contains the following subsections:

- 1.4.1 Fast Models design flow on page 1-19.
- 1.4.2 Project files on page 1-20.
- 1.4.3 Repository files on page 1-21.
- 1.4.4 File processing order on page 1-22.
- 1.4.5 Hierarchical systems on page 1-22.

1.4.1 Fast Models design flow

The basic design flow for Fast Models is:

1. Create or buy standard component models.
2. Use System Canvas to connect components and set parameters in the LISA+ source code.
3. Generate a new model using System Generator either from the command line (SimGen) or from within the System Canvas GUI.
4. Use the new model as input to a more complex system or distribute it as a standalone simulation environment.

![Fast Models design flow diagram](Figure 1-3)

The input to System Generator consists of:

**C++ library objects**

Typically these are models of processors or standard peripherals.

**LISA+ source code**

The source code files define custom peripheral components. They can be existing files in the Fast Models portfolio or new LISA+ files that were created in System Canvas. The LISA+ descriptions can be located in any directory. One LISA+ file can contain one or more component descriptions.

**Project file**

System Generator requires a .sgproj project file to configure the build.
After the required components have been added and connected, System Generator uses gcc or the Visual Studio C++ compiler to produce the output object as one of the following:

- One or more CADI libraries, which you can load into Model Shell or Model Debugger.
- An ISIM executable, for instance an FVP. You could run this standalone, or you could connect a CADI-enabled debugger to it, such as Model Debugger or Arm Development Studio Debugger.
- An EVS, which can be used as a building block for a SystemC system. It is generated using the Fast Models SystemC Export feature.

---

**Note**

To build ISIM executables or EVSs, a SystemC environment must be installed, and the `SYSTEMC_HOME` environment variable must be set.

1.4.2 Project files

System Canvas uses one project file (.sgproj) to describe the build options used for each host platform and the files that are required to build the model.

- There is no requirement to provide a makefile and a set of configuration files for each new project.
- Each project file references all files that System Canvas needs to build and run a simulation, including LISA, C, and C++ sources, libraries, files to deploy to the simulation directory, and nested repository files.

Repository files have the same format as project files.

You can add single files or a complete repository, such as the Fast Models Portfolio, to the project file.

---

**Figure 1-4 Example organization of project directories and files on Microsoft Windows**
The My_Projects directory contains the My_System.sgproj project file:
1. My_System.sgproj points to the standard Fast Models Portfolio repository file sglib.sgrepo.
2. The sglib.sgrepo repository file contains a list of repository locations such as components.sgrepo.
3. components.sgrepo lists the locations of the LISA files for the components and the location and type of libraries that are available for the components.
4. The project file lists My_System.lisa as the top-level LISA file for the system. The top-level LISA file lists the components in the system and shows how they interconnect.
5. This project uses a custom component in addition to the standard Fast Models Portfolio components. Custom components can exist anywhere in the directory structure. In this case, only the My_System component uses the custom component, so the My_custom_component.lisa file is in the same directory.
6. System Canvas generates the My_System.sgcanvas and My_custom_component.sgcanvas files to save display changes in the Workspace window. These files describe the display settings for a component such as:
   • Component location and size.
   • Label text, position and formatting.
   • Text font and size.
   • The moving of or hiding of ports.
   • Grid spacing.

The build process does not use .sgcanvas files. System Canvas uses them for its Block Diagram view.
7. My_System.sgproj defines Win64-Debug-VC2015 as the build directory for the selected platform. Other build options in the project file include:
   • The host platform, for instance “Win64”.
   • The compiler, for example “VC2015” and compiler options.
   • Additional linker options.
   • Additional options to be passed to SimGen.
   • The type of target to build, for example an ISIM executable, a CADI library, or a SystemC component.

Related references
5.3.17 Project Settings dialog on page 5-93
Project file contents on page 5-100

1.4.3 Repository files

Repository files group together references to commonly used files, eliminating the need to specify the path and library for each component in a project.

Repository files contain:
• A list of components.
• The paths to the LISA sources for the components.
• A list of library objects for the components.
• Optionally, lists of paths to other repository files. This enables a hierarchical structure.

System Canvas adds the default model repositories to a project when creating it. Changing these repository settings does not affect existing projects. The project_name.sgproj files contain the paths to the repositories as hard code. To change the repositories for an existing project, open the file and edit the paths.

Default repositories can also preset required configuration parameters for projects that rely on the default model library. These parameters are:
• Additional Include Directories.
• Additional Compiler Settings.
• Additional Linker Settings.
1.4.4 File processing order

The processing order enables a custom implementation of a Fast Models component.

An example of a project file

```java
/// project file
sgproject "MyProject.sgproj"
{
files
{
  path = "./MyTopComponent.lisa``;
  path = "./MySubComponent1.lisa``;
  path = "./repository.sgrepo``;
  path = "./MySubComponent2.lisa``;
}
}
```

An example of a repository file

```java
/// subrepository file
sgproject "repository.sgrepo"
{
files
{
  path = "./LISA/ASubComponent1.lisa``;
  path = "./LISA/ASubComponent2.lisa``;
}
}
```

System Canvas processes the files in sequence, expanding sub-repositories as it encounters them:

1. "./MyTopComponent.lisa``
2. "./MySubComponent1.lisa``
3. "./repository.sgrepo``
   a. "./LISA/ASubComponent1.lisa``
   b. "./LISA/ASubComponent2.lisa``
4. "./MySubComponent2.lisa``

Changing the processing order allows customization. If MySubComponent1.lisa and "./LISA/ASubComponent1.lisa`` both list a component with the same name, the application uses only the first definition.

The File List view of System Canvas shows the order of components in the project file. Use the application controls to re-order the files and repositories:

- The Up and Down context menu entries in the File List view of the Component window. The commands have keyboard shortcuts of Alt+Arrow Up and Alt+Arrow Down.

  - You can also drag-and-drop files inside a repository or between repositories.
- The Up and Down buttons on the Default Model Repository tab in the Properties dialog, for repositories in new projects.

1.4.5 Hierarchical systems

The terms system and component are both used to describe the output from System Canvas. The main difference is whether the output is intended as a standalone system or is to be used within a larger system.

The block diagram shows the advantage of using a hierarchical system with a complex model.
The main component in the system is a VE motherboard component. To open this item, select it and select **Open Component** from the **Object** menu. It is a complex object with many subcomponents.
Hiding the complexity of the VE motherboard in a component simplifies the drawing and enables the VE motherboard component to be shared between different FVP models.

For example, the ClockDivider component located at the top-left of Figure 1-6 Contents of VE motherboard component on page 1-24 has a connection to an external port called masterclk.

By double-clicking a component, in this case a clock divider, you can open it to see the LISA code, and the resulting Block Diagram window displays the external ports for that subcomponent.
The clock divider component contains only external ports, and it has no subcomponents. The behavior for this component is determined by the LISA code.

A component communicates with components in the higher-level system through its self ports. Self ports refer to ports in a system that are not part of a subcomponent, and are represented by a hollow rectangle with triangles to indicate data flow, and a text label in the rectangle.

Self ports can be internal or external.

**Internal ports**

These ports communicate with subcomponents and are not visible if the component is used in a higher-level system. Unlike hidden external ports, you cannot expose internal ports outside the subcomponent. Right-click on a port and select Object Properties... to identify or create internal ports. Set the port attributes to Internal for an internal self port.

**External ports**

These ports communicate with components in a higher-level system, and by default are external.

If you use the Block Diagram editor to make a connection between an external port and a subcomponent, the LISA code uses the keyword self to indicate the standalone port:

```plaintext
self.clk_in_master => clkdiv_ref25.clk_in;
```
1.5 Data collection in Fast Models

Arm periodically collects anonymous information about the usage of our products to understand and analyze what components or features you are using, with the goal of improving our products and your experience with them. Product usage analytics contain information such as system information, settings, and usage of specific features of the product. They do not include any personal information.

Host information includes:

- Operating system name, version, and locale.
- Number of CPUs.
- Amount of physical memory.
- Screen resolution.
- Processor and GPU type.

Feature tracking information includes:

<table>
<thead>
<tr>
<th>Category</th>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Platform</td>
<td>Types</td>
<td>Tracked: Name of the platform model being run and the version and build number of Fast Models that was used to build the model. Reported: Percentage of users using the different platforms. Data type: Text. Send policy: Every invocation. Trigger points: On starting the simulation.</td>
</tr>
</tbody>
</table>

Note:

- Use the `--enable-analytics` or `--disable-analytics` command-line options to enable or disable analytics gathering for the current invocation.
- Set the `ARM_DISABLE_ANALYTICS` environment variable to a non-zero value to disable analytics gathering, or to zero to enable it, for all invocations.
- Querying the list of parameters with `--list-parameters` or the available options using `--help` does not trigger reporting.
Chapter 2
Installing Fast Models

This chapter describes the system requirements for Fast Models and how to install and uninstall Fast Models.

It contains the following sections:
• 2.1 Requirements for Fast Models on page 2-28.
• 2.2 Installing Fast Models on page 2-30.
• 2.3 Installing the TPIP package on page 2-31.
• 2.4 Dependencies for Red Hat Enterprise Linux on page 2-32.
2.1 Requirements for Fast Models

This section describes the hardware and software that you need for Fast Models.

**Platform**

**Memory**
At least 2GB RAM, preferably 4GB.

**Processor**
2GHz Intel Core2Duo, or similar, that supports the MMX, SSE, SSE2, SSE3, and SSSE3 instruction sets.

**Linux**

**Operating system**
Red Hat Enterprise Linux 6 or 7 (for 64-bit architectures), Ubuntu 14.04 *Long Term Support* (LTS), or Ubuntu 16.04 LTS.

**Shell**
A shell compatible with sh, such as bash or tcsh.

**Compiler**
GCC 4.9.2, GCC 5.4.0, GCC 6.4.0.

--- Note ---
For full compatibility, it is highly recommended that all code that links against the Fast Models is compiled with C++11 support enabled. There are no known issues when linking non-C++11 code with the Fast Models. However, the compiler does not guarantee that the ABI is the same for both types of code. Compiling models with C++11 support disabled might cause data corruption or other issues when using them.

---

The following combinations of GCC and GNU binutils were used to build Fast Models libraries:

<table>
<thead>
<tr>
<th>GCC version</th>
<th>GNU binutils version</th>
</tr>
</thead>
<tbody>
<tr>
<td>4.9.2</td>
<td>2.17.50.0.17</td>
</tr>
<tr>
<td>5.4.0</td>
<td>2.27</td>
</tr>
<tr>
<td>6.4.0</td>
<td>2.27</td>
</tr>
</tbody>
</table>

--- Note ---
Fast Models binaries were built on Red Hat Enterprise Linux 6 and therefore were built against GLIBC 2.12.

---

**PDF Reader**
Adobe does not support Adobe Reader on Linux. Arm recommends system provided equivalents, such as Evince, instead.

**License management utilities**
The latest version of the FlexNet software that is available for download from

Note

• Set up a single arm1md license server. Spreading Fast Models license features over servers can cause feature denials.
• To run arm1md and lmgrd, install these libraries:

  Red Hat
  lsb, lsb-linux

  Ubuntu
  lsb

Microsoft Windows

Operating system
  Microsoft Windows 7 64-bit RTM or Service Pack 1, Professional, or Enterprise editions,
  Microsoft Windows 10 64-bit.

Compiler
  Microsoft Visual Studio 2015 update 3, Microsoft Visual Studio 2017 version 15.9.11 or later.

PDF Reader
  Adobe Reader 8 or higher.

License management utilities
  The latest version of the FlexNet software that is available for download from

Note

• Fast Models does not support Express or Community editions of Visual Studio.
• Set up a single arm1md license server. Spreading Fast Models license features over servers can cause feature denials.
• If you use Microsoft Windows Remote Desktop (RDP) to access System Canvas (or a simulation that it generated), your license type can restrict you:
  — Floating licenses require a license server, and have no RDP restrictions. Arm issues them on purchase.
  — Node locked licenses apply to specific workstations. Existing node locked licenses and evaluation licenses do not support running the product over RDP connections. Contact license.support@arm.com for more information.

Related information
Installing floating licenses
Installing node locked licenses
2.2 Installing Fast Models

This section describes how to install Fast Models.

Procedure
1. Unpack the installation package, if necessary, and execute ./setup.sh on Linux or Setup.exe on Windows.
   If the installer finds an existing installation, it displays a dialog to enable re-installation or uninstallation.
2. On Linux, source the correct script for your shell to set up the environment variables. Ideally, include it for sourcing into the user environment on log-in:

   ```bash/sh: . <install_directory>/FastModelTools_0.0/source_all.sh
   csh: source <install_directory>/FastModelTools_0.0/source_all.csh```

3. Optionally, install the Third Party IP (TPIP) add-on package.

   _________ Note _________
   • On Microsoft Windows 7, the Fast Models examples are installed in %PVLIB_HOME%/examples. The installer makes a copy of them in %USERPROFILE%/ARM/FastModelsPortfolio_%FM-VERSION%/examples. This copy allows you to save configuration changes to these examples without requiring Administrator permissions.
   • The Fast Models installation package includes the SystemC and TLM header files and libraries that you need to build an EVS, FVP, or SVP, and sets the SYSTEMC_HOME environment variable to their location.

Related tasks
2.3 Installing the TPIP package on page 2-31
2.3 Installing the TPIP package

This section describes how to install the Third Party IP (TPIP) package.

You need the TPIP package to use:
- Dhrystone.
- Linux images.
- The GDB Remote Connection plug-in.

Procedure
1. Download and install the TPIP package. It is available from the same location as the Fast Models package.

Related information
- Accellera Systems Initiative (ASI)
- Connect IP download site
- Arm Downloads
- Fast Models examples
- GDB remote connection
2.4 Dependencies for Red Hat Enterprise Linux

This section describes the dependencies for the supported versions of Red Hat Enterprise Linux.

This section contains the following subsections:

- 2.4.1 About Red Hat Enterprise Linux dependencies on page 2-32.
- 2.4.2 Dependencies for Red Hat Enterprise Linux 6 on page 2-32.

2.4.1 About Red Hat Enterprise Linux dependencies

Some library objects or applications depend on other library files. Fast Models requires some packages that are part of Red Hat Enterprise Linux, which you might need to install.

If you subscribed your Red Hat Enterprise Linux installation to the Red Hat Network, or if you are using CentOS rather than Red Hat Enterprise Linux, you can install dependencies from the internet. Otherwise, use your installation media.

Some packages might depend on other packages. If you install with the Add/Remove software GUI tool or the `yum` command line tool, these dependencies resolve automatically. If you install packages directly using the `rpm` command, you must resolve these dependencies manually.

To display the package containing a library file on your installation, enter:

```
rpm -qf library_file
```

For example, to list the package containing `/lib/tls/libc.so.6`, enter the following on the command line:

```
rpm -qf /lib/tls/libc.so.6
```

The following output indicates that the library is in version 2.3.2-95.37 of the `glibc` package:

```
glibc-2.3.2-95.37
```

2.4.2 Dependencies for Red Hat Enterprise Linux 6

Some package dependencies for Red Hat Enterprise Linux 6 are part of a base installation.

<table>
<thead>
<tr>
<th>Package</th>
<th>Required for</th>
</tr>
</thead>
<tbody>
<tr>
<td>glibc</td>
<td>Fast Models tools and virtual platforms</td>
</tr>
<tr>
<td>glibc-devel</td>
<td>Fast Models tools</td>
</tr>
<tr>
<td>libgcc</td>
<td>Fast Models tools and virtual platforms</td>
</tr>
<tr>
<td>make</td>
<td>Fast Models tools</td>
</tr>
<tr>
<td>libstdc++</td>
<td>Fast Models tools and virtual platforms</td>
</tr>
<tr>
<td>libstdc++-devel</td>
<td>Fast Models tools</td>
</tr>
<tr>
<td>libXext</td>
<td>Fast Models tools and virtual platforms</td>
</tr>
<tr>
<td>libX11</td>
<td>Fast Models tools and virtual platforms</td>
</tr>
<tr>
<td>libXau</td>
<td>Fast Models tools and virtual platforms</td>
</tr>
<tr>
<td>libxcb</td>
<td>Fast Models tools and virtual platforms</td>
</tr>
<tr>
<td>libSM</td>
<td>Fast Models tools and virtual platforms</td>
</tr>
<tr>
<td>libICE</td>
<td>Fast Models tools and virtual platforms</td>
</tr>
<tr>
<td>libuuid</td>
<td>Fast Models tools and virtual platforms</td>
</tr>
</tbody>
</table>
Table 2-2 Dependencies for Red Hat Enterprise Linux 6 (continued)

<table>
<thead>
<tr>
<th>Package</th>
<th>Required for</th>
</tr>
</thead>
<tbody>
<tr>
<td>libXcursor</td>
<td>Fast Models tools and virtual platforms</td>
</tr>
<tr>
<td>libXfixes</td>
<td>Fast Models tools and virtual platforms</td>
</tr>
<tr>
<td>libXrender</td>
<td>Fast Models tools and virtual platforms</td>
</tr>
<tr>
<td>libXft</td>
<td>Fast Models tools and virtual platforms</td>
</tr>
<tr>
<td>libXrandr</td>
<td>Fast Models tools and virtual platforms</td>
</tr>
<tr>
<td>libXt</td>
<td>Fast Models tools and virtual platforms</td>
</tr>
<tr>
<td>alsalib</td>
<td>Fast Models virtual platforms</td>
</tr>
<tr>
<td>xterm</td>
<td>Fast Models virtual platforms</td>
</tr>
<tr>
<td>telnet</td>
<td>Fast Models virtual platforms</td>
</tr>
</tbody>
</table>
Chapter 3
System Canvas Tutorial

This chapter describes using System Canvas to build a system model.

It contains the following sections:
• 3.1 About this tutorial on page 3-35.
• 3.2 Starting System Canvas on page 3-36.
• 3.3 Creating a new project on page 3-37.
• 3.4 Add and configure components on page 3-39.
• 3.5 Connecting components on page 3-43.
• 3.6 View project properties and settings on page 3-44.
• 3.7 Changing the address mapping on page 3-47.
• 3.8 Building the system on page 3-49.
• 3.9 Building SystemC ISIM targets on page 3-50.
• 3.10 Debugging with Model Debugger on page 3-51.
3.1 About this tutorial

This tutorial describes how to perform some basic operations in System Canvas to build a standalone system model that can run an application image.

It demonstrates how to:
- Create a System Canvas project.
- Add, connect, and modify components in the project. You can use the Block Diagram view in System Canvas to do this. You do not need to edit LISA source code directly.
- Build the project.
- Debug an application on the model using Model Debugger.
3.2 Starting System Canvas

This section describes how to start the application.

To start System Canvas:

- On Linux, enter `sgcanvas` in a terminal window and press Return.
- On Microsoft Windows, open the **System Canvas** application from the **Start** menu.

The application contains the following subwindows:

- A blank diagram window on the left-hand side of the application window.
- A component window at the right-hand side.
- An output window across the bottom.

![System Canvas at startup](image)

**Figure 3-1** System Canvas at startup

**Related references**

Preferences - Applications group on page 5-89

Chapter 5 System Canvas Reference on page 5-59
3.3 Creating a new project

This section describes how to create a new project. The project will be used to create a new system model.

Procedure

1. Select **New Project** from the **File** menu. Alternatively, click the **New** button on the toolbar.

   **Results:** The **New Project** dialog appears.

   ![New Project dialog](image)

2. Navigate to the directory to use for your project. Enter **MyProject** in the filename box and click the **Select** button.

   **Results:** A dialog appears for you to enter the name and location of the LISA+ file that represents your new system.

   ![Select Top Component LISA File dialog](image)

3. Enter **My_Top_Component.lisa** in the filename box and click the **Select** button.

   The component name for the top component is, by default, set to the name of the LISA+ file.

   **Results:** The Workspace area contains a blank block diagram with scroll bars. The Component window, to the right of the Workspace area, lists the components in the default repositories.

   These steps create a project file, **MyProject.sgproj** and a LISA+ source file, **My_Top_Component.lisa**. The project file contains:
   - System components.
   - Connections between system components.
• References to the component repositories.
• Settings for model generation and compilation.

Do not edit the project file. System Canvas modifies it if you change project settings.

The block diagram view of your system is a graphical representation of the LISA+ source. To display the contents of My_Top_Component.lisa, click the Source tab. This file is automatically updated if you add or rename components in the block diagram.

You can view the LISA+ source for many of the supplied components. To do so, double-click on a component in the Block Diagram. Alternatively, right click on a component in the Components window and select Open Component.
3.4 Add and configure components

This section describes how to add and configure the components required for the example system, introducing some key features of System Canvas.

This section contains the following subsections:
• 3.4.1 Adding the Arm® processor on page 3-39.
• 3.4.2 Naming components on page 3-40.
• 3.4.3 Resizing components on page 3-40.
• 3.4.4 Hiding ports on page 3-40.
• 3.4.5 Moving ports on page 3-41.
• 3.4.6 Adding components on page 3-41.
• 3.4.7 Using port arrays on page 3-41.

3.4.1 Adding the Arm® processor

This section describes how to add an Arm processor component to the system model.

Procedure
1. Click the Block Diagram tab in the Workspace window, unless the block diagram window is already visible.
   Results: A blank window with grid points appears.
2. Select the Components tab in the Components window to display the Fast Models Repository components.
3. Move the mouse pointer over the ARMCortexA8CT processor component in the Component window and press and hold the left mouse button.
4. Drag the component to the middle of the Workspace window.
   Note
   If you move the component within the Workspace window, the component automatically snaps to the grid points.
5. Release the left mouse button when the component is in the required location.
   Results:
   The system receives the component.

6. Save the file by selecting File > Save File or using Ctrl+S.
   The asterisk (*) at the end of the system name, in the title bar, shows unsaved changes.
These steps create a System Canvas file, *My_Top_Component.sgcanvas*, in the same location as the project and LISA+ files. It contains the block diagram layout information for your system. Do not edit this file.

### 3.4.2 Naming components

This section describes how to change the name of a component, for example the processor.

--- Note ---

Component names cannot have spaces in them, and must be valid C identifiers.

---

**Procedure**

1. Select the component and click the *Properties* button on the toolbar to display the *Component Instance Properties* dialog.
   - You can also display the dialog by either:
     - Right-clicking on the component and select *Object Properties* from the context menu.
     - Selecting the component and then selecting *Object Properties* from the *Object* menu.
   2. Click the *General* tab on the *Component Instance Properties* dialog.
   3. Enter *Arm* in the *Instance name* field.
   4. Click *OK* to accept the change. The instance name of the component, that is the name displayed in the processor component title, is now *Arm*.

### 3.4.3 Resizing components

This section describes how to resize components.

**Procedure**

1. Select the processor component and move the mouse pointer over one of the green resize control boxes on the edges of the component.
2. Hold the left mouse button down and drag the pointer to resize the component.
3. Release the mouse button to end the resize operation.
   - To vertically resize the component title bar to avoid truncating text, click the component and drag the lower handle of the shaded title bar.

![Figure 3-5 Processor component after changes](image)

### 3.4.4 Hiding ports

This section describes how to hide ports, for instance because they are not connected to anything.

If there are only a few ports to hide, use the port context menu. Right click on the port and select *Hide Port*. To hide multiple ports:

**Procedure**

1. Select the component and then select *Object Properties* from the *Object* menu.
2. Click the *Ports* tab on the dialog.
3. Click *Select All* to select all of the ports.
4. Click *Hide selected ports*.
5. Select the boxes next to clk_in and pvbus_m.
6. Click OK to accept the change, so that all ports except clk_in and pvbus_m are hidden in the Block Diagram view.

**Related tasks**

3.4.7 Using port arrays on page 3-41

### 3.4.5 Moving ports

This section describes how to move ports, for example to improve readability.

**Procedure**

1. Place the mouse pointer over the port. The mouse pointer changes shape to a hand with a pointing finger. This is the move-port mouse pointer.
2. Press and hold the left mouse button down over the port, and drag the port to the new location.
   
   This can be anywhere along the inner border of the component that is not on top of an existing port. If you select an invalid position, the port returns to its original location.
3. When the port is in position, release the mouse button.

   Arrange any other ports as needed. The clk_in port must be on the left side.

### 3.4.6 Adding components

This section describes how to add components to a project.

**Procedure**

1. Drag and drop the following components onto the Block Diagram window:
   - ClockDivider.
   - MasterClock.
   - PL340_DMC.
   - PVBusDecoder.
   - RAMDevice.

   The PL340_DMC component is included to demonstrate some features of System Canvas and is not part of the final example system.
2. Select the new components individually and use the General tab of the Component Instance Properties dialog to rename them to:
   - Divider.
   - Clock.
   - PL340.
   - BusDecoder.
   - Memory.

### 3.4.7 Using port arrays

This section describes how to expand, collapse, and hide port arrays.

**Procedure**

1. Right click on one of the axi_if_in ports in the PL340 component to open a context menu. Select **Collapse Port** to reduce the port array to a single visible item in the component.
2. Select the PL340 component and then select Object Properties from the Object menu.
3. Select the Ports tab in the Component Instance Properties dialog.
   
   The axi_if_in port is a port array as indicated by the + beside the port name. Click the + to expand the port tree view.
4. Deselect the checkboxes beside axi_if_in[2] and axi_if_in[3] to hide the chosen array ports so that expanding the port array still does not display them. Click OK to close the dialog.
You can also hide a port by using the port context menu and selecting **Hide Port**.

5. To expand the `axi_if_in` port in the PL340 component, you can:
   - Right click on the port and select **Expand Port** from the port context menu.
   - 1. Display the **Component Instance Properties** dialog.
   - 2. Select the **Ports** tab.
   - 3. Click the + next to the port array to expand the port tree view.
   - 4. Select the **Show as Expanded** radio button.

   **Results:** Only the `axi_if_in[0]` and `axi_if_in[1]` ports are shown.

6. To redisplay the `axi_if_in[2]` and `axi_if_in[3]` ports, you can:
   - Use the port context menu and select **Show All Ports**.
   - Reverse the deselection step, selecting the checkboxes next to the hidden ports, in the **Component Instance Properties** dialog.

   Ports with more than eight items are shown collapsed by default.

**Next Steps**

The rest of this tutorial does not require the PL340 component, so you can delete it.

![Figure 3-6 Example system with added components](image-url)
3.5 Connecting components

This section describes how to connect components.

Procedure
1. Select connection mode, by doing either of the following:
   - Click the Connect button.
   - Select Connect Ports Mode from the Edit menu.
2. Move the mouse pointer around in the Block Diagram window:

<table>
<thead>
<tr>
<th>Option</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Not over an object</td>
<td>The pointer changes to the invalid pointer, a circle with a diagonal line through it.</td>
</tr>
<tr>
<td>Over an object</td>
<td>The pointer changes to the start connection pointer and the closest valid port is highlighted.</td>
</tr>
</tbody>
</table>
3. Move the cursor so that it is over the Clock component and close to the clk_out port.
4. Highlight the clk_out port, then press and hold the left mouse button down.
5. Move the cursor over the clk_in port of the Divider component.
6. Release the mouse button to connect the two ports.
   The application remains in connect mode after the connection is made.
7. Make the remaining connections.

![Connected components](image)

Connections between the addressable bus ports have bold lines.
3.6 View project properties and settings

Before building the model, verify the toolchain configuration and top component using the Project Settings dialog.

This section contains the following subsections:
• 3.6.1 Viewing the project settings on page 3-44.
• 3.6.2 Specifying the Active Project Configuration on page 3-45.
• 3.6.3 Selecting the top component on page 3-46.

3.6.1 Viewing the project settings

Use the Project Settings dialog to view and edit the project configuration. Although no changes are required for this tutorial, this section demonstrates the steps to use if changes were necessary.

Procedure
1. Open the Project Settings dialog to inspect the project settings for the system, by doing either of the following:
   • Click the Settings button.
   • Select Project Settings from the Project menu.

Results:
The Project Settings dialog appears:
Figure 3-8 Project settings for the example, showing the Compiler options panel

The **Category View**, **List View**, and **Tree View** tabs present different views of the project parameters.

### 3.6.2 Specifying the Active Project Configuration

Use the **Select Active Project Configuration** drop-down menu on the main toolbar to display the configuration options that control how the target model is generated.

You can choose to:

- Build models with debug support.
- Build release models that are optimized for speed.

Display and edit the full list of project settings by selecting **Project Settings** from the **Project** menu. Inspect and modify a configuration for your operating system by selecting it from the **Configuration** drop-down list and clicking the different list elements to view the settings.

**Note**

- The configuration options available, including compilers and platforms, depend on the operating system.
- Projects that were created with earlier versions of System Generator might not have the compiler version specified in the **Project Settings** dialog, but are updateable.
3.6.3 Selecting the top component

The top component defines the root component of the system. Any component can be set as the top component. This flexibility enables building models from subsystems.

In the Project Settings dialog, click the Select From List... button. The Select Top Component dialog opens and lists all the components in the system.

Note

If the value in the Type column is System, the component has subcomponents.

Figure 3-9  Select Top Component dialog showing available components
3.7 Changing the address mapping

Addressable bus mappings, connections that have bold lines, have editable address maps.

Follow this procedure to change the address mapping.

Procedure

1. Double-click the `pvbus_m_range` port of the BusDecoder component to open the Port Properties dialog.

   Results:

   ![Port Properties dialog](image)

   Figure 3-10 Viewing the address mapping from the Port Properties dialog

2. Open the Edit Connection dialog by doing either of the following:
   - Select the Memory , pvbus Slave Port line, and click **Edit Connection**....
   - Double click on the entry.

   Results:

   ![Edit Connection dialog](image)

   Figure 3-11 Edit Connection dialog

3. Select the **Enable address mapping** checkbox to activate the address text fields.

   The address mapping for the master port is shown on the left side of the Edit Connection dialog. **Start**, **End**, and **Size** are all editable. If one value changes, the other values are automatically updated if necessary. The equivalent LISA statement is displayed at the bottom of the Edit Connection dialog.

4. Enter a **Start** address of `0x00000000` and an **End** address of `0x10FFFFFF` in the active left-hand side of the Edit Connection dialog. The **Size** of `0x11000000` is automatically calculated.
This step maps the master port to the selected address range. If mapping the master port to a different address range on the slave port is required, select **Enable slave port address range**. Checking it makes the parameters for the slave port editable. The default values are the same as for the master port when the slave address range is enabled. Disabling the slave address range is equivalent to specifying the address range 0...size-1, and not the master address range. In this case, a slave port address range is not required, so deselect the **Enable slave port address range** checkbox.

**Results:**

5. Click **OK** to close the **Edit Address Mapping** dialog for the *Memory.pvbus* slave port.
6. Click **OK** to close the **Port Properties** dialog.
3.8 Building the system

This section describes how to build the model as an .so or .dll library.

**Procedure**

1. Click the **Build** icon on the System Canvas toolbar to build the model.
   
   System Canvas might perform a system check, depending on your preference setting. If warnings or errors occur, a window might open. Click **Proceed** to start the build.

**Results:**

The progress of the build is displayed in the log window.

![Figure 3-13  Build process output](image)

Depending on the speed of your computer and the type of build selected, this process might take several minutes.

You can reduce compilation time by setting the SimGen options `--num-comps-file` and `--num-build-cpus` in the **Project Settings** dialog.

**Related concepts**

6.3 Decreasing compilation time with num-comps-file and num-build-cpus on page 6-113

**Related tasks**

3.9 Building SystemC ISIM targets on page 3-50
3.9 Building SystemC ISIM targets

Build a SystemC Integrated SIMulator (ISIM) target by ticking the SystemC integrated simulator checkbox under the Targets option in the Project Settings dialog.

A SystemC ISIM target is generated by statically linking the model with the SystemC framework.

All Model Shell command-line options apply to the ISIM executable except --model. This option is not required because the model is integrated into the executable.
3.10 Debugging with Model Debugger

This section describes how to use Model Debugger to debug an application that is running on the model.

**Procedure**

1. Click the **Debug** button on the System Canvas toolbar to open the **Debug Simulation** dialog:

   ![Debug Simulation dialog](image)

   **Figure 3-15 Debug Simulation dialog**

2. Select the **CADI library** radio button to attach Model Debugger to your CADI target. The radio buttons that are available depend on the target settings.

3. Specify the location of the application that you want to run, for example `dhrystone.axf`, in the **Application** field.
   `dhrystone.axf` is part of the Third-Party IP add-on package for the Fast Models Portfolio.

4. Click **OK** to start Model Debugger.

   **Results:**

   An instance of Model Debugger starts. The debugger loads the model library from the build directory of the active configuration. Model Debugger displays the **Configure Model Parameters** dialog containing the instantiation parameters for the top-level components in the model:
To display parameter sets:

- Select a Parameter category in the left-hand side of the dialog.
- Click a + next to a component name in the right-hand side.

For different views of the system parameters, select the **List View** or **Tree View** tabs.

5. Click **OK** to close the dialog.

   **Results:**

The **Select Targets** dialog displays the components to use in Model Debugger. The Arm processor component is the default.

6. Click **OK** to close the dialog.

7. Click **Run** to start the simulation.

   **Results:**

   The **Application Input** window appears:
8. Enter the required number of runs through the benchmark in the Application input field, for instance 1000000, and click OK.

Results:

After a short pause, the benchmark results are shown in the StdIO tab of Model Debugger.

**Related references**

- Chapter 4 Debugging LISA+ components on page 4-54

**Related information**

- Model Debugger for Fast Models User Guide
Chapter 4
Debugging LISA+ components

This chapter describes how to use GDB on Linux or Microsoft Visual Studio on Microsoft Windows to debug the LISA source code of models.

It contains the following sections:
• 4.1 Building a debuggable model on page 4-55.
• 4.2 Debugging with GDB at the source level on page 4-56.
• 4.3 Debugging with Microsoft Visual Studio at the source level on page 4-57.
• 4.4 Attaching Microsoft Visual Studio to a simulator process on page 4-58.
4.1 Building a debuggable model

This section describes how to build a model with debug information.

Procedure

1. Start System Canvas and load the project for the model.
2. In the Select Active Project Configuration drop-down menu, select a valid debug configuration for the build environment.
3. To build a debug version of the model, click the Build button.

The generated simulation code is annotated with line redirections to the corresponding LISA+ code. This feature enables the debugging of a LISA+ model with GDB or Microsoft Visual Studio.

You can start the model:

• Directly, in the debugger. This method is easier for an ISIM system. If the implementation of main() is a new one, this method is especially useful.
• Indirectly, dynamically connecting to a running simulation process.
4.2 Debugging with GDB at the source level

This section describes how to debug at the source level with GNU Debugger on Linux with System Canvas.

Prerequisites

Use GDB version 6.2 or higher.

Procedure

1. Start a GDB session with Model Debugger as the executable by typing `gdb modeldebugger` in the console.
2. Type `run` at the GDB prompt to start Model Debugger.
   
   You can load the model in the same step by adding the name of the model library, for example:

   ```
   run cadi_system_Linux-Debug-GCC-5.4.so
   ```

   **Results:** Model Debugger starts and the Configure Model Parameters dialog opens.

3. Configure any required parameters for the target, then click OK.
   **Results:** The Select Targets dialog opens. The Arm processor is selected by default.

4. Select any additional targets to load. Click OK to close the dialog.
   **Results:** Model Debugger creates a separate window for each target.

5. Load the applications to the targets that execute software:
   - For a single target, Model Debugger displays a dialog prompting for the application.
   - For more than one target, load the application to the targets in each debug view separately by selecting Load Application from the File menu.
   - Alternatively, select the application when Model Debugger starts by adding command-line option `-a` in line with the name of the target and application file:

   ```
   run cadi_system_Linux-Debug-GCC-5.4.so -a targetName1=application1 -a targetName2=application2
   ```

   GDB is now ready to debug the model source code.

This section contains the following subsection:
- 4.2.1 GDB LISA debug session on page 4-56.

4.2.1 GDB LISA debug session

This section describes how to set a breakpoint in the LISA source, and run to it.

1. In GDB, interrupt execution of Model Debugger by pressing Ctrl+C.
2. You can now type commands at the GDB prompt. To set a breakpoint in GDB, for example at line 123 in the MyCode.lisa file, enter:

   ```
   break MyCode.lisa:123
   ```

3. In GDB, continue execution of Model Debugger by typing:

   ```
   continue
   ```

4. In Model Debugger, click the Run button to start execution.

You can use GDB to perform any debug action on the LISA code such as printing values of variables or stepping through code.
4.3 Debugging with Microsoft Visual Studio at the source level

This section describes how to debug at the source level with Microsoft Visual Studio on Microsoft Windows and System Canvas.

Procedure

1. Start System Canvas and load the project for your model.
2. If you have not already done so, build a Debug version of your system by selecting one of the Win32-Debug-VC20xx or Win64-Debug-VC20xx options in the Select Active Project Configuration menu, and rebuilding your model.
   If you select Generate, you can use Microsoft Visual Studio later to build the model.
3. Launch Microsoft Visual Studio from System Canvas by clicking the Devenv button or pressing Alt + F5.
4. From Microsoft Visual Studio, select StartUp project to select the target to debug. The target file can be either an ISIM executable or CADI dll.
5. Use the Project Properties dialog to configure any required parameters for the target.
   For ISIM systems, all Model Shell command lines options are available, but specifying the model is unnecessary. For a CADI dll, you must specify either Model Debugger or Model Shell as the executable to run.

This section contains the following subsection:
• 4.3.1 Microsoft Visual Studio LISA debug session on page 4-57.

4.3.1 Microsoft Visual Studio LISA debug session

This section describes how to set a breakpoint in the LISA source, and run to it.

1. In Microsoft Visual Studio, select File > Open > File... to open the LISA source file to debug.
2. Set a breakpoint in the LISA source by double clicking on a source line.
3. In Model Debugger, click the Run button to start execution. When the breakpoint is reached, use Microsoft Visual Studio to perform host-level debugging of the LISA+ source code.
4.4 Attaching Microsoft Visual Studio to a simulator process

This section describes how to debug at the host level with Microsoft Visual Studio on Microsoft Windows and System Canvas.

To perform host-level debugging, attach Microsoft Visual Studio to a running Model Debugger process.

Procedure

1. Start System Canvas and load the project for your model.
2. Select a Win64-Debug-VC20XX option in the Select Active Project Configuration menu, and build your model.
3. Click the Debug button or press F5 to launch Model Debugger from System Canvas.
   Results: Model Debugger starts and the Configure Model Parameters dialog appears.
4. Configure any appropriate parameters for the target and click OK.
   Results: The Select Targets dialog opens. The Arm processor is selected by default.
5. Select any additional targets to load. Click OK to close the dialog.
   Results: Model Debugger creates a separate window for each target.
6. Load the applications to the targets that execute software:
   - For a single target, Model Debugger displays a dialog prompting for the application.
   - For more than one target, load the application to the targets in each debug view separately by selecting Load Application from the File menu.
7. Start Microsoft Visual Studio.
8. Select Tools > Attach to Process... . Select the ModelDebugger.exe process in the Attach to Process dialog. Click the Attach button.
   It might be necessary to close some dialogs to proceed.
   Results: Microsoft Visual Studio can now control the entire host level simulation.
Chapter 5
System Canvas Reference

This chapter describes the windows, menus, dialogs, and controls in System Canvas.

It contains the following sections:
• 5.1 Launching System Canvas on page 5-60.
• 5.2 System Canvas GUI on page 5-61.
• 5.3 System Canvas dialogs on page 5-75.
5.1 Launching System Canvas

Start System Canvas from the Microsoft Windows Start menu or from the command line on all supported platforms.

To start System Canvas from the command line, at the prompt type `sgcanvas`.

<table>
<thead>
<tr>
<th>Short form</th>
<th>Long form</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>-h</td>
<td>--help</td>
<td>Print help text and exit.</td>
</tr>
<tr>
<td>-v</td>
<td>--version</td>
<td>Print version and exit.</td>
</tr>
</tbody>
</table>
5.2 System Canvas GUI

This section describes System Canvas, the GUI to the Fast Models tools, which shows the components in a system, component ports, external ports (if the system itself is a component), and connections between ports.

This section contains the following subsections:
- 5.2.1 Application window on page 5-61.
- 5.2.2 Menu bar on page 5-62.
- 5.2.3 Toolbar on page 5-68.
- 5.2.4 Workspace window on page 5-70.
- 5.2.5 Component window on page 5-73.
- 5.2.6 Output window on page 5-74.

5.2.1 Application window

The main window of System Canvas contains several windows and various graphical elements.

![Figure 5-1 Layout of System Canvas](image)

**Main menu**

The available options with their corresponding keyboard shortcuts.

**Toolbar**

Buttons for frequently-used features.

**Workspace**

Tabs to select the views:

- **Block Diagram**
  The components, ports, and connections.

- **Source**
  The LISA code of the component.

You can edit every part of the system using these views.
Component list
All of the components and their protocols and libraries in the current project.

Output window
Displays status messages that are output from the build process.

Status bar
Displays information about menu items, commands, buttons, and component information.
The block diagram editor creates graphical representations of systems. It provides a rapid way to create and configure components or systems consisting of multiple components.
You can add new components to a single project or to a component repository for use in multiple projects. The Language for Instruction Set Architectures+ (LISA+) describes the components.

5.2.2 Menu bar
The main bar provides access to System Canvas functions and commands.

File menu
The File menu lists file and project operations.

New Project
Create a new model project.

Load Project
Open an existing project.

Close Project
Close a project. If there are pending changes, the Save changes dialog appears.

Save Project
Save the changes made to a project.

Save Project As
Save a project to a new location and name.

New File
Create a new file. The New File dialog appears. Select the type from the File type drop-down list.

Open File
This displays the Open File dialog. Filter the types to display by selecting the type from the File type drop-down list. Non-LISA files open as text in the source editor.

Close File
Close a LISA file. A dialog prompts to save any changes.

Save File
Save the changes made to the current LISA file.

Save File As
Save a LISA file to a new location and name.

Save All
Save the changes made to the project and the LISA files.
Print
Print the contents of the Block Diagram window.

Preferences
Modify the user preferences.

Recently Opened Files
Display the 16 most recently opened LISA files. Click on a list entry to open the file.
To remove a file from the list, move the mouse cursor over the filename and press the Delete key or right click and select Remove from list from the context menu.

Recently Opened Projects
Display the 16 most recently opened projects. Click on a list entry to open the project.
To remove a project from the list, move the mouse cursor over the project name and press the Delete key or right click and select Remove from list from the context menu.

Exit
Close System Canvas. A dialog prompts to save any changes. Disable it by selecting Do not show this message again. Re-enable it in the preferences.

Related references
5.3.13 New project dialogs on page 5-87
Preferences - Suppressed messages group on page 5-93

Edit menu
The Edit menu lists content operations.

Undo
Undo up to 42 of the latest changes to a file in the Source view or to the layout in the Block Diagram view. These actions are undoable:
• Add an object such as a component, label, or connection.
• Paste or duplicate.
• Cut or delete.
• Edit object properties.
• Move.
• Resize.

Note
Undo and Redo operations can affect Block Diagram view zoom and scroll actions.

Undo and Redo typically work normally. For example:
1. Change the system in the Block Diagram view by adding a RAMDevice component with name RAM.
2. Switch to Source view. The text RAM : RAMDevice(); is present in the composition section.
3. Change the code by removing the line RAM : RAMDevice();.
4. Change the code by adding, for example, the line PVS : PVBusSlave();.
5. Click on the Block Diagram tab. The change to the source code is reflected by the RAM component being replaced by the PVS component.
6. Select Undo from the Edit menu. The Block Diagram view shows that RAM is present but PVS is not.
7. Select Redo from the Edit menu. The Block Diagram view shows that PVS is present but RAM is not.
Redo

Redo the last undone change. This cancels the result of selecting Undo. Selecting Redo multiple times cancels multiple Undo actions.

Cut

Cut the marked element into the copy buffer.

Copy

Copy the marked element into the copy buffer.

Paste

Paste the content of the copy buffer at the current cursor position.

Duplicate

Duplicate the marked content.

Delete

Delete the marked element.

Select All

Select all elements.

Edit Mode

Change the Workspace to Edit mode. The cursor can select components.

Connect Ports Mode

Select Connection mode. The cursor can connect components.

Pan Mode

Select Movement mode. The cursor can move the entire system in the Workspace window.

Search menu

The Search menu lists find, replace and go to functions.

Find

Search for a string in the active window (with a thick black frame).

Find Next

Repeat the last search.

Find Previous

Repeat the last search, backwards in the document.

Replace

In the Source view, search for and replace strings in a text document.

Go To Line

In the Source view, specify a line number in the currently open LISA file to go to.

Note

Use the search icons at the top right of the application window to search for text. Entering text in the search box starts an incremental search in the active window.

Related references

5.3.10 Find and Replace dialogs on page 5-85
View menu
The View menu lists the Workspace window display options.

Show Grid
Using the grid simplifies component alignment.

Zoom In
Show more detail.

Zoom Out
Show more of the system.

Zoom 100%
Change the magnification to the default.

Zoom Fit
Fit the entire system into the canvas area.

Zoom Fit Selection
Fit the selected portion into the canvas area.

Object menu
The Object menu lists system and system component operations.

Open Component
Open the source for the selected component.

Add Component
Display all of the components available for adding to the block diagram.

Add Label
The mouse cursor becomes a default label. To add the label, move it to the required location in the Block Diagram window and click the left mouse button. The Label Properties dialog appears.

Add Port
Display the External Port dialog. Specify the type of port to add.

Mirror Self Port
Switch the direction that the external port image points in. It does not reverse the signal direction, so a master port remains a master port. If an unconnected port is not selected, this option is disabled.

Expand Port
For a port array, display all of the individual port elements. Expanded is the default for port arrays with eight or fewer ports. Collapsed is the default for port arrays with more than eight elements.

Note
Ports with many elements might expand so that elements appear on top of one another. Either: click and drag them apart, or collapse the port, increase the component size, then expand the port again.

Collapse Port
For a port array, hide the individual port elements and only display the top-level port name.
Hide Port
   Disable the selected port and make it invisible.

Hide All Unconnected Ports
   Hide all ports that are not connected to a component.

Show/Hide Ports of Protocol Types...
   Hide all ports that use a specified protocol. The Show/Hide Connection Types dialog appears. Select the protocols to filter.

Show All Ports
   Show all ports. Some might overlap if there is not enough space.

Autoroute Connection
   Redraw the selected connection.

Autoroute All Connections
   Redraw all of the connections.

Documentation
   Open the documentation for the selected component.

Object Properties
   Display the Component Instance Properties dialog to view and edit the properties for the selected component.

Project menu
The Project menu lists build, check, configure, run, and set options.

Check System
   Check for errors or missing information. This feature does not check everything, but does give useful feedback.

Generate System
   Generate the C++ source code, but do not compile it. After generation, click Build System and Debug to run the model.

Build System
   Generate and compile the generated C++ source code, producing a library or a runnable model.

Stop Build
   Cancel the active build process.

Clean
   Delete all generated files.

Launch Model Debugger
   Execute the simulation under the control of Model Debugger.

Run
   Run...
      Open the Run dialog to specify the run command.
Run in Model Shell
Execute the simulation under the control of Model Shell with command-line options taken from project settings and user preferences.

Run ISIM system
Execute the simulation as an ISIM executable with Model Shell command-line options taken from project settings and user preferences.

Clear History
Clear all recent run command entries.

Recent Command Entries (up to 10)
Call recent command entries.

Kill Running Command
Stop the running synchronous command.

Launch Host Debugger

Microsoft Windows
Launch Microsoft Visual Studio. Build the system there, and start a debug session.

Note
You can take the command-line arguments for ISIM systems or Model Shell from Microsoft Visual Studio by selecting Project > Properties > Configuration Properties > debugging.

Linux
Launch the executable or script set in the application preferences. The target must be an ISIM executable. Arm recommends this method for debugging at source-level.

Add Files
Add files to the system.

Add Current File
Add the currently open file to the system.

Refresh Component List
Update the Component List window to show all available components.

Setup Default Repository
Display the Default Model Repository section of the Preferences window, and select the default repositories for the next new project.

Note
This does not affect the currently open project.

Set Top Level Component
Displays the Select Top Component dialog that lists all available components in the system.
The top component defines the root component of the system. It can be any component. This enables building of models from subsystems.

Note
If the value in the Type column is System, the component has subcomponents.
Active Configuration
Select the system build configuration from the project file list.

Project Settings
Display the Project Settings dialog.

Related tasks
2.3 Installing the TPIP package on page 2-31

Related references
Preferences - Applications group on page 5-89

Help menu
The Help menu lists documentation, software and system information links.

Fast Model Tools User Guide
Display the Fast Models User Guide.

Model Shell Reference Manual

LISA+ Language Reference Manual

AMBA-PV Developer Guide
Display the AMBA-PV Extensions to TLM 2.0 Developer Guide.

CADI Developer Guide
Display the Component Architecture Debug Interface v2.0 Developer Guide.

Release Notes
Display this document.

Documents in $PVLIB_HOME/Docs
List the PDF files in the directory $PVLIB_HOME/Docs. The location syntax is the same on Microsoft Windows and Linux. The Fast Models Portfolio installation sets the PVLIB_HOME environment variable.

End User License Agreement (EULA)
Display the license agreement.

About
Display the version and license information.

System Information
Display information about the tools and loaded models.

5.2.3 Toolbar
The toolbar sets out frequently used menu functions.

New
Create a new project or LISA file.

Open
Open an existing project or file.
Save
Save current changes to the file.

All
Save project and all open files.

Undo
Undo the last change in the Source or Block Diagram view.

Redo
Undo the last undo.

Properties
Display the Properties dialog for the selected object:

Nothing
The Component Model Properties dialog, with the properties for the top-level component.

Component
The Component Instance Properties dialog.

Connection
The Connection Properties dialog.

Port
The Port Properties dialog.

Self port
The Self Port Properties dialog.

Label
The Label Properties dialog.

Note
The Properties button only displays properties for items in the block diagram.

Settings
Display the project settings.

Select Active Project Configuration
Select the build target for the project.

Refresh
Refresh the component and protocol lists.

Check
Perform a basic model error and consistency check.

Build
Generate a virtual system model using the project settings.

Stop
Stop the current generation process.

Clean
Delete all generated files.
Debug
Start Model Debugger to debug the generated simulator.

Run
Execute the most recent run command. The down arrow next to the button opens the Run dialog.

Kill
Stop Model Shell and end the simulation.

Devenv
Open the project in the compiler. For Microsoft Windows, Microsoft Visual Studio opens. For Linux, GDB opens.

Note
Set the project and debugger with the Select Active Project Configuration drop-down menu. Generate the project solution before using this button. Without a solution, the compiler opens without a project.

Edit
Edit mode: the cursor selects and moves components.

Connect
Connection mode: the cursor connects components.

Pan
Movement mode: the cursor moves the entire system in the Workspace window.

Zoom
Use the In, Out, 100%, and Fit buttons to change the system view zoom factor in the Workspace window.

Related tasks
3.6.1 Viewing the project settings on page 3-44

Related references
Edit menu on page 5-63
5.3.4 Component Instance Properties dialog on page 5-77
5.3.5 Component Model Properties dialog for the system on page 5-79
5.3.7 Connection Properties dialog on page 5-82
5.3.11 Label Properties dialog on page 5-85
5.3.12 New File dialog (File menu) on page 5-86
5.3.14 Open File dialog on page 5-87
5.3.15 Port Properties dialog on page 5-88
5.3.20 Self Port dialog on page 5-106

5.2.4 Workspace window
This section describes the Workspace window, which displays editable representations of the system.

Source view
The Source view displays the LISA source code of components. It can also display other text files.
The source text editor features:

• Similar operation to common Microsoft Windows text editors.
• Standard copy and paste operations on selected text, including with an external text editor.
• Undo/redo operations. Text changes can be undone by using Ctrl-Z or Edit > Undo. Repeat text changes with Ctrl-Y or Edit > Redo.
• Syntax highlighting for LISA, C++, HTML, Makefiles, project (*.sgproj) and repository (*.sgrepo) files.
• Auto-indenting and brace matching. Indenting uses four spaces not single tab characters.
• Auto-completion for LISA source. If you type a delimiter such as “.” or “:”, a list box with appropriate components, ports, or behaviors appears. Icons indicate master and slave ports.
• Call hint functionality. If you type a delimiter such as “(”, a tooltip appears with either a component constructor or behavior prototype, depending on the context. Enable call hints by enabling tooltips in the Appearance pane of the Preferences dialog.

Note

Every time System Canvas parses a LISA file, it updates lexical information for auto-completion and call hint functionality. This occurs, for example, when switching between the views.

Source view context menu

The Source view context menu lists text operations.

**Undo**

Undo the last change.

**Redo**

Undo the last undo.

**Cut**

Cut the selected text.

**Copy**

Copy the selected text.

**Paste**

Paste text from the global clipboard.

**Delete**

Delete the selected text.

**Select All**

Selects all of the text in the window.

Block Diagram view

The Block Diagram view displays a graphical representation of components. It enables the addition of components, connections, ports and labels to the system.

This view supports copy and paste operations on selected components, connections, labels, and self ports:

• Use the cursor to draw a bounding rectangle around the box.
• Press and hold shift while clicking on the components to copy.

Copied components will have different names. To copy connections, select both ends of the connection.
—— Note ——
Changes made in one view immediately affect the other view.

Open files have a named workspace tab at the top of the Workspace window. An asterisk after the name indicates unsaved changes. A question mark means that the file is not part of the project.

Click the right mouse button in the workspace to open the context menu for the view.

Displaying the block diagram fails if:
• The file is not a LISA file.
• The syntax of the LISA file is incorrect.
• The LISA file contains more than one component.
• The LISA file contains a protocol.

Block Diagram view context menu
The Block Diagram view context menu lists object operations.

Open Component
Open a new workspace tab for the selected component.

Delete
Delete the object under the mouse pointer.

Add Port...
Add a port to the component.

Mirror Self Port
Mirror the port image.

Expand Port
For a port array, display all of the individual port elements.

Collapse Port
For a port array, hide the individual port elements.

Hide Port
Disable the selected port and make it invisible.

Hide All Unconnected Ports
Hide all ports that are not connected to a component.

Show/Hide Ports of Protocol Types...
Hide all ports that use a specified protocol.

Show All Ports
Show all ports of the component.

Autoroute connection
Redraw the selected connection.

Documentation
Open the documentation for the selected component.

Object Properties
Open the object properties dialog.


**Related references**

5.3.14 Open File dialog on page 5-87
5.3.16 Preferences dialog on page 5-89

### 5.2.5 Component window

This section describes the **Component** window, which lists the available components and their protocols and libraries.

#### Component window views

The **Component** window has view tabs.

**Components**

The components, and their version numbers, types, and file locations. Drag and drop to place in the block diagram. Double click to open in the workspace.

**Protocols**

The protocols of these components, and their file locations. Double click to open in the workspace.

**Files**

The project files, in a fully expanded file tree with the project file as the root. Double click to open in the workspace. The project file can contain LISA files and component repositories. A repository can itself contain a repository.

---

**Note**

The order of file processing is from the top to the bottom. To move objects:

- Select and use **Up** and **Down** in the context menu, or use **Alt + Arrow Up** or **Alt + Arrow Down**.
- Drag and drop.

---

#### Component window context menu

The **Component** window context menu lists file operations and a documentation link.

**Open**

Open the associated file.

**Add...**

Add a repository, component or protocol file, or a library.

**Add New...**

Add a new file.

**Add Directory...**

Add an include path to be used by the compiler (**Files** tab only). To simplify navigation, the add dialog also shows the filename.

**Remove**

Remove an item.

**Up**

Move a file up the file list (**Files** tab only).

**Down**

Move a file down the file list (**Files** tab only).
Reload
Reload a component or protocol.

Refresh Component List
Refresh the entire component list.

Documentation
Open the documentation for the component.

Properties
Show the properties of the item.

5.2.6 Output window
The Output window displays the build or script command output.
The left side of the window has controls:
First
Go to the first message.

Previous
Go to the previous message.

Stop
Do not scroll automatically.

Next
Go to the next message.

Last
Go to the last message.
The right side of the window has controls:

Scroll bar
Move up and down in the output.

Stick
Force the window to show the latest output, at the bottom.
5.3 System Canvas dialogs

This section describes the dialog boxes of System Canvas.

This section contains the following subsections:

• 5.3.1 Add Existing Files and Add New File dialogs (Component window) on page 5-75.
• 5.3.2 Add Files dialog (Project menu) on page 5-76.
• 5.3.3 Add Connection dialog on page 5-77.
• 5.3.4 Component Instance Properties dialog on page 5-77.
• 5.3.5 Component Model Properties dialog for the system on page 5-79.
• 5.3.6 Component Properties dialog for a library component on page 5-81.
• 5.3.7 Connection Properties dialog on page 5-82.
• 5.3.8 Edit Connection dialog on page 5-83.
• 5.3.9 File/Path Properties dialog on page 5-83.
• 5.3.10 Find and Replace dialogs on page 5-85.
• 5.3.11 Label Properties dialog on page 5-85.
• 5.3.12 New File dialog (File menu) on page 5-86.
• 5.3.13 New project dialogs on page 5-87.
• 5.3.14 Open File dialog on page 5-87.
• 5.3.15 Port Properties dialog on page 5-88.
• 5.3.16 Preferences dialog on page 5-89.
• 5.3.17 Project Settings dialog on page 5-93.
• 5.3.18 Protocol Properties dialog on page 5-104.
• 5.3.19 Run dialog on page 5-105.
• 5.3.20 Self Port dialog on page 5-106.

5.3.1 Add Existing Files and Add New File dialogs (Component window)

This section describes these dialogs that add components, protocols, libraries, repositories, or source code to a project.

Displaying the Add Existing Files and Add New File dialogs (Component window)

This section describes how to display dialogs that add components, protocols, libraries, repositories, or source code to a project.

Procedure

1. Display a dialog by right-clicking in the Component window and selecting from the context menu:
   • Add.
   • Add New.

Using the Add Existing Files and Add New File dialogs (Component window)

This section describes how to add a file using the Component window context menu.

Procedure

1. Select the Components, Protocols, or Files tab in the Component window.
   To add a file at the top level of the file list, select the top entry. To add a file to an existing repository in the file list, select the repository.
2. Right-click in the Component window and select Add or Add New from the context menu.

<table>
<thead>
<tr>
<th>Option</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Add</td>
<td>In the Add Existing Files dialog, go to the file and select it.</td>
</tr>
<tr>
<td>Add New</td>
<td>In the Add New File dialog, go to the directory to contain the file and enter the name.</td>
</tr>
</tbody>
</table>

Save time with the Recently selected files drop-down list. To remove a file, mouse over it and press Delete, or right-click and select Remove from list from the context menu.
3. Click **Open** to add the file and close the dialog.

**Next Steps**

Library files, those with `.lib` or `.a` extensions, need build actions and a platform.

**Related references**

5.3.9 File/Path Properties dialog on page 5-83

**Using environment variables in filepaths**

Environment variables in filepaths enable switching to new repository versions without modifying the project.

For example, using `$(PVLIB_HOME)/etc/sglib.sgrepo` as the reference to the components of the Fast Models Portfolio enables migration to future versions of the library by modifying environment variable `PVLIB_HOME`.

**Note**

On Microsoft Windows, Unix syntax is valid for environment variables and paths, for example `$PVLIB_HOME/etc/my.sgrepo`.

Edit a filepath through the **File Properties** dialog:

**Procedure**

1. Select the file and click select **Properties** from the context menu.
2. Edit the **File** entry to modify the filepath.

**Related references**

5.3.9 File/Path Properties dialog on page 5-83

**Assigning platforms and compilers for libraries**

This section describes how to set the operating system that a library is for, and the compiler that built it.

**Procedure**

1. Use the **File Properties** dialog to specify the operating system and compilers by checking the appropriate boxes in the **Supported platforms** pane.

   Microsoft Visual Studio distinguishes between debug and release versions.

**Related references**

5.3.9 File/Path Properties dialog on page 5-83

5.3.17 Project Settings dialog on page 5-93

5.3.2 Add Files dialog (Project menu) on page 5-76

5.3.12 New File dialog (File menu) on page 5-86

**5.3.2 Add Files dialog (Project menu)**

Add files to a project with this dialog.

Select **Add File** from the **Project** menu to add a new file to the project.

The behavior of this dialog is identical to that of the **Add Existing Files** dialog.
To create a new file from code in the Source view, select Add Current File from the Projects menu to add the file to the project. No dialog appears.

---Note---

Save time with the Recently selected files drop-down list. To remove a file, mouse over it and press Delete, or right-click and select Remove from list from the context menu.

Related references
5.3.1 Add Existing Files and Add New File dialogs (Component window) on page 5-75
5.3.9 File/Path Properties dialog on page 5-83

5.3.3 Add Connection dialog

This dialog adds a connection to a component port.

To open the dialog:

1. Select a component port.
2. Display the Port Properties dialog by selecting Object Properties from the context menu or from the Object menu.
3. Click the Add Connection button.

The enabled fields for the dialog depend on whether a slave or master was displayed in the Port Properties dialog.

---Note---

This dialog also appears if you use the cursor in connect mode to connect two ports in the block diagram and one or more of the ports is a port array.

Related references
5.3.8 Edit Connection dialog on page 5-83

5.3.4 Component Instance Properties dialog

This dialog displays the properties of a component.

To open the dialog, select a component in the block diagram, and click on the Properties button in the toolbar or select Object Properties from the Object menu.

**General**

The component name, instance name, filename and path, and repository.

The Instance name field is editable.

---Note---

To view the properties of the top-level component, double-click in an area of the workspace that does not contain a component.

**Properties**

All properties for the component. If the properties are not editable, the tab says Properties (read only).

If the property is a Boolean variable, a checkbox appears next to it.

**Parameters**

All editable parameters for this component. Enter a new value in the Value edit box.

The following controls are present:
### Parameter name

The parameters for this component.

### Value

Select a parameter and then click the text box in the Value column to set the default value for the parameter.

Integer parameters in decimal format can contain binary multiplication suffixes. These left-shift the bits in parameter value by the corresponding power of two.

<table>
<thead>
<tr>
<th>Suffix</th>
<th>Name</th>
<th>Multiplier</th>
</tr>
</thead>
<tbody>
<tr>
<td>K</td>
<td>Kilo</td>
<td>$2^{10}$</td>
</tr>
<tr>
<td>M</td>
<td>Mega</td>
<td>$2^{20}$</td>
</tr>
<tr>
<td>G</td>
<td>Giga</td>
<td>$2^{30}$</td>
</tr>
<tr>
<td>T</td>
<td>Tera</td>
<td>$2^{40}$</td>
</tr>
<tr>
<td>P</td>
<td>Peta</td>
<td>$2^{50}$</td>
</tr>
</tbody>
</table>

### Ports

All the ports in the component.

For port arrays, display all of the individual ports or only the port array name by selecting Show as Expanded or Collapsed.

The properties of individual ports are editable:

1. Select a port from the list.
2. Click Edit and change the properties of the port.
3. Click OK to save the changes.

______ Note _______

If you click OK, the changes apply immediately.

Enable/disable individual ports with the checkboxes:

- Click Show selected ports to display the checked ports.
- Click Hide selected ports to hide the checked ports.

______ Note _______

Hiding the top level of a port array hides all of the individual ports but they retain their check mark setting.

### Methods

All the behaviors (component functions) that the component implements.

### Related references

- 5.3.5 Component Model Properties dialog for the system on page 5-79
- 5.3.6 Component Properties dialog for a library component on page 5-81
- 5.3.11 Label Properties dialog on page 5-85
- 5.3.15 Port Properties dialog on page 5-88
- 5.3.18 Protocol Properties dialog on page 5-104
- 5.3.20 Self Port dialog on page 5-106
5.3.5 Component Model Properties dialog for the system

This dialog displays the properties for the system.

To open the dialog, select a blank area in the block diagram, right-click and select **Object Properties** from the context menu to display the properties for the system or select **Object Properties** from the **Object** menu.

**General**

- The system name, filename and path, and repository.
- The **Component name** field is editable.
**Properties**

If the property is a Boolean variable, a checkbox appears next to it. Changes in these dialogs alter the LISA code in the model. Double-click in the **Value** column to change the property.

<table>
<thead>
<tr>
<th>Property</th>
<th>ID</th>
<th>Default</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Component name</td>
<td>component_name</td>
<td>&quot;&quot;</td>
<td>A string containing the name for the component.</td>
</tr>
<tr>
<td>Component category</td>
<td>component_type</td>
<td>&quot;&quot;</td>
<td>A string describing the type of component. This can be &quot;Processor&quot;, &quot;Bus&quot;, &quot;Memory&quot;, &quot;System&quot;, or any free-form category text.</td>
</tr>
<tr>
<td>Component description</td>
<td>description</td>
<td>&quot;&quot;</td>
<td>A textual component description.</td>
</tr>
<tr>
<td>Component documentation</td>
<td>documentation_file</td>
<td>&quot;&quot;</td>
<td>A filepath or an HTTP link to documentation. Supported file formats are PDF, TXT, and HTML.</td>
</tr>
<tr>
<td>Executes software</td>
<td>executes_software</td>
<td>0</td>
<td>The component executes software and can load application files. 1 for processor-like components, 0 for other components.</td>
</tr>
<tr>
<td>Hidden</td>
<td>hidden</td>
<td>0</td>
<td>1 for components hidden from the <strong>Component</strong> window. Otherwise, hidden components behave exactly as normal components, and they do appear in the <strong>Workspace</strong> window.</td>
</tr>
<tr>
<td>Has CADI interface</td>
<td>has_cadi</td>
<td>1</td>
<td>1 for components with a CADI interface, permitting connection to the target with a CADI-compliant debugger. 0 for components with no CADI interface.</td>
</tr>
<tr>
<td>Icon pixmap file</td>
<td>icon_file</td>
<td>&quot;&quot;</td>
<td>The XPM file that contains the system icon.</td>
</tr>
<tr>
<td>License feature</td>
<td>license_feature</td>
<td>&quot;&quot;</td>
<td>The license feature string required to run this system model. Example: &quot;.elf&quot; or &quot;.hex&quot;.</td>
</tr>
<tr>
<td>Load file extension</td>
<td>loadfile_extension</td>
<td>&quot;&quot;</td>
<td>The application filename extension for this target. Example: &quot;.elf&quot; or &quot;.hex&quot;.</td>
</tr>
</tbody>
</table>
Table 5-3  Component properties (continued)

<table>
<thead>
<tr>
<th>Property</th>
<th>ID</th>
<th>Default</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Small icon pixmap file</td>
<td>small_icon_file</td>
<td>&quot;&quot;</td>
<td>The XPM file that contains the 12x12 pixel system icon.</td>
</tr>
<tr>
<td>Component version</td>
<td>version</td>
<td>&quot;1.0&quot;</td>
<td>The version of the component.</td>
</tr>
</tbody>
</table>

Parameters

Parameter name

The parameters for this component.

Value

Select a parameter and then click the text box in the Value column to set the default value. Integer parameters in decimal format can contain binary multiplication suffixes. These left-shift the bits in parameter value by the corresponding power of two.

Table 5-4  Suffixes for parameter values

<table>
<thead>
<tr>
<th>Suffix</th>
<th>Name</th>
<th>Multiplier</th>
</tr>
</thead>
<tbody>
<tr>
<td>K</td>
<td>Kilo</td>
<td>$2^{10}$</td>
</tr>
<tr>
<td>M</td>
<td>Mega</td>
<td>$2^{20}$</td>
</tr>
<tr>
<td>G</td>
<td>Giga</td>
<td>$2^{30}$</td>
</tr>
<tr>
<td>T</td>
<td>Tera</td>
<td>$2^{40}$</td>
</tr>
<tr>
<td>P</td>
<td>Peta</td>
<td>$2^{50}$</td>
</tr>
</tbody>
</table>

Parameter ID in LISA code

The LISA ID for the component parameters.

Add

Click to add a new parameter.

Edit

Select a parameter and then click to change the name.

Delete

Select a parameter and then click to delete it.

Ports

All external ports.

If a port contains an array of ports, the Size column displays the number of ports in the array.

Enable/disable individual ports with the checkboxes:

- Click Show selected ports to display the checked ports.
- Click Hide selected ports to hide the checked ports.

Methods

The available LISA prototypes. The list is for reference only. It is not editable.

5.3.6  Component Properties dialog for a library component

This dialog displays the properties of a library component.
To open the dialog, select a component from the **Components** list, and right-click and select **Properties** from the context menu or select **Object Properties** from the **Object** menu.

**General**

**Component name**

The name of the component.

**Type**

The component category, for example Core or Peripheral.

**Version**

The revision number for the component.

**File**

The file that defines the component.

**Repository**

The repository that contains the component.

**Description**

Information about the component.

**Properties (read only)**

All the usable properties of the component.

**Note**

A valid license_feature string allows this component to work in a model.

**Parameters (read only)**

All the parameters for the component.

**Ports (read only)**

All the ports in the component.

**Note**

No port arrays are expandable here.

**Methods**

The LISA prototypes of the methods, that is, behaviors, of the component. The list is for reference only. It is not editable.

### 5.3.7 Connection Properties dialog

This dialog displays port connection properties.

To open the dialog, double click on a connection between components in the workspace.

**Name**

The name of the port.

**Type**

The type of port and the protocol.

To change the address mapping, click **Master Port Properties** or **Slave Port Properties**.

**Related references**

5.3.15 Port Properties dialog on page 5-88
5.3.8 Edit Connection dialog

This dialog controls port connection properties.

To open the dialog and change the connected port or the address mapping, select a connection from the Port Properties dialog and click Edit Connection....

Component
For a slave port, the source component is editable. For a master port, the destination component is editable.

Port
For a slave port, the master port is editable. For a master port, the slave port is editable.

Array index
For port arrays, an index value for the element to use.

Enable address mapping
Set the port address range with the Start and End boxes.

Start
The start address for the port.

End
The end address for the port.

Size
The size of the address region. Given the Start and End values, System Canvas calculates this value.

OK/Cancel
Click OK to modify the connection. Click Cancel to close the dialog without changing the connection.

LISA statement
The code equivalent to the address range.

5.3.9 File/Path Properties dialog

This dialog displays properties for the file and controls build and compile options.

——— Note ————

• On Microsoft Windows, the / and \ directory separators both appear as / . This simplification does not affect operation.
• Avoid using Japanese or Korean characters in filepaths. They can cause failure to find libraries.

———

Select a component from the Component window Files tab, right click on it to open the context menu, then click Properties to display the dialog.

General

File or path
The name of the file.

——— Note ————

The File Properties dialog is modeless. You can select a different file without closing the dialog. A warning message prompts to save any changes.
Absolute path
The full path to the file.

Repository
The repository file that contains this component entry.

Type
A brief description of the component type.

Info
The status of the file. For example, `file does not exist`.

Supported platforms
Select the platforms that the component supports:
- Linux64.
- Win64 (Release runtime library).
- Win64D (Debug runtime library).

Compiler
Select the compiler for this component from the drop-down list:
- No preference.
- Specific Microsoft Visual C++ compiler.
- gcc version found in $PATH at compile time.
- Specific gcc version.

Build actions
Default actions depending on file extension

.lisa
A LISA source file that SimGen parses.

.c .cpp .cxx
A C or C++ source file that the compiler compiles.

.a .o
A Linux object file that SimGen links to.

.lib .obj
A Microsoft Windows object file that SimGen links to.

.sgproj
A project file that SimGen parses.

.sgrepo
A component repository file that SimGen parses.

directory_path/
An include directory for the search path that the compiler uses. The trailing slash identifies it as an include path. For example, to add the directory that contains the *.sgproj file, specify `./` (dot slash), not only the dot.

All other files
Copy a deploy file to the build directory.

Note
Simulation Generator (SimGen) is one of the Fast Models tools.

Ignore
Exclude the selected file from build and deploy. This feature can be useful for examples, notes, or temporarily disabled files.

Customize actions
Ignore the file extension. Specify the actions with the check boxes:

**LISA - input file passed to Simulator Generator as LISA**
System Canvas passes the file to SimGen as a LISA file. Do not use this option for non-LISA files.

**Compile - compile as C/C++ source code**
To compile a file as C/C++ code during the build process, add it to this list of files.

**Link - input file for linker**
Link the file with the object code during the build process.

**Deploy - copy to build directory**
Copy the file into the build directory. This option can, for example, add dynamic link libraries for running the generated system model.

**Include path - add the file’s path to additional include directories**
Add the path of the parent directory that holds the file to the list of include directories for the compiler.

**Library path - add the file’s path to additional library directories**
Add the path of the parent directory that holds the file to the list of library directories for the compiler.

**Related references**
*Project parameter IDs* on page 5-98

### 5.3.10 Find and Replace dialogs

This dialog enables searching for and replacement of text in an editor window.

The **Find** dialog and the **Find and Replace** dialog are essentially the same dialog in two modes, find only, and find and replace. Switch modes by clicking the **Find mode** or **Find and replace mode** buttons. By default, matches are case sensitive but matches can appear as part of longer words. Change the default behavior by setting or clearing the relevant checkboxes in the dialog.

Open the **Find** dialog by clicking **Search > Find...** in the main menu. Type the text to find in the box and click the **Find Next** or **Find Previous** buttons to search upwards or downwards from the current cursor position. You can re-use previous search terms by clicking on the drop-down arrow on the right of the text entry box.

Open the **Find and Replace** dialog by clicking **Search > Replace** in the main menu. Replace the current match with new text by clicking the **Replace** button, or all matches by clicking the **Replace All** button. You can re-use previous find or replacement terms by clicking on the drop-down arrow on the right of the text entry boxes.

Find and Replace mode is only available if the current active window is a source editor. In that mode, additional replace controls appear. The dialog is modeless, so you can change views without closing it.

### 5.3.11 Label Properties dialog

This dialog controls the text and display properties for a label.

Double-click on a label to display the dialog. Select **Add Label** from the **Object** menu to add a label to the component.

**Label**
Specify the text to display on the label.

**Font**
The text font. Click **Select Font...** to change it.
Select Text Color...
Click to select a color for the text.

Select Background Color...
Click to select the background color for the label.

Check Transparent Background
Check to make objects behind the label visible, and to ignore the background color setting.

Horizontal
Set the horizontal justification for the label text.

Vertical
Set the vertical justification for the label text.

Rotation
Set the orientation for the label.

Frame Thickness
Set the thickness of the label border.

Shadow Thickness
Set the thickness of the label drop shadow.

Display on Top
Check to display the label on top of any components below it.

Use these settings as default
Check to use the current settings as the default settings for any new labels.

5.3.12 New File dialog (File menu)
This dialog creates new projects and LISA source files.
To display the dialog, select New File from the File menu or click the New button.

Look in
Specify the directory for the new file.

File name
Enter the name for the new file.

File type
• If a project is not open, this box displays .sgproj by default to create a project.
• If a project is open, this box displays .lisa by default to create a LISA source file.

Add to
Active for non-.sgproj files. Check to enable the adding of the created file to the open project.

Select
Click to accept the name and path.
If the new file is of type .sgproj, System Canvas prompts for the top level LISA file.
Save time with the Recently selected files drop-down list. To remove a file, mouse over it and press Delete, or right-click and select Remove from list from the context menu.

Note

---

5.3.13 New project dialogs

This section describes the dialogs that create new projects.

**New Project dialog**

This dialog creates new projects.

To display the dialog, select New Project from the File menu.

**Look in**

Specify the directory for the new project file.

**File name**

Enter the name for the new project.

If you select an existing file, the new project replaces the existing project.

**File type**

The default type for Fast Models projects is .sgproj.

**Select**

Click to accept the name and path.

For existing projects, System Canvas queries the replacement of the existing project with a new project of the same name.

After you click Select, the Select Top Component LISA File dialog appears.

Note

The project file includes the path to the model repositories from the Default Model Repositories pane of the Preferences dialog.

Related references

Select Top Component LISA File dialog on page 5-87

---

5.3.14 Open File dialog

This dialog opens project files, LISA source files, and text documents.

To display the dialog:

- Select Open File from the File menu.
- Select a file in the Component window and select Open from the context menu.
Look in
   Specify the directory.

File name
   Enter the name of the file.

File type
   Select the type of file.

Open
   Click to open the file.

Open project file as text in source editor
   Active for non-.lisa and for .sgproj files. Check to enable the opening of the file as plain text in the Source window.

   —— Note ——
   • Use this option, for example, for a .sgproj file to manually edit the list of repositories. Such changes take effect after you close and reopen the file.
   • If you select a .sgproj file without checking this box, the project loads.

   —— Note ——
   Save time with the Recently selected files drop-down list. To remove a file, mouse over it and press Delete, or right-click and select Remove from list from the context menu.

5.3.15 Port Properties dialog
   This dialog controls port properties.

   To display the Port Properties dialog, select a port or a connection.
   • Select a component port in the Block Diagram view and:
   — Double-click on the port.
   — Click the Properties button.
   — Select Object Properties from the Object menu.
   — Right-click and select Object Properties from the context menu.
   • Select a connection in the Block Diagram view and double-click to display the Connection Properties dialog. To display the Port Properties dialog:
   — Click the Master Port Properties button to display the properties for the master port.
   — Click the Slave Port Properties button to display the properties for the slave port.

Name
   The name of the port.

Type
   The type of port and the protocol.

Array size
   For port arrays, the number of elements.

Show connections for port array index
   For port arrays, enter an index value in the integer box to display only that element.

   For individual ports of port arrays, this box displays the index for the selected port.
Port connections

- Sort the connections: click on the column headings.
- Change the connected port or address mapping: select a connection and click Edit Connection.
- Add a connection: select a connection and click Add Connection.
- Delete a connection: select it and click Remove.
- Change the priority of a single connection: select it and click Increase Priority or Decrease Priority.

Related references

5.3.7 Connection Properties dialog on page 5-82

5.3.16 Preferences dialog

This section describes the Preferences dialog (File > Preferences), which configures the working environment of System Canvas.

Preferences - Appearance group

This group sets the appearance of System Canvas.

Show Tool Tips

Display all tool tips.

Display tool bar text labels

Display the status bar labels.

Word wrap in source windows

Wrap long lines to display them within the source window.

Show splash screen on startup

Show the splash screen on startup.

Reload recent layout on startup

Reload the layout settings from the last modified project.

Recent files and directories

Set the number of directories and files shown in System Canvas file dialogs and menus, up to 32 directories and 16 files.

Preferences - Applications group

This group sets the application paths.

--- Note ---

- On Microsoft Windows, environment variables appear as $MAXxxxx_HOME. You can use this format instead of %MAXxxxx_HOME%.
- The different path specifications enable the use of different versions of Model Debugger and provide more flexibility for installing Model Debugger separately from System Canvas.

Simulator Generator Executable

SimGen

Set the path to the simgen.exe file.

Command arguments

Set additional command-line options.

Model Debugger Executable
Model Debugger
Set the path to the Model Debugger executable.

Command arguments
Set additional command-line options.

Model Shell Executable
Model Shell
Set the path to the Model Shell executable.

Command arguments
Set additional command-line options.

Run Model Shell asynchronously with output to console in separate window
Check to enable starting a separate Model Shell instance with its own output window.

——— Note ————
To start the simulation, select the Run in Model Shell entry on the Projects menu.

Path to Microsoft Visual Studio application ‘devenv.com’
Select the path to the Microsoft Visual Studio devenv.com file. This application is the development environment and builds the model.

Reset to Defaults
Click to reset the application paths.

Apply
Click to save the changes.

Run Model Shell asynchronously
On Linux, check to use the command line:

```bash
xterm -e <Model Shell Executable> optional_command_arguments_list -m model.so
```

Host Debugger Command Line
On Linux, set the command-line options. The default text is:

```bash
xterm -e gdb --args %ISIM%
```

where %ISIM% is a placeholder for the isim_system executable file.

——— Note ————
On Linux, select the GCC compiler to build the model by using the SimGen command-line option --gcc-path.

Related references
Chapter 6 System Generator Reference on page 6-108
6.2 SimGen command-line options on page 6-110
5.3.17 Project Settings dialog on page 5-93

Related information
Model Debugger for Fast Models User Guide

Preferences - External Tools group
This group sets the tools that display the documentation.
use operating system file associations
   Check to inactivate the external tool edit fields and buttons. Clear to activate them.

   ___________ Note ___________
   This checkbox is not available on Linux.

Preferences - Fonts group
This group sets the application fonts.

Application
   The application font.

Base fixed font
   The Source view font.

Block Diagram Component Name
   The component title block font.

Fonts depend on $DISPLAY variable
   Check to use the font set in the $DISPLAY variable.

Reset to base size
   Reset all font sizes to the selected value.

Reset to defaults
   Click to reset the fonts to the factory settings.

   ___________ Note ___________
   If non-Latin characters are used in LISA code, the base fixed font must support them. The default font
   might not support non-Latin characters.

Preferences - Default Model Repository group
This group sets the default model repositories for new projects.
To incorporate components into a system, System Canvas requires information about them, such as their ports, protocols, and library dependencies. For convenience, model repositories, such as `sglib.sgrepo`, group multiple components together and specify the location of the LISA files and the libraries that are needed to build them.

Default repositories are added by default to new projects. To add a repository to an existing project, use the Component window context menu.

**Note**

To enable the immediate use of models in new projects, System Canvas has a default entry `$(PVLIB_HOME)/etc/sglib.sgrepo`. This entry is not deletable, but clearing the checkbox deactivates it.

- **Add**
  - Click **Add** to open a file selection dialog and add a new `.sgrepo` repository file to the list.
  - Select a directory to add all of the repositories in that directory to the list of repositories.

- **Edit Path**
  - Select a repository and click **Edit** to edit the path to it.
  - The path to the default repository `$(PVLIB_HOME)/etc/sglib.sgrepo` is not editable.

- **Remove**
  - Select a repository and click **Remove** to exclude the selected repository from new projects. This does not affect the repository itself.
  - The default repository `$(PVLIB_HOME)/etc/sglib.sgrepo` is not deletable.
File checkboxes

Check to automatically include the repository in new projects. Clear to prevent automatic inclusion, but to keep the path to the repository available.

Up/Down

Use the Up and Down buttons to change the order of repositories. File processing follows the repository order.

Related references

1.4.3 Repository files on page 1-21

Preferences - Suppressed messages group

This group lists the suppressed messages and controls their re-enabling.

Enable selected messages

Click to enable selected suppressed messages.

5.3.17 Project Settings dialog

This section describes the dialog (Project > Project Settings, or Settings toolbar button) that sets the project settings and customizes the generation process.

Project top-level settings

This part of the dialog sets the project build options.

Top level component

- Enter a name into the Top Level Component edit box.
- Click Use Current to set the component in the workspace as the top component.
- Click Select From List to open a dialog and select any component in the system.

Configuration

- Select an entry from the drop-down list to use an existing configuration.
- Click Add New to create a new configuration. A dialog prompts for the name and a description. Use Copy values from to select a configuration to copy the settings values from. This can be an existing configuration or a default set of configuration settings.
- Click Delete to delete the selected configuration from the list.

The values default to those of the active configuration.

Selecting a configuration in this dialog does not set the configuration in the Select Active Project Configuration drop-down box on the main window. System Canvas stores the configuration set in this dialog in the project file, to use if you specify it for a build. You can use this control to specify all of the configurations for a project, to simplify switching active configurations.

Note

If you build systems on Microsoft Windows workstations, other Microsoft Windows workstations need the matching support libraries to run the systems:

Debug builds

Microsoft Visual Studio.

Release builds

Microsoft Visual Studio redistributable package.

---

Copyright © 2014–2019 Arm Limited or its affiliates. All rights reserved.

Non-Confidential
Parameter category panel

This section describes the Parameter category panel, which lists parameters for the selected build, under different views.

Parameters - Category View

This view lists categories and the parameters for the selected category.

Top-level configuration details

Select the top-most category item to configure the project settings.

<table>
<thead>
<tr>
<th>Control name</th>
<th>Parameter</th>
</tr>
</thead>
<tbody>
<tr>
<td>Configuration name</td>
<td>CONFIG_NAME</td>
</tr>
<tr>
<td>Platform/Linkage</td>
<td>PLATFORM</td>
</tr>
<tr>
<td>Compiler</td>
<td>COMPILER</td>
</tr>
<tr>
<td>Configuration description</td>
<td>CONFIG_DESCRIPTION</td>
</tr>
<tr>
<td>Build directory</td>
<td>BUILD_DIR</td>
</tr>
</tbody>
</table>

Targets

Select the Targets item to configure the build target parameters.

<table>
<thead>
<tr>
<th>Control name</th>
<th>Parameter</th>
</tr>
</thead>
<tbody>
<tr>
<td>Integrated simulator with deprecated scheduler</td>
<td>TARGET_ISIM_DEPRECATED</td>
</tr>
<tr>
<td>CADI library*</td>
<td>TARGET_MAXVIEW</td>
</tr>
<tr>
<td></td>
<td>Note: Arm deprecates TARGET_MAXVIEW.</td>
</tr>
<tr>
<td>SystemC component</td>
<td>TARGET_SYSTEMC</td>
</tr>
<tr>
<td>SystemC component with auto-bridging</td>
<td>TARGET_SYSTEMC_AUTO</td>
</tr>
<tr>
<td>SystemC integrated simulator*</td>
<td>TARGET_SYSTEMC_ISIM</td>
</tr>
</tbody>
</table>
Debugging

Select the **Debugging** item in the panel to configure the debug parameters.

**Table 5-7 Debugging parameters in the Category View**

<table>
<thead>
<tr>
<th>Control name</th>
<th>Parameter</th>
</tr>
</thead>
<tbody>
<tr>
<td>Enable model debugging</td>
<td>ENABLE_DEBUG_SUPPORT</td>
</tr>
<tr>
<td>Source reference</td>
<td>GENERATE_LINEINFO</td>
</tr>
<tr>
<td>Verbosity</td>
<td>VERBOSITY</td>
</tr>
<tr>
<td>Model Debugger</td>
<td>MODEL_DEBUGGER_COMMAND_LINE</td>
</tr>
<tr>
<td>Model Shell and ISIM</td>
<td>MODEL_SHELL_COMMAND_LINE</td>
</tr>
<tr>
<td>SystemC executable</td>
<td>SYSTEMC_EXE</td>
</tr>
<tr>
<td>SystemC arguments</td>
<td>SYSTEMC_COMMAND_LINE</td>
</tr>
</tbody>
</table>

Sim Generator

Select the **Sim Generator** item in the panel to configure the Simulation Generator parameters.

**Table 5-8 Simulation Generator parameters in the Category View**

<table>
<thead>
<tr>
<th>Control name</th>
<th>Parameter</th>
</tr>
</thead>
<tbody>
<tr>
<td>Simgen options</td>
<td>SIMGEN_COMMAND_LINE</td>
</tr>
<tr>
<td>Warnings as errors</td>
<td>SIMGEN_WARNINGS_AS_ERRORS</td>
</tr>
<tr>
<td>Using namespace std</td>
<td>ENABLE_NAMESPACE_STD</td>
</tr>
<tr>
<td>Make options</td>
<td>MAKE_OPTIONS</td>
</tr>
</tbody>
</table>

Compiler

Select the **Compiler** item in the panel to configure the compiler parameters.

**Table 5-9 Compiler parameters in the Category View**

<table>
<thead>
<tr>
<th>Control name</th>
<th>Parameter</th>
</tr>
</thead>
<tbody>
<tr>
<td>Pre-Compile Actions</td>
<td>PRE_COMPILE_EVENT</td>
</tr>
<tr>
<td>Include Directories</td>
<td>INCLUDE_DIRS</td>
</tr>
<tr>
<td>Preprocessor Defines</td>
<td>PREPROCESSOR_DEFINES</td>
</tr>
<tr>
<td>Compiler Settings</td>
<td>ADDITIONAL_COMPILER_SETTINGS</td>
</tr>
<tr>
<td>Enable pre-compiling</td>
<td>ENABLE_PRECOMPILE_HEADER</td>
</tr>
</tbody>
</table>
Linker

Select the Linker item in the panel to configure the linker parameters.

Table 5-10  Linker parameters in the Category View tab

<table>
<thead>
<tr>
<th>Control name</th>
<th>Parameter</th>
</tr>
</thead>
<tbody>
<tr>
<td>Pre-Link Actions</td>
<td>PRE_LINK_EVENT</td>
</tr>
<tr>
<td>Linker Settings</td>
<td>ADDITIONAL_LINKER_SETTINGS</td>
</tr>
<tr>
<td>Post-Build Actions</td>
<td>POST_BUILD_EVENT</td>
</tr>
<tr>
<td>Post-Clean Actions</td>
<td>POST_CLEAN_EVENT</td>
</tr>
<tr>
<td>Disable suppression of symbols</td>
<td>DISABLE_SYMBOL_SUPRESSION</td>
</tr>
</tbody>
</table>

Parameters - List View

This view lists the parameters and their values. Reorder them by clicking on a column heading.

Parameters - Tree View

This view displays parameters in a tree structure, with expandable categories.

Parameters - setting the release options

This section describes how to set the build options for a project configuration using the Project Settings dialog.

Procedure

1. Click the Category View tab.
2. Select the Windows-Release entry and choose the operating system/link options from the Platform/Linkage drop-down menu.
   - **Option**   | **Description** |
   - Linux64      | 64-bit model for Linux. |
   - Win64        | 64-bit model using the release run-time library for Microsoft Windows. |
   - Win64D       | 64-bit model using the debug run-time library for Microsoft Windows. |
3. Select the compiler from the Compiler drop-down menu.
4. Enter a path into the Build directory field to select the directory to perform the builds in.
   This directory contains the source code and the build library for the system model. If the path is not absolute, System Canvas treats it as being relative to the directory that contains the project file.
5. Enter text into the Configuration description box that describes the configuration.

Parameters - overloading the main() function in the target

This section describes how to replace the default main() of an ISIM with a user-supplied main().

Caution

If you use the option USER_DEFINED_ISIM_MAIN and a user-supplied main(), you cannot build a CADI shared library from the project.

If a CADI shared library is required:
- Add a new configuration for isim_system that defines USER_DEFINED_MAIN.
- Add an #ifdef USER_DEFINED_MAIN test block around the main() in the user source file.
Procedure

1. Define the USER_DEFINED_ISIM_MAIN preprocessor option for the compiler in the Project Settings dialog.

Results:

2. Supply a C++ file or a library with a user-defined main() function.

A fragment of the standard IsimMain.cpp file:

```cpp
#ifndef USER_DEFINED_ISIM_MAIN // opposite logic to standard IsimMain.cpp
#include "SimGenTplMacros.h"
// function that performs command line parsing
#include "SimGenTplMacros.h"
// CADI system initialization and run
extern int LoadInitAndRunCADIModel(int argc, char *argv[],
    const char* topComponent, const char* pvLibVersion);
int main(int argc, char *argv[])
{
    return LoadInitAndRunCADIModel(argc, argv, SIMGEN_TOP_COMPONENT,
    PVLIB_VERSION_STRING);
}
#endif // #ifndef USER_DEFINED_ISIM_MAIN
```

You might define the USER_DEFINED_ISIM_MAIN preprocessor option, for example, so that you can implement processing of your own command-line options but must, after filtering out all user-defined switches, pass the remaining options to the Model Shell entry function LoadInitAndRunCADIModel().

![Figure 5-3 Specifying user-defined main() option](image-url)
3. Add the new source file containing the custom main() to the project.

*Related references*

5.3.1 Add Existing Files and Add New File dialogs (Component window) on page 5-75

**Project parameter IDs**

The parameters that configure a project, with IDs, names, defaults, and descriptions.

### Table 5-11 Full list of parameters shown in List View

<table>
<thead>
<tr>
<th>Parameter ID</th>
<th>Parameter name</th>
<th>Default</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>ADDITIONAL_COMPILER_SETTINGS</td>
<td>Compiler settings</td>
<td>&quot;&quot;</td>
<td>Compiler settings. If your C++ source code uses C++11 syntax, specify -std=c++11 in this parameter. For Microsoft Windows, consult the Visual Studio documentation.</td>
</tr>
<tr>
<td>ADDITIONAL_LINKER_SETTINGS</td>
<td>Linker settings</td>
<td>&quot;&quot;</td>
<td>Linker settings. For Microsoft Windows, consult the Visual Studio documentation.</td>
</tr>
<tr>
<td>BUILD_DIR</td>
<td>Build directory</td>
<td>&quot;&quot;</td>
<td>Build directory. If this path is not absolute, it is relative to the position of the project file. For Microsoft Windows, .\Windows-Debug or .\Windows-Release.</td>
</tr>
</tbody>
</table>
| COMPILER                              | Compiler             | -       | VC2017
VC2015
Microsoft Visual Studio 2015.  
gcc
The first gcc version in the Linux search path.  
gcc-4.9
GCC 4.9.2.  
gcc-5.4
GCC 5.4.  
gcc-6.4
GCC 6.4.                                                                                                                                 |
| CONFIG_DESCRIPTION                    | Configuration description | ""     | Description of the configuration, CONFIG_NAME.                                                                                             |
| CONFIG_NAME                           | Configuration name   | ""     | Name of the configuration.                                                                                                                  |
| ENABLE_DEBUG_SUPPORT                  | Enable model debugging | "0"    | Use implementation defined debug support.                                                                                                  |
| ENBALE_NAMESPACE_STD                  | Enable namespace std | "1"    | Use namespace std:  
1 (true)
Generate using namespace std and place in the code.  
0 (false)
Specify the namespace. This setting might reduce compilation time.                                          |
<table>
<thead>
<tr>
<th>Parameter ID</th>
<th>Parameter name</th>
<th>Default</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>ENBALE_PRECOMPILE_HEADER</td>
<td>Enable precompiling</td>
<td>&quot;0&quot;</td>
<td>Precompile headers if true.</td>
</tr>
<tr>
<td>GENERATE_LINEINFO</td>
<td>Source reference</td>
<td>&quot;LISA Code (incl. headers)&quot;</td>
<td>Control line redirection in the generated model source code:</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>&quot;LISA Code&quot; Source code.</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>&quot;LISA Code (incl. headers)&quot; Source and header.</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>&quot;Generated Code&quot; No line redirection at all.</td>
</tr>
<tr>
<td>INCLUDE_DIRS</td>
<td>Include directories</td>
<td>&quot;&quot;</td>
<td>Include directories. Separate multiple entries with semicolons.</td>
</tr>
<tr>
<td>MODEL_DEBUGGER_COMMAND_LINE</td>
<td>Model Debugger</td>
<td>&quot;&quot;</td>
<td>Options to pass on the command line.</td>
</tr>
<tr>
<td>MODEL_SHELL_COMMAND_LINE</td>
<td>Model Shell</td>
<td>&quot;&quot;</td>
<td>Options to pass on the command line.</td>
</tr>
<tr>
<td>PLATFORM</td>
<td>Platform/linkage</td>
<td>&quot;Linux64&quot;</td>
<td>64-bit Linux.</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>&quot;Win64&quot; 64-bit Microsoft Windows release.</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>&quot;Win64D&quot; 64-bit Microsoft Windows debug.</td>
</tr>
<tr>
<td>POST_BUILD_EVENT</td>
<td>Postbuild actions</td>
<td>&quot;&quot;</td>
<td>Commands to execute after building the model. Separate multiple entries with semicolons.</td>
</tr>
<tr>
<td>PRE_COMPILE_EVENT</td>
<td>Precompile actions</td>
<td>&quot;&quot;</td>
<td>Commands to execute before starting compilation. Applies to Microsoft Windows only. Separate multiple entries with semicolons.</td>
</tr>
<tr>
<td>PREPROCESSOR_DEFINES</td>
<td>Preprocessor defines</td>
<td>&quot;&quot;</td>
<td>Preprocessor defines. Separate multiple entries with semicolons.</td>
</tr>
<tr>
<td>PRE_LINK_EVENT</td>
<td>Prelink actions</td>
<td>&quot;&quot;</td>
<td>Commands to execute before starting linking. Applies to Microsoft Windows only. Separate multiple entries with semicolons.</td>
</tr>
<tr>
<td>SIMGEN_COMMAND_LINE</td>
<td>SimGen options</td>
<td>&quot;&quot;</td>
<td>Options to pass on the command line.</td>
</tr>
<tr>
<td>SIMGEN_WARNINGS_AS_ERRORS</td>
<td>Warnings as errors</td>
<td>&quot;1&quot;</td>
<td>If 1 (true), treat LISA parsing and compiler warnings as errors.</td>
</tr>
<tr>
<td>SYSTEMC_COMMAND_LINE</td>
<td>SystemC arguments</td>
<td>&quot;&quot;</td>
<td>Command-line arguments for System C executable.</td>
</tr>
<tr>
<td>SYSTEMC_EXE</td>
<td>SystemC executable</td>
<td>&quot;&quot;</td>
<td>Name of final SystemC executable. Call the file with ‘Run SystemC executable’.</td>
</tr>
<tr>
<td>TARGET_ISIM_DEPRECATED</td>
<td>Integrated simulator with deprecated scheduler</td>
<td>&quot;0&quot;</td>
<td>If 1 (true), build an executable with a statically linked CADI system and Model Shell, using the deprecated scheduler.</td>
</tr>
</tbody>
</table>
### Table 5-11 Full list of parameters shown in List View (continued)

<table>
<thead>
<tr>
<th>Parameter ID</th>
<th>Parameter name</th>
<th>Default</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>TARGET_MAXVIEW</td>
<td>CADI library</td>
<td>&quot;1&quot;</td>
<td>If 1 (true), build a CADI system dynamic library for running from Model Debugger.</td>
</tr>
<tr>
<td>Note:</td>
<td></td>
<td></td>
<td>Arm deprecates TARGET_MAXVIEW.</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>a</td>
</tr>
<tr>
<td>TARGET_SYSTEMC</td>
<td>SystemC component</td>
<td>&quot;0&quot;</td>
<td>If 1 (true), build a SystemC component library.</td>
</tr>
<tr>
<td>TARGET_SYSTEMC_AUTO</td>
<td>SystemC component</td>
<td>&quot;0&quot;</td>
<td>If 1 (true), build a SystemC component library with auto-bridging.</td>
</tr>
<tr>
<td></td>
<td>with auto-bridging</td>
<td></td>
<td></td>
</tr>
<tr>
<td>TARGET_SYSTEMC_ISIM</td>
<td>SystemC integrated</td>
<td>&quot;0&quot;</td>
<td>If 1 (true), build a SystemC ISIM executable.</td>
</tr>
<tr>
<td></td>
<td>simulator</td>
<td></td>
<td></td>
</tr>
<tr>
<td>VERBOSITY</td>
<td>Verbosity</td>
<td>&quot;Off&quot;</td>
<td>Verbosity level: &quot;Sparse&quot;, &quot;On&quot;, or &quot;Off&quot;.</td>
</tr>
</tbody>
</table>

#### Related concepts

7.4 Auto-bridging on page 7-123

### Project file contents

Project files describe the settings for each platform and the required files to build models.

System Generator generates the project file from the specified configuration options. File and directory names can be either absolute or relative to the project or repository file. You can use environment variables in filenames.

File or directory entries in project files can include filters to specify the following build options:

#### Host platform

- "Linux64"  
  64-bit Linux.
- "Win64"  
  64-bit Microsoft Windows release.
- "Win64D"  
  64-bit Microsoft Windows debug.

#### Compiler

- "VC2017"  
- "VC2015"  
  Microsoft Visual Studio 2015.
- "gcc"  
  The first gcc version in the Linux search path.
- "gcc-4.9"  
  GCC 4.9.2.
- "gcc-5.4"  
  GCC 5.4.

---

*If you select both TARGET_MAXVIEW and TARGET_SYSTEMC_ISIM, SimGen only generates an ISIM executable, not a CADI library.*
"gcc-6.4"
GCC 6.4.

--- Note ---
For Linux, the compiler version only affects the files that the project file and repositories identify. It does not select the gcc found in the search path. To enable the System Generator to automatically select the libraries that match the current gcc compiler, use the compiler option gcc.

**Action**

"lisa"
Process the file as a LISA file. This action is not applicable to directories.

"compile"
Process the file as a C++ file. If acting on a directory, the compiler compiles all *.c, *.cpp, and *.cxx files in the directory.

"ignore"
Exclude the file or directory from the build and deploy process, such as a disabled file or project notes.

"link"
Link the file with existing files. If acting on a directory on Microsoft Windows, System Generator adds all *.lib and *.obj files in the directory to the linker input. On Linux, it adds all *.a and *.o files.

"deploy"
Produce a deployable file. If acting on a directory, System Generator copies the entire directory and its subdirectories to the destination. This action is the only action that acts recursively on subdirectories.

"incpath"
Include the directory in the list of include search paths that the -I option for the compiler specifies. This action is the default action for directories.

"libpath"
Include the directory in the list of library search paths that the -L option for the compiler specifies. This action is the default action for directories.

The build options for the file or directory entries are not sensitive to case.

For example, the my_file.lib file can specify host, compiler, and action as:

```plaintext
path = my_file.lib, platform="WIN64|Win64D", compiler="VC2015", action="link|deploy";
```

Do not OR the compiler options together. Instead, omit them to permit more than one compiler:

```plaintext
path = ./src/my_windows_code.cpp, platform = "win64";
```

File entries in the project file can have a compiler filter in addition to the platform and action filters:

```plaintext
path = ../lib/release_2015/my_lib.lib, platform = "win64", compiler="VC2015";
path = ../lib/my_lib.lib, platform = "win64", compiler="VC2015";
path = ../src/my_windows_code.cpp, platform = "win64"; // Not specifying the compiler allows // more than one.
```

**Directories in path statements**

Differentiate directories from files with a trailing / character.

Project files can contain directories in the path statement. Platform and compiler filters might apply.
If you apply directory actions to files, System Generator applies them to the directories that contain the files, forming the directory path by removing the filename from the full path. This path specification:

```plaintext
path = MyFile.lisa, actions="lisa|incpath|libpath";
```

makes System Generator treat `MyFile.lisa` as the LISA source and add the parent directory of `MyFile.lisa` to the include and library search paths. If, for example, `MyFile.lisa` is in the directory `C:/ARM/MyProjects/Project_1/`, System Generator adds that directory path to the include and library search paths.

**Example project file**

An example project file that shows the use of different configuration sections.
Typical project file

```c
sgproject "exampleSystem.sgproj"
{
  TOP_LEVEL_COMPONENT = "exampleSystem";
  ACTIVE_CONFIG_LINUX = "Linux64-Release-GCC-4.9";
  ACTIVE_CONFIG_WINDOWS = "Win64-Release-VC2015"
  config "Linux64-Debug-GCC-4.9"
  {
    ADDITIONAL_COMPILER_SETTINGS = "-march=core2 -ggdb3 -Wall -std=c++11"
    ADDITIONAL_LINKER_SETTINGS = "-Wl,--no-undefined"
    BUILD_DIR = "./Linux64-Debug-GCC-4.9"
    COMPILER = "gcc-4.9"
    CONFIG_DESCRIPTION = "Default x86_64 Linux configuration for GCC 4.9 with debug information"
    CONFIG_NAME = "Linux64-Debug-GCC-4.9"
    ENABLE_DEBUG_SUPPORT = "1"
    PLATFORM = "Linux64"
    SIMGEN_COMMAND_LINE = "--num-comps-file 10"
    SIMGEN_WARNINGS_AS_ERRORS = "0"
  }
  config "Linux64-Release-GCC-4.9"
  {
    ADDITIONAL_COMPILER_SETTINGS = "-march=core2 -O3 -Wall -std=c++11"
    ADDITIONAL_LINKER_SETTINGS = "-Wl,--no-undefined"
    BUILD_DIR = "./Linux64-Release-GCC-4.9"
    COMPILER = "gcc-4.9"
    CONFIG_DESCRIPTION = "Default x86_64 Linux configuration for GCC 4.9, optimized for speed"
    CONFIG_NAME = "Linux64-Release-GCC-4.9"
    PLATFORM = "Linux64"
    PREPROCESSOR_DEFINES = "NDEBUG"
    SIMGEN_COMMAND_LINE = "--num-comps-file 10"
    SIMGEN_WARNINGS_AS_ERRORS = "0"
  }
  config "Linux64-Debug-GCC-5.4"
  {
    ADDITIONAL_COMPILER_SETTINGS = "-march=core2 -ggdb3 -Wall -std=c++11 -Wno-deprecated"
    ADDITIONAL_LINKER_SETTINGS = "-Wl,--no-undefined"
    BUILD_DIR = "./Linux64-Debug-GCC-5.4"
    COMPILER = "gcc-5.4"
    CONFIG_DESCRIPTION = "Default x86_64 Linux configuration for GCC 5.4 with debug information"
    CONFIG_NAME = "Linux64-Debug-GCC-5.4"
    ENABLE_DEBUG_SUPPORT = "1"
    PLATFORM = "Linux64"
    SIMGEN_COMMAND_LINE = "--num-comps-file 10"
    SIMGEN_WARNINGS_AS_ERRORS = "0"
  }
  config "Linux64-Release-GCC-5.4"
  {
    ADDITIONAL_COMPILER_SETTINGS = "-march=core2 -O3 -Wall -std=c++11 -Wno-deprecated -Wno-unused-function"
    ADDITIONAL_LINKER_SETTINGS = "-Wl,--no-undefined"
    BUILD_DIR = "./Linux64-Release-GCC-5.4"
    COMPILER = "gcc-5.4"
    CONFIG_DESCRIPTION = "Default x86_64 Linux configuration for GCC 5.4, optimized for speed"
    CONFIG_NAME = "Linux64-Release-GCC-5.4"
    PLATFORM = "Linux64"
    PREPROCESSOR_DEFINES = "NDEBUG"
    SIMGEN_COMMAND_LINE = "--num-comps-file 10"
    SIMGEN_WARNINGS_AS_ERRORS = "0"
  }
  config "Linux64-Debug-GCC-6.4"
  {
    ADDITIONAL_COMPILER_SETTINGS = "-march=core2 -ggdb3 -Wall -std=c++11 -Wno-deprecated -Wno-unused-function"
    ADDITIONAL_LINKER_SETTINGS = "-Wl,--no-undefined"
    BUILD_DIR = "./Linux64-Debug-GCC-6.4"
    COMPILER = "gcc-6.4"
    CONFIG_DESCRIPTION = "Default x86_64 Linux configuration for GCC 6.4 with debug information"
    CONFIG_NAME = "Linux64-Debug-GCC-6.4"
    ENABLE_DEBUG_SUPPORT = "1"
    PLATFORM = "Linux64"
    SIMGEN_COMMAND_LINE = "--num-comps-file 10"
    SIMGEN_WARNINGS_AS_ERRORS = "0"
  }
  config "Linux64-Release-GCC-6.4"
  {
    ADDITIONAL_COMPILER_SETTINGS = "-march=core2 -O3 -Wall -std=c++11 -Wno-deprecated -Wno-unused-function"
    ADDITIONAL_LINKER_SETTINGS = "-Wl,--no-undefined"
    BUILD_DIR = "./Linux64-Release-GCC-6.4"
    COMPILER = "gcc-6.4"
    CONFIG_DESCRIPTION = "Default x86_64 Linux configuration for GCC 6.4, optimized for speed"
    CONFIG_NAME = "Linux64-Release-GCC-6.4"
    PLATFORM = "Linux64"
    PREPROCESSOR_DEFINES = "NDEBUG"
  }
}
```
SIMGEN_COMMAND_LINE = "--num-comps-file 50";
SIMGEN_WARNINGS_AS_ERRORS = "0";
}

config "LinuxAArch64-Debug-GCC-5.4"
{
    ADDITIONAL_COMPILER_SETTINGS = "-march=armv8-a -ggdb3 -Wall -std=c++11 -Wno-deprecated";
    ADDITIONAL_LINKER_SETTINGS = "-Wl,--no-undefined";
    BUILD_DIR = "./LinuxAArch64-Debug-GCC-5.4";
    COMPILER = "gcc-5.4";
    CONFIG_DESCRIPTION = "Default aarch64 Linux configuration for GCC 5.4 with debug information";
    CONFIG_NAME = "LinuxAArch64-Debug-GCC-5.4";
    PLATFORM = "Linux64";
    SIMGEN_COMMAND_LINE = "--num-comps-file 10";
    SIMGEN_WARNINGS_AS_ERRORS = "0";
}

config "LinuxAArch64-Release-GCC-5.4"
{
    ADDITIONAL_COMPILER_SETTINGS = "-march=armv8-a -O3 -fomit-frame-pointer -Wall -std=c++11 -Wno-deprecated";
    ADDITIONAL_LINKER_SETTINGS = "-Wl,--no-undefined";
    BUILD_DIR = "./LinuxAArch64-Release-GCC-5.4";
    COMPILER = "gcc-5.4";
    CONFIG_DESCRIPTION = "Default aarch64 Linux configuration for GCC 5.4, optimized for speed";
    CONFIG_NAME = "LinuxAArch64-Release-GCC-5.4";
    PLATFORM = "Linux64";
    PREPROCESSOR_DEFINES = "NDEBUG";
    SIMGEN_COMMAND_LINE = "--num-comps-file 50";
    SIMGEN_WARNINGS_AS_ERRORS = "0";
}

config "Win64-Debug-VC2015"
{
    ADDITIONAL_COMPILER_SETTINGS = "/Od /RTCsu /Zi";
    ADDITIONAL_LINKER_SETTINGS = "/DEBUG";
    BUILD_DIR = "./Win64-Debug-VC2015";
    COMPILER = "VC2015";
    CONFIG_DESCRIPTION = "Default x86_64 Windows configuration for Visual Studio 2015 with debug information";
    CONFIG_NAME = "Win64-Debug-VC2015";
    ENABLE_DEBUG_SUPPORT = "1";
    PLATFORM = "Win64D";
    SIMGEN_COMMAND_LINE = "--num-comps-file 10";
    SIMGEN_WARNINGS_AS_ERRORS = "0";
}

config "Win64-Release-VC2015"
{
    ADDITIONAL_COMPILER_SETTINGS = "/O2";
    BUILD_DIR = "./Win64-Release-VC2015";
    COMPILER = "VC2015";
    CONFIG_DESCRIPTION = "Default x86_64 Windows configuration for Visual Studio 2015, optimized for speed";
    CONFIG_NAME = "Win64-Release-VC2015";
    PLATFORM = "Win64";
    SIMGEN_COMMAND_LINE = "--num-comps-file 50";
    SIMGEN_WARNINGS_AS_ERRORS = "0";
}

files
{
    path = "$\{PVLIB_HOME\}/etc/sglib.sgrepo";
    path = ".\..\LISA\exampleSystem.lisa";
    path = ".\..\LISA\exampleComponent.lisa";
}

Related references

1.4.4 File processing order on page 1-22
5.3.9 File/Path Properties dialog on page 5-83

5.3.18 Protocol Properties dialog

This dialog displays the properties of protocols.

Select a protocol from the Protocols list, right-click on it and select Properties to display the properties.

Protocol name

The name of the protocol.
File
The file that defines the protocol.

Repository
The repository that contains the reference to the file path.

Description
A description dating from the addition of the file to the project.

Methods
A panel that displays the LISA prototypes of methods, or behaviors, available for the protocol. The values are for reference only. They are not editable.

Properties
A panel that displays the properties for protocol. The values are for reference only. They are not editable.

5.3.19 Run dialog
This dialog specifies the actions that execute to run a selected target. There are actions for different targets, and additional options.

To display the dialog, click Run from the Project menu.

Select command to run
Select the executable to run.

Full command line
Adjust the command line that System Canvas generates, for example, add parameters or change the location of the application to load onto the executable.

Effective command line
Shows the complete command line with expanded macros and environment variables, ready for execution.

Model Debugger
Run the model in Model Debugger. The initial command line options come from project settings and user preferences.

Model Shell
Run the model with Model Shell. The initial command line options come from project settings and user preferences.

ISIM system
Run the model as an ISIM system. The initial command line options come from project settings and user preferences.

Custom
Specify the command line in Full command line.

Recent
Select a recent command.

Insert Placeholder Macro
Insert a macro or environment variable from drop-down list at the current cursor position in Full command line. System Generator expands them to build the complete command line.
%CADI%
The full absolute path of the CADI dynamic library.

%ISIM%
The full absolute path of the ISIM executable.

%BUILD_DIR%
The relative path to the build directory (relative to project path).

%DEPLOY_DIR%
The relative path to the deploy directory (identical to %BUILD_DIR%)

%PROJECT_DIR%
The full absolute path to the directory of the project.

Launch in background
Run an application asynchronously in a separate console window. Use this if the application requests user input or if the output is long.

Clear History
Remove all the recent entries from command history. This also removes corresponding items from the System Canvas main menu.

5.3.20 Self Port dialog
Use this dialog to add a port to the top-level component.

To display the dialog, without having anything selected in the Block Diagram view, click Add Ports, or click Add Port from the Object menu.

Instance name
The name of the port.

Array size
The number of ports, for a port array. Leave the box empty, or enter 1, for normal ports.

Protocol
The name of the protocol for the port. To display a list of protocols, click Select....

Type
Master port or Slave Port.

Attributes
- Addressable for bus ports.
- Internal for ports between subcomponents. The port is not visible if the component is added to a system.

Create LISA method templates according to selected protocol
Select an option from the drop-down list to create implementation templates for methods, or behaviors, for the selected protocol:
- Do not create method templates.
- Create only required methods. This is the default.
- Create all methods, including optional behaviors.

This creates only methods corresponding to the selected port type, that is, for either master or slave.

Editing the existing port might create new methods, but does not delete existing methods.
Mirror port image

Reverse the direction of the port image.
This chapter describes System Generator (SimGen).

It contains the following sections:

- 6.1 About System Generator on page 6-109.
- 6.2 SimGen command-line options on page 6-110.
- 6.3 Decreasing compilation time with num-comps-file and num-build-cpus on page 6-113.
- 6.4 Setting command-line options from System Canvas on page 6-114.
6.1 About System Generator

*System Generator* (SimGen) is a utility that generates a system model from a project.

To use SimGen, either:
- Press the **Build** icon in System Canvas to build the current project.
- Type `simgen` on a command line, specifying the project options.

To display help for the command-line options, start SimGen with no parameters or use the `--help` option:

```
simgen --help
```

SimGen uses gcc or Visual Studio C++ compiler during its operation. On Windows, if SimGen cannot find `devenv` for Visual Studio, the build fails. You can specify the path to `devenv` in the System Canvas **Preferences** dialog, or using the `--devenv-path` command-line option.

An example command line for building a platform simulator is:

```
simgen -p PROJECTFILE_NAME.sproj --configuration CONFIGURATION_NAME -b
```
### 6.2 SimGen command-line options

*System Generator* (SimGen) options, short forms, and descriptions.

<table>
<thead>
<tr>
<th>Option</th>
<th>Short form</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>--bridge-conf-file FILENAME</td>
<td>-</td>
<td>Set auto-bridging JSON configuration file <code>FILENAME</code>.</td>
</tr>
<tr>
<td>--build</td>
<td>-b</td>
<td>Build targets.</td>
</tr>
<tr>
<td>--build-directory DIR</td>
<td>-</td>
<td>Set build directory <code>DIR</code>.</td>
</tr>
<tr>
<td>--clean</td>
<td>-C</td>
<td>Clean targets.</td>
</tr>
<tr>
<td>--code-for-msvc</td>
<td>-m</td>
<td>Generate code and projects for Microsoft Visual Studio.</td>
</tr>
<tr>
<td>--config FILENAME</td>
<td>-</td>
<td>Set SimGen configuration file <code>FILENAME</code>. By default, <code>simgen.conf</code>.</td>
</tr>
<tr>
<td>--configuration NAME</td>
<td>-</td>
<td>The name for the configuration.</td>
</tr>
<tr>
<td>--cpp-flags-start</td>
<td>-</td>
<td>Ignore all parameters between this and <code>--cpp-flags-end</code>, except -D and -I.</td>
</tr>
<tr>
<td>--cpp-flags-end</td>
<td>-</td>
<td>See <code>--cpp-flags-start</code>.</td>
</tr>
<tr>
<td>--cxx-flags-start</td>
<td>-</td>
<td>Ignore all parameters between this and <code>--cxx-flags-end</code>, except -D.</td>
</tr>
<tr>
<td>--cxx-flags-end</td>
<td>-</td>
<td>See <code>--cxx-flags-start</code>.</td>
</tr>
<tr>
<td>--debug</td>
<td>-d</td>
<td>Enable debug mode.</td>
</tr>
<tr>
<td>--define SYMBOL</td>
<td>-D</td>
<td>Define preprocessor <code>SYMBOL</code>. You can also use <code>SYMBOL=DEF</code>.</td>
</tr>
<tr>
<td>--devenv-path ARG</td>
<td>-</td>
<td>Path to Visual C++ development environment, <code>devenv</code>.</td>
</tr>
<tr>
<td>--disable-warning NUM</td>
<td>-</td>
<td>Disable warning number <code>NUM</code>.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>This overrides the <code>--warning-level LEVEL</code> option.</td>
</tr>
<tr>
<td>--dumb-term</td>
<td>-</td>
<td>The terminal in which SimGen is running is dumb, so instead of fancy progress indicators, use simpler ones.</td>
</tr>
<tr>
<td>--enable-warning NUM</td>
<td>-</td>
<td>Enable warning number <code>NUM</code>.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>This overrides the <code>--warning-level LEVEL</code> option.</td>
</tr>
<tr>
<td>--gcc-path PATH</td>
<td>-</td>
<td>Under Linux, the GCC C++ compiler that builds the model. Passes the full path of the chosen <code>g++</code> executable to SimGen. Match this GCC version to the GCC version in the model configuration. By default, SimGen uses the <code>g++</code> in the search path.</td>
</tr>
<tr>
<td>--gen-sysgen-lib</td>
<td>-</td>
<td>Generate system library.</td>
</tr>
<tr>
<td>--help</td>
<td>-h</td>
<td>Print help message with a list of command-line options then exit.</td>
</tr>
<tr>
<td>--ignore-compiler-version</td>
<td>-</td>
<td>Do not stop on a compiler version mismatch. Try to build anyway.</td>
</tr>
<tr>
<td>--include INC_PATH</td>
<td>-I</td>
<td>Add include path <code>INC_PATH</code>.</td>
</tr>
</tbody>
</table>
Table 6-1  SimGen command-line options (continued)

<table>
<thead>
<tr>
<th>Option</th>
<th>Short form</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>--indir_tpl DIR</td>
<td>-</td>
<td>Set directory DIR where SimGen finds its template data files.</td>
</tr>
<tr>
<td>--link-against LIBS</td>
<td>-</td>
<td>Final executable will be linked against debug or release libraries. LIBS can be debug or release, does certain consistency checks.</td>
</tr>
<tr>
<td>--MSVC-debuginfo-type ARG</td>
<td>-</td>
<td>Set the debug info type for MSVC projects. ARG can be one of: none, /Zi, /Zd</td>
</tr>
<tr>
<td></td>
<td></td>
<td>none: No debug info. /Zi: Program Database. /Zd: Line numbers only.</td>
</tr>
<tr>
<td>--no-deploy</td>
<td>-</td>
<td>Prevent SimGen from copying deployed files from their original location to the location of the model. For example, when this option is specified, SimGen does not copy armctmodel.dll or libarmctmodel.so from the model library to the location of the generated model. This option is for advanced users who are building models in a batch system, or as part of another tool where they are taking responsibility for making sure all the required libraries are present.</td>
</tr>
<tr>
<td>--no-lineinfo</td>
<td>-c</td>
<td>Do not generate line number redirection in generated source and header files.</td>
</tr>
<tr>
<td>--num-build-cpus NUM</td>
<td>-</td>
<td>The number of processors used during the build.</td>
</tr>
<tr>
<td>--num-comps-file NUM</td>
<td>-</td>
<td>The number of components generated into one file.</td>
</tr>
<tr>
<td>--outdir_arch DIR</td>
<td>-</td>
<td>Set output directory DIR for file with variable filenames.</td>
</tr>
<tr>
<td>--outdir_fixed DIR</td>
<td>-</td>
<td>Set output directory DIR for file with constant filenames.</td>
</tr>
<tr>
<td>--override-config-parameter PARAM=VALUE</td>
<td>-P</td>
<td>Override the configuration parameter from the *.sgproj file.</td>
</tr>
<tr>
<td>--print-config</td>
<td>-</td>
<td>Print out configuration parameters in file .ConfigurationParameters.txt.</td>
</tr>
<tr>
<td>--print-preprocessor-output</td>
<td>-E</td>
<td>Print preprocessor output, then exit.</td>
</tr>
<tr>
<td>--print-resource-mapping</td>
<td>-</td>
<td>Print flat resource mapping when generating a simulator.</td>
</tr>
<tr>
<td>--project-file FILENAME</td>
<td>-p</td>
<td>Set SimGen project file FILENAME.</td>
</tr>
<tr>
<td>--replace-strings</td>
<td>-</td>
<td>Replace strings in files, then exit. Ignore binary files. Usage: simgen --replace-strings FOO BAR [FOO2 BAR2]... -- FILES...</td>
</tr>
<tr>
<td>--replace-strings-bin</td>
<td>-</td>
<td>Replace strings in files, then exit. Do not ignore binary files. Usage: simgen --replace-strings-bin FOO BAR [FOO2 BAR2]... -- FILES...</td>
</tr>
<tr>
<td>--top-component COMP</td>
<td>-</td>
<td>Top level component (system).</td>
</tr>
<tr>
<td>--user-MSVC-libs-start</td>
<td>-</td>
<td>Set additional libraries for MSVC projects. The list is terminated by --user-MSVC-libs-end.</td>
</tr>
</tbody>
</table>
Table 6-1 SimGen command-line options (continued)

<table>
<thead>
<tr>
<th>Option</th>
<th>Short form</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>--user-MSVC-libs-end</td>
<td>-</td>
<td>See --user-MSVC-libs-start.</td>
</tr>
<tr>
<td>--user-sourcefiles-start</td>
<td>-</td>
<td>Add source files listed between this option and --user-sourcefiles-end to the executable.</td>
</tr>
<tr>
<td>--user-sourcefiles-end</td>
<td>-</td>
<td>See --user-sourcefiles-start.</td>
</tr>
<tr>
<td>--verbose ARG</td>
<td>-v</td>
<td>Verbosity. ARG can be: on, sparse (default), off.</td>
</tr>
<tr>
<td>--version</td>
<td>-V</td>
<td>Print the version and exit.</td>
</tr>
<tr>
<td>--warning-level LEVEL</td>
<td>-w</td>
<td>Warning level LEVEL.</td>
</tr>
<tr>
<td>--warnings-as-errors</td>
<td>-</td>
<td>Treat LISA parsing and compiler warnings as errors.</td>
</tr>
</tbody>
</table>
6.3 Decreasing compilation time with `num-comps-file` and `num-build-cpus`

The `--num-comps-file` SimGen command-line option sets the maximum number of components for each C++ file. If the build system has multiple processors, you can use `--num-build-cpus` to distribute the build.

The `--num-comps-file` option specifies how many components SimGen places into each C++ file. The default is 1, which means SimGen places each component in a separate C++ file, along with generic code. For development work, where you frequently modify the source, setting `--num-comps-file` to 1 can reduce the compilation time after each change, because SimGen recompiles only the changed file.

Increasing the value of `--num-comps-file` reduces the total amount of generic code, so that the total time to compile the whole project is less. For example, if you increase `--num-comps-file` to a setting of 1000, then the first 1000 components go into the first file, the second 1000 components go into the second file, and so on until all components are in files.

--- Note ---

Large systems might be larger than the compiler can place in one C++ file. In this case, increase the value of `--num-comps-file`.

The `--num-build-cpus` option sets the maximum number of groups of files to split the build into. Have one group for each build system processor in order to fully use each processor. This makes the build quicker.

For example, if there are 40 components, and `--num-build-cpus` is set to 4, and `--num-comps-file` is set to 25, then SimGen produces four groups of files, each with 10 components. This is less than `--num-comps-file`, so each group contains only one C++ file. So:

- SimGen produces four C++ files.
- Each C++ file has 10 components instead of the maximum value of 25.

Using only the `--num-comps-file` option to calculate the components per file would result in two C++ files, with one file containing 25 components and the second file containing 15. This split would only use two build system processors. Accurately setting `--num-build-cpus` allows SimGen to fully use all processors.
6.4 Setting command-line options from System Canvas

You can also set the command-line options for SimGen using the System Canvas GUI.

Procedure
1. Select Preferences from the File menu.
2. If it is not selected, select the Applications tab.
3. Add the command-line options in the Command arguments text box in the Simulation Generator Executable panel.
Chapter 7
SystemC Export with Multiple Instantiation

This chapter describes the Fast Models SystemC Export feature with *Multiple Instantiation* (MI) support.

It contains the following sections:

- 7.1 *About SystemC Export with Multiple Instantiation* on page 7-116.
- 7.2 *Building a SystemC subsystem with System Canvas* on page 7-117.
- 7.3 *Adding a SystemC subsystem to a SystemC system* on page 7-122.
- 7.4 *Auto-bridging* on page 7-123.
- 7.5 *SystemC Export generated ports* on page 7-124.
- 7.6 *SystemC Export API* on page 7-128.
- 7.7 *Scheduler API* on page 7-146.
- 7.8 *SystemC Export limitations* on page 7-163.
7.1 About SystemC Export with Multiple Instantiation

SystemC Export wraps the components of a SystemC-based virtual platform into an Exported Virtual Subsystem (EVS). Multiple Instantiation (MI) enables the generation and integration of multiple EVS instances into a single SystemC simulation.

SystemC Export with MI enables the generation of EVSs as first-class SystemC components:

- Capable of running any number of instances, alongside other EVSs.
- Providing one `SC_THREAD` per core component (that is, one `SC_THREAD` per core component in a cluster Code Translation (CT) model).

MI enables the generation and integration of multiple EVS instances into a virtual platform with SystemC as the single simulation domain. A single EVS can appear in multiple virtual platforms. Equally, multiple EVSs can combine to create a single platform.

SystemC components (including Fast Models ones) can exchange data via the Direct Memory Interface (DMI) or normal (blocking) Transaction Level Modeling (TLM) transactions.

Fast Models supports SystemC 2.3.2, including integrated TLM 2.0.4. In this version, the TLM and SystemC headers are in the same place, and some filenames are different.

If you use them, set `SYSTEMC_HOME` and `TLM_HOME` to valid directories when running `simgen`. If the “include” directories do not exist when `simgen` runs, they cannot be in the include path.

--- Note ---

When loading an image on an EVS, you might see the following warning:

```
Warning: Base.cluster0.cpu0: Uncaught exception, thread terminated
In file: gen/scx_scheduler_mapping.cpp:523
In process: Base.thread_p_5 @ 0 s
```

This warning means that the image is attempting to run from DRAM, but this is access-controlled by the TZC_400 component. To disable security checking by the TZC_400, specify `-C Base.bp.secure_memory=false` when running the EVS.

---

Related information

- Accellera Systems Initiative (ASI)
7.2 **Building a SystemC subsystem with System Canvas**

This section describes how to build an *Exported Virtual Subsystem* (EVS) with System Canvas, using the Fast Models tool, System Generator.

This section contains the following subsections:

- **7.2.1 License implications** on page 7-117.
- **7.2.2 Building the EVS** on page 7-117.
- **7.2.3 Header files and libraries for Linux export** on page 7-118.
- **7.2.4 Header files and libraries for Microsoft Windows export** on page 7-119.

### 7.2.1 License implications

Arm recommends that you closely follow the instructions for building EVSs with System Generator, as well as for integrating them into virtual platforms, so that simulations behave as expected. Not following the instructions might affect the number of licenses that the simulation checks out.

Arm recommends statically linking the required libraries with the final virtual platform. Dynamic linking of those libraries is possible, but the built platforms require the check-out of additional licenses to run correctly.

**Related references**

- Libraries to build the virtual platform on Linux on page 7-119
- Libraries to build the virtual platform on Microsoft Windows on page 7-120

### 7.2.2 Building the EVS

This section describes how to build an *Exported Virtual Subsystem* (EVS).

System Canvas automatically uses the include path $SYSTEMC_HOME/include. On Windows, the Fast Models installer automatically sets the SYSTEMC_HOME environment variable. On Linux, you need to run the appropriate setup script. To use a different copy of SystemC, modify the variable before starting System Canvas.

**Procedure**

1. In System Canvas, select the target option SystemC component under the menu entry Project > Project Settings > Targets. To use auto-bridging, select both target options SystemC component and SystemC component with auto-bridging.
2. Add the ports you require on the SystemC component as normal Fast Models ports.
3. Add the headers and library files specified for your operating system.
4. Build the SystemC component outside of System Canvas by invoking System Generator.

**Example:**

```
simgen -p MySubsystem.sgproj --configuration AConfig -b
```

**Note**

SystemC executables that are made from the SystemCExport examples do not use the Post Build Actions for the linker in the Project Settings dialog. It is not possible to build and run these examples directly from System Canvas. Instead, the examples produce a SystemC executable by invoking System Generator from a Makefile. For Linux, your SystemC files are linked against the EVS generated files by the Makefile. For Microsoft Windows, the file is nMakefile.

**Results:**

The following filenames assume:

- MySubsystem is the name of the top-level component of the Fast Models subsystem.
- AConfig is the build configuration name, for example, Linux64-Release-GCC-5.4.

The built files are:
### Linux

- **Shared library**
  - libMySubsystem-AConfig.so.

- **Static library**
  - libscx-MySubsystem-AConfig.a.

- **Header file**
  - AConfig/gen/scx_evs_MySubsystem.h.

- **Static library**
  - libscx.a, containing the MI simulation infrastructure scheduler mapping, simulation controller, and report handler default implementations.

### Microsoft Windows

- **Shared library**
  - MySubsystem-AConfig.dll.

- **Static library**
  - MySubsystem-AConfig.lib.

- **Header file**
  - AConfig/gen/scx_evs_MySubsystem.h.

- **Static library**
  - scx.lib, containing the MI simulation infrastructure scheduler mapping, simulation controller, and report handler default implementations.

### Related references

7.2.3 Header files and libraries for Linux export on page 7-118
7.2.4 Header files and libraries for Microsoft Windows export on page 7-119
7.5 SystemC Export generated ports on page 7-124

#### 7.2.3 Header files and libraries for Linux export

This section describes additional header files and libraries for SystemC export. Some are for build time, and others are for building and packaging the final virtual platform.

**Header files and libraries to build the EVS on Linux**

The EVS requires specific header files and libraries.

- Fast Models header files from `$PVLIB_HOME/include/fmruntime/`.
- SystemC header files from `$SYSTEMC_HOME/include/`.
- AMBA-PV header files from `$MAXCORE_HOME/AMBA-PV/include/` if AMBA-PV ports are used.
- `$PVLIB_HOME/lib/Linux64_GCC-4.x.x/libfmruntime.a` for release builds using GCC x.x where x.x is 4.9, 5.4, or 6.4.
- `$PVLIB_HOME/lib/Linux64_GCC-4.x.x/libpvbus.a`.
- `$PVLIB_HOME/lib/Linux64_GCC-4.x.x/libcomponents.a`.
- Compiler and linker option: `-pthread`.

System Generator needs access to these header files and libraries for building. The tools automatically include and use them, without them being in the `MySubsystem.sgproj` project file. Depending on the EVS, more libraries might be necessary.

---

**Note**

On Windows, the Fast Models installer automatically sets the `SYSTEMC_HOME` environment variable. On Linux, you need to run the appropriate setup script.
Libraries to build the virtual platform on Linux

The virtual platform requires specific libraries in addition to the ones the EVS requires.

- Header file and libraries from the generation of the EVS:
  - Shared library libMySubsystem-AConfig.so.
  - Static library libscx-MySubsystem-AConfig.a.
  - Header file AConfig/gen/scx_evs_MySubsystem.h.
  - Static library libscx.a for the MI simulation infrastructure scheduler mapping, simulation controller, and report handler default implementations, in the directory where the EVS is built.
- $PVLIB_HOME/lib/Linux64_GCC-x.x/libpvbus.a for release builds using GCC x.x, where x.x is 4.9, 5.4, or 6.4.
- $PVLIB_HOME/lib/Linux64_GCC-x.x/libcomponents.a.
- $PVLIB_HOME/lib/Linux64_GCC-x.x/libarmctmodel.a.
- $PVLIB_HOME/lib/Linux64_GCC-x.x/libfmruntime.a.
- SystemC library from $SYSTEMC_HOME/lib/Linux64_GCC-x.x/libsystemc.a.
- Compiler and linker option: -pthread.

Note

On Windows, the Fast Models installer automatically sets the SYSTEMC_HOME environment variable. On Linux, you need to run the appropriate setup script.

Related tasks

7.2.2 Building the EVS on page 7-117

Libraries to package the virtual platform on Linux

The virtual platform requires specific libraries.

- Shared library from the generation of the EVS, libMySubsystem-AConfig.so
- libMAXCOREInitSimulationEngine.so.2
- libarmctmodel.so
- libSDL-1.2.so.0.11.4 (required only if your model uses the PL041 AACI component or any visualization components)
- librui_5.2.0.x64.so
- libarmfastmodelsanalytics.1.0.0.so

Depending on the example, additional libraries may be necessary (for example the EVS_LinuxBoot example requires the SDL library). The framework searches for such libraries alongside the virtual platform executable that loaded the EVS .so.

Related tasks

7.2.2 Building the EVS on page 7-117

7.2.4 Header files and libraries for Microsoft Windows export

This section describes additional header files and libraries for SystemC export. Some are for build time, and others are for building and packaging the final virtual platform.

Header files and libraries to build the EVS on Microsoft Windows

The EVS requires specific header files and libraries.

- Fast Models header files from %PVLIB_HOME%/include/fmruntime.
- SystemC header files from %SYSTEMC_HOME%/include.
- TLM header files from %TLM_HOME%/include/tlm.
- AMBA-PV header files from %MAXCORE_HOME%/AMBA-PV/include if AMBA-PV ports are used.
System Generator needs access to these header files and libraries for building. The tools automatically include/use them, without them being in the MySubsystem.sgproj project file. Depending on the EVS, additional libraries might be necessary.

--- Note ---

- Use the /vmg option in the project settings to correctly compile source code for use with SystemC.
- On Windows, the Fast Models installer automatically sets the SYSTEMC_HOME environment variable.
  On Linux, you need to run the appropriate setup script.

### Libraries to build the virtual platform on Microsoft Windows

The virtual platform requires specific libraries in addition to the ones the EVS requires.

- Shared library from the generation of the EVS, MySubsystem-AConfig.dll.
- Static library scx-MySubsystem-AConfig.lib.
- Static library scx.lib for the MI simulation infrastructure scheduler mapping, simulation controller, and report handler default implementations, in the directory where the EVS is built.
- libMAXCOREInitSimulationEngine.2.dll.
- armctmodel.dll.
- ruiSDK_5.2.0.x64.dll.
- SDL.dll (required only if your model uses the PL041 AACI component or any visualization components).

Depending on the example, additional libraries may be necessary (for example the EVS_LinuxBoot example requires the SDL library). The application searches for such libraries alongside the virtual platform executable that loaded the EVS .dll.
Related tasks

7.2.2 Building the EVS on page 7-117
Adding a SystemC subsystem to a SystemC system

This section describes how to add a generated SystemC subsystem to a SystemC virtual platform.

Procedure

1. Link the generated libraries to the existing SystemC virtual platform using the appropriate additional libraries.

2. Include the generated header file of the generated SystemC components into a suitable C++ file.

   Example:
   ```
   #include "scx_evs_MySubsystem.h"
   #include "scx_evs_MySubsystem2.h"
   ```

3. Initialize the MI simulation infrastructure for the virtual platform.

   Example:
   ```
   scx::scx_initialize("myPlatform");
   ```

4. Instantiate the generated SystemC component.

   Example:
   ```
   scx_evs_MySubsystem mySubsystem("mySubsystem");
   scx_evs_MySubsystem2 mySubsystem2("mySubsystem2");
   ```

5. Optionally, you might configure the simulation from the virtual platform executable command-line arguments for loading applications, setting arguments, for example.

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

6. Load application files into them.

   Example:
   ```
   scx::scx_load_application("mySubsystem.cpu0", "myApplication.elf");
   ```

   where cpu0 is the instance name of a core in the Fast Models subsystem mySubsystem. The DualDhrystone example in the SystemCExport directory uses this convention.

7. If required, set parameters for the configurable components of the Fast Models system.

   Example: To set parameter PAR of component instance cpu0 to value true:
   ```
   scx::scx_set_parameter("mySubsystem2.cpu0.PAR", true);
   ```

8. Bind the master port and slave ports (exports) of the generated SystemC component to the other components in the SystemC virtual platform. The generated ports are native SystemC ports, so binding the ports works in the same way as between all other SystemC components.

Related references

7.6.13 scx::scx_parse_and_configure on page 7-132
Libraries to build the virtual platform on Linux on page 7-119
Libraries to build the virtual platform on Microsoft Windows on page 7-120
7.4 Auto-bridging

Auto-bridging is a Fast Models feature that SimGen uses to automatically convert between LISA+ protocols and their SystemC equivalents. It helps to automate the generation of SystemC wrappers for LISA+ subsystem models.

A bridge is a LISA component that converts transactions from one protocol to another. A wide variety of bridges are available to convert between various LISA+ protocols and their SystemC equivalents. For example, PVBus2AMBA PV converts from PVBus to AMBA-PV protocols.

When auto-bridging is enabled, you do not need to manually add bridges to your LISA+ file. Auto-bridging causes SimGen to apply the protocol-to-bridge mappings that are defined in a configuration file to the LISA+ components and generate a single EVS component.

Enable auto-bridging by selecting both the TARGET_SYSTEMC and TARGET_SYSTEMC_AUTO build targets in the .sgproj file. Or, in System Canvas Project Settings, select both targets SystemC component and SystemC component with auto-bridging.

Use the --bridge-conf-file SimGen command-line option to select your own auto-bridging configuration file. Alternatively, edit the file $PVLIB_HOME/etc/bridges_conf.json, which SimGen uses if you do not specify this option. The syntax is:

```json
"<protocol_name>" : {
    "master" : {
        "name" : "<bridge_name>"
    },
    "slave" : {
        "name" : "<bridge_name>"
    },
    "peer" : {
        "name" : "<bridge_name>"
    }
},
```

--- Note ---

- SimGen ignores any bridges whose name is empty in the configuration file.
- Auto-bridging is not applied to any ports that are marked as internal in the LISA+ file.
- SimGen reports an error if auto-bridging is enabled and a top-level port in a LISA+ component uses a protocol that is not listed in the configuration file.
- SimGen reports an error if auto-bridging is enabled and it cannot find the configuration file.
- You do not need to specify bridges for the following LISA+ protocols. When ports that use these protocols are exported to SystemC, SimGen can automatically generate the TLM sockets for them, without the need for bridging:
  - AudioControl
  - CounterInterface
  - GICv3Comms
  - InstructionCount
  - KeyboardStatus
  - LCD
  - MouseStatus
  - VECBProtocol
7.5 SystemC Export generated ports

This section describes the generated ports and the associated port protocols.

This section contains the following subsections:

- 7.5.1 Protocol definition on page 7-124.
- 7.5.2 TLM 1.0 protocol for an exported SystemC component on page 7-124.
- 7.5.3 TLM 2.0 bus protocol for an exported SystemC component on page 7-124.
- 7.5.4 Properties for TLM 1.0 based protocols on page 7-125.
- 7.5.5 Properties for TLM 2.0 based protocols on page 7-126.

7.5.1 Protocol definition

The ports of the top level Fast Models component, used to create SystemC ports, have protocols.

The behaviors in a Fast Models protocol definition must match exactly the functions in the SystemC port class. System Canvas does not check this for consistency, but the C++ compiler can find inconsistencies when compiling the generated SystemC component.

The set of functions and behaviors, their arguments, and their return value must be the same. The order of the functions and behaviors does not matter.

All behaviors in the Fast Models protocol must be slave behaviors. There is no corresponding concept of master behaviors.

The protocol definition also contains a properties section that contains the properties that describe the SystemC C++ classes that implement the corresponding ports on the SystemC side.

Related information

7.5.2 TLM 1.0 protocol for an exported SystemC component

Here is an example of a TLM 1.0 signal protocol.

```cpp
protocol MySignalProtocol
{
    includes
    {
        #include <mySystemCClasses.h>
    }
    properties
    {
        sc_master_port_class_name = "my_signal_base<bool>";
        sc_slave_base_class_name = "my_slave_base<bool>";
        sc_slave_export_class_name = "my_slave_export<bool>";
    }
    slave behavior set_state(const bool & state);
}
```

7.5.3 TLM 2.0 bus protocol for an exported SystemC component

Here is an example of a TLM 2.0 bus protocol.

```cpp
protocol MyProtocol
{
    includes
    {
        #include <mySystemCClasses.h>
    }
    properties
    {
        sc_master_base_class_name = "my_master_base";
        sc_master_socket_class_name = "my_master_socket<64>";
        sc_slave_base_class_name = "my_slave_base<64>";
        sc_slave_socket_class_name = "my_slave_socket<64>";
    }
    slave behavior read(uint32_t addr, uint32_t &data);
    slave behavior write(uint32_t addr, uint32_t data);
}```
This protocol enables declaring ports that have read() and write() functions. This protocol can declare master and slave ports.

### 7.5.4 Properties for TLM 1.0 based protocols

TLM 1.0 based protocols map to their SystemC counterparts using properties in the LISA protocol definition. The protocol description must set these properties.

#### sc_master_port_class_name

The sc_master_port_class_name property is the class name of the SystemC class that the generated SystemC component instantiates for master ports on the SystemC side. This class must implement the functions defined in the corresponding protocol, for example:

```c
void my_master_port<bool>::set_state(bool state)
```

#### sc_slave_base_class_name

The sc_slave_base_class_name property is the class name of the SystemC class that the generated SystemC component specializes for slave ports on the SystemC side. This class must declare the functions defined in the corresponding protocol, for example:

```c
void my_slave_base<bool>::set_state(const bool &state)
```

The SystemC component must define it to forward the protocol functions from the SystemC component to the Fast Models top level component corresponding port. It must also provide a constructor taking the argument:

```c
const std::string &name
```

#### sc_slave_export_class_name

The sc_slave_export_class_name property is the class name of the SystemC class that the generated SystemC component instantiates for slave ports (exports) on the SystemC side. The component binds to the derived sc_slave_base_class_name SystemC class, and forwards calls from the SystemC side to the bound class.

### AMBAPV Signal protocol in Fast Models

```c
protocol AMBAPVSignal {
    includes {
        #include <amba_pv.h>
    }
    properties {
        description = "AMBA-PV signal protocol";
        sc_master_port_class_name = "amba_pv::signal_master_port<bool>";
        sc_slave_base_class_name = "amba_pv::signal_slave_port<bool>";
        sc_slave_export_class_name = "amba_pv::signal_slave_export<bool>";
    }
    ...
}
```

sc_slave_export_class_name and sc_master_port_class_name describe the type of the port instances in the SystemC domain.

sc_slave_base_class_name denotes the base class from which the SystemC component publicly derives.

### AMBAPV Signal protocol in SystemC component class

The SystemC module ports must use the corresponding names in the SystemC code.

```c
class pv_dma: public sc_module,
    public amba_pv::signal_slave_base<bool> {
```
The SystemC port names must also match the Fast Models port names. For example, `signal_out` is the instance name for the master port in the Fast Models AMBAPVBus component and the SystemC port.

**Properties for TLM 2.0 based protocols**

The TLM 2.0 protocol provides forward and backward paths for master and slave sockets. Protocols that use TLM 2.0 must specify properties in the protocol declaration.

**sc_master_socket_class_name**

This is the class name of the SystemC class that the generated SystemC component instantiates for master sockets on the SystemC side. The component binds to the derived `sc_master_base_class_name` SystemC class and forwards calls from:

- The bound class to SystemC (forward path).
- The SystemC side to the bound class (backward path).

**sc_master_base_class_name**

This is the class name of the SystemC class that the generated SystemC component specializes for master sockets on the SystemC side. This class must declare the master behavior functions defined in the corresponding protocol, for example:

```cpp
global_move_base::global_move_to(uint32_t addr)
```

The SystemC component must define it to forward the protocol functions from the SystemC component (backward path) to the System Generator top level component corresponding socket. It must also provide a constructor taking the argument:

```cpp
const std::string &
```

**sc_slave_socket_class_name**

This is the class name of the SystemC class that the generated SystemC component instantiates for slave sockets on the SystemC side. The component binds to the derived `sc_slave_base_class_name` SystemC class and forwards calls from:

- The bound class to SystemC (backward path).
- The SystemC side to the bound class (forward path).
sc_slave_base_class_name
This is the class name of the SystemC class that the generated SystemC component specializes for slave sockets on the SystemC side. It must also provide a constructor taking the argument:

```
const std::string &
```

**AMBAPV protocol in System Generator**

```cpp
protocol AMBAPVSignal {
  includes {
    #include <amba_pv.h>
  }

  properties {
    description = "AMBA-PV protocol";
    sc_master_base_class_name = "amba_pv::amba_pv_master_base";
    sc_master_socket_class_name = "amba_pv::amba_pv_master_socket<64>";
    sc_slave_base_class_name = "amba_pv::amba_pv_slave_base<64>";
    sc_slave_socket_class_name = "amba_pv::amba_pv_slave_socket<64>";
  }
}
```

**AMBAPV protocol in SystemC component class**

The SystemC module sockets must use the corresponding names in the SystemC code.

```cpp
class pv_dma: public sc_module,
    public amba_pv::amba_pv_slave_base<64>,
    public amba_pv::amba_pv_master_base {

  /* Module ports */
  amba_pv::amba_pv_slave_socket<64> amba_pv_s;
  amba_pv::amba_pv_master_socket<64> amba_pv_m;
  ...
}
```

**Related concepts**

*A.1 About SystemC Export generated ports on page Appx-A-219*
7.6 SystemC Export API

This section describes the SystemC eXport (SCX) API provided by Fast Models Exported Virtual Subsystems (EVSs). Each description of a class or function includes the C++ declaration and the use constraints.

This section contains the following subsections:

- 7.6.1 SystemC Export header file on page 7-129.
- 7.6.2 scx::scx_initialize on page 7-129.
- 7.6.3 scx::scx_load_application on page 7-129.
- 7.6.4 scx::scx_load_application_all on page 7-130.
- 7.6.5 scx::scx_load_data on page 7-130.
- 7.6.6 scx::scx_load_data_all on page 7-130.
- 7.6.7 scx::scx_set_parameter on page 7-131.
- 7.6.8 scx::scx_get_parameter on page 7-131.
- 7.6.9 scx::scx_get_parameter_list on page 7-132.
- 7.6.10 scx::scx_set_cpi_file on page 7-132.
- 7.6.11 scx::scx_cpulimit on page 7-132.
- 7.6.12 scx::scx_timelimit on page 7-132.
- 7.6.13 scx::scx_parse_and_configure on page 7-132.
- 7.6.14 scx::scx_start_cadi_server on page 7-134.
- 7.6.15 scx::scx_enable_cadi_log on page 7-135.
- 7.6.16 scx::scx_prefix_appli_output on page 7-135.
- 7.6.17 scx::scx_print_port_number on page 7-135.
- 7.6.18 scx::scx_print_statistics on page 7-135.
- 7.6.19 scx::scx_load_trace_plugin on page 7-135.
- 7.6.20 scx::scx_load_plugin on page 7-135.
- 7.6.21 scx::scx_get_global_interface on page 7-136.
- 7.6.22 scx::scx_evs_base on page 7-136.
- 7.6.23 scx::load_application on page 7-136.
- 7.6.24 scx::load_data on page 7-136.
- 7.6.25 scx::set_parameter on page 7-137.
- 7.6.26 scx::get_parameter on page 7-137.
- 7.6.27 scx::get_parameter_list on page 7-137.
- 7.6.28 scx::constructor on page 7-137.
- 7.6.29 scx::destructor on page 7-138.
- 7.6.30 scx::before_end_of_elaboration on page 7-138.
- 7.6.31 scx::end_of_elaboration on page 7-138.
- 7.6.32 scx::start_of_simulation on page 7-138.
- 7.6.33 scx::end_of_simulation on page 7-138.
- 7.6.34 scx::simcallback_if on page 7-138.
- 7.6.35 scx::notify_running on page 7-139.
- 7.6.36 scx::notify_stopped on page 7-139.
- 7.6.37 scx::notify_debuggable on page 7-139.
- 7.6.38 scx::notify_idle on page 7-139.
- 7.6.39 scx::simcallback_if_destructor on page 7-139.
- 7.6.40 scx::simcontrol_if on page 7-139.
- 7.6.41 scx::get_scheduler on page 7-140.
- 7.6.42 scx::get_report_handler on page 7-140.
- 7.6.43 scx::run on page 7-140.
- 7.6.44 scx::stop on page 7-140.
- 7.6.45 scx::is_running on page 7-140.
- 7.6.46 scx::stop_acknowledge on page 7-141.
- 7.6.47 scx::process_debuggable on page 7-141.
- 7.6.48 scx::process_idle on page 7-141.
- 7.6.49 scx::shutdown on page 7-141.
7.6.1 SystemC Export header file

To use the SystemC Export feature, an application must include this C++ header file at appropriate positions in the source code as required by the scope and linkage rules of C++.

The header file `$PVLIB_HOME/include/fmruntime/scx/scx.h` adds the namespace scx to the declarative region that includes it. This inclusion declares all definitions related to the SystemC Export feature of Fast Models within that region.

```cpp
#include "scx.h"
```

7.6.2 scx::scx_initialize

This function initializes the simulation.

Initialize the simulation before constructing any exported subsystem.

```cpp
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.

7.6.3 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`.
7.6.4 scx::scx_load_application_all

This function loads an application in the memory of instances that execute software, across all EVSs in the platform.

```cpp
void scx_load_application_all(const std::string &application);
```

application

the application to load.

Note

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

7.6.5 scx::scx_load_data

This function loads binary data in the memory of an instance at a memory address.

```cpp
void scx_load_data(const std::string &instance,
                   const std::string &data,
                   const std::string &address);
```

instance

the name of the instance to load into. The parameter `instance` must start with an EVS instance name, or with "*" to load data into the instance on all EVSs in the platform. On an SMP processor, if `instance` specifies "*" for the core instead of its index, the binary data loads only on the first processor.

data

the filename of the binary data to load.

address

the memory address at which to load the data. The parameter `address` might start with a memory space specifier.

Note

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

7.6.6 scx::scx_load_data_all

This function loads binary data in the memory of instances that execute software, across all EVSs in the platform, at a memory address. On SMP processors, the data loads only on the first core.

```cpp
void scx_load_data_all(const std::string &data,
                        const std::string &address);
```

data

the filename of the binary data to load.

address

the memory address at which to load the data. The parameter `address` might start with a memory space specifier.

Note

The loading of the binary data happens at `start_of_simulation()` call-back, at the earliest.
7.6.7 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.

7.6.8 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.
7.6.9 **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.

```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()`.

7.6.10 **scx::scx_set_cpi_file**

Sets the Cycles Per Instruction (CPI) file for CPI class functionality.

```cpp
void scx_set_cpi_file(const std::string & cpi_file_path);
```

**cpi_file_path**

the path to the CPI file.

Use this function to activate the CPI class functionality.

--- **Note** ---

This function must be called before any call to a platform parameter function.

7.6.11 **scx::scx_cpulimit**

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

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

**t**

the number of seconds to run. Defaults to unlimited.

7.6.12 **scx::scx_timelimit**

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

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

**t**

the number of seconds to run. Defaults to unlimited.

7.6.13 **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 the help message (-h option).

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

The application must pass the values of the options from function sc_main() as arguments to this function.

-a, --application
application to load, format: -a [INST=]FILE. For SMP cores: -a INST*=FILE.

-A, --iris-allow-remote
allow remote connections from another machine to the Iris server. Defaults to not allowed.

-b, --break
set a breakpoint, format: -b [INST=]ADDRESS. This option can be specified multiple times.

-C, --parameter
set a parameter, format: -C INST.PARAM=VALUE. This option can be specified multiple times.

--check-regs
the same as --list-regs but does more consistency checks on the CADI register API.

--cpi-file
use FILE to set Cycles Per Instruction (CPI) classes, format: --cpi-file FILE

--cpulimit
maximum number of CPU (User + System) seconds to run, excluding startup and shutdown, format: --cpulimit NUM. Defaults to unlimited.

--cyclelimit
number of cycles to run, ignored if the debug server has started, format: --cyclelimit NUM. Defaults to unlimited.

-D, --allow-debug-plugin
allow a plug-in to debug the simulation.

--data
raw data to load, format: --data [INST=]FILE@[MEMSPACE:]ADDRESS

--dump
dump a section of memory into FILE, format: --dump [INST=]FILE@[MEMSPACE:]ADDRESS,SIZE. This option can be specified multiple times.

-f, --config-file
load model parameters from configuration file FILE, format: --config-file FILE

-h, --help
print help message and exit.

-i, --iris-log
Iris log level. This option can be specified multiple times, for example: -ii for log level 2.

-I, --iris-server
start an Iris server, allowing debuggers to connect to targets in the simulation.

-K, --keep-console
keep the console window open after completion. This option applies to Microsoft Windows only.

-l, --list-params
print the list of platform parameters to standard output and exit.

-L, --cadi-log
log all CADI calls to XML log files.

--list-instances
print list of target instances to standard output.

--list-memory
print model memory information to standard output.

--list-regs
print model register information to standard output.

-o, --output
redirect parameters, memory and instance lists to output file FILE, format: --output FILE
-p, --print-port-number
    print the TCP port number the CADI server is listening to.
-P, --prefix
    prefix semihosting output with the name of the instance.
--plugin
    plug-in to load, format: --plugin [NAME=]FILE
-q, --quiet
    suppress informational output.
-r, --restore
    restore a checkpoint from DIR on simulation startup, format: --restore DIR
-R, --run
    run the simulation immediately after the CADI server starts.
-s, --save
    save a checkpoint to DIR on simulation exit, format: --save DIR
-S, --cadi-server
    start a CADI server, allowing debuggers to connect to targets in the simulation.
--simlimit
    maximum number of seconds to simulate, ignored if the debug server has started, format: --simlimit NUM. Defaults to unlimited.
--start
    set initial PC to application start address, format: --start [INST=]ADDRESS
--stat
    print run statistics on simulation exit.
-T, --timelimit
    maximum number of seconds to run, excluding startup and shutdown, ignored if the debug server has started, format: --timelimit NUM. Defaults to unlimited.
--trace-plugin
    deprecated, use --plugin instead.

This function treats all other command-line arguments as applications to load.
This function calls std::exit(EXIT_SUCCESS) to exit, for options --list-params and --help. 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.

7.6.14 scx::scx_start_cadi_server

This function specifies whether to start a CADI server.

```cpp
void scx_start_cadi_server(bool start = true, bool run = true, bool debug = false);
```

start
    true to start a CADI server, false otherwise.
run
    true to run the simulation immediately after CADI server has been started, false otherwise.
debug
    true to enable debugging through a plug-in, false otherwise.

Starting a CADI server enables the attachment of a debugger to debug targets in the simulation.
When debug is set to true, the CADI server does not start, but a plug-in can implement an alternative debugging mechanism in place of it.
When start is set to true, it overrides debug.

Note

- A CADI server cannot start once simulation starts.
- You do not need to call this function if you have called scx_parse_and_configure() and parsed at most one of -S or -D into sc_main().
7.6.15  **scx::scx_enable_cadi_log**

This function specifies whether to log all CADI calls to XML files.

```c
void scx_enable_cadi_log(bool log = true);
```

`log`
true to log CADI calls, false otherwise.

——— **Note** ———
You cannot enable logging once simulation starts.

7.6.16  **scx::scx_prefix_appli_output**

This function specifies whether to prefix semihosting output with the name of the CADI target instance.

```c
void scx_prefix_appli_output(bool prefix = true);
```

`prefix`
true to prefix semihosting output, false otherwise.

7.6.17  **scx::scx_print_port_number**

This function specifies whether to enable printing of the TCP port number that the CADI server is listening to.

```c
void scx_print_port_number(bool print = true);
```

`print`
true to enable printing of the TCP port number, false otherwise.

——— **Note** ———
You cannot enable printing of the TCP port number once simulation starts.

7.6.18  **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.

7.6.19  **scx::scx_load_trace_plugin**

Arm deprecates this function. Use `scx_load_plugin()` instead.

7.6.20  **scx::scx_load_plugin**

This function specifies a plug-in to load.

```c
void scx_load_plugin(const std::string &file);
```

`file`
the file of the plug-in to load.
The plug-in loads at `end_of_elaboration()`, at the latest, or as soon as a platform parameter function is called.

**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.

### 7.6.21 `scx::scx_get_global_interface`

This function accesses the global interface.

```cpp
eslapi::CAInterface *scx_get_global_interface();
```

The global interface allows access to all of the registered interfaces in the simulation.

### 7.6.22 `scx::scx_evs_base`

This class is the base class for EVSs. EVSs are the principal subsystems of the Fast Models SystemC Export feature.

```cpp
class scx_evs_base {
  public:
    void load_application(const std::string &, const std::string &);
    void load_data(const std::string &, const std::string &, const std::string &);
    bool set_parameter(const std::string &, const std::string &);
    template<class T>
    bool set_parameter(const std::string &, T);
    bool get_parameter(const std::string &, std::string &) const;
    template<class T>
    bool get_parameter(const std::string &, T &) const;
    std::string get_parameter(const std::string &) const;
    std::map<std::string, std::string> get_parameter_list() const;
  protected:
    scx_evs_base(const std::string &, sg::ComponentFactory *);
    virtual ~scx_evs_base();
    void before_end_of_elaboration();
    void end_of_elaboration();
    void start_of_simulation();
    void end_of_simulation();
};
```

### 7.6.23 `scx::load_application`

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

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

**Note**

The application loads at `start_of_simulation()`, at the earliest.

### 7.6.24 `scx::load_data`

This function loads raw data in the memory of an instance at a memory address.

```cpp
void load_data(const std::string &instance, const std::string &data, const std::string &address);
```

instance

the name of the instance to load into.
data
the file name of the raw data to load.

address
the memory address at which to load the raw data. The parameter address might start with a memory space specifier.

——— Note ————
The raw data loads at start_of_simulation(), at the earliest.

7.6.25 scx::set_parameter
This function sets the value of a parameter from components present in the EVS.

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

name
the name of the parameter to change.

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 set_parameter().
- You can set parameters during the construction phase, and before the elaboration phase. Calls to set_parameter() after the construction phase are ignored.
- You can change run-time parameters after the construction phase with the debug interface.

7.6.26 scx::get_parameter
This function retrieves the value of a parameter from components present in the EVS.

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

name
the name of the parameter to retrieve.

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.

7.6.27 scx::get_parameter_list
This function retrieves a list of all parameters in all components present in the EVS.

std::map<std::string, std::string> get_parameter_list();

7.6.28 scx::constructor
This function constructs an EVS.

scx_evs_base(const std::string &, sg::ComponentFactory *);
name

the name of the EVS instance.

factory

the `sg::ComponentFactory` to use to instantiate the corresponding LISA subsystem. The factory initializes within the generated derived class.

EVS instance names must be unique across the virtual platform. The EVS instance name initializes using the value passed as an argument to the constructor of the generated derived class.

### 7.6.29 `scx::destructor`

This function destroys an EVS including the corresponding subsystem, and frees the associated resources.

```
~scx_evs_base();
```

### 7.6.30 `scx::before_end_of_elaboration`

This function calls the `instantiate()`, `configure()`, `interconnect()`, and `populateCADIMap()` LISA behaviors of the corresponding exported subsystem.

```
void before_end_of_elaboration();
```

The generated derived class calls this function, after the SystemC simulation call-backs.

### 7.6.31 `scx::end_of_elaboration`

This function initializes the simulation framework.

```
void end_of_elaboration();
```

The generated derived class calls this function, after the SystemC simulation call-backs.

### 7.6.32 `scx::start_of_simulation`

This function calls the `reset()` LISA behaviors of the corresponding exported subsystem. It then loads applications.

```
void start_of_simulation();
```

The generated derived class calls this function, after the SystemC simulation call-backs.

### 7.6.33 `scx::end_of_simulation`

This function shuts down the simulation framework.

```
void end_of_simulation();
```

The generated derived class calls this function, after the SystemC simulation call-backs.

### 7.6.34 `scx::scx_simcallback_if`

This interface is the base class for simulation control call-backs.

```
class scx_simcallback_if {
    public:
        virtual void notify_running() = 0;
        virtual void notify_stopped() = 0;
        virtual void notify_debuggable() = 0;
        virtual void notify_idle() = 0;
    protected:
        virtual ~scx_simcallback_if() {}
};
```

The simulation framework implements this interface. The simulation controller uses the interface to notify the simulation framework of changes in the simulation state.
7.6.35 **scx::notify_running**

This function notifies the simulation framework that the simulation is running.

```cpp
void notify_running();
```

The simulation controller calls this function to notify the simulation framework that the simulation is running. The simulation framework then notifies debuggers of the fact.

7.6.36 **scx::notify_stopped**

This function notifies the simulation framework that the simulation has stopped.

```cpp
void notify_stopped();
```

The simulation controller calls this function to notify the simulation framework that the simulation has stopped. The simulation framework then notifies debuggers of the fact.

7.6.37 **scx::notify_debuggable**

This function notifies the simulation framework that the simulation is debuggable.

```cpp
void notify_debuggable();
```

The simulation controller periodically calls this function to notify that the simulation is debuggable. This typically occurs while the simulation is stopped, to allow clients to process debug activity, for instance memory or breakpoint operations.

This version of the function does nothing.

7.6.38 **scx::notify_idle**

This function notifies the simulation framework that the simulation is idle.

```cpp
void notify_idle();
```

The simulation controller periodically calls this function to notify the simulation framework that the simulation is idle, typically while the simulation is stopped, to allow clients to process background activity, for example, GUI events processing or redrawing.

7.6.39 **scx::simcallback_if destructor**

This function destroys simulation callback interfaces.

```cpp
~scx_simcallback_if();
```

This version of the function does not allow destruction of instances through the interface.

7.6.40 **scx::scx_simcontrol_if**

This is the simulation control interface.

```cpp
class scx_simcontrol_if {
    public:
        virtual eslapi::CAInterface *get_scheduler() = 0;
        virtual scx_report_handler_if *get_report_handler() = 0;
        virtual void run() = 0;
        virtual void stop() = 0;
        virtual bool is_running() = 0;
        virtual void stop_acknowledge(sg::SchedulerRunnable *runnable) = 0;
        virtual void process_debuggable() = 0;
        virtual void process_idle() = 0;
        virtual void shutdown() = 0;
        virtual void add_callback(scx_simcallback_if *callback_obj) = 0;
        virtual void remove_callback(scx_simcallback_if *callback_obj) = 0;
    protected:
        virtual ~scx_simcontrol_if();
};
```
The simulation controller, which interacts with the simulation framework, must implement this interface. The simulation framework uses this interface to access current implementations of the scheduler and report handler, as well as to request changes to the state of the simulation.

Unless otherwise stated, requests from this interface are asynchronous and can return immediately, whether the corresponding operation has completed or not. When the operation is complete, the corresponding notification must go to the simulation framework, which in turn notifies all connected debuggers to allow them to update their states.

Unless otherwise stated, an implementation of this interface must be thread-safe, that is it must not make assumptions about threads that issue requests.

The default implementation of the simulation controller provided with Fast Models is at: $MAXCORE_HOME/lib/template/tpl_scx_simcontroller.{h,cpp}.

### 7.6.41 scx::get_scheduler

This function returns a pointer to the implementation of the simulation scheduler.

```cpp
eslapi::CAInterface *get_scheduler();
```

The simulation framework calls the `get_scheduler()` function to retrieve the scheduler implementation for the simulation at construction time.

---

**Note**

Implementations of this function need not be thread-safe.

---

### 7.6.42 scx::get_report_handler

This function returns a pointer to the current implementation of the report handler.

```cpp
scx_report_handler_if *get_report_handler();
```

`scx_initialize()` calls the `get_report_handler()` function to retrieve the report handler implementation for the simulation at construction time.

---

**Note**

Implementations of this function need not be thread-safe.

---

### 7.6.43 scx::run

This function requests to run the simulation.

```cpp
void run();
```

The simulation framework calls `run()` upon receipt of a CADI run request from a debugger.

### 7.6.44 scx::stop

This function requests to stop the simulation as soon as possible, that is at the next `wait()`.

```cpp
void stop();
```

The simulation framework calls `stop()` upon receipt of a CADI stop request from a debugger, a component, or a breakpoint hit.

### 7.6.45 scx::is_running

This function returns whether the simulation is running.

```cpp
bool is_running();
```

The return value is true when the simulation is running, `false` when it is paused or stopped.
The simulation framework calls is_running() upon receipt of a CADI run state request from a debugger.

### 7.6.46 scx::stop_acknowledge

This function blocks the simulation while the simulation is stopped.

```c
void stop_acknowledge(sg::SchedulerRunnable *runnable);
```

`runnable`

a pointer to the scheduler thread calling stop_acknowledge().

The scheduler thread calls this function to effectively stop the simulation, as a side effect of calling stop() to request that the simulation stop.

An implementation of this function must call clearStopRequest() on runnable (when not NULL).

### 7.6.47 scx::process_debuggable

This function processes debug activity while the simulation is at a debuggable point.

```c
void process_debuggable()
```

This function is called by the scheduler thread whenever the simulation is at a debuggable point, to enable debug activity to be processed.

An implementation of this function might simply call scx_simcallback_if::notify_debuggable() on all registered clients.

This version of the function does nothing.

### 7.6.48 scx::process_idle

This function processes idle activity while the simulation is stopped.

```c
void process_idle();
```

The scheduler thread calls this function whenever idle to enable the processing of idle activity.

An implementation of this function might simply call scx_simcallback_if::notify_idle() on all registered clients.

### 7.6.49 scx::shutdown

This function requests to stop the simulation.

```c
void shutdown();
```

The simulation framework calls shutdown() to notify itself that it wants the simulation to stop. Once the simulation has shut down it cannot run again.

--- Note ---

There are no call-backs associated with shutdown().

---

### 7.6.50 scx::add_callback

This function registers call-backs with the simulation controller.

```c
void add_callback(scx_simcallback_if *callback_obj);
```

`callback_obj`

a pointer to the object whose member functions serve as call-backs.

Clients call this function to register with the simulation controller a call-back object that handles notifications from the simulation.
7.6.51 **scx::remove_callback**

This function removes call-backs from the simulation controller.

```c
void remove_callback(scx_simcallback_if *callback_obj);
```

callback_obj

a pointer to the object to remove.

Clients call this function to unregister a call-back object from the simulation controller.

7.6.52 **scx::destructor**

This function destroys simulation controller interfaces.

```c
~scx_simcontrol_if();
```

This version of the function does not allow destruction of instances through the interface.

7.6.53 **scx::scx_get_default_simcontrol**

This function returns a pointer to the default implementation of the simulation controller provided with Fast Models.

```c
scx_simcontrol_if *scx_get_default_simcontrol();
```

7.6.54 **scx::scx_report_handler_if**

This interface is the report handler interface.

```c
class scx_report_handler_if {
public:
    virtual void set_verbosity_level(int verbosity) = 0;
    virtual int get_verbosity_level() const = 0;
    virtual void report_info(const char *id,
        const char *file,
        int line,
        const char *fmt, ...) = 0;
    virtual void report_info _verb(int verbosity,
        const char *id,
        const char *file,
        int line,
        const char *fmt, ...) = 0;
    virtual void report_warning(const char *id,
        const char *file,
        int line,
        const char *fmt, ...) = 0;
    virtual void report_error(const char *id,
        const char *file,
        int line,
        const char *fmt, ...) = 0;
    virtual void report_fatal(const char *id,
        const char *file,
        int line,
        const char *fmt, ...) = 0;
protected:
    virtual ~scx_report_handler_if() {}
};
```

This interface provides run-time reporting facilities, similar to the ones provided by SystemC. It has the additional ability to specify a format string in the same way as the `std::vprintf()` function, and associated variable arguments, for the report message.

The Fast Models simulation framework for SystemC Export uses this interface to report various messages at run-time.

The default implementation of the report handler provided with Fast Models is in: `$MAXCORE_HOME/lib/template/tpl_scx_report.cpp`.

**Related information**

7.6.55 **scx::scx_get_default_report_handler**

This function returns a pointer to the default implementation of the report handler provided with Fast Models.

```cpp
scx_report_handler_if *scx_get_default_report_handler();
```

7.6.56 **scx::scx_get_curr_report_handler**

This function returns a pointer to the current implementation of the report handler.

```cpp
scx_report_handler_if *scx_get_curr_report_handler();
```

7.6.57 **scx::scx_sync**

This function adds a future synchronization point.

```cpp
void scx_sync(double sync_time);
```

- **sync_time**
  - the time of the future synchronization point relative to the current simulated time, in seconds.

SystemC components call this function to hint to the scheduler when a system synchronization point will occur.

The scheduler uses this information to determine the quantum sizes of threads.

Threads that have run their quantum are unaffected; all other threads (including the current thread) run to the `sync_time` synchronization point.

Calling `scx_sync()` again adds another synchronization point.

Synchronization points automatically vanish when the simulation time passes.

7.6.58 **scx::scx_set_min_sync_latency**

This function sets the minimum synchronization latency for this scheduler.

```cpp
void scx_set_min_sync_latency(double t);
void scx_set_min_sync_latency(sg::ticks_t t);
```

- **t**
  - the minimum synchronization latency. Measured in seconds.

The minimum synchronization latency helps to ensure that sufficient simulated time has passed between two synchronization points for synchronization to be efficient.

A small latency increases accuracy but decreases simulation speed.

A large latency decreases accuracy but increases simulation speed.

The scheduler uses this information to compute the next synchronization point as returned by `sg::SchedulerInterfaceForComponents::getNextSyncPoint()`.

**Related references**

7.6.59 **scx::scx_get_min_sync_latency** on page 7-143

7.6.59 **scx::scx_get_min_sync_latency**

This function returns the minimum synchronization latency, measured in seconds, for this scheduler.

```cpp
double scx_get_min_sync_latency();
sg::ticks_t scx_get_min_sync_latency(sg::Tag<sg::ticks_t> *);
```

**Related references**

7.6.58 **scx::scx_set_min_sync_latency** on page 7-143
7.6.60 **scx::scx_simlimit**

This function sets the maximum number of seconds to simulate.

```cpp
void scx_simlimit(double t);
```

`t` is the number of seconds to simulate. Defaults to unlimited.

7.6.61 **scx::scx_create_default_scheduler_mapping**

This function returns a pointer to a new instance of the default implementation of the scheduler mapping that is provided with Fast Models.

```cpp
sg::SchedulerInterfaceForComponents * scx_create_default_scheduler_mapping(scx_simcontrol_if * simcontrol);
```

`simcontrol` is a pointer to an existing simulation controller. When this is NULL, this function returns NULL.

7.6.62 **scx::scx_get_curr_scheduler_mapping**

This function returns a pointer to the current implementation of the scheduler mapping interface.

```cpp
sg::SchedulerInterfaceForComponents * scx_get_curr_scheduler_mapping();
```

7.6.63 **scx::scx_enable_iris_server**

This function specifies whether to start an Iris server. If a server is started, it listens on the first available port found in the range `DefaultIrisServerPortMin` to `DefaultIrisServerPortMax`.

```cpp
void scx_enable_iris_server(bool enable = true);
```

`enable` is true to start an Iris server (default), otherwise false.

7.6.64 **scx::scx_set_iris_server_port_range**

This function starts an Iris server and sets the range of ports to scan. The server uses the first available port found in the range.

```cpp
void scx_set_iris_server_port_range(uint16_t port_min, uint16_t port_max);
```

`port_min` is the port number at the start of the range.

`port_max` is the port number at the end of the range.

7.6.65 **scx::scx_set_iris_server_port**

This function starts an Iris server and sets a specific port for it to listen on.

```cpp
inline void scx_set_iris_server_port(uint16_t port)
```

`port` is the port number for the Iris server to listen on.

7.6.66 **scx::scx_enable_iris_log**

This function enables Iris message logging and specifies the log level.

```cpp
void scx_enable_iris_log(unsigned level = 0);
```

`level` is the log level.
the log level. The possible values are:

0
   Logging is disabled, the default.
1
   Log messages in JSON format.
2
   Additionally, log U64JSON data in hex.

7.6.67  scx::scx_get_iris_connection_interface
   This function retrieves the IrisConnectionInterface for the simulation. This can be used to create and register IrisInstances.

   iris::IrisConnectionInterface *scx_get_iris_connection_interface();

7.6.68  scx::scx_register_synchronous_thread
   This function registers a new thread in the simulation engine which is implicitly synchronized with the simulation thread.

   void scx_register_synchronous_thread(std::thread::id thread_id);

   thread_id
      ID of the newly registered thread.

   The caller must make sure that the simulation thread and the newly registered thread do not run concurrently.

   Calling this function for a thread $X$ completely disables the thread synchronization for thread $X$, that is, marshaling of function calls from the calling thread onto the simulation thread, for example Iris calls.

   This function is useful for debugger threads which are blocking the simulation thread and which still want to issue Iris calls while the simulation thread is blocked.
7.7 Scheduler API

This section describes the Fast Models Scheduler API. To explain the API, this section also describes the intended use, some simple use cases, and the relationship of this API to other APIs.

This section contains the following subsections:

- 7.7.1 Scheduler API - about on page 7-146.
- 7.7.2 Scheduler API - use cases and implementation on page 7-147.
- 7.7.3 sg::SchedulerInterfaceForComponents class on page 7-149.
- 7.7.4 sg::SchedulerRunnable class on page 7-155.
- 7.7.5 sg::SchedulerThread class on page 7-158.
- 7.7.6 sg::ThreadSignal class on page 7-159.
- 7.7.7 sg::Timer class on page 7-160.
- 7.7.8 sg::TimerCallback class on page 7-161.
- 7.7.9 sg::FrequencySource class on page 7-161.
- 7.7.10 sg::FrequencyObserver class on page 7-161.
- 7.7.11 sg::SchedulerObject class on page 7-161.
- 7.7.12 sg::scx_create_default_scheduler_mapping on page 7-162.
- 7.7.13 sg::scx_get_curr_scheduler_mapping on page 7-162.

7.7.1 Scheduler API - about

This API makes modeling components and systems accessible in different environments, with or without a built-in scheduler. Examples are a SystemC environment or a standalone simulator.

The Fast Models Scheduler API is a C++ interface consisting of a set of abstract base classes. The header file that defines these classes is $PVLIB_HOME/include/fmruntime/sg/SGSchedulerInterfaceForComponents.h. This header file depends on other header files under $PVLIB_HOME/include.

All Scheduler API constructs are in the namespace sg.

The interface decouples the modeling components from the scheduler implementation. The parts of the Scheduler API that the modeling components use are for the scheduler or scheduler adapter to implement. The parts that the scheduler or scheduler adapter use are for the modeling components to implement.

`class SchedulerInterfaceForComponents`

The scheduler (or an adapter to the scheduler) must implement an instance of this interface class for Fast Models components to work. Fast Models components use this interface to talk to the scheduler, for example, to create threads and timers. This class is the main part of the interface.

`class SchedulerThread`

An abstract Fast Models thread class, which `createThread()` creates instances of. For example, CT core models use this class. The scheduler implements it. Threads have co-routine semantics.

`class SchedulerRunnable`

The counterpart of the `SchedulerThread` class. The modeling components, which contain the thread functionality, implement it.

`class ThreadSignal`

A class of event that threads can wait on. It has `wait()` and `notify()` but no timing functions. The scheduler implements it.

`class Timer`

An abstract interface for one-shot or continuous timed events, which `createTimer()` creates instances of. The scheduler implements it.

`class TimerCallback`

The counterpart of the `Timer` class. The modeling components, which contain the functionality for the timer callback, implement it. Arm deprecates this class.
class SchedulerCallback
A callback function class. The modeling components, which use addCallback() (asynchronous callbacks), implement it.

class FrequencySource
An abstract interface class that provides a frequency in Hz. The modeling components implement it. The scheduler uses it to determine the time intervals for timed events. Arm deprecates this class.

class FrequencyObserver
An abstract interface class for observing a FrequencySource and changes to the frequency value. The scheduler implements it for objects that have access to a FrequencySource (Timer and SchedulerThread). Arm deprecates this class.

class SchedulerObject
The base class for all scheduler interface objects, which provides getName().

7.7.2 Scheduler API - use cases and implementation
This section describes uses of the Scheduler API.

Accessing the SchedulerInterfaceForComponents from within a modeling component
This section describes ways of accessing the global interfaces.

LISA component for accessing the SchedulerInterfaceForComponents
A way to access the global interfaces with getGlobalInterface().

```cpp
#include "sg/SGSchedulerInterfaceForComponents.h"
#include "sg/SGComponentRegistry.h"

behavior init
{
    sg::SchedulerInterfaceForComponents *scheduler = 
        sg::obtainComponentInterfacePointer<sg::SchedulerInterfaceForComponents>(getGlobalInterface(), "scheduler");
}
```

C++ component for accessing the SchedulerInterfaceForComponents
A way to access the global interfaces with simulationContext->getGlobalInterface(). C++ components have an sg::SimulationContext pointer passed into their constructor.

```cpp
#include "sg/SGSchedulerInterfaceForComponents.h"
#include "sg/SGComponentRegistry.h"

sg::SchedulerInterfaceForComponents *scheduler = 
    sg::obtainComponentInterfacePointer<sg::SchedulerInterfaceForComponents>(simulationContext->getGlobalInterface(), "scheduler");
```

SystemC component for accessing the SchedulerInterfaceForComponents
A way to access the global interfaces with scx::scx_get_global_interface().

```cpp
#include "sg/SGSchedulerInterfaceForComponents.h"
#include "sg/SGComponentRegistry.h"

sg::SchedulerInterfaceForComponents *scheduler = 
    sg::obtainComponentInterfacePointer<sg::SchedulerInterfaceForComponents>(scx::scx_get_global_interface(), "scheduler");
```
Using the default scheduler mapping in the SystemC export use case

Call the global function scx_initialize() to initialize the simulation infrastructure.

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

If you do not specify the ctrl parameter, the default implementations of the simulation controller and of the scheduler mapping onto SystemC apply.

--- Note ---

The namespace for interfaces, classes, and functions in this SystemC export case is scx, except for those of the Scheduler API.

---

Providing a custom mapping of the scheduler functionality onto SystemC

This section describes how to map the SchedulerInterfaceForComponents onto SystemC scheduling primitives by passing a custom system controller, scx::scx_simcontrol_if, as the second parameter, ctrl, into scx_initialize(). The system controller must return the custom scheduler mapping in get_scheduler().

Minimalistic example of a custom mapping of the scheduler functionality onto SystemC

This section describes how to register a custom scheduler mapping, using the default scheduler mapping for simplicity. A realistic scheduler mapper would reimplement all functionality.

It consists of:

- A custom scheduler mapping implementation, my_scheduler_mapping.
  - Forwards all calls to the default scheduler mapping.
  - The wait() function prints a verbose message in addition, to make the effect visible.
- A custom simulation controller implementation, my_simulation_controller.
  - Forwards all calls to the default scx::scx_simcontrol_if implementation.
  - Implements only get_scheduler() differently and returns an instance of my_scheduler_mapping.
- Creating an instance of my_simulation_controller, my_sim_controller.
- Passing a pointer to my_sim_controller to scx_initialize() as the second parameter, ctrl.

This example adds a verbose message to sg::SchedulerInterfaceForComponents::wait() calls.

Intended mapping of the Scheduler API onto SystemC/TLM

How Scheduler API functionality might map onto SystemC functionality.

sg::SchedulerInterfaceForComponents::wait(time)

Call sc_core::wait(time) and handle all pending asynchronous events that are scheduled with sg::SchedulerInterfaceForComponents::addCallback() before waiting.

sg::SchedulerInterfaceForComponents::wait(sg::ThreadSignal)

Call sc_core::wait(sc_event) on the sc_event in sg::ThreadSignal and handle all pending asynchronous events that are scheduled with sg::SchedulerInterfaceForComponents::addCallback() before waiting.

sg::SchedulerInterfaceForComponents::getCurrentSimulatedTime()

Return the current SystemC scheduler time in seconds as in sc_core::sc_time_stamp().to_seconds().

sg::SchedulerInterfaceForComponents::addCallback(), removeCallback()

SystemC has no way to trigger simulation events from alien (non-SystemC) host threads in a thread-safe way: buffer and handle these asynchronous events in all regularly re-occurring scheduler events. Handling regular simulation wait() and timerCallback() calls is sufficient.
sg::SchedulerInterfaceForComponents::stopRequest(), stopAcknowledge()
Pause and resume the SystemC scheduler. This function is out of scope of SystemC/TLM functionality, but in practice every debuggable SystemC implementation has ways to pause and resume the scheduler. Do not confuse these functions with sc_core::sc_stop(), which exits the SystemC simulation loop. They work with the sg::SchedulerRunnable instances and the scx::scx_simcontrol_if interface.

sg::SchedulerInterfaceForComponents::createThread(), createThreadSignal(), createTimer()
Map these functions onto SystemC threads created with sc_spawn() and sc_events. You can create and destroy sg::SchedulerThread, sg::ThreadSignal, and sg::Timer objects during elaboration, and delete them at runtime, unlike their SystemC counterparts. This process requires careful mapping. For example, consider what happens when you remove a waited-for sc_event.

sg::ThreadSignal
Map onto sc_event, which is notifiable and waitable.

sg::SchedulerThread
Map onto a SystemC thread that was spawned with sc_core::sc_spawn(). The thread function can call sg::SchedulerThread::threadProc().

sg::QuantumKeeper
Map onto the tlm_quantumkeeper utility class because the semantics of these classes are similar. Arm deprecates this class.

sg::Timer
Map onto a SystemC thread that, after the timer is set(), issues calls to the call-backs in the intervals (according to the set() interval).

7.7.3 sg::SchedulerInterfaceForComponents class
This section describes the main scheduler interface class.

About sg::SchedulerInterfaceForComponents
The modeling components use this interface class, which gives access to all other parts of the Scheduler API, directly or indirectly. The scheduler must implement this class.

```cpp
// Main scheduler interface class
class sg::SchedulerInterfaceForComponents {
public:
    static eslapi::if_name_t IFNAME() { return "sg.SchedulerInterfaceForComponents"; }
    static eslapi::if_rev_t IFREVISION() { return 1; }
    virtual eslapi::CAInterface * ObtainInterface(eslapi::if_name_t, eslapi::if_rev_t, eslapi::if_rev_t *) = 0;
    virtual sg::Timer * createTimer(const char *, sg::TimerCallback *) = 0;
    virtual sg::SchedulerThread * createThread(const char *, sg::SchedulerRunnable *) = 0;
    virtual sg::SchedulerThread * currentThread();
    virtual sg::ThreadSignal * createThreadSignal(const char *) = 0;
    virtual void wait(sg::ticks_t);
    virtual void wait(sg::ThreadSignal *) = 0;
    virtual void setGlobalQuantum(sg::ticks_t);
    virtual sg::ticks_t getGlobalQuantum(sg::Tag<sg::ticks_t> *);
    virtual double getGlobalQuantum();
    virtual void setMinSyncLatency(sg::ticks_t);
    virtual sg::ticks_t getMinSyncLatency(sg::Tag<sg::ticks_t> *);
    virtual double getMinSyncLatency();
    virtual void addSynchronisationPoint(sg::ticks_t);
    virtual sg::ticks_t getNextSyncPoint(sg::Tag<sg::ticks_t> *);
    virtual sg::ticks_t getNextSyncPoint();
    virtual double getNextSyncRange(sg::ticks_t &, sg::ticks_t &);
    virtual void setSimulatedTimeResolution(double);
    virtual void setSimulatedTimeResolution(double resolution);
    virtual void stopRequest();
};
```
virtual void stopAcknowledge(sg::SchedulerRunnable *) = 0;
}

--- Note ---

Pass a null pointer to the extra Tag<> argument in getGlobalQuantum(), getMinSyncLatency(),
getNextSyncPoint(), and getCurrentSimulatedTime.

---

Arm deprecates these API functions:

virtual void wait(sg::ticks_t, sg::FrequencySource *)
virtual void setGlobalQuantum(sg::ticks_t, sg::FrequencySource *)
virtual void setMinSyncLatency(sg::ticks_t, sg::FrequencySource *)
virtual void addSynchronisationPoint(sg::ticks_t, sg::FrequencySource *)

Arm deprecates classes sg::FrequencySource and sg::FrequencyObserver. Modeling components
must not use these classes to directly communicate with the Scheduler API. Use the sg::Time class
instead.

Modeling components use this interface to create threads, asynchronous and timed events, system
synchronization points, and to request a simulation stop. Examples of components that access this
interface are:
• CT core models.
• Timer peripherals.
• Peripheral components with timing or that indicate system synchronization points.
• Peripheral components that can stop the simulation for certain conditions (external breakpoints).
• GUI components.

Passive components that do not interact with the scheduler (and that do not need explicit scheduling)
usually do not access this interface.

Related references
Accessing the SchedulerInterfaceForComponents from within a modeling component on page 7-147
Providing a custom mapping of the scheduler functionality onto SystemC on page 7-148

eslapi::CAInterface and eslapi::ObtainInterface

The CAInterface base class and the ObtainInterface() function make the interface discoverable at
runtime through a runtime mechanism. All interfaces in Fast Models that must be discoverable at runtime
derive from CAInterface.

The functions IFNAME(), IFREVISION(), and ObtainInterface() belong to the base class
eslapi::CAInterface. IFNAME() and IFREVISION() return static information (name and revision)
about the interface (not the interface implementation). An implementation of the interface cannot re-
implement these functions. To access this interface, code must pass these two values to the
ObtainInterface() function to acquire the SchedulerInterfaceForComponents

Use ObtainInterface() to access the interfaces that the scheduler provides. As a minimum
requirement, the implementation of ObtainInterface() must provide the
SchedulerInterfaceForComponents interface itself and also the eslapi::CAInterface interface. The
easiest way to provide these interfaces to use the class eslapi::CAInterfaceRegistry and register
these two interfaces and forward all ObtainInterface() calls to this registry. See the default
implementation of the Scheduler API over SystemC for an example.

--- Note ---

CAInterface and ObtainInterface() are not part of the scheduler functionality but rather of the
simulation infrastructure. The information here is what is necessary to understand and implement
ObtainInterface(). For more details on the eslapi::CAInterface class, see the header file
$PVLIB_HOME/include/fmruntime/eslapi/CAInterface.h.
sg::SchedulerInterfaceForComponents::addCallback

This method schedules a callback in the simulation thread. AsyncSignal uses it.

```cpp
virtual void addCallback(SchedulerCallback *callback)=0;
```

callback

Callback object to call. If callback is NULL, the call has no effect.

Any host thread can call this method. It is thread safe. It is always the simulation thread (host thread which runs the simulation) that calls the callback function (callback->schedulerCallback()). The scheduler calls the callback function when it can respond to the addCallback() function.

Multiple callbacks might be pending. The scheduler can call them in any order. Do not call addCallback() or removeCallback() from a callback function.

Callbacks automatically vanish once called. Removing them deliberately is not necessary unless they become invalid, for example on the destruction of the object implementing the callback function.

**Related references**
sg::SchedulerInterfaceForComponents::removeCallback on page 7-153

sg::SchedulerInterfaceForComponents::addSynchronisationPoint

This method adds synchronization points.

```cpp
virtual void addSynchronisationPoint(ticks_t ticks);
```

ticks

Simulated time for synchronization relative to the current simulated time, in ticks relative to simulated time resolution.

Modeling components can call this function to hint to the scheduler when a potentially useful system synchronization point will occur. The scheduler uses this information to determine the quantum sizes of threads.

Calling this function again adds another synchronization point.

Synchronization points automatically vanish when reached.

sg::SchedulerInterfaceForComponents::createThread

CT core models and modeling components call this method to create threads. This method returns an object implementing SchedulerThread. (Not NULL except when runnable is NULL.)

```cpp
virtual SchedulerThread *createThread(const char *name, SchedulerRunnable *runnable)=0;
```

name

Instance name of the thread. Ideally, the hierarchical name of the component that owns the thread is included in the name. If name is NULL, it receives the name '(anonymous thread)'. The function makes a copy of name.

runnable

Object that implements the SchedulerRunnable interface. This object is the one that contains the actual thread functionality. The returned thread uses this interface to communicate with the thread implementation in the modeling component. If runnable is NULL, the call returns NULL, which has no effect.

Having created the thread, start it with a call to SchedulerThread::start().

Destroying the returned object with the SchedulerThread destructor might not kill the thread.

**Related concepts**
sg::SchedulerRunnable - about on page 7-155
sg::SchedulerThread - about on page 7-158
**Related references**

sg::SchedulerInterfaceForComponents::currentThread on page 7-152
sg::SchedulerThread::destructor on page 7-158
sg::SchedulerThread::start on page 7-159

sg::SchedulerInterfaceForComponents::createThreadSignal

CT core models use this method to create thread signals. A thread signal is a nonschedulable event that threads wait for. Giving the signal schedules all waiting threads to run.

```cpp
virtual ThreadSignal* createThreadSignal(const char* name)=0;
```

- **name**

  Instance name of the thread. Ideally, the hierarchical name of the component that owns the thread is included in the name. If `name` is `NULL`, it receives the name '`(anonymous thread signal)'`. The function makes a copy of `name`.

Destroying the returned object while threads are waiting for it leaves the threads unscheduled.

sg::SchedulerInterfaceForComponents::createTimer

Modeling components call this method to create objects of class `Timer`. They use timers to trigger events in the future (one-shot or repeating events).

```cpp
virtual Timer* createTimer(const char* name, TimerCallback* callback)=0;
```

sg::SchedulerInterfaceForComponents::currentThread

This method returns the currently running scheduler thread, which `createThread()` created, or null if not in any `threadProc()` call.

```cpp
virtual SchedulerThread* currentThread();
```

**Related references**

sg::SchedulerInterfaceForComponents::createThread on page 7-151

sg::SchedulerInterfaceForComponents::getCurrentSimulatedTime

This method returns the simulated time in ticks relative to simulated time resolution, since the creation of the scheduler. `ClockDivider` and `MasterClock(ClockSignalProtocol::currentTicks())` use it.

```cpp
virtual ticks_t getCurrentSimulatedTime(Tag<ticks_t>*);
```

This clock accurately reflects the time on the last timer callback invocation or the last return from `SchedulerThread::wait()`, whichever was last. The return values monotonically increase over (real or simulated) time.

sg::SchedulerInterfaceForComponents::getGlobalQuantum

This method returns the global quantum in ticks relative to simulated time resolution.

```cpp
virtual ticks_t getGlobalQuantum(Tag<ticks_t>*);
```

**Related references**

sg::SchedulerInterfaceForComponents::setGlobalQuantum on page 7-153

sg::SchedulerInterfaceForComponents::getMinSyncLatency

This method returns the minimum synchronization latency in ticks relative to simulated time resolution.

```cpp
virtual ticks_t getMinSyncLatency(Tag<ticks_t>*);
```

**Related references**

sg::SchedulerInterfaceForComponents::setMinSyncLatency on page 7-153
**sg::SchedulerInterfaceForComponents::getNextSyncPoint**

This method returns the next synchronization point relative to the current simulated time. The next synchronization point is expressed in ticks relative to simulated time resolution.

```cpp
virtual ticks_t getNextSyncPoint(Tag<ticks_t>*);
```

Modeling components can call this function for a hint about when a potentially useful system synchronization point will occur. Core threads use this information to determine when to synchronize.

**sg::SchedulerInterfaceForComponents::getSimulatedTimeResolution**

This method returns the simulated time resolution in seconds.

```cpp
virtual double getSimulatedTimeResolution();
```

**sg::SchedulerInterfaceForComponents::removeCallback**

This method removes all callbacks that are scheduled using `addCallback()` for this callback object. AsyncSignal uses it.

```cpp
virtual void removeCallback(SchedulerCallback *callback)=0;
```

**callback**

The callback object to remove. If `callback` is NULL, an unknown callback object, or a called callback, then the call has no effect.

Any host thread can call this method. It is thread safe.

The scheduler will not call the specified callback after this function returns. It can, however, call it while execution control is inside this function.

Callbacks automatically vanish after being called. Removing them deliberately is not necessary unless they become invalid, for example on the destruction of the object implementing the callback function.

*Related references*

- `sg::SchedulerInterfaceForComponents::addCallback` on page 7-151

**sg::SchedulerInterfaceForComponents::setGlobalQuantum**

This method sets the global quantum.

```cpp
virtual void setGlobalQuantum(ticks_t ticks);
```

**ticks**

Global quantum value, relative to simulated time resolution. The global quantum is the maximum time that a thread can run ahead of simulation time.

All threads must synchronize on timing points that are multiples of the global quantum.

*Related references*

- `sg::SchedulerInterfaceForComponents::getGlobalQuantum` on page 7-152

**sg::SchedulerInterfaceForComponents::setMinSyncLatency**

This method sets the minimum synchronization latency.

```cpp
virtual void setMinSyncLatency(ticks_t ticks);
```

**ticks**

Minimum synchronization latency value, relative to simulated time resolution.

The minimum synchronization latency helps to ensure that sufficient simulated time has passed between two synchronization points for synchronization to be efficient. A small latency increases accuracy but decreases simulation speed. A large latency decreases accuracy but increases simulation speed.
The scheduler uses this information to set the minimum synchronization latency of threads with `sg::SchedulerRunnable::setThreadProperty()`, and to compute the next synchronization point as returned by `getNextSyncPoint()`.

**Related references**

*sg::SchedulerInterfaceForComponents::getMinSyncLatency* on page 7-152

### sg::SchedulerInterfaceForComponents::setSimulatedTimeResolution

This method sets the simulated time resolution in seconds.

```cpp
virtual void setSimulatedTimeResolution(double resolution);
```

**resolution**

Simulated time resolution in seconds.

Setting simulated time resolution after the start of the simulation or after setting timers is not possible.

### sg::SchedulerInterfaceForComponents::stopAcknowledge

This function blocks the simulation thread until being told to resume.

```cpp
virtual void stopAcknowledge(SchedulerRunnable *runnable)=0;
```

**runnable**

Pointer to the runnable instance that called this function, or `NULL` when not called from a runnable instance. If not `NULL` this function calls `runnable->clearStopRequest()` once it is safe to do so (with respect to non-simulation host threads).

CT core models call this function from within the simulation thread in response to a call to `stopRequest()` or spontaneously (for example, breakpoint hit, debugger stop). The call must always be from the simulation thread. The scheduler must block inside this function. The function must return when the simulation is to resume.

The scheduler usually implements a thread-safe mechanism in this function that allows blocking and resuming of the simulation thread from another host thread (usually the debugger thread).

Calling this function from a nonsimulation host thread is wrong by design and is forbidden.

This function must clear the stop request that led to calling this function by calling `runnable->clearStopRequest()`.

This function must have no effects other than blocking the simulation thread.

### sg::SchedulerInterfaceForComponents::stopRequest

This function requests the simulation of the whole system to stop (pause).

```cpp
virtual void stopRequest()=0;
```

You can call this function from any host thread, whether the simulation is running or not. The function returns immediately, possibly before the simulation stops. This function will not block the caller until the simulation stops. The simulation stops as soon as possible, depending on the `syncLevel` of the threads in the system. The simulation calls the function `stopAcknowledge()`, which blocks the simulation thread to pause the simulation. This function must not call `stopAcknowledge()` directly. It must only set up the simulation to stop at the next sync point, defined by the `syncLevels` in the system. Reset this state with `stopAcknowledge()`, which calls `SchedulerRunnable::clearStopRequest()`.

Debuggers and modeling components such as CT cores and peripherals use this function to stop the simulation from within the simulation thread (for example for external breakpoints) and also asynchronously from the debugger thread. Calling this function again (from any host thread) before
stopAcknowledge() has reset the stop request, using SchedulerRunnable::clearStopRequest() is harmless. The simulation only stops once.

Note

The simulation can stop (that is, call stopAcknowledge()) spontaneously without a previous stopRequest(). This stop happens for example when a modeling component hits a breakpoint. A stopRequest() is sufficient, but not necessary, to stop the simulation.

The scheduler implementation of this function is to forward this stopRequest() to the running runnable object, but only for stopRequest() calls from the simulation thread. When the runnable object accepts the stopRequest() (SchedulerRunnable::stopRequest() returns true), the scheduler need do nothing more because the runnable object will respond with a stopAcknowledge() call. If the runnable object did not accept the stopRequest() (SchedulerRunnable::stopRequest() returns false) or if this function call is outside of the context of a runnable object (for example, from a call-back function) or from a non-simulation host thread, then the scheduler is responsible for handling the stopRequest() itself by calling stopAcknowledge() as soon as possible.

The stop handling mechanism should not change the scheduling order or model behavior (non-intrusive debugging).

Related references

sg::SchedulerRunnable::stopRequest on page 7-158

sg::SchedulerInterfaceForComponents::wait(ThreadSignal)

This method waits on a thread signal.

```cpp
virtual void wait(ThreadSignal* threadSignal)=0;
```

**threadSignal**

Thread signal object to wait for. A call with threadSignal of NULL is valid, but has no effect.

wait() blocks the current thread until it receives ThreadSignal::notify(). This function returns when the calling thread can continue to run.

Only call this method from within a SchedulerRunnable::threadProc() context. Calling this method from outside of a threadProc() context is valid, but has no effect.

sg::SchedulerInterfaceForComponents::wait(ticks_t)

This method blocks the running thread and runs other threads for a specified time.

```cpp
virtual void wait(ticks_t ticks);
```

**ticks**

Time to wait for, in timebase units. ticks can be 0.

Only call this method from within a SchedulerRunnable::threadProc() context. Calls from outside of a threadProc() context are valid, but have no effect.

This method blocks a thread for a time while the other threads run. It returns when the calling thread is to continue, at the co-routine switching point. Typically, a thread calls wait(ticks) in its loop when it completes ticks ticks of work. ticks is a “quantum”.

7.7.4 sg::SchedulerRunnable class

This section describes the SchedulerRunnable class.

sg::SchedulerRunnable - about

This class is a thread interface on the runnable side. The modeling components create and implement SchedulerRunnable objects and pass a pointer to a SchedulerRunnable interface to
SchedulerInterfaceForComponents::createThread(). The scheduler uses this interface to run the thread.

**Related references**
*sg::SchedulerInterfaceForComponents::createThread on page 7-151*

**sg::SchedulerRunnable::breakQuantum**
This function breaks the quantum. Arm deprecates this function.

**sg::SchedulerRunnable::clearStopRequest**
This function clears stop request flags.

```cpp
void clearStopRequest();
```

Only SchedulerInterfaceForComponents::stopAcknowledge() calls this function, so calls are always from the simulation thread.

**Related references**
*sg::SchedulerRunnable::stopRequest on page 7-158*

**sg::SchedulerRunnable::getName**
This function returns the name of the instance that owns the object.

```cpp
const char *getName() const;
```

By convention, this is the name that createThread() received. SchedulerRunnable inherits this function from sg::SchedulerObject.

**sg::SchedulerRunnable::setThreadProperty, sg::SchedulerRunnable::getThreadProperty**

These functions set and get thread properties.

```cpp
bool setThreadProperty(ThreadProperty property, uint64_t value);
based on the property, this can return false.
```

**Scheduler-configures runnable properties**

<table>
<thead>
<tr>
<th>Property</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>TP_BREAK_QUANTUM</td>
<td>Arm deprecates this property. SchedulerInterfaceForComponents::getNextSyncPoint() gives the next quantum size.</td>
</tr>
<tr>
<td>TP_DEFAULT_QUANTUM_SIZE</td>
<td>Arm deprecates this property. Use SchedulerInterfaceForComponents::set/getGlobalQuantum().</td>
</tr>
<tr>
<td>TP_COMPILER_LATENCY</td>
<td>set Compiler latency, the maximum interval in which generated straight-line code checks for signals and the end of the quantum.</td>
</tr>
<tr>
<td></td>
<td>get Compiler latency.</td>
</tr>
<tr>
<td></td>
<td>default 1024 instructions.</td>
</tr>
<tr>
<td>TP_MIN_SYNC_LATENCY</td>
<td>set Synchronization latency, the minimum interval in which generated straight-line code inserts synchronization points.</td>
</tr>
</tbody>
</table>
get
Synchronization latency.
default
64 instructions.

TP_MIN_SYNC_LEVEL
set
syncLevel to at least $N$ (0-3).

get
Minimum syncLevel.
default
min_sync_level CADI parameter and the syncLevel* registers also determine the syncLevel. If nothing else is set, the default is 0 (SL_OFF).

TP_LOCAL_TIME
set
Local time of temporally decoupled thread.

get
Current local time.

TP_LOCAL_QUANTUM
set
Local quantum of temporally decoupled thread.

get
Current local quantum.

Note
The temporarily decoupled thread usually retrieves the local quantum by calling SchedulerInterfaceForComponents::getNextSyncPoint().

Runnable-configures-scheduler properties
TP_STACK_SIZE
set
Return false and ignore the value. Not for a scheduler to call.

get
Intended stack size for the thread in bytes. If this field returns false or a low value, this field uses the default stack size that the scheduler determines. Not all schedulers use this field. If a scheduler supports setting the stack size, it requests this field from SchedulerInterfaceForComponents::createThread() or SchedulerThread::start(). Is to return a constant value.

default
2MB.

Schedulers need not use all fields, and runnable objects need not provide all fields. If a runnable object does not support a property or value, it must return false.

Related references
sg::SchedulerRunnable::breakQuantum on page 7-156
sg::SchedulerRunnable::stopRequest

This function requests the simulation of the whole system to stop (pause) as soon as possible by setting a request flag. This might be to inspect a runnable, for example to pause at an instruction boundary to inspect a processor component with a debugger.

```cpp
bool stopRequest();
```

You can call this function from any host thread, whether the simulation is running or not. The function returns immediately, before the simulation stops. This function will not block the caller until the simulation stops. The simulation stops as soon as possible, depending on the `syncLevel` of the runnable. The simulation calls the function `SchedulerInterfaceForComponents::stopAcknowledge()`, which blocks the simulation thread to pause the simulation. The function must not call `stopAcknowledge()` directly but only set up a state such that the simulation stops at the next sync point, defined by the `syncLevel` of this runnable. Reset this state with `stopAcknowledge()`, which calls `clearStopRequest()`.

Modeling components use this function to stop the simulation from within the simulation thread (for example for external breakpoints) and also asynchronously from from the debugger thread. Calling this function again (from any host thread) before `stopAcknowledge()` has reset the stop request using `clearStopRequest()` is harmless. The simulation only stops once.

Returns `true` when the runnable accepts the stop request and will stop later. Returns `false` when the runnable does not accept the stop request. In this case, the scheduler must stop the simulation when the runnable returns control to the scheduler (for example, by use of `wait()`).

**Related references**

* sg::SchedulerRunnable::clearStopRequest on page 7-156

sg::SchedulerRunnable::threadProc

This is the main thread function, the thread entry point.

```cpp
void threadProc();
```

When `threadProc()` returns, the thread no longer runs and this `SchedulerThread` instance will not call `threadProc()` again. The thread usually does not return from this function while the thread is running. `threadProc()` is to call `SchedulerInterfaceForComponents::wait(0, ...) after completing initialization. `threadProc()` is to call `SchedulerInterfaceForComponents::wait(t>=0, ...) after completing t ticks worth of work.

Do not create/destroy any other threads or scheduler objects within the context of this function.

### 7.7.5 sg::SchedulerThread class

This section describes the `SchedulerThread` class.

sg::SchedulerThread - about

This class is a thread interface on the thread instance/scheduler side. The `SchedulerInterfaceForComponents::createThread()` function creates the `SchedulerThread` objects. Modeling components use this interface to talk to the scheduler.

**Related references**

* sg::SchedulerInterfaceForComponents::createThread on page 7-151

sg::SchedulerThread::destructor

This method destroys `SchedulerThread` objects.

```cpp
~SchedulerThread();
```
This destructor kills threads if the underlying scheduler implementation supports it. Killing threads without their cooperation is unclean because it might leak resources. To end a thread cleanly, signal the thread to return from its threadProc() function, for example by using an exception that is caught in threadProc(). Destroying this object before calling start() must not start the thread. Destroying this object after calling start() might kill the thread immediately or leave it running until it returns from its threadProc().

SchedulerThread inherits this method from sg::SchedulerObject.

Related references
sg::SchedulerInterfaceForComponents::createThread on page 7-151

sg::SchedulerThread::getName
This method returns the name of the instance that owns the object.

```
const char *getName() const;
```
This is the name that createThread() received.
SchedulerThread inherits this method from sg::SchedulerObject.

sg::SchedulerThread::setFrequency
This method sets the frequency source to be the parent clock for the thread. Arm deprecates this function.

sg::SchedulerThread::start
This method starts the thread.

```
void start();
```
This method calls the threadProc() function immediately, which must call wait(0, ...) after initialization in order for start() to return. start() only runs the threadProc() of the associated thread and no other threads. Calling start() on a running thread has no effect. Calling start() on a terminated thread (threadProc() returned) has no effect.

Note
The modeling component counterpart of the sg::SchedulerThread class is sg::SchedulerRunnable. Runnable objects must call sg::QuantumKeeper::sync() regularly to pass execution control on to other threads.

Related references
sg::SchedulerInterfaceForComponents::createThread on page 7-151

7.7.6 sg::ThreadSignal class
This section describes the ThreadSignal class. It represents a nonschedulable event on which threads can wait. When the event is signaled, all waiting threads can run.

sg::ThreadSignal::destructor
This method destroys ThreadSignal objects, thread signals.

```
~ThreadSignal();
```
Destroying these objects while threads are waiting for them leaves the threads unscheduled.

sg::ThreadSignal::notify
This method notifies the system of the event, waking up any waiting threads.

```
void notify();
```
SchedulerRunnable::threadProc() can call this method, but calls can come from outside of threadProc(). Calling this method when no thread is waiting for the signal is valid, but has no effect.

**sg::ThreadSignal::getName**

This method returns the name of the instance that owns the object.

```c
const char *getName() const;
```

This is the name that createThreadSignal() received.

ThreadSignal inherits this method from sg::SchedulerObject.

### 7.7.7 **sg::Timer class**

This section describes the Timer interface class. The SchedulerInterfaceForComponents::createTimer() method creates Timer objects.

**sg::Timer::cancel**

This method unsets the timer so that it does not fire.

```c
void cancel();
```

If the timer is not set, this method has no effect.

**sg::Timer::destructor**

This method destroys Timer objects.

```c
~Timer();
```

The timer must not call TimerCallback::timerCallback() after the destruction of this object.

**sg::Timer::getName**

This method returns the name of the instance that owns the object.

```c
const char *getName() const;
```

This is the name that createTimer() received.

Timer inherits this method from sg::SchedulerObject.

**sg::Timer::isSet**

This method returns true if the timer is set and queued for call-back, otherwise false.

```c
bool isSet();
```

This method has no side effects.

**sg::Timer::remaining**

This method requests the remaining number of ticks relative to simulated time resolution until a timer makes a signal.

```c
ticks_t remaining();
```

This method returns 0 if there are no ticks remaining or if the timer is not set.

This method has no side effects.

**sg::Timer::set**

This method sets a timer to make a signal.

```c
bool set(ticks_t ticks);
```
ticks
the number of ticks after which the timer is to make a signal.

The signal that this method makes is a call to the user call-back function. If the return value \( t \) is 0, the timer does not repeat, otherwise it repeats after \( t \) ticks. The latest \texttt{set()} overrides the previous one.

This method returns \texttt{false} if ticks is too big to schedule the timer.

\texttt{sg::Timer::setFrequency}

This method sets the frequency source clock for the timer. Arm deprecates this function. Simulated time is relative to global time resolution. See
\texttt{SchedulerInterfaceForComponents::getSimulatedTimeResolution()} and
\texttt{SchedulerInterfaceForComponents::setSimulatedTimeResolution()}.

7.7.8 \texttt{sg::TimerCallback} class

This section describes the \texttt{TimerCallback} base class. This interface does not allow object destruction.

\texttt{sg::TimerCallback::getName}

This method returns the name of the instance that owns the object.

\begin{verbatim}
const char *getName() const;
\end{verbatim}

Conventionally, this is the name that \texttt{createTimer()} received.

\texttt{TimerCallback} inherits this method from \texttt{sg::SchedulerObject}.

\texttt{sg::TimerCallback::timerCallback}

The \texttt{createTimer()} method receives a \texttt{timerCallback} instance. This \texttt{timerCallback()} method is called whenever the timer expires. This method returns a value \( t \). If \( t \) is 0, the timer does not repeat, otherwise it is to call \texttt{timerCallback()} again after \( t \) ticks.

\begin{verbatim}
ticks_t timerCallback();
\end{verbatim}

7.7.9 \texttt{sg::FrequencySource} class

\texttt{FrequencySource} objects provide clock frequencies, and notify frequency observers of frequency changes. This interface does not allow object destruction. Arm deprecates this class. Simulated time is relative to global time resolution. See
\texttt{SchedulerInterfaceForComponents::getSimulatedTimeResolution()} and
\texttt{SchedulerInterfaceForComponents::setSimulatedTimeResolution()}.

7.7.10 \texttt{sg::FrequencyObserver} class

\texttt{FrequencySource} instances notify \texttt{FrequencyObserver} instances of \texttt{FrequencySource} instance changes. This interface does not allow object destruction. Arm deprecates this class. Simulated time is relative to global time resolution. See
\texttt{SchedulerInterfaceForComponents::getSimulatedTimeResolution()} and
\texttt{SchedulerInterfaceForComponents::setSimulatedTimeResolution()}.

7.7.11 \texttt{sg::SchedulerObject} class

This section describes the \texttt{SchedulerObject} class. It is the base class for scheduler objects and interfaces. This interface does not allow object destruction.

\texttt{sg::SchedulerObject::getName}

This method returns the name of the instance that implements the object or interface. The intended use is debugging.

\begin{verbatim}
const char *getName() const;
\end{verbatim}
Although Arm does not guarantee this name to be unique or hierarchical, Arm recommends including or using the hierarchical component name. The caller must not free/delete the returned string. This object owns the string. The pointer is valid as long as the object implementing this interface exists. If the caller cannot track the lifetime of this object and wants to remember the name, it must copy it.

7.7.12 **sg::scx_create_default_scheduler_mapping**

This function returns a pointer to a new instance of the default implementation of the scheduler mapping provided with Fast Models.

```c
sg::SchedulerInterfaceForComponents *scx_create_default_scheduler_mapping(scx_simcontrol_if *simcontrol);
```

- `simcontrol` is a pointer to an existing simulation controller. If this is `NULL`, this function returns `NULL`.

7.7.13 **sg::scx_get_curr_scheduler_mapping**

This function returns a pointer to the scheduler mapping interface.

```c
sg::SchedulerInterfaceForComponents *scx_get_curr_scheduler_mapping();
```
7.8 SystemC Export limitations

This section describes the limitations of the current release of SystemC Export.

The Exported Virtual Subsystems (EVSs) are deliberately not time or cycle accurate, although they are accurate on a functional level.

This section contains the following subsections:

• 7.8.1 SystemC Export limitation on reentrancy on page 7-163.
• 7.8.2 SystemC Export limitation on calling wait() on page 7-163.
• 7.8.3 SystemC Export limitation on code translation support for external memory on page 7-163.
• 7.8.4 SystemC Export limitation on Fast Models versions for MI platforms on page 7-163.

7.8.1 SystemC Export limitation on reentrancy

Processor models, and the CCI400, MMU_400, and MMU_500 component models support reentrancy.

Reentrancy occurs when a component in an EVS issues a blocking transaction to a SystemC peripheral that in turn generates another blocking transaction back into the same component. This generation might come directly or indirectly from a call to wait() or by another SystemC peripheral.

Virtual platforms including EVSs that comprise a processor model do support such reentrancy.

For models that do not support reentrancy, the virtual platform might show unpredictable behavior because of racing within the EVS component.

7.8.2 SystemC Export limitation on calling wait()

Arm only supports calling wait() on bus transactions.

When a SystemC peripheral must really issue a wait() in reaction to a signal that is changing, buffer the signal in the bridge between the EVS and SystemC. On the next activation of the bridge, set the signal with the thread context of the EVS.

_________ Note _________

The EVS runs in a temporally decoupled mode using a time quantum. Transaction Level Modeling (TLM) 2.0 targets using the Loosely-Timed coding style do not call wait().

7.8.3 SystemC Export limitation on code translation support for external memory

EVS core components use code translation for speed. Not enabling Direct Memory Interface (DMI) reduces performance.

The core components in EVSs use code translation for high simulation speed. Therefore they fetch data from external memory to translate it into host machine code. Changing the memory contents outside of the scope of the core makes the data inconsistent.

Enable DMI accesses to instruction memory to avoid dramatic performance reductions. Otherwise, EVSs:

• Model all accesses.
• Perform multiple spurious transactions.
• Translate code per instruction not per block of instructions.

7.8.4 SystemC Export limitation on Fast Models versions for MI platforms

SystemC Export with Multiple Instantiation (MI) supports virtual platforms with multiple EVSs made with the same version of Fast Models. Integrating EVSs from different versions of Fast Models might result in unpredictable behavior.
Chapter 8
Generic Graphics Accelerator

This chapter describes the Generic Graphics Accelerator (GGA) feature in Fast Models.

It contains the following sections:

• 8.1 Introduction on page 8-165.
• 8.2 Using the Generic Graphics Accelerator on page 8-169.
• 8.3 The configuration file on page 8-187.
• 8.4 Feedback on the Generic Graphics Accelerator on page 8-188.
8.1 Introduction

The Generic Graphics Accelerator (GGA) provides hardware acceleration for rendering work done by applications running on an Android target. It can help you debug the target graphics APIs and can also help you debug the integration of the target graphics software stack with a target subsystem that contains a Mali GPU model.

This section contains the following subsections:
- 8.1.2 Prerequisites on page 8-167.
- 8.1.3 Installing the Arm® Mali™ OpenGL ES Emulator on page 8-167.

8.1.1 Overview of the Generic Graphics Accelerator

This section describes the components of the Generic Graphics Accelerator, and how they perform hardware acceleration and graphics integration for Android targets.

Hardware acceleration for Android targets

The Generic Graphics Accelerator maps OpenGL ES or Vulkan APIs on an Android target to OpenGL or Vulkan APIs on the host. The host renders Android images with these APIs using its own GPU.

The following figure shows the hardware rendering process:

![Figure 8-1 Graphics APIs mapping between the target and host](image)

Signals in this figure are:
• 1 is the SWI signal.
• 2 is the CADIMemRead signal from the Sidechannel plug-in to read from target memory into the write buffer.
• 3 is the CADIMemWrite signal from the Sidechannel plug-in to write to target memory from the read buffer.
• The buffer inside the target covers both the usual buffer to implement graphics APIs, and the frame buffer as the rendering target.

Components in this figure are:

**Shim layer**

Plays the role of the GPU driver. Implements the OpenGL ES or Vulkan APIs used by the application running on the target, and communicates with the Sidechannel plug-in. The Shim layer is implemented by the Shim library.

**Sidechannel plug-in**

Provides a communication channel for messages and data passed between the target and host. The application that calls OpenGL ES or Vulkan APIs on the target is in a different address space from the graphics APIs running on the host. The Sidechannel plug-in communicates between these different address spaces.

**Reconciler**

Bridges the communication between the Sidechannel plug-in and the graphics APIs.

The Generic Graphics Accelerator comprises the Shim layer and the Reconciler.

**Arm Mali™ OpenGL ES Emulator**

Simulates the OpenGL ES APIs by using the OpenGL APIs of the graphics card on the host. This emulator is used only with OpenGL ES APIs.

The workflow in this figure is:
1. An application running on the Fast Models target makes an OpenGL ES or a Vulkan API call.
2. The Shim layer receives this call and informs the Sidechannel plug-in that it needs to send data to the host.
3. The Shim layer writes the data into the Write Buffer inside the Sidechannel plug-in when a software interrupt is invoked in the Fast Model.
4. The Reconciler wakes up after the data writing completes.
5. For an OpenGL ES API call, the Reconciler passes the data to the Arm Mali OpenGL ES Emulator on the host and waits until the call is completed.
   For a Vulkan API call, the Reconciler passes the data to the host graphics driver and waits until the call is completed.
6. The Reconciler writes the output back to the Read Buffer of the Sidechannel plug-in.
7. The Shim layer sends another software interrupt to read the data from the Read Buffer when the application needs the data returned from the API.
8. The Sidechannel plug-in reads the data from the read buffer and returns directly from the interrupt.
9. The application continues to run on the target.

**Graphics integration for Android targets**

The Generic Graphics Accelerator supports the integration of the Android target graphics software stack with the Arm Mali GPU register models, Mali-G71, Mali-G72, and Mali-G51.

The integration work flow is:
1. On the target, the application APIs (OpenGL ES and EGL) interact with both the shim libraries and the Mali graphics drivers.
2. Shim libraries pass the API data to the Generic Graphics Accelerator on the host. Then, the Generic Graphics Accelerator works with the host GPU and Mali OpenGL ES Emulator to render this data using the host GPU.
   Images are saved in the Generic Graphics Accelerator.
3. In parallel with rendering on the host, on the target, the Mali driver executes APIs and passes commands, addresses, and the other settings to the GPU register model.
4. When the model receives the start command from the driver, it sends the rendering-related destination addresses, MMU settings, and other data to the Generic Graphics Accelerator.
5. On receiving this data, the Generic Graphics Accelerator combines it with the stored rendering images, and outputs the combined result to the memory model on the target.
6. The Generic Graphics Accelerator reports to the target GPU that it has completed rendering.
7. The target GPU reports to the driver that rendering is complete.

![Figure 8-2 Graphics integration with the GPU register model](image)

### 8.1.2 Prerequisites

The Generic Graphics Accelerator is available in Fast Models version 10.0 and later. It is supported on both Windows and Linux hosts.

To use the Generic Graphics Accelerator, the Arm Mali OpenGL ES Emulator must be installed on the host. For information, see [8.1.3 Installing the Arm® Mali™ OpenGL ES Emulator on page 8-167](#).

The target must use:
- Android version 4.4.2, 5.0.1, 6.0, or 7.0.
- OpenGL ES version 2.0, 3.0, 3.1, or Vulkan 1.0.

**Note**

The Generic Graphics Accelerator does not support OpenVG or OpenCL.

GGA has been validated on NVIDIA GT 730 or later graphics cards, with driver versions:
- 390.77 and above for Ubuntu.
- 390.77 and above for Windows.

If the Fast Models version is 10.3.016 or later, the Generic Graphics Accelerator supports integration of the Android target graphics software stack with one of the GPU register models, for example Mali_G71. For details about these models, see [Fast Models components in Fast Models Reference Manual](#).

**Related references**

[2.1 Requirements for Fast Models on page 2-28](#)

### 8.1.3 Installing the Arm® Mali™ OpenGL ES Emulator

This section walks you through the installation step by step.
Procedure

2. Install and configure the emulator.
   For the instructions, see the Mali OpenGL ES Emulator User Guide, contained in the installation package.
3. Verify the installation by running the mali-cube application.
   For details, see the emulator user guide.

Results:

When the installation is successful, you can see a spinning cube, as shown in the following figure:

![Mali Cube Application](image-url)
8.2 Using the Generic Graphics Accelerator

The Generic Graphics Accelerator (GGA) makes Android targets boot faster and makes them easier to debug.

You can use GGA to:
• Boot an Android target with hardware acceleration.
• Debug the target graphics APIs.
• Debug the target graphics software integration with a Mali GPU model.

This section contains the following subsections:
• 8.2.1 Booting Android targets for the first time on page 8-169.
• 8.2.2 Booting Android targets on page 8-175.
• 8.2.3 Verifying Android targets on page 8-178.

8.2.1 Booting Android targets for the first time

The first boot is performed with both software and hardware acceleration.

Before you begin

Make sure that both the host and the target meet the conditions described in 8.1.2 Prerequisites on page 8-167.

Procedure

   Fast Models emulates the target with software graphics acceleration. For details, see Booting Android targets with software acceleration on page 8-169.
2. Enable the Generic Graphics Accelerator for the target.
   To do this, push the Shim libraries from the host to the target. For details, see Enabling the Generic Graphics Accelerator on page 8-170.
3. Boot the target with the Generic Graphics Accelerator support.
   With the Generic Graphics Accelerator enabled, Fast Models employs the host graphics hardware to accelerate the target boot. For more details, see Booting Android targets with the Generic Graphics Accelerator on page 8-173.
4. Test the target graphics with test applications.
   For the installation and execution of applications on Android targets, see Installing applications on Android targets on page 8-174.
5. If the target platform includes one of the GPU register models, Mali-G71, Mali-G72, or Mali-G51, and the Fast Models version is 10.3.016 or later, to debug the target graphics software stack integration, boot the target again as described in Integrating the graphics driver with the GPU model on page 8-179.
   For information about debugging the integration, see Debugging graphics software stack integration on page 8-178.

What to do next

Use hardware acceleration independently to boot the target, in which case you can use the boot command as described in 8.2.2 Booting Android targets on page 8-175.

Booting Android targets with software acceleration

Fast Models boots Android targets to use software acceleration when the Generic Graphics Accelerator is disabled.

Before you begin
Make sure:
• Both the host and the target systems meet the conditions as described in 8.1.2 Prerequisites on page 8-167.
• Both the Android binary image and the target platform model are available.

**Procedure**
1. On the host, enter the boot command from the directory of the Android target.

    __________ Note __________

    For convenience, you can use the same command as to boot the target with the Generic Graphics Accelerator. The command options that are related to the Generic Graphics Accelerator are skipped when the Generic Graphics Accelerator is disabled.

    For the command, see Example: Booting an Android target to verify the OS on page 8-176.

**Results**
If the boot is successful, the default Android desktop is displayed on the screen.

**What to do next**
Enable the Generic Graphics Accelerator to reboot Android with hardware acceleration.

For details, see the subsequent sections:
• Enabling the Generic Graphics Accelerator on page 8-170
• Booting Android targets with the Generic Graphics Accelerator on page 8-173

**Enabling the Generic Graphics Accelerator**
To enable the Generic Graphics Accelerator for the Android target, push the Shim libraries from the host to the target.

**About this task**
The Shim libraries in the Generic Graphics Accelerator act as the graphics driver for the Android target. They pass OpenGL ES API data from the target to the host to enable the host GPU to render images for these APIs.

**Before you begin**
Make sure that:
• The Android SDK is installed.
• The Android target has booted on Fast Models using software acceleration. For instructions, see Booting Android targets with software acceleration on page 8-169.

**Procedure**
1. Copy the `settings.ini` configuration file from either of the following locations to the directory where you booted the Android target with software acceleration:
   On Linux, `<installation_directory>/GGA/reconciler/linux-x86_64/gcc-x.x.x/rel/` (where `x.x.x` is the GCC version number).
   On Windows, `<installation_directory>\GGA\reconciler\win_32-x86_64\cl-18.00.31101\rel`.
2. From the host, enter the following commands:
   **Example:**
   ```
   export PATH=$PATH:<path_of_android_sdk_linux>/platform-tools/
   adb connect localhost:5212
   ```
On Windows:

```
set PATH=%PATH%;<installation_directory>\<path_of_android_sdk_windows>\platform-tools
adb connect localhost:5212
```

--- **Note** ---

In the `adb` command, the port number must use the first value of the boot command option `board.hostbridge.userNetPorts`.

For the boot command, see *Example: Booting an Android target to verify the OS on page 8-176.*

3. From the host, connect to the target using the following command:

```
adb connect localhost:5212
```

4. Remove `libEGL*.so` and `libGLES*.so` from the target.

--- **Note** ---

Remember to back up files before removal.

If Vulkan is used, the commands are:

```
adb shell rm /system/vendor/lib/egl/libEGL*.so
adb shell rm /system/vendor/lib/egl/libGLES*.so
adb shell rm /system/vendor/lib64/egl/libEGL*.so
adb shell rm /system/vendor/lib64/egl/libGLES*.so
```

Otherwise, the commands are:

```
adb shell rm /system/vendor/lib/egl/libEGL*.so
adb shell rm /system/vendor/lib/egl/libGLES*.so
adb shell rm /system/vendor/lib64/egl/libEGL*.so
adb shell rm /system/vendor/lib64/egl/libGLES*.so
```

5. If Android uses customized native window handles, generate a customized `libnwhal.so` file. For details, see *Generating libnwhal.so on page 8-172.*

Skip this step if Android follows the standard Android Open Source Project, or uses the Gralloc module provided from [https://developer.arm.com/products/software/mali-drivers/android-gralloc-module](https://developer.arm.com/products/software/mali-drivers/android-gralloc-module).

6. From the Generic Graphics Accelerator directory, push the Shim libraries to the target.

The following examples show the instructions on different hosts:

* **Example:**

--- **Note** ---

For Android versions later than 6.0, the command uses the directory `/system/lib/egl/` or `/system/lib64/egl/` instead of `/system/vendor/lib/egl/` or `/system/vendor/lib64/egl/`.

---

* To push Shim libraries from a Linux host to a 32-bit Android target:

```
adb remount
adb push GGA/shim/linux-armv7sfl/rel/libGLES.so /system/vendor/lib/egl/libGLES_vimpl.so
adb push GGA/shim/linux-armv7sfl/rel/HAL/libnwhal.so /system/lib/libnwhal.so
```

```
adb shell chmod 0644 /system/vendor/lib/egl/libGLES_vimpl.so
adb shell chmod 0644 /system/lib/libnwhal.so
```
If Vulkan is used, enter one more command:

```
adb shell ln -s /vendor/lib/egl/libGLES_vimpl.so /system/vendor/lib/hw/vulkan.<ro.board.platform>.so
```

- To push Shim libraries from a Linux host to a 64-bit Android target:

```
adb remount
adb push GGA/shim/linux-armv7sfl/rel/libGLES.so /system/vendor/lib/egl/libGLES_vimpl.so
adb push GGA/shim/linux-armv8l_64/rel/libGLES.so /system/vendor/lib64/egl/libGLES_vimpl.so
adb shell chmod 0644 /system/lib/libnwhal.so
adb shell chmod 0644 /system/vendor/lib/egl/libGLES_vimpl.so
adb shell chmod 0644 /system/vendor/lib64/egl/libGLES_vimpl.so
```

If Vulkan is used, enter one more command:

```
adb shell ln -s /vendor/lib64/egl/libGLES_vimpl.so /system/vendor/lib64/hw/vulkan.<ro.board.platform>.so
```

- To push Shim libraries from a Windows host to a 32-bit Android target:

```
adb remount
adb push GGA\shim\linux-armv7sfl\rel\libGLES.so /system/vendor/lib/egl/libGLES_vimpl.so
adb push GGA\shim\linux-armv8l_64\rel\libGLES.so /system/vendor/lib64/egl/libGLES_vimpl.so
adb shell chmod 0644 /system/lib/libnwhal.so
adb shell chmod 0644 /system/vendor/lib/egl/libGLES_vimpl.so
```

If Vulkan is used, enter one more command:

```
adb shell ln -s /vendor/lib64/egl/libGLES_vimpl.so /system/vendor/lib/hw/vulkan.<ro.board.platform>.so
```

- To push Shim libraries from a Windows host to a 64-bit Android target:

```
adb remount
adb push GGA\shim\linux-armv7sfl\rel\libGLES.so /system/vendor/lib/egl/libGLES_vimpl.so
adb push GGA\shim\linux-armv8l_64\rel\libGLES.so /system/vendor/lib64/egl/libGLES_vimpl.so
adb shell chmod 0644 /system/lib64/libnwhal.so
adb shell chmod 0644 /system/vendor/lib/egl/libGLES_vimpl.so
```

If Vulkan is used, enter one more command:

```
adb shell ln -s /vendor/lib64/egl/libGLES_vimpl.so /system/vendor/lib64/hw/vulkan.<ro.board.platform>.so
```

---

**Note**

In these commands, the folder name `<HAL>` can be either of the following:
- **stock**, if the Android version follows the standard Android Open Source Project.
- The Gralloc version number used, such as `bfst-r7p0-01r1e10`, if the Android version uses a Gralloc module provided from [https://developer.arm.com/products/software/mali-drivers/android-gralloc-module](https://developer.arm.com/products/software/mali-drivers/android-gralloc-module).

---

**What to do next**

Boot the target with the Generic Graphics Accelerator through the host GPU acceleration. For details, see *Booting Android targets with the Generic Graphics Accelerator* on page 8-173.

**Generating libnwhal.so**

Create a customized `libnwhal.so` file for an Android target if it uses customized native window handles.

**About this task**
Create this file before you push the Shim libraries to the Android target. This file parses and transfers the target OpenGL ES data from the Android HAL to the Shim libraries.

**Procedure**

1. Open the file `$PVLIB_HOME/GGA/HAL/<subdirectory>/jni/src/nw_hal.h`
   Here, `<subdirectory>` can be the folder stock or a folder named with a Gralloc version number.
2. Follow the descriptions in the file to define a customized data structure `android_private_handle_t`.
3. From the directory, `$PVLIB_HOME/GGA/HAL/<subdirectory>/`, run the following build command to generate the `libnwhal.so` file:

   `<ANDROID_NDK_HOME>/ndk-build`

**What to do next**

Push the Shim libraries from the host to the Android target as described in:

- *Enabling the Generic Graphics Accelerator on page 8-170*, if this is the first time to boot the Android target.
- *Pushing the Shim libraries to the Android target for the driver integration on page 8-180*, if you want to debug the graphics software stack integration on the Android target.

**Booting Android targets with the Generic Graphics Accelerator**

The Generic Graphics Accelerator enables hardware acceleration to boot an Android target in Fast Models.

**Before you begin**

Make sure that the Generic Graphics Accelerator is enabled for the Android target. For details, see *Enabling the Generic Graphics Accelerator on page 8-170*.

**Procedure**

1. On the host, enter the command to disable the target OpenGL ES driver preload feature:

   ```
   adb remount
   adb shell sed -i '/ro.zygote.disable_gl_preload=/d' /system/build.prop
   adb shell "echo \"ro.zygote.disable_gl_preload=true\" >> /system/build.prop"
   ```

2. On the host, enter the command to enable rendering through the host GPU.
   Commands are different for Linaro, and 32-bit and 64-bit Android:
   - **64-bit Android:**
     ```
     adb remount
     adb shell sed -i '1,/ro.kernel.qemu/s/ro.kernel.qemu/No.kernel.qemu/' /system/lib/libEGL.so
     adb shell sed -i '1,/ro.kernel.qemu/s/ro.kernel.qemu/No.kernel.qemu/' /system/lib64/libEGL.so
     ```
   - **32-bit Android:**
     ```
     adb remount
     adb shell sed -i '1,/ro.kernel.qemu/s/ro.kernel.qemu/No.kernel.qemu/' /system/lib/libEGL.so
     ```
   - Some Linaro distributions of 64-bit Android:
     ```
     adb remount
     adb shell sed -i '1,/ro.kernel.qemu/s/ro.kernel.qemu/No.kernel.qemu/' /system/lib/libEGL.so
     adb shell sed -i '1,/ro.kernel.qemu/s/ro.kernel.qemu/No.kernel.qemu/' /system/lib64/libEGL.so
     adb shell sed -i 's/ro.nohardwaregfx=true/ro.nohardwaregfx=false/' /system/build.prop
     ```
   - Some Linaro distributions of 32-bit Android:
     ```
     adb remount
     adb shell sed -i '1,/ro.kernel.qemu/s/ro.kernel.qemu/No.kernel.qemu/' /system/lib/libEGL.so
     adb shell sed -i 's/ro.nohardwaregfx=true/ro.nohardwaregfx=false/' /system/build.prop
     ```
3. On the host, enter the command to reboot the target:

   ```
   adb shell sync
   adb shell reboot
   ```
If failures occur, kill Fast Models and then repeat the procedure as described in 8.2.1 Booting Android targets for the first time on page 8-169.

Results
If the Android target has booted:
- The host displays the default Android desktop.
- The host terminal shows interaction details between the Reconciler and the OpenGL ES Emulator.
- By entering the logcat command on the target terminal, you can view more information from the target, such as:
  - The interaction between the Shim Layer and applications.
  - The interaction between the Shim Layer and the Reconciler.

What to do next
Test the target by installing and running test applications on it.

Installing applications on Android targets
Install and run applications on an Android target to test its graphics.

About this task
You can find test applications for both OpenGL ES and Vulkan in the directory <installation_directory>/GGA/examples/.

Before you begin
Boot the Android target. If this is the first time, follow the procedure described in 8.2.1 Booting Android targets for the first time on page 8-169. Otherwise, use the boot command described in 8.2.2 Booting Android targets on page 8-175.

Procedure
1. Install the application, for example, the Cube test application.
   **Example:**
   On Linux, for 32-bit Android:
   ```bash
   adb install <installation_directory>/GGA/examples/opengles/linux-armv7sf1/Cube.apk
   ```
   **Example:**
   On Windows, for 32-bit Android:
   ```bash
   adb install <installation_directory>\GGA\examples\opengles\linux-armv7sf1\Cube.apk
   ```
   **Example:**
   On Linux, for 64-bit Android:
   ```bash
   adb install <installation_directory>/GGA/examples/opengles/linux-armv8l_64/Cube.apk
   ```
   **Example:**
   On Windows, for 64-bit Android:
   ```bash
   adb install <installation_directory>\GGA\examples\opengles\linux-armv8l_64\Cube.apk
   ```
2. Run the application.

Results
If the Android target is rendering correctly, a spinning cube is displayed on the screen as the following figure shows:
What to do next
You can:

• Debug the target graphics APIs at the application level. You can use the debug method described in *Debugging graphics APIs for applications on page 8-178.*

• Debug the target graphics software stack integration if the target subsystem contains a GPU register model. Use the debug methods described in *Debugging graphics software stack integration on page 8-178.*

**Related information**

*Android Debug Bridge*

### 8.2.2 Booting Android targets

To boot Android after the initial boot, use the command described in this section.

**Before you begin**

Make sure that the initial boot was successful, see 8.2.1 *Booting Android targets for the first time on page 8-169.*

**Procedure**

1. On the host, from the same directory as for the initial boot, enter the boot command. There are different commands for:

   • Booting the target to test the target OS and graphics APIs.
   • Booting the target to test the target graphics driver integration with the GPU model.

   For these different commands, see the following sections.

   After the target has booted:
• The host displays the default Android desktop for the target. You can operate Android using the mouse.
• The host terminal shows details of the interaction between the Reconciler and the OpenGL ES Emulator.
• By entering the logcat command on the target terminal, you can view more information from the target, such as:
  — The interaction between the Shim Layer and applications.
  — The interaction between the Shim Layer and the Reconciler.

What to do next

You can:
• Run test applications to test the graphics.
  For details, see Installing applications on Android targets on page 8-174.
• Debug the target graphics APIs and the graphics driver integration with the modeled GPU.
  For details, see 8.2.3 Verifying Android targets on page 8-178.

Example: Booting an Android target to verify the OS

In this example, a command line is used to boot an Android target with graphics hardware acceleration. The Generic Graphics Accelerator is used to support the hardware acceleration.

The following examples show different commands for different FVPs and host systems:

• To boot a 32-bit Android target on a Linux host:

```
../../../models/Linux64_GCC-5.4/FVP_VE_Cortex-A15x1 \
boot/rtsm/linux-system-semi.axf \ 
--plugin ../../../plugins/<path_of_Sidechannel.so> \ 
-C DEBUG.Sidechannel.interceptor=<path_of_libReconciler.so> \ 
-C motherboard.smsc_91c111.enabled=1 \ 
-C motherboard.vis.disable_visualisation=0 \ 
-C motherboard.hostbridge.userNetworking=1 \ 
-C motherboard.hostbridge.userNetPorts=5212=6565 \ 
-C motherboard.mmc.p_mmc_file=linaro-android-vexpress-lsk-14.10.img
```

• To boot a 64-bit Android target on a Linux host:

```
../../../models64/Build_AEMv8A-AEMv8A/Linux64_GCC-5.4/FVP_Base_AEMv8A-AEMv8A \ 
--plugin ../../../plugins/<path_of_Sidechannel.so> \ 
-C pctl.startup=0.0.0.0 \ 
-C bp.secure_memory=0 \ 
-C cluster0.NUM_CORES=1 \ 
-C cluster1.NUM_CORES=0 \ 
-C cache_state_modelled=0 \ 
-C bp.hostbridge.userNetworking=1 \ 
-C bp.hostbridge.userNetPorts=5212=6565 \ 
-C bp.smsc_91c111.enabled=1 \ 
-C bp.smsc_91c111.mac_address=auto \ 
-C bp.p18f1_uart0.untimed_fifos=1 \ 
-C bp.secureflashloader.fname=bl1.bin \ 
-C bp.flashloader0.fname=fvp_fip.bin \ 
-C bp.flashloader1.fname=uefi-vars.fd \ 
-C bp.virtioblockdevice.image_path=../linaro-android-fvp_v8-lcr-14.12_build.img
```

• To boot a 32-bit Android target on a Windows host:

```
 ../../../models\FVP_VE_Cortex-A17x1.exe \ 
 boot/rtsm\linux-system-semi.axf \ 
--plugin ../../../plugins/<path_of_Sidechannel.dll> \ 
-C DEBUG.Sidechannel.interceptor=<path_of_Reconciler.dll> \ 
-C motherboard.smsc_91c111.enabled=1 \ 
-C motherboard.vis.disable_visualisation=0 \ 
-C motherboard.hostbridge.userNetworking=1 \ 
-C motherboard.hostbridge.userNetPorts=5212=6565 \ 
-C motherboard.mmc.p_mmc_file=linaro-android-vexpress-lsk-14.10.img \ 
-C cluster.cpu0.semihosting-cmd_line="--kernel boot\uImage \ 
--dbt boot\rtsm\ve-ca15x1-t6xx.dbt"
```
To boot a 64-bit Android target on a Windows host:

```bash
..\..\models\FVP_Base_Cortex-A57x1.exe
--plug...
-C C.. DEBUG.Sidechannel.interceptor=\(path_of_Sidechannel.dll\)
-C pctl.startup=0.0.0.0
-.. bp.secure_memory=0
-C cache_state_modelled=0
-.. bp..hostbridge.userNetworking=1
-.. bp..hostbridge..userNetPorts=5212=6565
-. bp..smsc_91c111..enabled=1
-.. bp..smsc_91c111..mac_address=auto
-. bp.pl011_uart0.un...ferin...es=1
-. bp..secureflashloader..fname=b1.bin
-. bp..flashloa..fname=fvp_fip.bin
-. bp..ve_sysregr..mbSiteDefault=0
-. bp..vintiblockdeivce..image_path=..\linaro-android-6.0.0_r26-fvp_v8-1cr-15.11_build.jpg
```

In these examples:
- The first line specifies the FVP.
- The parameters used for the Generic Graphics Accelerator are:

  ```bash
  --plug...
  C DEBUG.Sidechannel.interceptor=\(path_of_Sidechannel.dll\)
  ```

  Instructs Fast Models to load the Sidechannel plugin, which communicates between the host and the target. The plugin name is case-sensitive.

  ```bash
  -C DEBUG.Sidechannel.interceptor=\(path_of_Reconciler.dll\)
  ```

  Specifies the location of the Reconciler plugin to be loaded. It is named `libReconciler.so` on Linux and `Reconciler.dll` on Windows.

Example: Booting an Android target to verify graphics drivers

In this example, a command line is used to boot an Android target with integration between its graphics driver and the GPU register model. The Generic Graphics Accelerator is enabled to test the graphics integration.

The following examples show the commands on a Linux and a Windows host respectively. They use the same FVP:
- To boot a 64-bit Android target on a Linux host:

```bash
${PATH_ModelFile}\ 
-C css.trustedBootROMloader.fname=${BINDIR}/tf-b1.bin 
-C css.scp.ROMloader.fname=${BINDIR}/scp-rom.bin 
-C soc.pl011_uart0.out_file=${OUTPUT_DIR}/soc-uart0 
-C soc.pl011_uart1.out_file=${OUTPUT_DIR}/soc-uart1 
-C config_id=0 
-C displayController=${Controller} 
-C SOC.display$SUBSYS$hostbridge..interfaceName=..subsys0 
-C board.hostbridge..userNetworking=1 
-C board.hostbridge..userNetPorts=6901=5555 
-C board.smsc_91c111..enabled=1 
-C board.smsc_91c111..mac_address=auto 
-S -R 
-C css..cache_state_modelled=0 
-C board.flashloader0.fname=${BINDIR}/fip-uboot.bin 
--data css.cluster0.cpu0=${CPU0_dtb} 
--data css.cluster0.cpu0=${CPU0_Initrd} 
--data css.cluster0.cpu0=${CPU0_uImage} 
-C board.virtio...device..image_path=${BIN(Image)} 
--plug...
-C C DEBUG.Sidechannel.interceptor=\(PATH\_Reconciler\)/libReconciler.so
```

- To boot a 64-bit Android target on a Windows host:

```bash
${PATH_ModelFile}\ 
-C css.trustedBootROMloader.fname=${BINDIR}/tf-b1.bin 
-C css.scp.ROMloader.fname=${BINDIR}/scp-rom.bin 
-C soc.pl011_uart0.out_file=${OUTPUT_DIR}/soc-uart0 
-C soc.pl011_uart1.out_file=${OUTPUT_DIR}/soc-uart1 
```
Note

- The parameter `board.hostbridge.userNetPorts` maps a host port to a model port. This example sets up a networking connection from the host `adb` with port number 5212 to the Android target `adbd` with port number 6565.
- Command options might vary on different platform models. To see the supported options, run the model with the `--list-params` option.

8.2.3 Verifying Android targets

You can verify an Android target at both the application level and the graphics software stack integration level.

This section walks you through this verification with the Generic Graphics Accelerator.

Debugging graphics APIs for applications

To show the execution of target graphics APIs, firstly set the configuration file of the Generic Graphics Accelerator, then boot the Android target with the Generic Graphics Accelerator.

Before you begin

Make sure that the Android target boots successfully with the Generic Graphics Accelerator. For more details, see 8.2.1 Booting Android targets for the first time on page 8-169.

Procedure

1. In the `settings.ini` configuration file, set `LogLevel` to either of the following values:
   - 6565: Represents LOG_LEVEL_INFO to show information about the important stages in executing APIs.
   - 6566: Represents LOG_LEVEL_DEBUG to show the names and parameters of each API that is called.

   For more details about `settings.ini`, see 8.3 The configuration file on page 8-187.

2. Boot the Android target again to show execution details of the graphics APIs. For the boot command, see Example: Booting an Android target to verify the OS on page 8-176.

   Note

   If you find issues, try to reproduce them using a different platform model. Report bugs in the Generic Graphics Accelerator to the support team as described in 8.4 Feedback on the Generic Graphics Accelerator on page 8-188.

Debugging graphics software stack integration

The Generic Graphics Accelerator enables integration between the Android graphics software stack and the GPU model. You can also debug this integration using the Generic Graphics Accelerator.
In this section, you will learn how to perform and debug the graphics integration using the Generic Graphics Accelerator.

**Integrating the graphics driver with the GPU model**

The integration is performed when booting the Android target on a host when the Generic Graphics Accelerator is set for integration test.

**About this task**

Perform this task to verify the integration of the Mali Mali graphics driver with a GPU model on an Android target. After the integration, rendering is performed using the host GPU, because the Mali GPU models only model registers, not the shader core.

For background information, see *Graphics integration for Android targets on page 8-166.*

**Before you begin**

Make sure that:

- The target virtual platform contains one of the GPU models, for example Mali_G71. For details about these models, see *Fast Models components* in *Fast Models Reference Manual.*
- The Android image file contains the Mali graphics driver.
- The working directory of the virtual platform contains the nomali library. If not, copy the library from the following location to this directory:
  - On Windows, `<installation_directory>\lib\Win64_VCxxxx\Release\nomali.dll`, where xxxx is the Visual Studio version number.
  - On Linux: `<installation_directory>/lib/Linux64_GCC-x.x/libnomali.so`, where x.x is the GCC version number.
- If you are building your own virtual platform, your platform project .sgproj file must contain:
  ```
  path="$(PVLIB_HOME)/etc/nomali.sgrepo"
  ```
- In the device tree, GPU is enabled.
- The Android target has booted successfully with the Generic Graphics Accelerator, see *8.2.1 Booting Android targets for the first time on page 8-169.*

**Procedure**

1. From the directory in which you booted Android with the Generic Graphics Accelerator, open the settings.ini file.
   For details about settings.ini, see *8.3 The configuration file on page 8-187.*
2. In the file, set callOnTargetAPI to 1.
3. On the host, boot the target by using the command shown in *Example: Booting an Android target to verify graphics drivers on page 8-177.*
   Terminals open to emulate the target.
4. From the host, push Shim libraries to the target with new locations given to the target Mali GLES libraries.
   For details, see *Pushing the Shim libraries to the Android target for integration test on page 8-180.*
5. On the host, enter the following commands to disable the OpenGL ES preload feature from the target driver:

   ```
   adb remount
   adb shell sed -i '/ro.zygote.disable_gl_preload=/d' /system/build.prop
   adb shell "echo \"ro.zygote.disable_gl_preload=true\"\" >> /system/build.prop"
   ```
6. On the host, enter the following commands to enable rendering on the host GPU.
   The commands are different for Linaro, and 32-bit and 64-bit Android:
• 64-bit Android:

```bash
adb remount
adb shell sed -i '1,/ro.kernel.qemu/s/ro.kernel.qemu/No.kernel.qemu/' /system/lib/libEGL.so
adb shell sed -i '1,/ro.kernel.qemu/s/ro.kernel.qemu/No.kernel.qemu/' /system/lib64/libEGL.so
```

• 32-bit Android:

```bash
adb remount
adb shell sed -i '1,/ro.kernel.qemu/s/ro.kernel.qemu/No.kernel.qemu/' /system/lib/libEGL.so
```

• Some Linaro distributions of 64-bit Android:

```bash
adb remount
adb shell sed -i '1,/ro.kernel.qemu/s/ro.kernel.qemu/No.kernel.qemu/' /system/lib/libEGL.so
adb shell sed -i 's/ro.nohardwaregfx=true/ro.nohardwaregfx=false/' /system/build.prop
```

• Some Linaro distributions of 32-bit Android:

```bash
adb remount
adb shell sed -i '1,/ro.kernel.qemu/s/ro.kernel.qemu/No.kernel.qemu/' /system/lib/libEGL.so
adb shell sed -i 's/ro.nohardwaregfx=true/ro.nohardwaregfx=false/' /system/build.prop
```

7. On the host, reboot the target:

```bash
adb shell sync
adb shell reboot
```

Results

• During booting:
  — The host terminal shows the interaction between the Reconciler and the OpenGL ES Emulator.
  — A graphical window opens to display the target system desktop.
  — From the target terminal, enter the logcat command to view more details from the target, such as:
    ◦ The interaction between the Shim Layer and applications.
    ◦ The interaction between the Shim Layer and the Reconciler.

• After the target has booted, you see:
  — The message, mali 2d000000 gpu: Probed as mali0, on the target terminal. It implies that the driver has recognized the GPU model and starts the control.
  — The default Android desktop for this target. You can operate it using the mouse.

• If the boot fails, study the output messages carefully and debug the driver.

To view more log information, modify the log settings in settings.ini. For details, see 8.3 The configuration file on page 8-187.

What to do next

You can:

• Run the test applications, such as Cube.
  
  For installation information, see Installing applications on Android targets on page 8-174.

• Debug the integration.
  
  You can use the Generic Graphics Accelerator to debug the graphics driver's execution of the graphics APIs and accesses to the GPU register model, as described in the following sections.

• Quit the boot by using the kill command from the host.

Pushing the Shim libraries to the Android target for the driver integration

Use the adb command to push the Shim libraries to the target.

About this task

This task transfers the OpenGL ES API data from both the Shim libraries and the graphics driver to the host and the GPU register model respectively. The host GPU is used to perform fast rendering for the Android target.

Before you begin
Make sure that the Android SDK is installed.

**Procedure**

1. From the host, enter the following commands:
   **Example:**
   - On Linux:
     ```bash
     export PATH=$PATH:<path of android-sdk-linux>/platform-tools/
     adb connect localhost:6901
     ```
   - On Windows:
     ```bash
     set PATH=%PATH%;
     <installation_directory>\<path of android-sdk-windows>\platform-tools
     adb connect localhost:6901
     ```

   ———— **Note** ————
   The port number here must be identical to the first value of the option `board.hostbridge.userNetPorts` in the boot command that is shown in *Example: Booting an Android target to verify graphics drivers on page 8-177.*

2. From the host, connect to the target using:
   ```bash
   adb connect localhost:6901
   ```

3. From the host, change the Mali GLES library directories on the target using:
   ```bash
   adb shell mv /system/vendor/lib64/egl/libGLES_mali.so /system/lib64/libmali.so
   adb shell mv /system/vendor/lib/egl/libGLES_mali.so /system/lib/libmali.so
   ```

4. Remove `libEGL*.so` and `libGLES*.so` from the target.
   ———— **Note** ————
   Back up the files before removal.
   The commands are:
   ```bash
   adb shell rm /system/vendor/lib/egl/libEGL*.so
   adb shell rm /system/vendor/lib/egl/libGLES*.so
   adb shell rm /system/vendor/lib64/egl/libEGL*.so
   adb shell rm /system/vendor/lib64/egl/libGLES*.so
   ```

5. If Android uses customized native window handles, create a customized `libnwhal.so` file as described in *Generating libnwhal.so on page 8-172.*
   ———— **Note** ————
   Skip this step if Android follows the standard Android Open Source Project, or uses the default Gralloc module provided at https://developer.arm.com/products/software/mali-drivers/android-gralloc-module.

6. On the host, change to the `$PVLIB_HOME` directory that contains the Generic Graphics Accelerator package.
7. Enter the following command to push the Shim libraries from the host to the target.
   Commands are different for 32-bit and 64-bit Android.
   **Example:**
   - For 64-bit:
     ```bash
     • The Linux commands are:
     ```
adb push GGA/shim/linux-armv7sfl/rel/libGLES.so /system/vendor/lib/egl/libGLES_vimpl.so
adb push GGA/shim/linux-armv7sfl/rel/\<HAL\>/libnwhal.so /system/lib/libnwhal.so
adb push GGA/shim/linux-armv8l_64/rel/libGLES.so /system/vendor/lib/egl/libGLES_vimpl.so
adb push GGA/shim/linux-armv8l_64/rel/\<HAL\>/libnwhal.so /system/lib/libnwhal.so
adb shell chmod 0644 /system/lib/libnwhal.so
adb shell chmod 0644 /system/vendor/lib/egl/libGLES_vimpl.so
adb shell chmod 0644 /system/lib64/libnwhal.so
adb shell chmod 0644 /system/vendor/lib64/egl/libGLES_vimpl.so

- The Windows commands are:

adb remount
adb push GGA\shim\linux-armv7sfl\rel\libGLES.so /system/vendor/lib/egl/libGLES_vimpl.so
adb push GGA\shim\linux-armv7sfl\rel\\<HAL\>\libnwhal.so /system/lib/libnwhal.so
adb push GGA\shim\linux-armv8l_64\rel\libGLES.so /system/vendor/lib/egl/libGLES_vimpl.so
adb push GGA\shim\linux-armv8l_64\rel\\<HAL\>\libnwhal.so /system/lib/libnwhal.so
adb shell chmod 0644 /system/vendor/lib/egl/libGLES_vimpl.so
adb shell chmod 0644 /system/lib/libnwhal.so
adb shell chmod 0644 /system/vendor/lib64/egl/libGLES_vimpl.so

For 32-bit:
- The Linux commands are:

adb remount
adb push GGA/shim/linux-armv7sfl/rel/libGLES.so /system/vendor/lib/egl/libGLES_vimpl.so
adb push GGA/shim/linux-armv7sfl/rel/\<HAL\>/libnwhal.so /system/lib/libnwhal.so
adb shell chmod 0644 /system/vendor/lib/egl/libGLES_vimpl.so
adb shell chmod 0644 /system/lib/libnwhal.so
adb shell chmod 0644 /system/vendor/lib64/egl/libGLES_vimpl.so
adb shell chmod 0644 /system/lib64/libnwhal.so

adb remount
adb push GGA\shim\linux-armv7sfl\rel\libGLES.so /system/vendor/lib/egl/libGLES_vimpl.so
adb push GGA\shim\linux-armv7sfl\rel\\<HAL\>\libnwhal.so /system/lib/libnwhal.so
adb shell chmod 0644 /system/vendor/lib/egl/libGLES_vimpl.so
adb shell chmod 0644 /system/lib/libnwhal.so
adb shell chmod 0644 /system/vendor/lib64/egl/libGLES_vimpl.so
adb shell chmod 0644 /system/lib64/libnwhal.so

- The Windows commands are:

adb remount
adb push GGA\shim\linux-armv7sfl\rel\libGLES.so /system/vendor/lib/egl/libGLES_vimpl.so
adb push GGA\shim\linux-armv7sfl\rel\\<HAL\>\libnwhal.so /system/lib/libnwhal.so
adb shell chmod 0644 /system/vendor/lib/egl/libGLES_vimpl.so
adb shell chmod 0644 /system/lib/libnwhal.so
adb shell chmod 0644 /system/vendor/lib64/egl/libGLES_vimpl.so
adb shell chmod 0644 /system/lib64/libnwhal.so

Note
In these commands, the folder name <HAL> can be either of the following:
- stock, if the Android version follows the standard Android Open Source Project.
- The Gralloc version number used, such as bfrst-r7p0-01rel0, if the Android version uses a published Gralloc module.

Examining OpenGL ES execution in the graphics driver
The Error Code Check function in the Generic Graphics Accelerator examines the execution of OpenGL ES API in the target graphics driver.

About the task
Error Code Check reports APIs for which the host driver and the target driver return different error codes. For the error report that it generates, see Error messages from Error Code Check on page 8-183.

Before you begin
Make sure:
- The integration between the graphics driver and the GPU model for your Android target is successful.

For details, see Integrating the graphics driver with the GPU model on page 8-179.

Procedure
1. Open the settings.ini file.
2. In the file, assign a value to LogLevel other than 0 or 1.
For the allowed values for LogLevel, see 8.3 The configuration file on page 8-187.

3. Set checkErrorCode to 1 to enable the Error Code Check function.

   #1:Enable. 0:Disable.
   checkErrorCode 1

4. If you want to examine all OpenGL ES APIs, set:

   #1:Enable. 0:Disable.
   enableErrorCheckWhiteList 0

5. If you want to examine only specific APIs:
   a. Set enableErrorCheckWhiteList to 1.
   b. Open the checkerrcode.ini file, and set the APIs you are interested in to 1.

   Note

   checkerrcode.ini is located in the same directory as settings.ini, which is
   <installation_directory>/GGA/reconciler/linux-x86_64/gcc-x.x.x/rel/ on Linux, or
   <installation_directory>/GGA/reconciler/win_32-x86_64\c1-18.00.31101\rel\ on
   Windows.

6. Boot Android to show the API execution in the driver.
   You can use the boot command as shown in Example: Booting an Android target to verify graphics
   drivers on page 8-177.

Results

If abnormal APIs are detected, the host shows errors like the following:

ERROR [RECONCILER] gles20_g1CopyTexSubImage2D Inconsistent error code detected. host=0x0501, target=0x0502

For more details about this and other error messages, see Error messages from Error Code Check
on page 8-183.

Error messages from Error Code Check

The error messages show problematic OpenGL ES APIs for which the host driver and the target driver
have returned different error codes.

The following examples show errors that are generated by the target GPU drivers, the Generic Graphics
Accelerator, and the Mali OpenGL ES Emulator:

• Errors from the target graphics drivers:

ERROR [RECONCILER] gles20_g1CopyTexSubImage2D Inconsistent error code detected. host=0x0501, target=0x0502

   Here:
   — gles20_g1CopyTexSubImage2D is the problematic API.
   — 0x0501 and 0x0502 are the error codes retrieved from the host driver and the target driver
   respectively.

   Note

   API error codes are defined in the OpenGL ES header file.

• Errors from the Generic Graphics Accelerator:

FATAL [RECONCILER] glProgramParameteri() Could not find program object descriptor for target-side program id [0]
Here, `glProgramParameteri()` is the problematic API.

--- Note ---

Report Generic Graphics Accelerator bugs directly to the Arm account team. For more details, see 8.4 Feedback on the Generic Graphics Accelerator on page 8-188.

--- Note ---

Errors from the Mali OpenGL ES Emulator:

```
FATAL-Exception thrown in GLES32Api::glUniformMatrix4fv -> Underlying OpenGL error in GL33Backend.
See Fatal error logs for full details. This is probably a programming error, please report it.
```

--- Note ---

Report Mali emulator errors directly to the Arm account team.

For information about the Mali OpenGL ES Emulator, see *Mali OpenGL ES Emulator User Guide*.

Tracing the driver accesses to the GPU registers

You can use the Trace and Dump function provided by the Generic Graphics Accelerator to trace the modeled GPU register accesses to the graphics driver.

**Before you begin**

- You must have integrated the graphics driver with the GPU model for your Android target. For details, see *Integrating the graphics driver with the GPU model* on page 8-179.
- You require the ListTraceSources and GenericTrace plug-ins. These plug-ins list the available trace sources and specify which events should be traced, respectively. They are located in `$PVLIB_HOME/plugins/<OS_compiler>/`.

**Procedure**

1. On the host, execute the following command to list the trace sources that are provided by the GPU model:

   ```
   ${PATH_Model} --plugin $PVLIB_HOME/plugins/Linux64_GCC-5.4/ListTraceSources.so
   ```

   **Results**: The terminal shows:

   - The GPU model:

     ```
     Component (292) providing trace: Kits3_Subsys.css.gpu
     ```

   - Trace sources provided by the Mali GPU model:

     **INFO_ReadRegister**
     Outputs the access time, addresses, data, and names of the read registers.

     **INFO_Reset**
     Outputs the GPU reset data.

     **INFO_WriteRegister**
     Outputs the access time, addresses, and names of the write registers, and the data before and after write actions.

     **INFO_IrqGpuControl**
     Outputs the ID, name, and the state of the IRQ signal from the GPU. The state can be `Y` for Set, or `N` for Clear.

     **INFO_IrqJobControl**
     Outputs the ID, the name, and the state of the IRQ signal from the Job Manager on the GPU. The state can be `Y` for Set, or `N` for Clear.

     **INFO_IrqMmuControl**
     Outputs the ID, the name, and the state of the IRQ signal from the MMU on the GPU. The state can be `Y` for Set, or `N` for Clear.
WARN_ReadToWriteOnlyRegister
Outputs warning messages and addresses for the write-only registers that have been read by the graphics driver.

WARN_WriteToReadOnlyRegister
Outputs warning messages and addresses for the read-only registers that have been written by the graphics driver.

WARN_AccessToUnimplementedRegister
Outputs warning messages and addresses for the invalid registers that have been accessed by the graphics driver.

2. Boot the Android target with the command shown in Example: Booting an Android target to verify graphics drivers on page 8-177.

When you enter the command, add the following options:

```bash
--plugin $PVLIB_HOME/plugins/Linux64_GCC-5.4/GenericTrace.so \
-C TRACE.GenericTrace.trace-sources=Kits3_Subsys.css.gpu.* \
-C TRACE.GenericTrace.enabled=1 \
-C TRACE.GenericTrace.verbose=1 \
-C TRACE.GenericTrace.print-timestamp=1 \
-C TRACE.GenericTrace.trace-file=dp-trace-generic.log
```

In these options:
- Kits3_Subsys.css.gpu is the GPU model obtained from Step 1.
  - To trace all the GPU-supported registers, add the suffix "*" to this GPU. For instance, `Kits3_Subsys.css.gpu.*`.
  - To trace one type of register, add the register type name as a suffix to this GPU. For instance, `Kits3_Subsys.css.gpu.INFO_ReadRegister`.
  - To trace different types of registers, separate each type with ",". For instance, `Kits3_Subsys.css.gpu.INFO_ReadRegister, Kits3_Subsys.css.gpu.INFO_WriteRegister`

- dp-trace-generic.log is the log file used in this example to save the trace results.

--- Note ---
If the trace-file option is not used, the trace results are shown on the host terminal.

For more details about these trace options, see GenericTrace in the Fast Models Reference Manual.

3. (Optional) Run test applications to continue examining the driver accesses to the GPU.

For how to install an application, see Installing applications on Android targets on page 8-174.

**Results:**

The host terminal or the log file shows details about the driver-accessed registers, such as the register addresses, contained data, and the access time, as shown in the following output:

```
HOST_TIME=1557460.545195s INFO_ReadRegister: REG_OFFSET=0x0000000000000000 VALUE=0x60000000 REG_NAME="GPU_ID"
HOST_TIME=1557460.545266s INFO_ReadRegister: REG_OFFSET=0x000000000000000c VALUE=0x07130206 REG_NAME="L2_FEATURES"
HOST_TIME=1557460.545279s INFO_ReadRegister: REG_OFFSET=0x0000000000000000 VALUE=0x00000000 REG_NAME="SUSPEND_SIZE"
HOST_TIME=1557460.545303s INFO_ReadRegister: REG_OFFSET=0x0000000000000008 VALUE=0x00000001 REG_NAME="MEM_FEATURES"
HOST_TIME=1557460.545316s INFO_ReadRegister: REG_OFFSET=0x000000000000000c VALUE=0x00000002 REG_NAME="TILER_FEATURES"
HOST_TIME=1557460.545325s INFO_ReadRegister: REG_OFFSET=0x0000000000000010 VALUE=0x00000007 REG_NAME="JS_PRESENT"
HOST_TIME=1557460.545334s INFO_ReadRegister: REG_OFFSET=0x0000000000000014 VALUE=0x00000009 REG_NAME="MMU_FEATURES"
HOST_TIME=1557460.545345s INFO_ReadRegister: REG_OFFSET=0x0000000000000018 VALUE=0x0000000a REG_NAME="AS_PRESENT"
HOST_TIME=1557460.545359s INFO_ReadRegister: REG_OFFSET=0x000000000000001c VALUE=0x0000000b REG_NAME="JS0_FEATURES"
HOST_TIME=1557460.545370s INFO_ReadRegister: REG_OFFSET=0x0000000000000020 VALUE=0x0000000c REG_NAME="JS1_FEATURES"
HOST_TIME=1557460.545383s INFO_ReadRegister: REG_OFFSET=0x0000000000000024 VALUE=0x0000000d REG_NAME="JS2FEATURES"
HOST_TIME=1557460.545397s INFO_ReadRegister: REG_OFFSET=0x0000000000000028 VALUE=0x0000000e REG_NAME="JS3_FEATURES"
HOST_TIME=1557460.545410s INFO_ReadRegister: REG_OFFSET=0x000000000000002c VALUE=0x0000000f REG_NAME="JS4_FEATURES"
```

HOST_TIME=1557460.545424s INFO_WriteRegister: REG_OFFSET=0x0000000000000070 VALUE=0x00000000
```
Migrating the target graphics software stack to real hardware

This migration directly integrates the graphics software stack with the GPU register model without deploying the host GPU for rendering the Android graphics. This minimizes the relevance of the host to the target integration and yields a more reliable integration test.

About this task

This is the final integration check. The Generic Graphics Accelerator is disabled so the host GPU is not used. The migration test can expose sequence issues in the target graphics driver.

Before you begin

Make sure that the Android target:
• Booted with a successful integration between the graphics driver and the GPU model.
• Passed the other tests as described in this chapter.
• In the device tree, the GPU is enabled.

Procedure

1. Use the kill command to quit the current process.
2. Boot the target with the graphics integration. For details, see Integrating the graphics driver with the GPU model on page 8-179.

---------- Note ----------

When using the boot command, remove --plugin <filepath>/Sidechannel.so to disable the Generic Graphics Accelerator.

For the command details, see Example: Booting an Android target to verify graphics drivers on page 8-177.

----------

3. Resolve all issues.
4. Repeat the above steps until no issue is found.

What to next

Integrate the Android graphics software stack with the real hardware.

Related references

8.4 Feedback on the Generic Graphics Accelerator on page 8-188
8.3 The configuration file

The configuration file settings.ini configures the information to be logged by the Generic Graphics Accelerator and configures whether OpenGL ES calls are handled only by the Generic Graphics Accelerator, or additionally by the Mali driver on the target.

Location of the configuration file
You must copy settings.ini from the following directory into the directory to boot your Android target:

- On Linux: `<installation_directory>/GGA/reconciler/linux-x86_64/gcc-xxx.x.x/rel/`, where x.x.x is the GCC version number.
- On Windows: `<installation_directory>\GGA\reconciler\win_32-x86_64\cl-18.00.31101\rel\`

Options in the configuration file
The configuration options are:

- **LogLevel**: Specifies the information to be logged. It uses numbers to indicate log levels. The default setting is LogLevel 1.
  
  The numbers are:
  - 0: Represents LOG_LEVEL_OFF, which disables logs from the Generic Graphics Accelerator.
  - 1: Represents LOG_LEVEL_FATAL, which logs fatal issues from the Generic Graphics Accelerator.
  - 2: Represents LOG_LEVEL_ERROR, which only logs errors generated by the Generic Graphics Accelerator.
  - 3: Represents LOG_LEVEL_WARN, which only logs warnings generated by the Generic Graphics Accelerator.
  - 6565: Represents LOG_LEVEL_INFO, which logs the OpenGL ES API execution sequences.
  - 6566: Represents LOG_LEVEL_DEBUG, which logs the executed API names and parameters.
  - 6567: Represents LOG_LEVEL_TRACE, which logs more detailed information generated by the Generic Graphics Accelerator.

- **callOnTargetAPI**: Specifies whether the OpenGL ES APIs interact with only the Generic Graphics Accelerator or with both the Generic Graphics Accelerator and the Mali graphics driver.
  
  Values for this option are:
  - 0: APIs interact only with the Generic Graphics Accelerator.
  - 1: APIs interact with both the Generic Graphics Accelerator and the Mali graphics driver.

- **checkErrorCode**: Enables or disables the Error Code Check function. This function examines the execution of OpenGL ES APIs in the target graphics driver. This option is valid only if callOnTargetAPI is set to 1.
  
  Values for this option are:
  - 0: Disables Error Code Check.
  - 1: Enables Error Code Check.

- **enableErrorCheckWhiteList**: Specifies whether the Error Code Check function should check errors from all OpenGL ES APIs or from specific OpenGL ES APIs. This option is valid only if both callOnTargetAPI and checkErrorCode are set to 1.
  
  Values for this option are:
  - 0: To examine all APIs.
  - 1: To examine specific APIs.

For more details about the use of the Error Code Check function, see Examining OpenGL ES execution in the graphics driver on page 8-182.
8.4 Feedback on the Generic Graphics Accelerator

To report issues or bugs, send the following information to support-esl@arm.com for diagnostic purposes:

- The specific version of Fast Models.
- The Fast Models virtual platform.
- The OS of the host.
- The graphics card that is used on the host.
- Driver information for the graphics card.
- A brief description of the application, including the language that it is written in.
- The version of Android running on the target.
- A description of the issue, with the expected output and the output you observed.
- If possible, the application that is failing, or a cutdown application that reproduces the issue.
- Debug logs.
This chapter describes timing annotation. You can use this set of Fast Models features to estimate the time spent on various operations, for example instruction execution and prefetch. It allows the model to be used for basic benchmarking.

It contains the following sections:

- 9.1 Timing annotation on page 9-190.
- 9.2 Enabling and disabling timing annotation on page 9-191.
- 9.3 CPI files on page 9-192.
- 9.5 BNF specification for CPI files on page 9-198.
- 9.6 Instruction and data prefetching on page 9-200.
- 9.8 Timing annotation tutorial on page 9-203.
Timing annotation enables you to perform high-level performance estimation on Fast Models. Fast Models are Programmers View (PV) models that are targeted at software development. They sacrifice timing accuracy to achieve fast simulation execution speeds. By default, each instruction takes a single simulator clock cycle, with no delays for memory accesses.

Timing annotation enables you to perform more accurate performance estimation on SystemC models with minimal simulation performance impact. You can use it to show performance trends and to identify test cases for further analysis on approximately timed or cycle-accurate models.

You can configure the following aspects of timing annotation:

- The time that processors take to execute instructions. This can be modeled in either of the following ways:
  - As an average Cycles Per Instruction (CPI) value, using the cpi_mul and cpi_div model parameters.
  - By assigning CPI values to different instruction classes, using CPI files.
- Branch predictor type and misprediction latency. For details, see BranchPrediction in the Fast Models Reference Manual.
- Instruction and data prefetching.
- Cache and TLB latency.
- Latency caused by pipeline stalls. For details, see PipelineModel in the Fast Models Reference Manual.

Note
Timing annotation is supported on all SystemC-based platforms. However, it is disabled by default on SystemC ISIMs. To enable timing annotation for a SystemC ISIM, set the environment variable FASTSIM_DISABLE_TA to 0.

Related concepts
9.8 Timing annotation tutorial on page 9-203
9.2 Enabling and disabling timing annotation

The environment variable `FASTSIM_DISABLE_TA` can be used to enable or disable timing annotation latency.

For example, if you set `FASTSIM_DISABLE_TA` to 1 and you load a timing annotation plug-in, or use a timing annotation feature, for example CPI or cache latency modeling, none of the timing annotation latencies that are computed are injected into the model. In other words, the simulated CPU time is the same for all instructions, that is 1 cycle per instruction.

--- Note ---
You can view timing statistics by using the `--stat` parameter.

---

Disabling timing annotation does not prevent timing annotation plug-ins from working. For example, the PipelineModel plug-in continues to process instructions and generate statistics, and the BranchPrediction plug-in continues to predict branches and generate statistics files. However, any pipeline stall latencies or branch misprediction penalties that they calculate are ignored by the Fast Models simulation engine.

--- Note ---
By default, timing annotation is disabled for SystemC ISIMs. To enable it, set `FASTSIM_DISABLE_TA` to 0.
9.3 CPI files

CPI files define classes of instructions and assign CPI values to them. They increase the accuracy of the estimated number of cycles that are required to run a binary on a SystemC model.

--- Note ---

• An alternative to using CPI files is to use `cpi_mul` and `cpi_div` processor model parameters. These are integer values that represent a CPI multiplication or division factor. They are used together to represent non-integer CPI values. For example, to achieve a CPI of 1.25, use `cpi_mul = 5`, `cpi_div = 4`.
• If a CPI file is present, it overrides the `cpi_mul` and `cpi_div` parameters.
• If you do not set these parameters and do not specify a CPI file, a CPI value of 1.0 is used for all instructions.

A CPI file can support multiple instruction sets, including A64, A32, and T32. Also, it can support multiple processor types, including pre-defined and user-defined types.

Arm provides pre-defined CPI instruction classes which you can include in your CPI files, or you can define your own classes. The following files are located in `$PVLIB_HOME/etc/CPIPredefines/`:

- `ARMv8A_A64_Predefines.txt`
- `ARMv8A_A32_Predefines.txt`
- `ARMv8A_T32_Predefines.txt`
- `ARMv7M_Thumb_Predefines.txt`

You can find typical CPI values and instruction groups in the Arm Software Optimization Guides, which are available at [http://infocenter.arm.com/](http://infocenter.arm.com/).

Specify a CPI file when launching a SystemC platform model by using the `--cpi-file` command-line parameter, for example:

```bash
./EVS_Base_Cortex-A73x1 -a $PVLIB_HOME/images/brot.axf \
-C cache_state_modelled=1 \
-C bp.secure_memory=false \
--cpi-file /CPI_file.txt --stat
```

--- Note ---

The `--stat` parameter displays timing statistics on simulation exit.

Alternatively, you can specify the CPI file in your code by calling the SystemC Export API function `scx::scx_set_cpi_file()`. You must call `scx::scx_set_cpi_file()` before any call to a platform parameter function.

The `CPIValidator` command-line executable is provided to help you create valid CPI files. It is located in the `$MAXCORE_HOME/bin/` folder. Use the `--help` command-line switch to list the available options. For example, the following command parses and builds the evaluation tree for `CPI_file.txt`, and prints it in plain text to a file called `CPIEvaluationTree.txt`:

```bash
$MAXCORE_HOME/bin/CPIValidator --input-file ./CPI_file.txt --output-file ./CPIEvaluationTree.txt
```

--- Related references ---

9.4 CPI file syntax on page 9-193
9.5 BNF specification for CPI files on page 9-198
9.4 CPI file syntax

CPI files are plain text files that contain a series of statements, one per line. Any lines that begin with a # character are ignored.

In the following syntax definitions, square brackets [] enclose optional attributes. Attributes that can be repeated are followed by an ellipsis …

The valid statements in a CPI file are:

**DefineCpi**

Defines the CPI value to use for an instruction class or group. The syntax is:

```
DefineCpi class_or_group ISet=iset [CpuType=cputype] Cpi=cpi
```

where:

* **class_or_group**
  
  The name of an instruction class or group. This name can contain wildcards.
  
  A decoded instruction is matched against all DefineCpi statements in the order they appear in the CPI file from top to bottom. The first instruction class match is used and all following statements are ignored.

* **ISet=iset**
  
  Specifies which instruction set this CPI value refers to. This parameter is one of A32, A64, Thumb, or T2EE, or use the * character to specify all instruction sets.

* **CpuType=cputype**
  
  Specifies which Arm processor type this CPI value refers to. This parameter can be a user-defined type, or one of the following pre-defined types:

  - ARM_Cortex-A12
  - ARM_Cortex-A17
  - ARM_Cortex-A15
  - ARM_Cortex-A7
  - ARM_Cortex-ASMP
  - ARM_Cortex-M4
  - ARM_Cortex-M7
  - ARM_Cortex-A57
  - ARM_Cortex-A72
  - ARM_Cortex-A53
  - ARM_Cortex-R7
  - ARM_Cortex-R5
  - ARM_Cortex-A9MP
  - ARM_Cortex-A9UP
  - ARM_Cortex-A8
  - ARM_Cortex-R4
  - ARM_Cortex-M3
  - ARM_Cortex-M0+
  - ARM_Cortex-M0

  Use the * character to specify any processor type. Specifying no CpuType is equivalent to specifying CpuType=*.

* **Cpi=cpi**
  
  The CPI value to assign to this instruction class or group.

For example:

```
DefineCpi Load_instructions ISet=A64 CpuType=ARM_Cortex-A53 Cpi=2.15
```
**DefineClass**

Defines an instruction class. The syntax is:

```
DefineClass class Mask=mask Value=value [ProhibitedMask=pmask
ProhibitedValue=pvalue ...] ISet=iset [CpuType=cputype]
```

where:

- **class**
  The name of the instruction class to define. It must be unique in the CPI file. It can be used in a subsequent DefineCpi statement.

- **Mask=mask**
  A bitmask to apply to an instruction encoding before comparing the result with the Value attribute. This parameter identifies which bits in the encoding are relevant for comparing with Value.
  
  For example, the value `0000xxxx1xxx100x` is represented as `Mask=0xF08E` and `Value=0x0088`.

- **Value=value**
  The binary value to compare with the instruction encodings. A match indicates that the instruction belongs to this class, unless the encoding also matches the ProhibitedValue.

- **ProhibitedMask=pmask**
  A bitmask to apply to an instruction encoding before comparing the result with the ProhibitedValue attribute. It identifies which bits in the encoding are relevant for comparing with ProhibitedValue.

- **ProhibitedValue=pvalue**
  The binary value to compare with the instruction encodings. A match indicates that the instruction does not belong to this class.

- **ISet=iset**
  Specifies which instruction set this class refers to. See DefineCpi for the possible values.

- **CpuType=cputype**
  Specifies which Arm processor type this class refers to. See DefineCpi for the possible values.

**Note**

A DefineClass statement must include a single Mask and Value attribute pair, but can include any number of ProhibitedMask and ProhibitedValue attribute pairs.

For example:

```
DefineClass Media_instructions Mask=0x0E000010 Value=0x06000010
ProhibitedMask=0xF0000000 ProhibitedValue=0xF0000000 ISet=A32
```

**DefineGroup**

Defines a group of instruction classes. The syntax is:

```
DefineGroup group Classes=class[,class,...] ISet=iset [CpuType=cputype]
[Mix=mix[,mix,...]]
```

where:
**group**

The name of the group to define. It must be unique in the CPI file. It can be used in a subsequent `DefineCpi` statement.

**Classes=class[,class,...]**

A comma-separated list of instruction classes that belong to this group.

**ISet=iset**

Specifies which instruction set this group refers to. See `DefineCpi` for the possible values.

**CpuType=cputype**

Specifies which Arm processor type this group refers to. See `DefineCpi` for the possible values.

**Mix=mix[,mix,...]**

A comma-separated list of mixin names that cause additional instruction groups and classes to be automatically defined.

For example:

```plaintext
DefineGroup Divide_instructions Classes=SDIV,UDIV CpuType=ARM_Cortex-A73 ISet=A32
DefineMixIn my_mixin Mask=0xF0000000 Value=0xE0000000 Suffix=AL
DefineClass my_class Mask=0x0FF00000 Value=0x03000000 ISet=A32
```

**DefineMixIn**

Defines a single mask/value pair and suffix that can optionally be used in `DefineGroup` statements to automatically define new instruction groups and classes. Applying a mixin to a group causes a new instruction group or class to be defined for every instruction group or class that is included in the group, and also for the group itself. The names of these newly-defined groups and classes is the original group or class name followed by an underscore character, then the mixin suffix.

The syntax is:

```plaintext
DefineMixIn mix Mask=mask Value=value Suffix=suffix
```

where:

**mix**

The name of the mixin to define. It must be unique in the CPI file. It can be used in subsequent `DefineGroup` statements.

**Mask=mask**

A bitmask to apply to an instruction encoding before comparing the result with the `Value` attribute.

**Value=value**

The binary value to compare with the instruction encodings. A match indicates that the instruction belongs to this group or class.

**Suffix=suffix**

After applying a mixin to a group, this suffix is appended to the names of the automatically-defined groups and classes.

In the following example, the `DefineGroup` statement defines `my_group`, but also automatically defines `my_group_AL` and `my_class_AL`:

```plaintext
DefineMixIn my_mixin Mask=0xF0000000 Value=0xE0000000 Suffix=AL
... DefineClass my_class Mask=0x0FF00000 Value=0x03000000 ISet=A32
DefineGroup my_group Classes=my_class ISet=A32 Mix=my_mixin
```
Defines a processor type. The syntax is:

```
DefineCpuType cputype ISets=iset[,iset,...]
```

where:

- **cputype**
  The name of the processor type to define. It must be unique in the CPI file. It can be used in subsequent DefineCpi, DefineClass, DefineGroup, and MapCpu statements.

- **ISets=iset[,iset,...]**
  A comma-separated list of instruction sets that this processor type supports. See DefineCpi for the possible values.

For example:

```
DefineCpuType ARM_Cortex-A73 ISets=*
```

**MapCpu**

Maps a CPU instance by name to a CPU type. The syntax is:

```
MapCpu cpuinstance ToCpuType=cputype
```

where:

- **cpuinstance**
  The name of the CPU instance to map to a processor type. It can contain wildcards.

- **ToCpuType=cputype**
  The processor type to map the CPU instance onto. See the list of CpuTypes in DefineCpi for the possible values.

For example:

```
MapCpu SVP_Base_AEMv8A-AEMv8A_AEMv8A_Primary.cluster.cpu0 ToCpuType ARM_Cortex-A73
```

**Defaults**

Defines the default CPI value to be used for instructions that do not match any class or group. This statement is optional and can occur more than once in the CPI file. The syntax is:

```
Defaults ISet=iset [CpuType=cputype] Cpi=cpi
```

where:

- **ISet=iset**
  Specifies which instruction set this value refers to. See DefineCpi for the possible values.

- **CpuType=cputype**
  Specifies which Arm processor type this value refers to. See DefineCpi for the possible values.

- **Cpi=cpi**
  The default CPI value for the specified instruction set and processor type.

For example:

```
Defaults ISet=* CpuType=* Cpi=0.82
```
Include

Includes a supplementary CPI file at this point in the file. This is equivalent to the \#include preprocessor directive in C. The evaluation of the FilePath attribute is to first treat it as an absolute path, then as a relative path, and finally as relative to the PVLIB_HOME environment variable. The syntax is:

```
Include FilePath=path
```

For example:

```
Include FilePath=etc/CPIPredefines/ARMv8A_A32_Predefines.txt
```
9.5  BNF specification for CPI files

CPI files have the following BNF specification:

```bnf
cpi-file ::= statements

statements ::= statement statements | statement

statement ::= comment | define-cpi-statement | defaults-statement | map-cpu-statement | define-class-statement | define-group-statement | include-statement | define-mixin-statement

define-cpi-statement ::= "DefineCpi" <instruction-class-or-group> define-cpi-attributes

defaults-statement ::= "Defaults" <define-cpi-attributes> <eol>
define-cpu-type-statement ::= "DefineCpuType" <user-cpu-type> define-cpu-type-attributes <eol>
map-cpu-statement ::= "MapCpu" <cpu-instance> <map-cpu-attributes> <eol>
define-class-statement ::= "DefineClass" <instruction-class> define-class-attributes <eol>
define-group-statement ::= "DefineGroup" <instruction-group> define-group-attributes <eol>
include-statement ::= "Include" <include-attributes> <eol>
define-mixin-statement ::= "DefineMixIn" <mix-in-type> define-mixin-attributes <eol>
define-cpi-attributes ::= define-cpi-attribute define-cpi-attributes | define-cpi-attribute

define-cpi-attribute ::= i-set-attribute | cpu-type-attribute | cpu-type | cpu-type-attribute-attributes ::= define-cpi-attribute define-cpi-attributes | define-cpi-attribute

i-set-attribute ::= "ISet" = <iset-or-star>

iset-or-star ::= iset | *
iset ::= "A32" | "A64" | "Thumb" | "T2EE"
cpu-type ::= "ARM Cortex-A12" | "ARM Cortex-A17"
               | "ARM Cortex-A15" | "ARM Cortex-A7"
               | "ARM Cortex-A5MP" | "ARM Cortex-M4"
               | "ARM Cortex-M7" | "ARM Cortex-A57"
               | "ARM Cortex-A72" | "ARM Cortex-A53"
               | "ARM Cortex-R7" | "ARM CortexRS"
               | "ARM Cortex-A9MP" | "ARM Cortex-A9UP"
               | "ARM Cortex-A8" | "ARM Cortex-R4"
               | "ARM Cortex-M3" | "ARM Cortex-M0" | <user-cpu-type> | *

define-cpu-type-attributes ::= define-cpu-type-attribute define-cpu-type-attributes | define-cpu-type-attribute

define-cpu-type-attribute ::= i-set-attributes | cpu-type-attributes

i-set-attributes ::= i-sets attributes | i-sets

i-sets ::= i-set <iset-or-star>
i-set ::= "A32" | "A64" | "Thumb" | "T2EE" | "ARM Cortex-A12" | "ARM Cortex-A17"

cpu-type-attributes ::= define-cpu-type-attributes define-cpu-type-attributes

define-class-statement ::= define-class-attributes <eol>
define-class-attributes ::= define-class-attribute define-class-attributes | define-class-attribute

define-class-attribute ::= classes-attribute | i-set-attribute | cpu-type-attribute | mix-attribute

classes-attribute ::= "Classes" = <instruction-class-or-groups>

i-set-attribute ::= i-set-attributes | cpu-type-attribute

cpu-type-attribute ::= cpu-type-attributes

mix-attribute ::= "Mix" = <mix-in-types>

define-group-statement ::= define-group-attributes <eol>
define-group-attributes ::= define-group-attribute define-group-attributes | define-group-attribute

define-group-attribute ::= classes-attribute | i-set-attribute | cpu-type-attribute | mix-attribute

classes-attribute ::= "Classes" = <instruction-class-or-groups>

i-set-attribute ::= i-set-attributes | cpu-type-attribute

cpu-type-attribute ::= cpu-type-attributes

mix-attribute ::= mix-in-types

include-attributes ::= file-path-attribute

file-path-attribute ::= "FilePath" = <file-path>
define-mixin-attributes ::= define-mixin-attributes define-mixin-attribute | define-mixin-attribute

define-mixin-attribute ::= define-mixin-attributes define-mixin-attribute | define-mixin-attribute
```

9 Timing Annotation
9.5 BNF specification for CPI files
<table>
<thead>
<tr>
<th>DefineMixInAttribute</th>
<th>::=</th>
<th>MaskAttribute</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>ValueAttribute</td>
</tr>
<tr>
<td></td>
<td></td>
<td>SuffixAttribute</td>
</tr>
<tr>
<td>SuffixAttribute</td>
<td>::=</td>
<td>&quot;Suffix&quot; &quot;=&quot; String</td>
</tr>
<tr>
<td>FilePath</td>
<td>::=</td>
<td>String</td>
</tr>
<tr>
<td>InstructionClass</td>
<td>::=</td>
<td>Symbol</td>
</tr>
<tr>
<td>InstructionGroup</td>
<td>::=</td>
<td>Symbol</td>
</tr>
<tr>
<td>UserCpuType</td>
<td>::=</td>
<td>Symbol</td>
</tr>
<tr>
<td>CpuInstance</td>
<td>::=</td>
<td>quotedString   { Supports use of wild cards }</td>
</tr>
<tr>
<td>Cpi</td>
<td>::=</td>
<td>Double</td>
</tr>
<tr>
<td>Mask</td>
<td>::=</td>
<td>UnsignedInteger</td>
</tr>
<tr>
<td>Value</td>
<td>::=</td>
<td>UnsignedInteger</td>
</tr>
</tbody>
</table>
9.6 Instruction and data prefetching

Arm Cortex-A series processors implement prefetching of instructions and data into caches to improve the cache hit rate and to reduce the number of cycles required to execute code. Fast Models supports modeling instruction and data prefetching independently, through the use of model parameters.

This section contains the following subsections:

• 9.6.1 Configuring instruction prefetching on page 9-200.
• 9.6.2 Configuring data prefetching on page 9-200.

9.6.1 Configuring instruction prefetching

Configure instruction cache prefetching for AEMv8-A processor models by using the following cluster-level parameters:

\texttt{icache-prefetch\_enabled}

\texttt{true} to enable simulation of instruction cache prefetching, \texttt{false} otherwise. Defaults to \texttt{false}.

The execution of a branch instruction causes the model to prefetch instructions from the memory region starting at the branch target address into a number of sequential cache lines. If \texttt{true}, the following extra parameters are available:

\texttt{icache-prefetch\_level}

Specifies the zero-indexed cache level into which instructions are prefetched. Defaults to 0, which means L1.

\texttt{icache-nprefetch}

Specifies the number of additional, sequential instruction cache lines to prefetch. Defaults to 1.

\texttt{Note}

These parameters only have an effect when cache state modeling is enabled, which is controlled by the model parameter \texttt{icache\_state\_modelled}.

\texttt{Example}

The following command-line enables instruction cache prefetching and displays \texttt{WAYPOINT} trace events in the console:

\texttt{Note}

A \texttt{WAYPOINT} is a point at which instruction execution by the processor might change the program flow.

```
./FVP\_Base\_AEMv8A.exe \-a $PVLIB\_HOME/images/brot.axf \n\-C cache\_state\_modelled=1 \n\-C cluster0.icache\_prefetch\_enabled=1 \n\-C bp.secure\_memory =false \n\-\-plugin $PVLIB\_HOME/plugins/Linux64\_GCC\_5.4/GenericTrace.so \n\-C TRACE.GenericTrace.trace\_sources=WAYPOINT
```

\texttt{Related information}

\texttt{Loading a plug-in into a model}

9.6.2 Configuring data prefetching

The purpose of data prefetch modeling is to make the contents of the data cache more closely resemble those on a system with a hardware prefetcher. A default data prefetcher is supplied, which is relatively configurable. It is not intended to match any specific processor.

To run the model with data prefetch modeling enabled, using the default data prefetcher with default parameters, use the following parameters:

```
-C cache\_state\_modelled=true \-plugin "<<internal<<DataPrefetch>>" \-C cluster0.dcache\_prefetch\_enabled=1
```
When the model exits, it reports how many prefetches were issued and how many cache hits on recently-prefetched data were detected. The performance impact is about 10% compared to running with cache state modeling enabled.

By default, a data prefetch plug-in attaches to all processors and clusters in a system, and maintains independent internal state for each processor. To change this, for example if you want a different number of tracked streams on big and LITTLE cores, load the plug-in twice and pass a different .cluster parameter to each instance, for example:

```
--plugin "DP_BIG=<<internal><DataPrefetch>>" --plugin "DP_LITTLE=<<internal><DataPrefetch>>"
-C DataPrefetch.DP_BIG.cluster=0 -C DataPrefetch.DP_LITTLE.cluster=1
-C DataPrefetch.DP_BIG.lfb_entries=16 -C DataPrefetch.DP_LITTLE.lfb_entries=4
```

The names DP_BIG and DP_LITTLE are examples. They can be any names you choose.

The example preficher is a basic stride-detecting preficher, but relatively configurable using the following parameters:

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>history_length</td>
<td>Length of history to maintain.</td>
</tr>
<tr>
<td>history_threshold</td>
<td>Number of misses to allow in history before issuing a prefetch.</td>
</tr>
<tr>
<td>lfb_entries</td>
<td>Number of access streams to track.</td>
</tr>
<tr>
<td>mbs_expire</td>
<td>Number of non-hitting loads to allow before the preficher stops tracking a potential access stream.</td>
</tr>
<tr>
<td>pf_count</td>
<td>Number of prefetch streams available.</td>
</tr>
<tr>
<td>pf_tracker_count</td>
<td>Number of prefetches tracked.</td>
</tr>
<tr>
<td>pf_initial_number</td>
<td>Initial number of prefetches to issue for a new stream.</td>
</tr>
<tr>
<td>prefetch_all_levels</td>
<td>Prefetch to all cache levels rather than just the lowest level.</td>
</tr>
</tbody>
</table>

An access stream is created whenever a load is made to an address which is not within three cache lines of a previously-observed load. This might overwrite a previously created access stream. When a consistent stride has been observed, that is, when addresses \( N, N+\delta, N+2\delta \) are seen, a prefetch stream is allocated with stride \( \delta \) and a lifetime of \( \text{pf}._{\text{initial}}.\text{number} \).

Prefetches are issued in a round-robin fashion from active prefetch streams (the lifetime goes down by one each time a prefetch is issued) whenever there have been fewer than \( \text{history}.\text{threshold} \) cache misses among the last \( \text{history}.\text{length} \) loads. The rationale is that if lots of cache hits are occurring, there should be available bandwidth on the memory interface to be used by prefetching.

Issued prefetches are tracked in a circular list of size \( \text{pf}._{\text{tracker}}.\text{count} \), and if the preficher sees a load to an address in this circular list, it increments the lifetime of the prefetch stream that issued the successful prefetch.

--- Note ---

Prefetches are to physical addresses, and as a result, a prefetch stream expires when it reaches the end of a 4KB region.
9.7 Configuring cache and TLB latency

You can configure latency for different cache operations for Cortex-A processor models by setting model parameters.

The following parameters are available:

- Read access latency for L1 D-cache, L1 I-cache, or L2 cache. For example `dcache-read_access_latency`.
- Separate latencies for read hits and misses in L1 D-cache, L1 I-cache, or L2 cache. For example `dcache-hit_latency` and `dcache-miss_latency`. The total latency for a read access is the sum of the read access latency and the hit or miss latency.
- Write access latency for L1 D-cache or L2 cache. For example `dcache-write_access_latency`.
- Latency for cache maintenance operations for L1 D-cache, L1 I-cache, or L2 cache. For example `dcache-maintenance_latency`.
- Latency for snoop accesses that perform a data transfer for L1 D-cache or L2 cache. For example `dcache-snoop_data_transfer_latency`.
- Latency for snoop accesses that are issued by L2 cache. For example `l2cache-snoop_issue_latency`.
- TLB and page table walk latencies. For example `tlb_latency`.

Note

- These parameters can only be used when cache state modeling is enabled. This is controlled using parameters, for example `dcache-state_modelled` and `icache-state_modelled`.
- All of these latency values are measured in clock ticks.
- For reads and writes, latency can be specified per access, for example `dcache-read_access_latency`, or per byte, for example `dcache-read_latency`. If both parameters are set, the per-access value takes precedence over the per-byte value.
9.8 Timing annotation tutorial

This tutorial shows how to use the Cycles Per Instruction (CPI) specification and branch prediction modeling features with a Fast Models example platform model, and how to measure their impact on code execution time. The commands shown are for Linux, although the process is the same on Windows.

This section contains the following subsections:
• 9.8.1 Setting up the environment on page 9-203.
• 9.8.2 Modeling Cycles Per Instruction (CPI) on page 9-205.

9.8.1 Setting up the environment

This tutorial runs some example applications on the EVS_Base_Cortex-A73x1 example virtual platform to show different timing annotation features. This section describes the prerequisites for using timing annotation.

Prerequisites

These are the prerequisites for using timing annotation:
• A SystemC virtual platform.
• An application that enables caches.
• A way of calculating the execution time of individual instructions.
• A way of determining the total execution time of the simulation.
• A way of calculating the average Cycles Per Instruction (CPI) value for the simulation.

Building the EVS_Base_Cortex-A73x1 example

The EVS_Base_Cortex-A73x1 example includes a single EVS that is connected to SystemC components that model a timer, and an application memory component that supports individual configuration of read and write latencies. The example is not provided pre-built in the Fast Models Portfolio installation, so you must first build it.

To build the example, run the following commands:

```
cd $PVLIB_HOME/examples/SystemCExport/EVS_Platforms/EVS_Base/Build_Cortex-A73x1/
./build.sh
```

About the example applications

This tutorial uses some bare-metal example applications to show timing annotation features. The applications are provided in the $PVLIB_HOME/images/ directory.

They are based on the Arm Development Studio example project startup_AEMv8-FVP_AArch64_AC6. The relevant source code and the compiled binaries are provided. The binaries were compiled using Arm Compiler version 6.6.

<table>
<thead>
<tr>
<th>Application</th>
<th>Source directory</th>
<th>Binary file</th>
</tr>
</thead>
<tbody>
<tr>
<td>CPI example</td>
<td>$PVLIB_HOME/images/source/ta_cpi/</td>
<td>$PVLIB_HOME/images/ta_cpi.axf</td>
</tr>
<tr>
<td>Branch prediction example</td>
<td>$PVLIB_HOME/images/source/ta_brpred/</td>
<td>$PVLIB_HOME/images/ta_brpred.axf</td>
</tr>
</tbody>
</table>

Calculating the execution time of an instruction

The INST MTI trace source displays every instruction that is executed while running a program. When used with an EVS or an SVP, it also displays the current simulation time after an instruction has completed executing.
To determine the number of ticks an instruction takes to execute, you can use the difference between the times of two consecutive instructions. The default is one tick (on the core) for each instruction. With the default clock speed of 100MHz, this gives a default execution time for an instruction of 10000 picoseconds. Any changes to latency due to branch mispredictions, memory accesses, or CPI specifications can be observed by comparison with this value.

This tutorial uses the `INST` trace source to measure the time it takes to execute an instruction. To generate trace, you can use the `GenericTrace` plugin. This plugin allows you to output any number of MTI trace sources to a text file.

Use the following extra parameters when launching the model to collect the `INST` trace source:

```bash
--plugin=$PVLIB_HOME/plugins/Linux64_GCC-5.4/GenericTrace.so \
-C TRACE.GenericTrace.trace-sources=INST \
-C TRACE.GenericTrace.trace-file=/path/to/trace/file.txt
```

In an application that uses timing annotation, the trace that is produced for the first two instructions might look like this:

```
INST: PC=0x0000000080000000 OPCODE=0x58001241 SIZE=0x04 MODE=EL3h ISET=AArch64
PADDR=0x0000000000000000 NSDESC=0x00 PADDR2=0x0000000000000000 NSDESC2=0x00 NS=0x00
ITSTATE=0x00 INST_Count=0x00000000000001388 LOCAL_TIME=0x00000000000001388 CORE_NUM=0x00
DISASS="LDR x1,(pc)+0x248 ; 0x80000248"

INST: PC=0x0000000080000004 OPCODE=0xd518c001 SIZE=0x04 MODE=EL3h ISET=AArch64
PADDR=0x0000000000000000 NSDESC=0x00 PADDR2=0x0000000000000000 NSDESC2=0x00 NS=0x00
ITSTATE=0x00 INST_Count=0x00000000000001388 LOCAL_TIME=0x00000000000001388 CORE_NUM=0x00
DISASS="MSR VBAR_EL1,x1"
```

The `CURRENT_TIME` value for the first instruction is 0x1388, or 5000ps. This value shows that the instruction took 0.5 ticks to execute. Timing annotation has halved the execution time of this instruction.

The difference between the `CURRENT_TIME` values of the two instructions is 0x2710, or 10000 picoseconds. This value shows that the second instruction took one tick to execute.

**Related references**

- `MTI trace sources` on page 9-212

**Related information**

- `GenericTrace`

**Displaying the total execution time of the simulation**

You can use MTI trace to calculate the execution time of individual instructions. However, to determine the overall simulation time, use the command-line option `--stat` instead.

This option causes the model to print statistics about the simulation to the terminal on exiting. The statistics include `Simulated time`, which is the total simulation time in seconds. For example:

```
--- Base statistics: -----------------------------------------------
Simulated time : 0.001206s
User time : 0.276000s
System time : 0.136000s
Wall time : 0.700834s
Performance index : 0.42 MIPS (172289 Inst)
Base.cluster0.cpu0
```

---

**Note**

The MIPS value is based on the host system time, not the simulated time.

This tutorial uses the `--stat` option to compare the model’s performance for an application with various timing annotation configurations.
Calculating the average CPI value

You can calculate the average CPI value for the simulation by using the instruction count and the simulated time value, which are displayed using the --stat option.

Use the following formula:

\[ \text{average_cpi} = \frac{\text{simulated_time_in-picoseconds}}{10000 \times \text{instruction_count}} \]

In this example, the average CPI value is 0.69999, which is calculated as follows:

\[ \text{average_cpi} = \frac{(0.001206 \times 10^{12})}{(10000 \times 172289)} = 0.69999 \]

9.8.2 Modeling Cycles Per Instruction (CPI)

This section demonstrates how to precisely model the simulated time per instruction by using the CPI timing annotation feature.

CPI parameters

You can specify a single CPI value for all instructions that execute within a cluster. This value is referred to as a fixed CPI value. Alternatively, you can use a custom CPI file to define individual CPI values for specific instructions. Use a fixed CPI value instead of a CPI file when precise per-instruction modeling is not required.

When running a simulation with either of these options, you can calculate the average CPI value using the formula that is shown in Calculating the average CPI value on page 9-205.

Note

You can combine the CPI specification with other timing annotation features. Therefore, the average CPI value that you observe can be different from the fixed CPI value that you specify.

Specifying a fixed CPI value

You can specify a fixed CPI value by using the per-cluster model parameters cpi_mul and cpi_div.

These parameters are documented in the Fast Models Reference Manual. By default, a fixed CPI value of 1.00 is used. The values that you specify in these parameters must be integers. Using them, any arbitrary value can be generated and is applied to all instructions during execution within that cluster. The value is used in a way that \( \text{core_clock_period} \times \text{fixed_cpi_value} \) is rounded to the nearest picosecond.

Related concepts

Running the example with a fixed CPI value on page 9-209

Example CPI file

CPI files can be large because they have to cover multiple encodings for many of the instructions that are included. Various predefined encodings are provided under \$PVLIB_HOME/etc/CPIPredefines/ that can help you to create CPI files. This tutorial does not use predefined encodings.

The following example defines CPI values for the instructions ADRP, ADR, ADD, CMP, ORR, LDP, STR, branches, exception generating instructions, and system instructions. It defines a default CPI value of 0.75 for all other instructions. It applies to the A64 instruction set, and does not restrict the values to a specific core.

Note

These CPI values are an example only. They are arbitrary and are not representative of any Arm processor.
Related references

9.4 CPI file syntax on page 9-193

Defining CPI values in a CPI file

To define CPI values in a CPI file, use the following procedure for each instruction or set of instructions:

Procedure
1. Create an instruction class for each encoding of an instruction or set of instructions by using the DefineClass keyword.
2. Group instruction classes by using the DefineGroup keyword.
3. Set a CPI value for each instruction class or group of classes by using the DefineCpi keyword.

The encodings for each instruction in the A64 instruction set are provided by the Armv8-A Architecture Reference Manual, section C6.2. Also, groups of instructions that share encodings are described in chapter C4. You can use these encodings to define the Mask and Value fields in the CPI file.

The Mask field must cover all bits that are fixed in the encoding of an instruction. The Value field must specify the value of these bits. For example, section C4.2.6 of the Armv8-A Architecture Reference Manual defines a set of instructions called PC-rel. addressing. In the example CPI file, the following statements specify a common CPI value for these instructions:

```
DefineClass ADRP Mask=0x9F000000 Value=0x90000000 ISet=A64
DefineClass ADR Mask=0x9F000000 Value=0x10000000 ISet=A64
```
DefineGroup PC_rel_addr_instr Classes=ADRP,ADR ISet=A64
DefineCpi PC_rel_addr_instr ISet=A64 Cpi=0.25

For both instruction classes, the Mask value has bit[31] set to 0b1 and bits [28:24] set to 0b1111. As shown in the reference manual, a value of 0b1000 for bits [28:24] identifies the instruction as being ADR or ADRP. Therefore, both Value fields set bits [28:24] to 0b1000. Bit[31] distinguishes between ADR and ADRP, so bit[31] in the Value field for ADR is set to 0b0 and to 0b1 for ADRP.

This specification allows the model to specify a CPI value of 0.25 for the PC_rel_addr_instr group of instructions. A similar process has been followed to determine the Mask and Value fields for the other instructions in the CPI file example.

Related references
9.4 CPI file syntax on page 9-193

Related information
Arm Architecture Reference Manual

Validating a CPI file
To validate CPI files, use the CPIValidator tool. You can find this tool with the Fast Models Tools under $MAXCORE_HOME/bin/. The tool can detect missing or incompatible instruction groups and classes, but cannot validate the encodings themselves.

For example, if you remove the DefineClass statement for the B_gen_except_sys instruction class, and validate the example CPI file by using the following command:

CPIValidator --input-file /path/to/custom_cpi.txt --output-file cpi_evaluation.txt

the tool produces the following output:

ERROR: Instruction Class 'B_gen_except_sys' has no definition, when Instruction Set is 'A64' and the CPU Type is 'Default ARM Core'.
ERROR: Processing error in file /path/to/custom_cpi.txt

Using the tool with the complete CPI file produces the following output:

Core Performance Profile: Default ARM Core

Instruction Set: A32 Default Cpi:0.75
Instruction Set: A64 Default Cpi:0.75

The example CPI file and the CPIValidator output are provided in $PVLIB_HOME/images/source/ta_cpi/, see custom_cpi.txt and cpi_evaluation.txt respectively.

CPI class example program
The example program is designed to show the effect of the CPI values that are specified in the example CPI file that was described previously.
It includes the following sequence of embedded assembly code that uses instructions for which specific CPI values were defined:

```
.section asm_func, "ax"
.global  asm_cpi
.type    asm_cpi, "function"
asm_cpi:
    ldp  w1, w2, [x0]
    cmp  w1, w2
    b.gt skip
    orr  w1, w1, w2
    str  w1, [x0]
skip:
    ret
```

This sequence checks if the second value in a two-element array pointed to by the address in x0 is greater than the first value. If so, it performs a bitwise OR operation using the two values, storing the result as the new first value. The rest of this section examines this sequence by running the example on the EVS_Base_Cortex-A73x1 platform model with the following CPI configurations:

- Using the default CPI value.
- Using the custom CPI file that was described earlier in the tutorial.
- Using a fixed CPI value.

The CPI class example is based on the Arm Development Studio example startup_AEMv8-FVP_AArch64_AC6. The binary file is $PVLIB_HOME/images/ta_cpi.axf, and the source code is available under $PVLIB_HOME/images/source/ta_cpi/.

### Running the example with the default CPI value

If you do not specify any CPI parameters, a default CPI value of 1.00 is used. This value establishes a baseline to compare with the other CPI configurations.

To use the default CPI value of 1.00, launch the model using the following command:

```
$PVLIB_HOME/examples/SystemCExport/EVS_Platforms/EVS_Base/Build_Cortex-A73x1/EVS_Base_Cortex-A73x1.x \
-C Base.bp.secure_memory=0 \ 
--plugin=$PVLIB_HOME/plugins/Linux64_GCC-5.4/GenericTrace.so \ 
-C TRACE.GenericTrace.trace-sources=INST \ 
-C TRACE.GenericTrace.trace-file=trace.txt \ 
-a $PVLIB_HOME/images/ta_cpi.axf \ 
--stat
```

In the trace file that the GenericTrace plugin produces, find the instruction at address 0x800005a4. The trace for this instruction and the one before it is as follows:

```
INST: PC=0x00000000000005a0 OPCODE=0x910003fd SIZE=0x04 MODE=El1h ISET=AArch64
      PADDR=0x00000000000005a0 NSDESC=0x0f PADADDR=0x00000000000005a0 NSDESC2=0x0f NS=0x01
      ITSTATE=0x00 INST_COUNT=0x0000000000000000 LOCAL_TIME=0x000000000000007530
      CURRENT_TIME=0x0000000001000000 CORE_NUM=0x00 DISASS="MOV      x29,sp"

INST: PC=0x0000000000005a4 OPCODE=0x90000020 SIZE=0x04 MODE=El1h ISET=AArch64
      PADDR=0x0000000000005a4 NSDESC=0x0f PADADDR=0x0000000000005a4 NSDESC2=0x0f NS=0x01
      ITSTATE=0x00 INST_COUNT=0x000000000000007bc LOCAL_TIME=0x000000000000007530
      CURRENT_TIME=0x0000000000000000 CORE_NUM=0x00 DISASS="ADRP     x0,{pc}+0x4000 ; 0x800045a4"
```

Using the CURRENT_TIME values, it can be observed that the instruction took 10000ps or 1 tick to complete, which shows the default CPI value of 1.00 is being used. You can verify that all other instructions are also using the default CPI value by examining the trace.

### Running the example with a custom CPI file

To use the custom CPI file, launch the model using the following command:

```
$PVLIB_HOME/examples/SystemCExport/EVS_Platforms/EVS_Base/Build_Cortex-A73x1/EVS_Base_Cortex-A73x1.x \
-C Base.bp.secure_memory=0 \ 
--plugin=$PVLIB_HOME/plugins/Linux64_GCC-5.4/GenericTrace.so \ 
-C TRACE.GenericTrace.trace-sources=INST \ 
-C TRACE.GenericTrace.trace-file=trace.txt \ 
-a $PVLIB_HOME/images/ta_cpi.axf \ 
--cpi-file $PVLIB_HOME/images/source/ta_cpi/custom_cpi.txt \ 
--stat
```

Using the trace output that the GenericTrace plugin produces for the 10 instructions starting at address 0x800005a4, and the --stat output, the following information can be obtained for the embedded assembly code sequence in the example program:

### Table 9-3  CPI values for embedded assembly instructions

<table>
<thead>
<tr>
<th>Address</th>
<th>Instruction</th>
<th>Simulated time (ps)</th>
<th>CPI value observed</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x800005a4</td>
<td>ADRP x0,{pc}+0x4000</td>
<td>2500</td>
<td>0.25</td>
</tr>
<tr>
<td>0x800005a8</td>
<td>ADD x0,x0,#0x9f0</td>
<td>5000</td>
<td>0.50</td>
</tr>
<tr>
<td>0x800005ac</td>
<td>ADD x1,x0,#4</td>
<td>5000</td>
<td>0.50</td>
</tr>
<tr>
<td>0x800005b0</td>
<td>BL {pc}+0x4294</td>
<td>10000</td>
<td>1.00</td>
</tr>
<tr>
<td>0x80004844</td>
<td>LDP w1,w2,[x0,#8]</td>
<td>20000</td>
<td>2.00</td>
</tr>
<tr>
<td>0x80004848</td>
<td>CMP w1,w2</td>
<td>7500</td>
<td>0.75</td>
</tr>
<tr>
<td>0x8000484c</td>
<td>B.GT {pc}+0xc</td>
<td>10000</td>
<td>1.00</td>
</tr>
<tr>
<td>0x80004850</td>
<td>ORR w1,w1,w2</td>
<td>5000</td>
<td>0.50</td>
</tr>
<tr>
<td>0x80004854</td>
<td>STR w1,[x8,#0]</td>
<td>10000</td>
<td>1.00</td>
</tr>
<tr>
<td>0x80004858</td>
<td>RET</td>
<td>10000</td>
<td>1.00</td>
</tr>
</tbody>
</table>

This table shows that the CPI values that are defined in the example CPI file have been applied to the appropriate instructions.

The following information can be obtained for the simulation as a whole:

### Table 9-4  Statistics for the whole simulation

<table>
<thead>
<tr>
<th>Total number of instructions</th>
<th>Overall simulated time in seconds</th>
<th>Average CPI value</th>
</tr>
</thead>
<tbody>
<tr>
<td>47701</td>
<td>0.000362</td>
<td>0.75889</td>
</tr>
</tbody>
</table>

**Note**

The average CPI value being close to the default CPI value specified in the CPI file does not signify anything by itself. To draw any conclusions from it, further analysis on the distribution of instructions would be required.

**Running the example with a fixed CPI value**

The average CPI value that was observed when running the example program with the custom CPI file is approximately 0.75889. Fractionally, the exact value is 36200/47701.

This fraction can be applied to the simulation by using the cpi_mul and cpi_div model parameters as follows:

```
$PVLIB_HOME/examples/SystemCExport/EVS_Platforms/EVS_Base/Build_Cortex-A73x1/EVS_Base_Cortex-A73x1.x \
-C Base.bp.secure_memory=0 \
--plugin=$PVLIB_HOME/plugins/Linux64_GCC-5.4/GenericTrace.so \
-C TRACE.GenericTrace.trace-sources=INST \
-C TRACE.GenericTrace.trace-file=trace.txt \
-C Base.cluster0.cpi_mul=36200 \
-C Base.cluster0.cpi_div=47701 \
-a $PVLIB_HOME/images/ta_cpi.axf \
--stat
```
For each instruction, a simulated time of 7589ps or 0.7589 ticks can be observed using the GenericTrace plugin. The --stat output is as follows and shows the same simulated time value as that obtained using the custom CPI file:

```
--- Base statistics: ----------------------------------------------------------
Simulated time                          : 0.000362s
User time                               : 0.171601s
System time                             : 0.015601s
Wall time                               : 0.196000s
Performance index                       : 0.00
Base.cluster0.cpu0                      : 0.25 MIPS (47701 Inst)
```

In this case, because the same application was run with the custom CPI file and with the average CPI value, an approximation of the average CPI value shows the same overall simulated time. However, the average CPI value for one application is not necessarily an accurate approximation of the average CPI value for a different application.

For example, running the branch prediction example application, described in the next section, clearly shows this difference. Specifying a branch misprediction latency increases the overall simulated time, and therefore gives a different average CPI value to the fixed CPI value that was specified. Using the custom CPI file produces a more accurate average CPI value for the branch prediction example.

### Table 9-5 CPI values for simulation with branch prediction latency

<table>
<thead>
<tr>
<th>Branch prediction example CPI configuration</th>
<th>Overall simulated time in seconds</th>
<th>Average CPI value</th>
</tr>
</thead>
<tbody>
<tr>
<td>Using the average CPI value that was observed in the CPI class example program.</td>
<td>0.001726</td>
<td>1.00754</td>
</tr>
<tr>
<td>Using the custom CPI file.</td>
<td>0.001945</td>
<td>1.13538</td>
</tr>
</tbody>
</table>

**Related concepts**

*Branch prediction example program on page 9-213*

### 9.8.3 Modeling branch prediction

This section demonstrates various techniques for measuring the effectiveness of different branch prediction algorithms.

**Branch predictor types and parameters**

The BranchPrediction plugin allows you to select the branch prediction algorithm to use, the type of statistics to collect, and the misprediction latency.

The plugin parameters that are used in this tutorial are as follows:

### Table 9-6 BranchPrediction plugin parameters

<table>
<thead>
<tr>
<th>Plugin parameter</th>
<th>Purpose in this example</th>
<th>Values that are used in this example</th>
</tr>
</thead>
</table>
| predictor-type   | Comparing the impact of different branch prediction algorithms. | • FixedDirectionPredictor  
• BiModalPredictor  
• GSharePredictor  
• CortexA53Predictor |
| mispredict-latency | Simulating the additional latency due to a pipeline flush that is caused by a branch misprediction. | 11. This value is the minimum pipeline flush length for a Cortex-A73 processor. |
| bpstat-pathfilename | Providing statistics about the branch prediction behavior, to determine per-branch and overall predictor accuracy. | stats.txt |
The different predictor types that are used in this example behave as follows:

**FixedDirectionPredictor**
Always predicts branches as **TAKEN**.

**BiModalPredictor**
Uses a 2-bit state machine to classify branches as one of **STRONGLY_NOT_TAKEN**, **WEAKLY_NOT_TAKEN**, **WEAKLY_TAKEN**, or **STRONGLY_TAKEN**, and predicts accordingly. Tracks up to 512 individual branch instructions by address.

**GSharePredictor**
Uses the history of the eight most recently executed branch instructions to classify a set of branch instructions, based on the instruction address, as one of **STRONGLY_NOT_TAKEN**, **WEAKLY_NOT_TAKEN**, **WEAKLY_TAKEN**, or **STRONGLY_TAKEN**, and predicts accordingly. Unlike the **BiModalPredictor**, it is not limited to a specific number of branch instruction addresses, but it is less precise than **BiModalPredictor**.

**CortexA53Predictor**
Implements the Cortex-A53 branch prediction algorithm.

To help you understand the algorithms in more detail, the source code for these branch predictors, except **CortexA53Predictor**, is provided under `$PVLIB_HOME/plugins/source/BranchPrediction/`.

**Related information**

**BranchPrediction**

**Generating branch misprediction statistics**

There are two ways to trace branch mispredictions when running an application:

- Use the statistics that are produced by the **BranchPrediction** plugin to get an overall picture, without context about the execution order.
- Load the **BranchPrediction** plugin and use the MTI trace sources **INST**, **BRANCH_MISPREDICT**, and **WAYPOINT** to see branch misprediction details for individual instructions in execution order.

**BranchPrediction plugin statistics**

The statistics feature of the **BranchPrediction** plugin provides overall and per-branch statistics, which are saved to a file when the model exits. You can specify the filename and location using the **bpstat-pathfilename** parameter.

The overall branch prediction statistics are described in the following table:

<table>
<thead>
<tr>
<th>Statistic</th>
<th>Description</th>
<th>Example</th>
</tr>
</thead>
<tbody>
<tr>
<td>Processor Core</td>
<td>Name of the core to which the branch prediction plugin was connected.</td>
<td>ARM_Cortex-A73</td>
</tr>
<tr>
<td>Cluster instance</td>
<td>The cluster number in the processor.</td>
<td>0</td>
</tr>
<tr>
<td>Core instance</td>
<td>The core number in the cluster.</td>
<td>0</td>
</tr>
<tr>
<td>Mispredict Latency</td>
<td>The branch misprediction latency as specified using the mispredict-latency parameter.</td>
<td>11</td>
</tr>
<tr>
<td>Image executed</td>
<td>The name of the application file that was executed.</td>
<td>ta_bpred.axf</td>
</tr>
<tr>
<td>PredictorType</td>
<td>The branch prediction algorithm as specified using the predictor-type parameter.</td>
<td>FixedDirectionPredictor</td>
</tr>
</tbody>
</table>
Table 9-7  Overall statistics (continued)

<table>
<thead>
<tr>
<th>Statistic</th>
<th>Description</th>
<th>Example</th>
</tr>
</thead>
<tbody>
<tr>
<td>Total branch calls</td>
<td>The total number of times all branch instructions were executed.</td>
<td>37434</td>
</tr>
<tr>
<td>Total Mispredictions</td>
<td>The total number of mispredictions for all executed branch instructions.</td>
<td>5106</td>
</tr>
<tr>
<td>Average prediction</td>
<td>The fraction of all branch instructions that were correctly predicted.</td>
<td>0.8636</td>
</tr>
<tr>
<td>accuracy</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Conditional Branches</td>
<td>The total number of unique conditional branch instructions. This figure does not include the instructions CBZ and CBNZ.</td>
<td>123</td>
</tr>
<tr>
<td>Total unique branch</td>
<td>The total number of unique conditional and unconditional branch instructions.</td>
<td>300</td>
</tr>
</tbody>
</table>

The following table shows the BranchPrediction plugin statistics for each unique branch instruction. They can be used to analyze how a given branch prediction algorithm behaves with a particular type of branch instruction. The branch prediction example program uses this information to determine how effectively the different branch prediction algorithms predict different types of branches.

Table 9-8  Per-branch statistics

<table>
<thead>
<tr>
<th>Statistic</th>
<th>Description</th>
<th>Example</th>
</tr>
</thead>
<tbody>
<tr>
<td>PC Addr</td>
<td>The address of the branch instruction.</td>
<td>0x8000062c</td>
</tr>
<tr>
<td>Calls</td>
<td>The total number of times the branch was called.</td>
<td>2100</td>
</tr>
<tr>
<td>Mispredict</td>
<td>The total number of times the branch was mispredicted.</td>
<td>260</td>
</tr>
<tr>
<td>Accuracy</td>
<td>The fraction of calls to the branch instruction that were correctly predicted.</td>
<td>0.87619</td>
</tr>
</tbody>
</table>

Related concepts
Branch prediction example program on page 9-213

Related references
Branch predictor types and parameters on page 9-210

MTI trace sources

The INST trace source, described earlier in this tutorial, can be used to show the latency that is added to the instruction execution time by a branch misprediction.

Whenever the BranchPrediction plug-in makes a branch misprediction, the BRANCH_MISPREDICT trace source prints the address of the branch instruction that was mispredicted. This address can be compared with the address from the corresponding INST trace event to determine the exact branch instruction involved. The number of BRANCH_MISPREDICT entries for a given branch address at the end of the simulation matches the Mispredict count for that address that is shown in the BranchPrediction plug-in statistics file.

The WAYPOINT trace source prints an event whenever an effective branch operation takes place. This event includes the address of the branch instruction, the target address of the branch, whether the branch is conditional, and whether it was taken. This trace source requires instruction prefetching to be enabled. Combined with a BRANCH_MISPREDICT trace event, it can be used to determine whether a branch was mispredicted as TAKEN or NOT_TAKEN.

To collect trace from these sources, run the model with the GenericTrace and BranchPrediction plugins. For example:

```
$PVLIB_HOME/examples/SystemCExport/EVS_Platforms/EVS_Base/Build_Cortex-A73x1/EVS_Base_Cortex-A73x1.x \
-C Base.bpsecure_memory=0 \
```
Related concepts

Calculating the execution time of an instruction on page 9-203

Example trace for a branch misprediction

The following example trace is for a branch misprediction with a misprediction latency of 11 ticks:

```
INST: PC=0x0000000000000062 OP CODE=0x7100655f SIZE=0x04 MODE=EL1h ISET=AArch64
PADDR=0x0000000000000028 NSDESC=0x01 PADDR2=0x0000000000000062 NSDESC2=0x01
ITSTATE=0x00 INST_COUNT=0x00000000000180b LOCAL_TIME=0x000000000003f7a0
CURRENT_TIME=0x000000000002eb3a0 CORE_NUM=0x00 DISASS="CMP w10,#0x19"

INST: PC=0x0000000000000062c OP CODE=0x54000168 SIZE=0x04 MODE=EL1h ISET=AArch64
PADDR=0x000000000000002c NSDESC=0x01 PADDR2=0x000000000000002c NSDESC2=0x01
ITSTATE=0x00 INST_COUNT=0x00000000000180c LOCAL_TIME=0x0000000000041eb9
CURRENT_TIME=0x000000000002eb3a0 CORE_NUM=0x00 DISASS="B.HI \{pc}+0x2c ; 0x80000658"
WAYPOINT: PC=0x0000000000000062c ISET=AArch64 TARGET=0x0000000000000658
TARGET_ISET=AArch64 TAKEN=N IS_COND=Y CORE_NUM=0x00
BRANCH_MISPREDICT: PC=0x0000000000000062c

INST: PC=0x00000000000000630 OP CODE=0x7100151f SIZE=0x04 MODE=EL1h ISET=AArch64
PADDR=0x000000000000006c NSDESC=0x01 PADDR2=0x000000000000006c NSDESC2=0x01
ITSTATE=0x00 INST_COUNT=0x00000000000180d LOCAL_TIME=0x0000000000041eb0
CURRENT_TIME=0x000000000002eb3a0 CORE_NUM=0x00 DISASS="CMP w8,#5"
```

The following information can be gathered from this trace:

- The branch instruction at address 0x800062c was mispredicted, as shown by the BRANCH_MISPREDICT trace event.
- The branch was conditional, and was incorrectly predicted as TAKEN, as shown by the TAKEN=N field in the WAYPOINT trace event. The PC field value from this source must correspond to the PC field value from the BRANCH_MISPREDICT source.
- As a result of the misprediction, the instruction following the branch instruction took 120,000 picoseconds, or 12 ticks to complete. The misprediction latency was defined as 11 ticks, so the instruction would have taken only 1 tick to complete if the branch had been predicted correctly. The execution time is the difference between:
  - The CURRENT_TIME value for the INST trace before the BRANCH_MISPREDICT trace.
  - The CURRENT_TIME value for the INST trace after the BRANCH_MISPREDICT trace.

The branch instruction itself took 10,000 picoseconds, or one tick to complete. This is important, as it shows that the misprediction latency is added to the instruction after the mispredicted branch instruction, not to the branch instruction itself. The execution time is the difference between the CURRENT_TIME values for the INST traces corresponding to the branch instruction and the instruction before.

The rest of this tutorial uses these techniques to compare the different branch prediction algorithms.

Branch prediction example program

The example is designed to use various types of branch operations that can take place during the execution of a program.

These operations are:
- A branch to skip a loop after a fixed number of iterations has completed.
- A branch to skip a code sequence, depending on the value of a variable.
• A branch to skip a code sequence, which can only be executed a limited number of times consecutively, if a previous branch was taken.
• A branch for a condition that is always true if the conditions for two previous branches were true.
• A branch for a condition that is always true if the conditions for two previous branches were false.

The code operation is trivial. It looks for acronyms of a set maximum length within a constant string, and loops over this operation a set number of times. The string is:

```
Timing annotation can be used with an SVP, Split Virtual Platform, or an EVS, Exported Virtual Subsystem.
```

The code prints the acronyms SVP and EVS during each search operation. The complete source is provided in the file `src/ta_brpred/main.c`. The following code snippet shows the branch operations of interest:

```
// A: loop not entered 1/LOOP_COUNT times
for(j = 0; j < LOOP_COUNT; j++) {
    printf("Starting iteration \#%d\n", j);
    blockCount = 0;
    c = 0;
    resetOnly(&acronymLength, acronym);
    // B: loop not entered 1/length times
    for(i = 0; i < length; i++) {
        c = string[i];
        // C: condition true
        // (number_of_block_letters)/(total_characters_in_string) times
        if (c >= 'A' && c <= 'Z') {
            blockCount++;
        }
        // D: condition true up to MAX_LENGTH times consecutively
        if (acronymLength < MAX_LENGTH) {
            acronym[acronymLength] = c;
        }
        // E: condition true up to MAX_LENGTH+1 times consecutively
        if (acronymLength <= MAX_LENGTH) {
            acronymLength++;
        }
    }
    else {
        // F: condition true if E was true then C was false
        if (acronymLength > 1 && acronymLength <= MAX_LENGTH) {
            printAndReset(&acronymLength, acronym);
        }
        // G: condition true if E was false then C was false
        else if (acronymLength != 0) {
            resetOnly(&acronymLength, acronym);
        }
    }
}
```

The branch instructions that are assembled for the conditions A to G in this code snippet can be examined using the branch prediction statistics and trace sources described previously.

The conditions are described in the following table. The branch behavior column describes the relationship between the condition and the associated branch instruction.

<table>
<thead>
<tr>
<th>Condition</th>
<th>Description</th>
<th>Compiled instruction</th>
<th>Branch behavior</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>Outer loop for processing string LOOP_COUNT times. Loop not entered 1/LOOP_COUNT times.</td>
<td>B.NE 0x800005f4 at address 0x80000698.</td>
<td>Backwards branch. Taken to start of loop if more iterations remain.</td>
</tr>
<tr>
<td>B</td>
<td>Inner loop for iterating through characters in the string.</td>
<td>B.NE 0x80000618 at address 0x8000068c.</td>
<td>Backwards branch. Taken to start of loop if more iterations remain.</td>
</tr>
<tr>
<td>C</td>
<td>Condition true if the character being processed is upper case.</td>
<td>B.HI 0x80000658 at address 0x8000062c.</td>
<td>Forwards branch. Taken if the condition is false. Skips code that handles upper case characters.</td>
</tr>
<tr>
<td>D</td>
<td>Condition true up to MAX_LENGTH times consecutively.</td>
<td>B.GE 0x80000644 at address 0x80000634.</td>
<td>Forwards branch. Taken if the condition is false. Skips code that appends a letter to an acronym.</td>
</tr>
</tbody>
</table>
Table 9-9 Branch behavior for each condition (continued)

<table>
<thead>
<tr>
<th>Condition</th>
<th>Description</th>
<th>Compiled instruction</th>
<th>Branch behavior</th>
</tr>
</thead>
<tbody>
<tr>
<td>E</td>
<td>Condition true up to MAX_LENGTH +1 times consecutively.</td>
<td>B.GT 0x80000684 at address 0x80000648.</td>
<td>Forwards branch. Taken if the condition is false. Skips code that increments the acronym length.</td>
</tr>
<tr>
<td>F</td>
<td>Condition true if E was true, after which C was false.</td>
<td>B.HI 0x80000674 at address 0x80000660.</td>
<td>Forwards branch. Never taken if the condition was true, that is, branch E was not taken and then branch C was taken. Skips the code to print a completed acronym.</td>
</tr>
<tr>
<td>G</td>
<td>Condition true if E was false, after which C was false.</td>
<td>CBZ w8,0x80000684 at address 0x80000674.</td>
<td>Forwards branch. Never taken if the condition was true, that is, branch E was taken then branch C was taken. Skips the code to clear the saved acronym.</td>
</tr>
</tbody>
</table>

LOOP_COUNT and MAX_LENGTH are defined using a preprocessor macro and can be configured. This tutorial assumes that LOOP_COUNT is 20 and MAX_LENGTH is 5, as defined in the pre-compiled binary.

Running the simulation

To generate trace and statistics for comparing the performance of the different branch predictors, run the simulation with the BranchPrediction plug-in parameters shown here.

For example, to use the FixedDirectionPredictor, launch the model using the following command:

```
$PVLIB_HOME/examples/SystemCExport/EVS_Platforms/EVS_Base/Build_Cortex-A73x1/EVS_Base_Cortex-A73x1.x \
-Dbp.secure_memory=0 \
-Dbase.cache_state_modelled=1 \
--plugin=$PVLIB_HOME/plugins/Linux64_GCC-5.4/BranchPrediction.so \
-Dbp.BranchPrediction.BranchPrediction.predictor-type=FixedDirectionPredictor \
-Dbp.BranchPrediction.BranchPrediction.mispredict-latency=11 \
-Dbp.BranchPrediction.BranchPrediction.bpstat-pathfilename=stats.txt \
--plugin=$PVLIB_HOME/plugins/Linux64_GCC-5.4/GenericTrace.so \
-Dbp.TRAY.GenericTrace.trace-sources=INST,BRANCH_MISPREDICT,WAYPOINT \
-Dbp.TRAY.GenericTrace.trace-file=trace.txt \
-a $PVLIB_HOME/images/ta_brpred.axf \
--stat
```

The program prints the following output to the terminal:

```
Looking for acronyms of maximum length 5 in the string:
Timing annotation can be used with an SVP, Split Virtual Platform, or an EVS, Exported Virtual Subsystem.
Starting iteration #0 
SVP 
EVS
Starting iteration #19 
SVP 
EVS
Info: /OSCI/SystemC: Simulation stopped by user.
--- Base statistics: -----------------------------------------------
Simulated time : 0.002275s
User time : 0.343203s
System time : 0.202801s
Wall time : 0.648064s
Performance index : 0.38
Base.cluster0.cpu0 : 0.31 MIPS ( 171308 Inst)
```

You can now analyze the end of simulation statistics, the branch prediction statistics file stats.txt, and the MTI trace file trace.txt, that are generated for each branch predictor type.

Related references

Branch predictor types and parameters on page 9-210
Comparison of branch predictor types

Statistics about the accuracy of the different branch predictors for the various types of branch instructions can now be compared.

These statistics are shown in the following table:

<table>
<thead>
<tr>
<th>Branch predictor</th>
<th>Statistic</th>
<th>A</th>
<th>B</th>
<th>C</th>
<th>D</th>
<th>E</th>
<th>F</th>
<th>G</th>
</tr>
</thead>
<tbody>
<tr>
<td>All</td>
<td>Calls</td>
<td>20</td>
<td>2100</td>
<td>2100</td>
<td>260</td>
<td>260</td>
<td>1840</td>
<td>1800</td>
</tr>
<tr>
<td>TAKEN</td>
<td></td>
<td>19</td>
<td>2080</td>
<td>1840</td>
<td>0</td>
<td>0</td>
<td>1800</td>
<td>1800</td>
</tr>
<tr>
<td>NOT_TAKEN</td>
<td></td>
<td>1</td>
<td>20</td>
<td>260</td>
<td>260</td>
<td>40</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>FixedDirectionPredictor</td>
<td>Mispredictions</td>
<td>1</td>
<td>20</td>
<td>260</td>
<td>260</td>
<td>40</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td></td>
<td>Mispredicted as TAKEN</td>
<td>1</td>
<td>20</td>
<td>280</td>
<td>260</td>
<td>40</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td></td>
<td>Mispredicted as NOT_TAKEN</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td></td>
<td>Accuracy (%)</td>
<td>95</td>
<td>99</td>
<td>88</td>
<td>0</td>
<td>0</td>
<td>98</td>
<td>100</td>
</tr>
<tr>
<td>BiModalPredictor</td>
<td>Mispredictions</td>
<td>1</td>
<td>20</td>
<td>341</td>
<td>1</td>
<td>1</td>
<td>40</td>
<td>0</td>
</tr>
<tr>
<td></td>
<td>Mispredicted as TAKEN</td>
<td>1</td>
<td>20</td>
<td>220</td>
<td>1</td>
<td>1</td>
<td>40</td>
<td>0</td>
</tr>
<tr>
<td></td>
<td>Mispredicted as NOT_TAKEN</td>
<td>0</td>
<td>0</td>
<td>121</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td></td>
<td>Accuracy (%)</td>
<td>95</td>
<td>99</td>
<td>84</td>
<td>100</td>
<td>100</td>
<td>98</td>
<td>100</td>
</tr>
<tr>
<td>GSharePredictor</td>
<td>Mispredictions</td>
<td>1</td>
<td>20</td>
<td>279</td>
<td>241</td>
<td>241</td>
<td>40</td>
<td>0</td>
</tr>
<tr>
<td></td>
<td>Mispredicted as TAKEN</td>
<td>1</td>
<td>20</td>
<td>260</td>
<td>241</td>
<td>241</td>
<td>40</td>
<td>0</td>
</tr>
<tr>
<td></td>
<td>Mispredicted as NOT_TAKEN</td>
<td>0</td>
<td>0</td>
<td>19</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td></td>
<td>Accuracy (%)</td>
<td>95</td>
<td>99</td>
<td>87</td>
<td>7</td>
<td>7</td>
<td>98</td>
<td>100</td>
</tr>
<tr>
<td>CortexA53Predictor</td>
<td>Mispredictions</td>
<td>1</td>
<td>23</td>
<td>324</td>
<td>2</td>
<td>1</td>
<td>40</td>
<td>0</td>
</tr>
<tr>
<td></td>
<td>Mispredicted as TAKEN</td>
<td>1</td>
<td>20</td>
<td>221</td>
<td>2</td>
<td>1</td>
<td>40</td>
<td>0</td>
</tr>
<tr>
<td></td>
<td>Mispredicted as NOT_TAKEN</td>
<td>0</td>
<td>3</td>
<td>103</td>
<td>0</td>
<td>0</td>
<td>9</td>
<td>0</td>
</tr>
<tr>
<td></td>
<td>Accuracy (%)</td>
<td>95</td>
<td>99</td>
<td>85</td>
<td>99</td>
<td>100</td>
<td>97</td>
<td>100</td>
</tr>
</tbody>
</table>

The accuracy figures have been rounded to the nearest percentage. For each branch instruction type, A to G, the entry for the best accuracy is shown in gray. As expected, different branch prediction algorithms are better suited to different types of branch instructions.

With the FixedDirectionPredictor, all branches are predicted as TAKEN, so the accuracy is equal to the percentage of calls to that branch that were TAKEN.

With the BiModalPredictor and GSharePredictor algorithms, only the random branch C was mispredicted both as TAKEN and NOT_TAKEN. With the other systematic branches, the misprediction was always in one direction. The result is different for the more complex algorithm of the CortexA53Predictor, which has mispredictions in both directions for systematic branches as well.

The BiModalPredictor is able to store the history of individual branches, and is therefore most accurate with predicting branches with a deterministic ratio between the number of times they are TAKEN and NOT_TAKEN. This accuracy can be seen with branches A, B, D, and E. With a more random branch, such
as C, which depends entirely on the contents of a user-defined string, relying on the history of the branch proves ineffective.

Interestingly, the GSharePredictor appears to be highly inaccurate at predicting branches D and E. These branches are NOT_TAKEN a fixed number of times consecutively. However, since there are calls to many other branches between consecutive calls to these branches, the GSharePredictor’s global history is not able to use the specific outcome of these branches to update their prediction values effectively.

Overall, the BiModalPredictor and the CortexA53Predictor have predicted these branch instructions most accurately, as shown in the following table:

<table>
<thead>
<tr>
<th>Predictor type</th>
<th>Overall accuracy (%)</th>
</tr>
</thead>
<tbody>
<tr>
<td>FixedDirectionPredictor</td>
<td>86</td>
</tr>
<tr>
<td>BiModalPredictor</td>
<td>98</td>
</tr>
<tr>
<td>GSharePredictor</td>
<td>86</td>
</tr>
<tr>
<td>CortexA53Predictor</td>
<td>98</td>
</tr>
</tbody>
</table>

**Impact of branch misprediction on simulation time**

You can directly observe the impact of mispredictions on the overall simulation time, as shown in the --stat output after the model exits.

The simulated execution times with the different branch predictors are shown in the following table.

Note: The execution times also include the impact of branch mispredictions that occur in other parts of the code, as well as in the startup and shutdown sequences.

<table>
<thead>
<tr>
<th>Predictor type</th>
<th>Simulation time with mispredict-latency=11</th>
<th>Simulation time with mispredict-latency=0</th>
</tr>
</thead>
<tbody>
<tr>
<td>FixedDirectionPredictor</td>
<td>0.002275s</td>
<td>0.001713s</td>
</tr>
<tr>
<td>BiModalPredictor</td>
<td>0.001805s</td>
<td>0.001713s</td>
</tr>
<tr>
<td>GSharePredictor</td>
<td>0.002289s</td>
<td>0.001713s</td>
</tr>
<tr>
<td>CortexA53Predictor</td>
<td>0.001806s</td>
<td>0.001713s</td>
</tr>
</tbody>
</table>
Appendix A
SystemC Export generated ports

This appendix describes Fast Models SystemC Export generated ports.

It contains the following section:
A.1 About SystemC Export generated ports

The generated SystemC component must have SystemC ports to communicate with the SystemC world. The SystemC Export feature automatically generates these ports from the Fast Models ports of the top-level component.

--- Caution ---
Although it is possible to export your own protocols, Arm strongly recommends using the AMBA-PV protocols provided and bridge from these in SystemC, if needed.

The SystemC export feature automatically generates port wrappers that bind the SystemC domain to the Fast Models virtual platform.

![Figure A-1 Port wrappers connect Fast Models and SystemC components](image)

Each master port in the Fast Models top level component results in a master port on the SystemC side. Each slave port in the Fast Models top level component results in a slave port (export) on the SystemC side.

For Fast Models to instantiate and use the ports, it requires protocol definitions that:
• Correspond to the equivalent SystemC port classes.
• Refer to the name of these SystemC port classes.

This effectively describes the mapping from Fast Models port types (protocols) to SystemC port types (port classes).

Related information