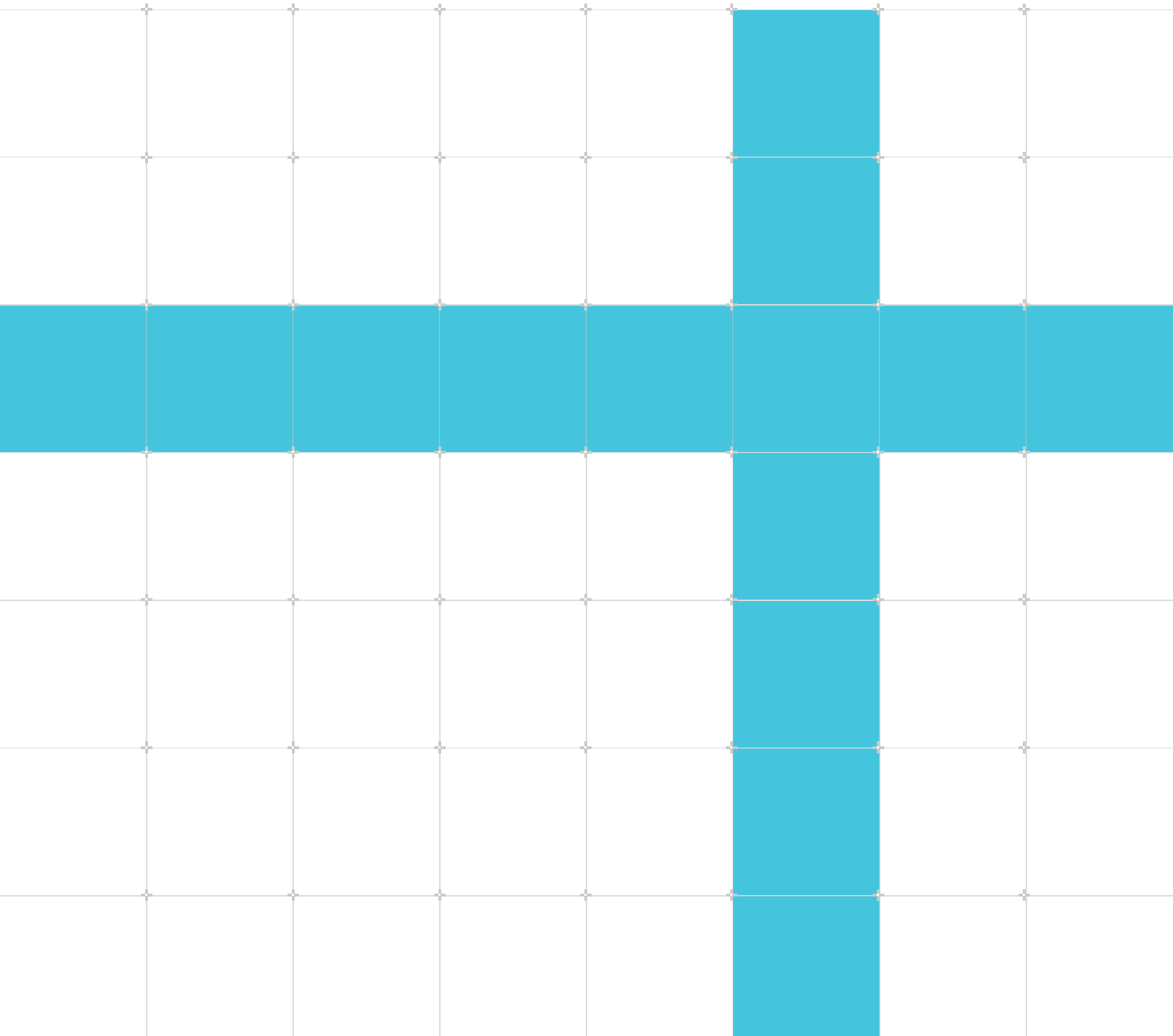
Fast Models

Version 11.17

User Guide

Issue 00
100965_1117_00_en



Fast Models

User Guide

Release Information

Document history

Issue	Date	Confidentiality	Change
A	31 May 2014	Non-Confidential	New document for Fast Models v9.0, from DUI0370R for v8.3.
B	30 November 2014	Non-Confidential	Update for v9.1.
C	28 February 2015	Non-Confidential	Update for v9.2.
D	31 May 2015	Non-Confidential	Update for v9.3.
E	31 August 2015	Non-Confidential	Update for v9.4.
F	30 November 2015	Non-Confidential	Update for v9.5.
G	29 February 2016	Non-Confidential	Update for v9.6.
H	31 May 2016	Non-Confidential	Update for v10.0.
I	31 August 2016	Non-Confidential	Update for v10.1.
J	11 November 2016	Non-Confidential	Update for v10.2.
K	17 February 2017	Non-Confidential	Update for v10.3.
1100-00	31 May 2017	Non-Confidential	Update for v11.0. Document numbering scheme has changed.
1101-00	31 August 2017	Non-Confidential	Update for v11.1.

Issue	Date	Confidentiality	Change
1102-00	17 November 2017	Non-Confidential	Update for v11.2.
1103-00	23 February 2018	Non-Confidential	Update for v11.3.
1104-00	22 June 2018	Non-Confidential	Update for v11.4.
1104-01	17 August 2018	Non-Confidential	Update for v11.4.2.
1105-00	23 November 2018	Non-Confidential	Update for v11.5.
1106-00	26 February 2019	Non-Confidential	Update for v11.6.
1107-00	17 May 2019	Non-Confidential	Update for v11.7.
1108-00	5 September 2019	Non-Confidential	Update for v11.8.
1108-01	3 October 2019	Non-Confidential	Update for v11.8.1.
1109-00	28 November 2019	Non-Confidential	Update for v11.9.
1110-00	12 March 2020	Non-Confidential	Update for v11.10.
1111-00	9 June 2020	Non-Confidential	Update for v11.11.
1112-00	22 September 2020	Non-Confidential	Update for v11.12.
1113-00	9 December 2020	Non-Confidential	Update for v11.13.
1114-00	17 March 2021	Non-Confidential	Update for v11.14.
1115-00	29 June 2021	Non-Confidential	Update for v11.15.
1116-00	6 October 2021	Non-Confidential	Update for v11.16.

Issue	Date	Confidentiality	Change
1117-00	16 February 2022	Non-Confidential	Update for v11.17.

Proprietary Notice

and names mentioned in this document may be the trademarks of their respective owners. Please follow Arm's trademark usage guidelines at https://www.arm.com/company/policies/trademarks.

Arm Limited. Company 02557590 registered in England.

110 Fulbourn Road, Cambridge, England CB1 9NJ.

(LES-PRE-20349)

Confidentiality Status

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

Unrestricted Access is an Arm internal classification.

Product Status

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

Feedback

Arm welcomes feedback on this product and its documentation. To provide feedback on the product, create a ticket on https://support.developer.arm.com

To provide feedback on the document, fill the following survey: https://developer.arm.com/documentation-feedback-survey.

Inclusive language commitment

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

This document includes language that can be offensive. We will replace this language in a future issue of this document.

To report offensive language in this document, email terms@arm.com.

Contents

1 Introduction

1.1 Conventions

The following subsections describe conventions used in Arm documents.

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: developer.arm.com/glossary.

Typographic conventions

Convention	Use
italic	Citations.
bold	Interface elements, such as menu names. Signal names. Terms in descriptive lists, where appropriate.
`monospace`	Text that you can enter at the keyboard, such as commands, file and program names, and source code.
`monospace bold`	Language keywords when used outside example code.
`monospace underline`	A permitted abbreviation for a command or option. You can enter the underlined text instead of the full command or option name.
`<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	Terms that have specific technical meanings as defined in the *Arm® Glossary*. For example, **IMPLEMENTATION DEFINED**, **IMPLEMENTATION SPECIFIC**, **UNKNOWN**, and **UNPREDICTABLE**.
	Recommendations. Not following these recommendations might lead to system failure or damage.
	Requirements for the system. Not following these requirements might result in system failure or damage.
	Requirements for the system. Not following these requirements will result in system failure or damage.
	An important piece of information that needs your attention.

Convention	Use
	A useful tip that might make it easier, better or faster to perform a task.
	A reminder of something important that relates to the information you are reading.

1.2 Other information

See the Arm® website for other relevant information.

- Arm® Developer.
- Arm® Documentation.
- Technical Support.
- Arm® Glossary.

2 Introduction to Fast Models

This chapter provides a general introduction to Fast Models.

2.1 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 Arm®v8-A, Arm®v8-R, and Arm®v8-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 information

System Canvas GUI on page 72
LISA+ Language for Fast Models Reference Guide
About Model Debugger

2.2 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 some free of charge FVPs, separately from the Fast Models package.

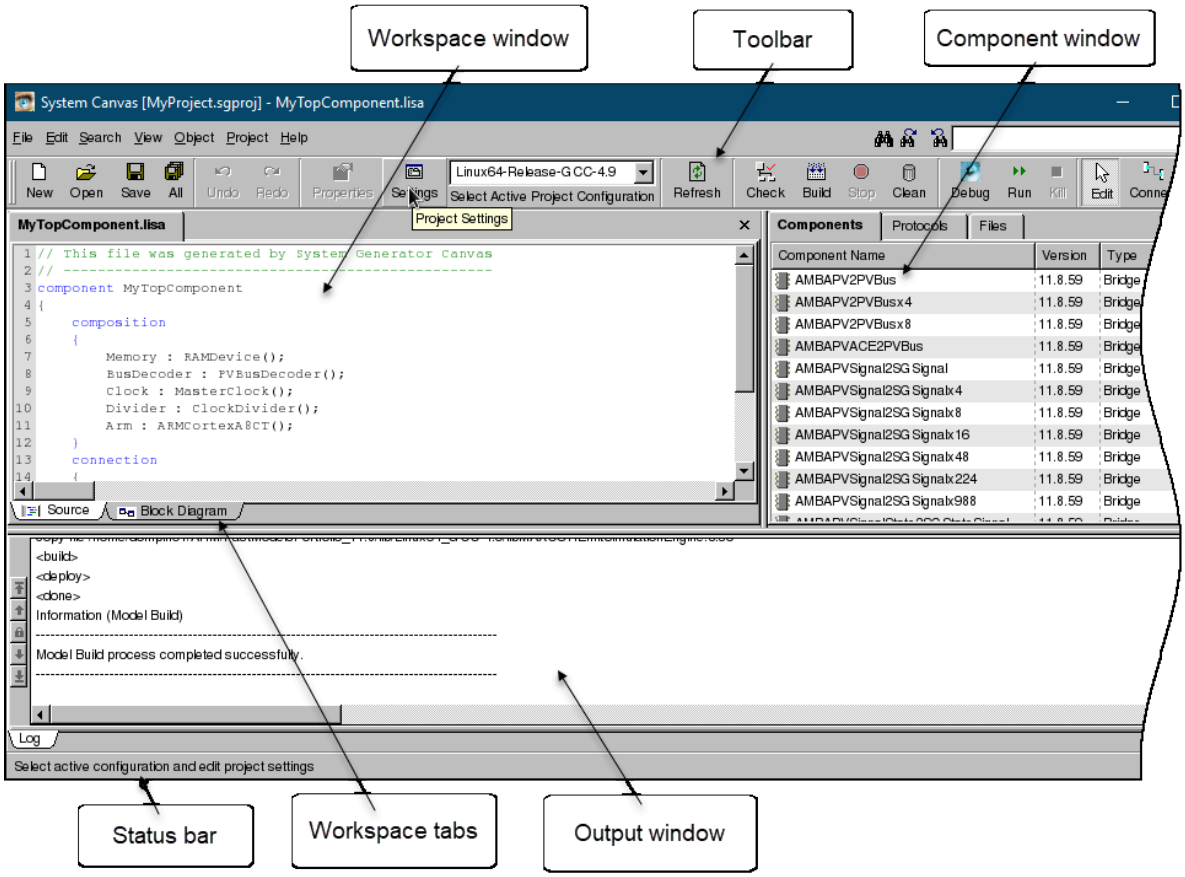
2.2.1 Fast Models tools

Fast Models tools enable you to create custom system models from the library of component models supplied in the Fast Models portfolio.

System Canvas or `sgcanvas`

A GUI design tool for developing new model components written in LISA+. It can also be used for building and launching system models. To launch System Canvas from the command line, type `sgcanvas`. It displays the model as either LISA+ source code, or graphically, in a block diagram editor:

Figure 2-1: System Canvas



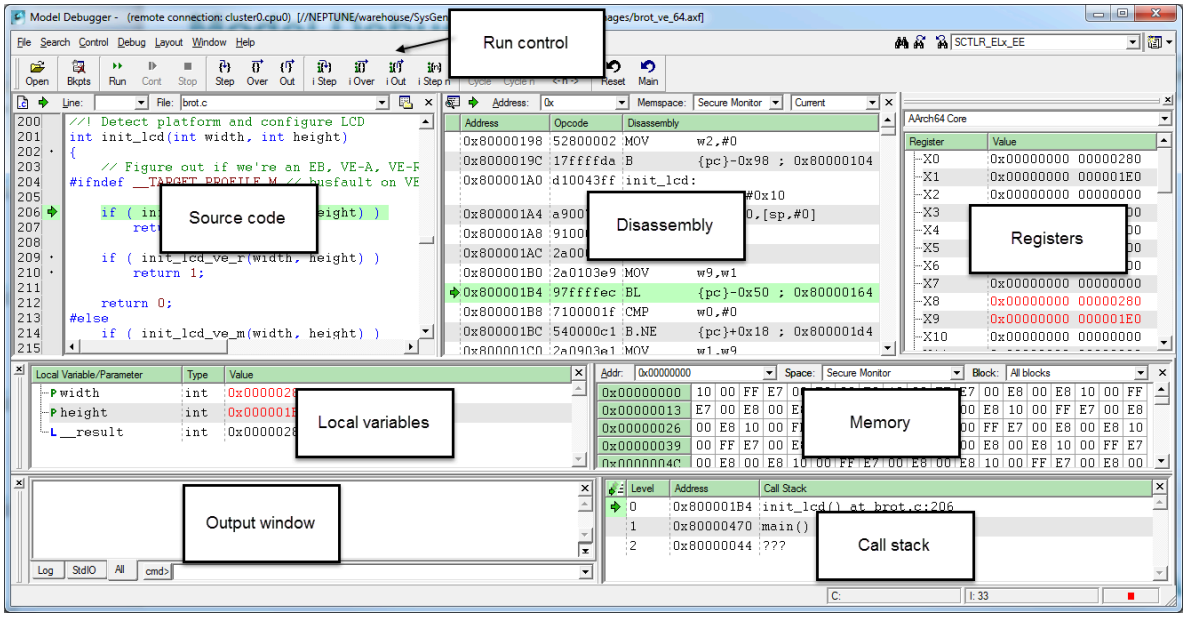
System Generator or `simgen`

A backend tool that handles system model 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 symbolic debugger with a GUI that communicates with models using the CADI interface. It enables you to launch a model or connect to a running model, and debug code running on the model. The following screen capture shows the different views available in the GUI:

Figure 2-2: Model Debugger



Model Shell

A command-line tool for launching simulations that are implemented as CADI libraries. It can also run a CADI debug server to enable CADI-enabled debuggers to connect to the model.

Models can alternatively be implemented as standalone executables called ISIMs, which do not require Model Shell.

Note

Arm deprecates Model Shell in Fast Models version 11.2 and later. We recommend you use ISIMs instead.

2.2.2 Fast Models portfolio

Fast Models portfolio is a library of component models of Arm® IP.

It includes the following:

- A collection of models and protocols, provided as LISA+ components. You can use them to create 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, 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 that show how to integrate the model components. They are 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:
    - Systems based on Arm®v8-A and Arm®v8-R Base Platform.
    - Systems based on Arm® Versatile™ Express development boards for Arm®v7-A and Arm®v7-R processors.
    - Systems based on MPS2 development boards for Arm®v6-M, Arm®v7-M, and Arm®v8-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.

  Some trace plug-ins are provided in source form as programming examples.
- 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.

2.2.3 Other Fast Models products

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

Fixed Virtual Platforms (FVPs)

FVPs are models of Arm® platforms, including processors, memory, and peripherals. They are supplied as pre-built executables for Linux and Windows. Their composition is fixed, although you can configure their behavior using parameters.

Arm provides different types of FVP, based on the following platforms:

- Arm®v8-A Base Platform.

- Arm®v8-R BaseR Platform.
- Arm® Versatile™ Express development boards.
- Arm® MPS2 or Arm® MPS2+ platforms, for Cortex®-M series processors.

FVPs are available for all Cortex®-A, Cortex®-R, and Cortex®-M processors, and they support the CADI, MTI, and Iris interfaces, so can be used for debugging and for trace output.

The most commonly used FVPs are supplied in a single package which is downloadable from Arm Developer, see Fixed Virtual Platforms.

Arm provides validated Linux and Android deliverables for the Arm®v8-A AEM Base Platform FVP and for the Foundation Platform. These are available on the Arm Development Platforms wiki on Arm Community. To get started with Linux on Arm®v8-A FVPs, see FVPs on the Arm Development Platforms wiki.

Foundation Platform

A simple FVP that includes an Arm®v8-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 Arm Ecosystem Models on Arm Developer. Registration and login are required.

System Guidance platforms

These FVPs include documentation to guide SoC design and a reference software stack that is validated on the FVP. They are also known as Reference Design FVPs. For more information, see Reference Design on Arm Developer.

Third party IP

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

2.3 Fast Models glossary

This glossary defines some 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)

An architectural model that aims to expose software bugs by modeling the extremes of behavior that the Arm® architecture allows. There are several AEMs available, including AEMvA for both Arm®v8-A and Arm®v9-A.

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 7.2 Auto-bridging on page 125.

Base Platform

An example platform that is provided as part of Fast Models which can boot Linux and Android. Variations of this platform are available including different cores, and with additional system IP. It is often used together with Linux images from Linaro.

See Base Platform.

Component Architecture Debug Interface (CADI)

A legacy C++ debug interface that enables run control and inspection of models. It has been replaced by Iris.

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 Fast Models component or subsystem that is exported as a SystemC module for use within a SystemC environment.

See 7.1 About SystemC Export with Multiple Instantiation on page 124.

Fast Models

High performance software models of components of Arm® SoCs, for example processors or peripherals. Components can have subcomponents to form a hierarchy, and can 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 supports the Arm®v8-A and Arm®v9-A architectures. It can be downloaded from Arm Ecosystem Models on Arm Developer, registration and login are required. It was formerly known as Foundation Model.

IMP DEF

Used in register descriptions in the *Fast Models Reference Guide* 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. SimGen generates ISIMs by statically linking the model with the SystemC framework.

See 5.10 Building a SystemC ISIM target on page 70.

Iris

An interface for debugging and tracing model behavior. Iris is the replacement for CADI.

See Iris User 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.

See LISA+ Language for Fast Models Reference Guide.

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

A command-line utility for configuring and running CADI-compliant models. Arm deprecates Model Shell from Fast Models version 11.2. Use ISIM executables instead.

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.

See Model Trace Interface Reference Manual.

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 name 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 6.2 System Canvas GUI on page 72.

SystemC Virtual Platform (SVP)

A Fast Models platform that consists of components and subsystems that are individually exported to SystemC as a collection of multiple EVSs.

System Generator

A utility that generates a platform model by processing LISA files. You can run System Generator from the command line by invoking simgen, or from the System Canvas GUI. It is also referred to as SimGen.

See 4.1 System Generator (SimGen) on page 40.

System Model

An alternative term for *Platform Model*.

Tarmac trace

A format for tracing the execution on code on an Arm® core. For Fast Models, there is a TarmacTrace plug-in that can consume and display tarmac trace.

See TarmacTrace.

Timing Annotation

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

See 9 Timing Annotation on page 215.

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 a model of the Versatile™ Express hardware platform, with a single Cortex®-A5 processor.

Related information

Arm Glossary

2.4 Fast Models design

This section describes the design of Fast Models systems.

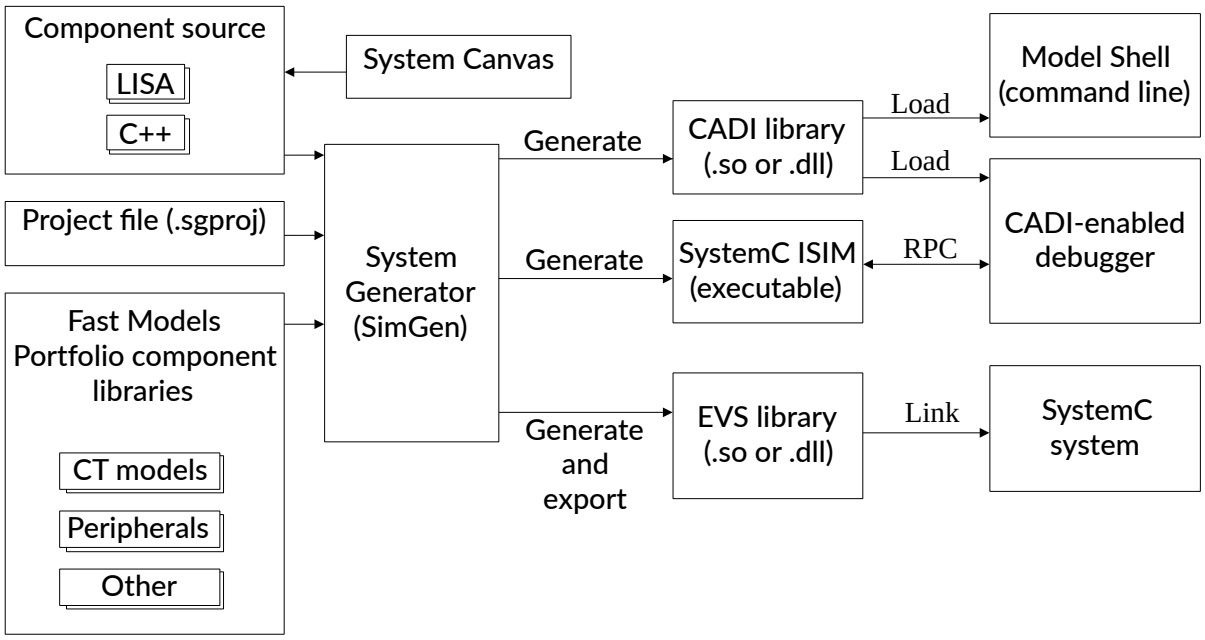
2.4.1 Fast Models design flow

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.

Figure 2-3: Fast Models design flow



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.

2.4.2 Project files

A single project file (`.sgproj`) describes to System Generator (SimGen) the build configuration to use for each host platform and the files that are required to build the model.

The build configuration includes:

- The compiler version to use
- Whether to build release or debug binaries
- Linker and compiler flags
- SimGen flags
- Build target, for example EVS library or ISIM

It also specifies the location of the LISA source files for the project.

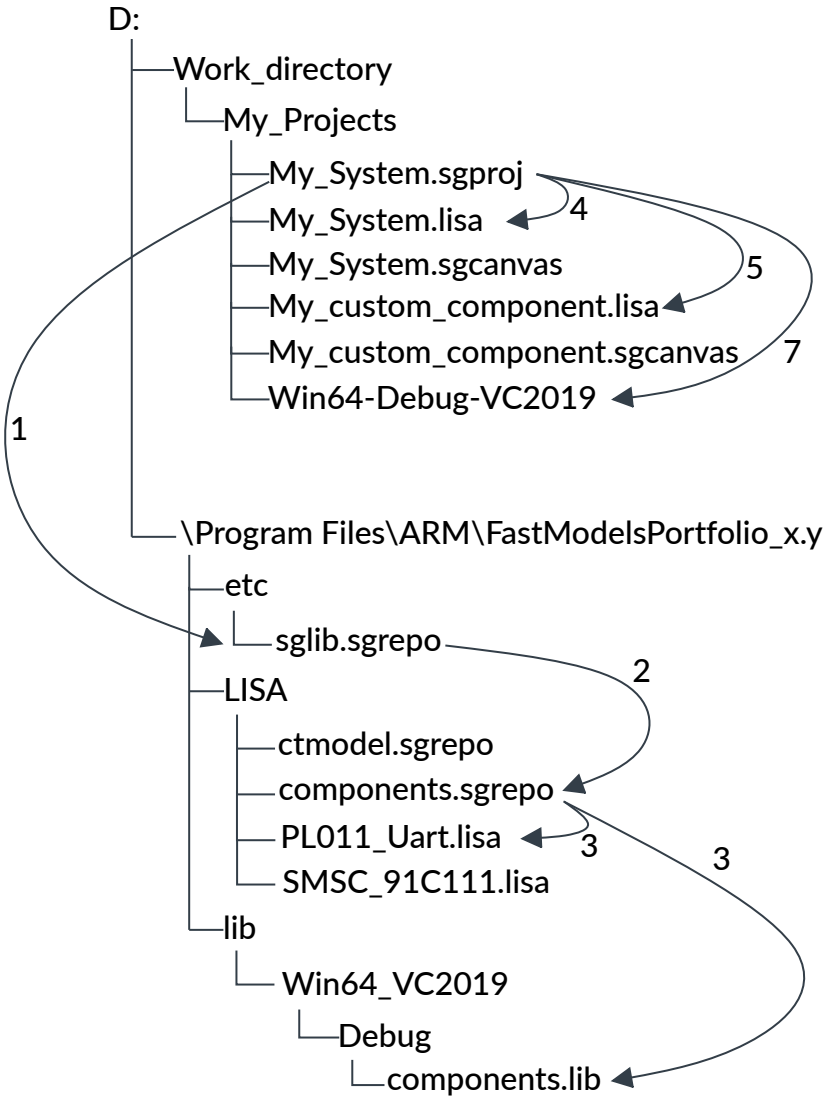
There is no requirement to provide a makefile and a set of configuration files for each new project.

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 (`.sgrepo`) 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 2-4: 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-VC2019` 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 `"VC2019"` and compiler options.
   - Additional linker options.
   - Additional options to be passed to `SimGen`.
   - The type of target to build, for example an ISIM executable or a SystemC component.

Related information

Project Settings dialog on page 109
Project file contents on page 117

2.4.3 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`.

2.4.4 File processing order

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

An example of a project file

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

An example of a repository file

```
/// 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**.

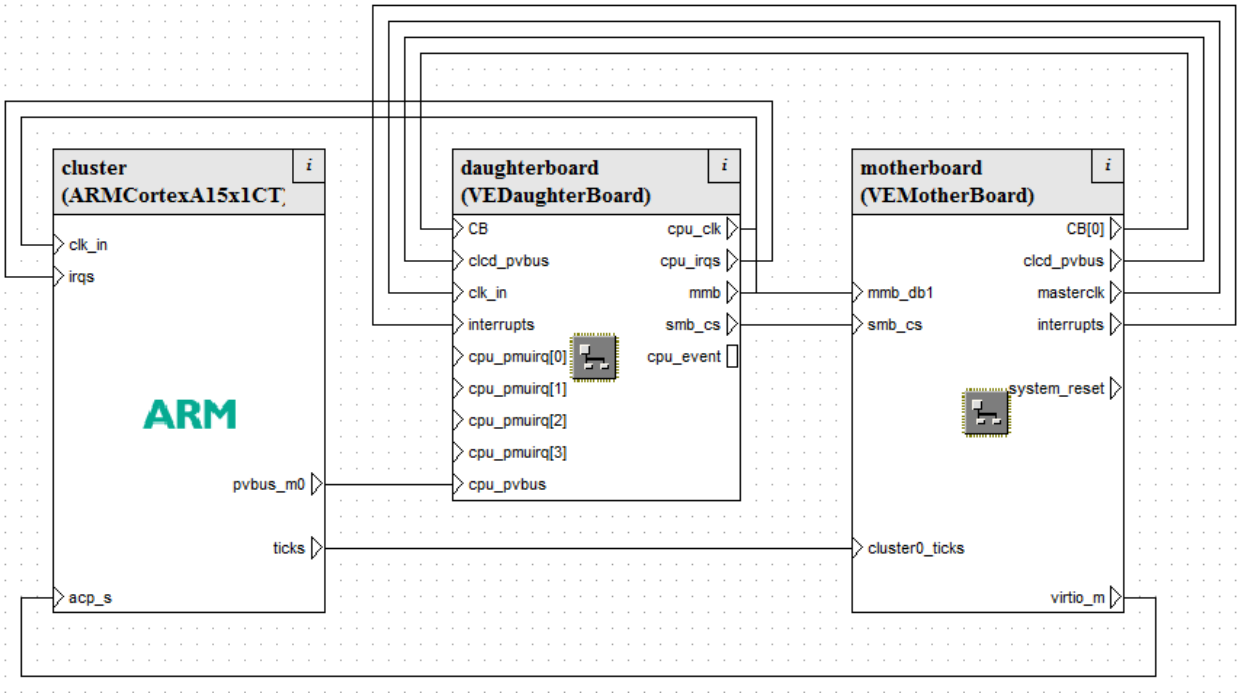
  You can also drag-and-drop files inside a repository or between repositories.
- The **Up** and **Down** buttons on the **Default Model Repository** tab in the Properties dialog, for repositories in new projects.

2.4.5 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.

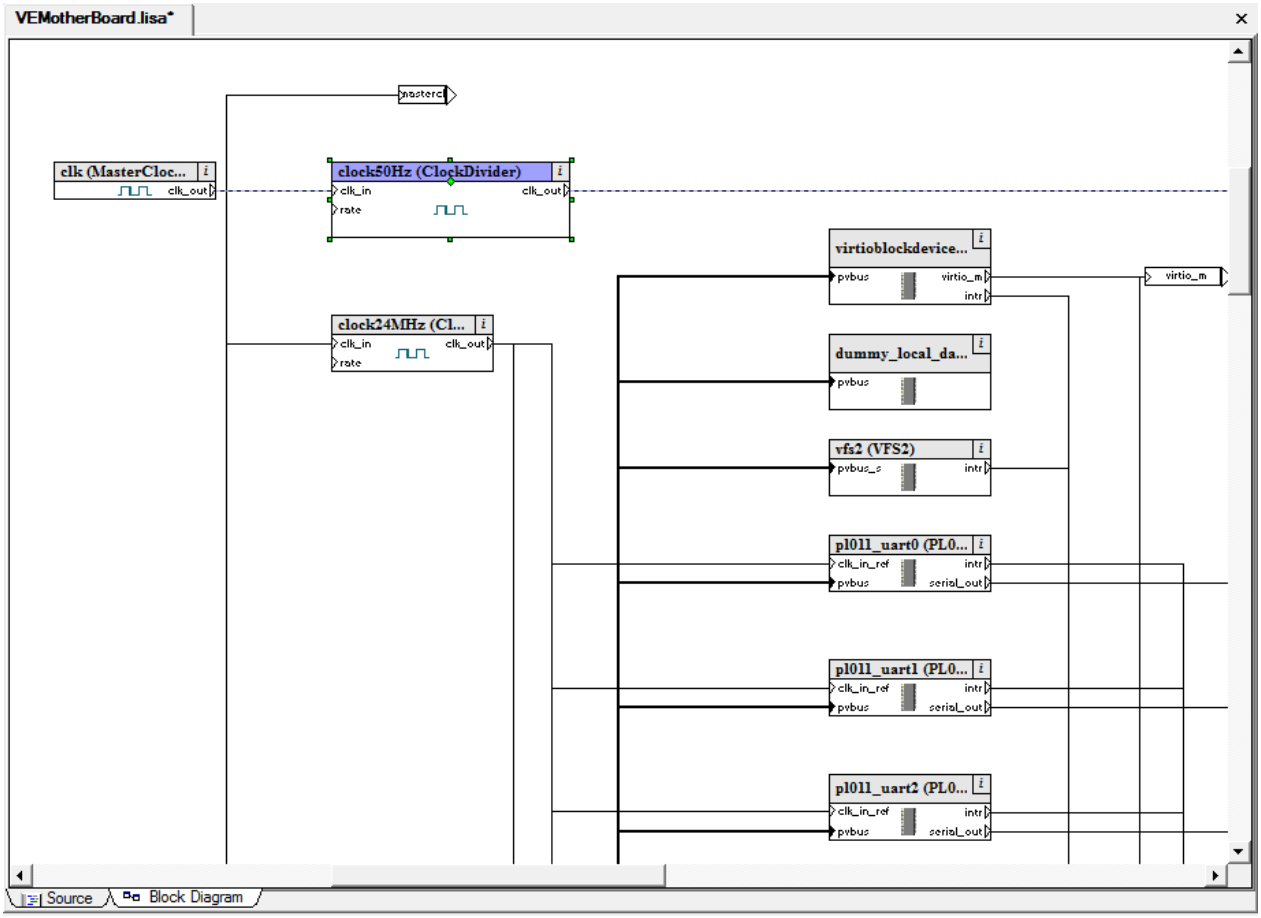
The block diagram shows the advantage of using a hierarchical system with a complex model.

Figure 2-5: Block diagram of top-level VE 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.

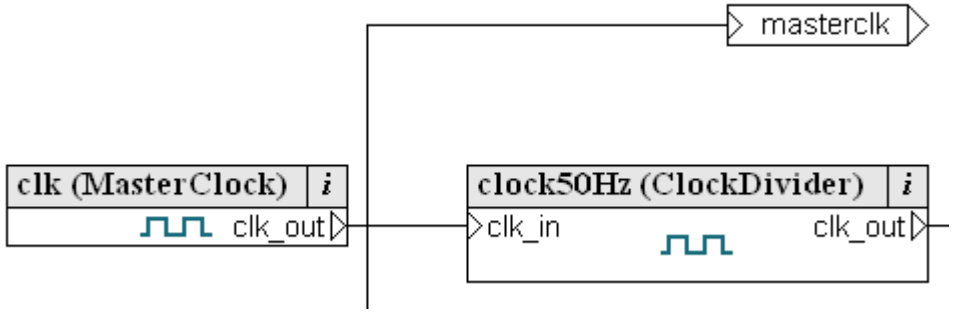
Figure 2-6: Contents of VE motherboard component



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.

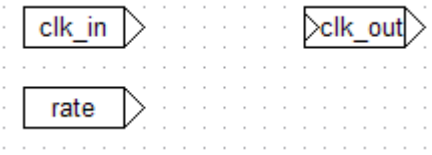
For example, the `ClockDivider` component located at the top-left of Figure 2-6: Contents of VE motherboard component on page 32 has a connection to an external port called `masterclk`.

Figure 2-7: Self port detail



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.

Figure 2-8: Clock divider component external ports



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:

```
self.clk_in_master => clkdiv_ref25.clk_in;
```

3 Installing Fast Models

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

3.1 Requirements for Fast Models

This section describes the host hardware and software requirements for using 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 7 or 8 (for 64-bit architectures), Ubuntu 16.04, 18.04, or 20.04 *Long Term Support* (LTS).

Shell

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

Compiler

GCC 7.3.0, GCC 9.3.0.

Table 3-1: Supported GCC versions on Linux

OS	GCC versions supported
RHEL 7	GCC 7.3.0, GCC 9.3.0
RHEL 8	GCC 9.3.0
Ubuntu 16.04 LTS	GCC 7.3.0
Ubuntu 18.04 LTS	GCC 7.3.0
Ubuntu 20.04 LTS	GCC 9.3.0

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 3-2: GCC and binutils versions

GCC version	GNU binutils version
7.3.0	2.29
9.3.0	2.32

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 License Server Management Software.

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

  Red Hat

  `lsb`, `lsb-linux`

  Ubuntu

  `lsb`

Microsoft Windows

Operating system

Microsoft Windows 10 64-bit.

Compiler

Microsoft Visual Studio 2019 version 16.7.3 or later.

Table 3-3: Supported versions of Visual Studio and Windows SDK

Visual Studio version supported	Windows SDK version required
Visual Studio 2019	Windows SDK version 10.0.16299.0 or later

The following Visual Studio components are required:

- Visual C++ ATL for x86 and x64
- Visual C++ MFC for x86 and x64

PDF Reader

Adobe Reader 8 or higher.

License management utilities

The latest version of the FlexNet software that is available for download from License Server Management Software.

- To build models using Visual Studio requires you to install the Visual Studio redistributable package which contains the runtime libraries for Visual Studio. Fast Models does not provide these libraries. Download the libraries for Visual Studio free of charge from Microsoft, from https://www.microsoft.com/en-gb/download/details.aspx?id=48145.
- On Windows, Fast Models libraries are built with one of the following MSVC compiler options:
  - `/MD` for release builds
  - `/MDd` for debug builds


Note

Any objects or libraries that link against the Fast Models libraries must also be built with the same `/MD` or `/MDd` option.

- Fast Models does not support Express or Community editions of Visual Studio.
- Set up a single `armlmd` 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 Arm Support for more information.

3.2 Installation

This section describes how to install the Fast Models package.

Procedure

1. Unpack the installation package, if necessary, and execute `./setup.sh` on Linux or `Setup.exe` on Windows.
   To install the package without the need for user interaction, use the `--i-accept-the-end-user-license-agreement` command-line option.

   
   Note

   Using this option means you have read and accepted the terms and conditions of the End User License Agreement for the product and version installed.

   This option can be followed by either or both of these options:

   --basepath <*path*>
   Set the base directory for the installation.

   --licpath <*path*>
   Set the location of the license file.

   If the installer finds an existing installation, it displays a dialog to enable re-installation or uninstallation.

   On Windows, the installer automatically defines the following environment variables:

   MAXCORE_HOME
   Points to the installation directory of the Fast Models Tools. It is set by installing the Fast Models Tools package.

   PVLIB_HOME
   Points to the installation directory of the Fast Models Portfolio. It is set by installing the Fast Models Portfolio package.

   SYSTEMC_HOME
   Points to the Accellera SystemC library installation directory. It is set by installing the Accellera SystemC Library package. This package includes the SystemC and TLM header files and libraries that you need to build an EVS, FVP, or SVP.
2. On Linux, source the appropriate script for your shell to set up these environment variables. Ideally, include it for sourcing into the user environment on log-in:

   bash/sh

   ```
   . <install_directory>/FastModelTools_x.x/source_all.sh
   ```

   csh

   ```
   source <install_directory>/FastModelTools_x.x/source_all.csh
   ```
3. Optionally, download and install the Third-Party IP (TPIP) add-on package from Product Download Hub. It contains third party add-ons for Fast Models, including ELF images that you

can run on the example platforms for evaluation purposes and the GDB Remote Connection plug-in.

Results


Note

On Microsoft Windows, 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.

3.3 Uninstallation

On Linux, uninstall Fast Models Tools and Fast Models Portfolio by deleting the installation directories.

On Windows, uninstall Fast Models Tools and Fast Models Portfolio by selecting the **Uninstall** option for each product from the **Start** > **Settings** > **Apps** > **Apps & features** list.

3.4 Dependencies for Red Hat Enterprise Linux

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
```

Table 3-4: Dependencies for Red Hat Enterprise Linux

Package	Required for
`glibc`	Fast Models tools and virtual platforms
`glibc-devel`	Fast Models tools
`libgcc`	Fast Models tools and virtual platforms
`make`	Fast Models tools
`libstdc++`	Fast Models tools and virtual platforms
`libstdc++-devel`	Fast Models tools
`libXext`	Fast Models tools and virtual platforms
`libX11`	Fast Models tools and virtual platforms
`libXau`	Fast Models tools and virtual platforms
`libxcb`	Fast Models tools and virtual platforms
`libSM`	Fast Models tools and virtual platforms
`libICE`	Fast Models tools and virtual platforms
`libuuid`	Fast Models tools and virtual platforms
`libXcursor`	Fast Models tools and virtual platforms
`libXfixes`	Fast Models tools and virtual platforms
`libXrender`	Fast Models tools and virtual platforms
`libXft`	Fast Models tools and virtual platforms
`libXrandr`	Fast Models tools and virtual platforms
`libXt`	Fast Models tools and virtual platforms
`alsa-lib`	Fast Models virtual platforms
`xterm`	Fast Models virtual platforms
`telnet`	Fast Models virtual platforms

4 Building Fast Models

This chapter explains the process for using Fast Models to build different types of platform models.

It assumes a Linux host and GCC 7.3, although Windows hosts and other versions of GCC are also supported, see 3.1 Requirements for Fast Models on page 34 for details.

It refers to the following platform examples that are installed with Fast Models:

- EVS platforms and SVPs, located in `$PVLIB_HOME/examples/SystemCExport/EVS_Platforms/` and `$PVLIB_HOME/examples/SystemCExport/SVP_Platforms/`
- ISIMs, located in `$PVLIB_HOME/examples/LISA/FVP_*/`

4.1 System Generator (SimGen)

All types of platform model are built using a utility called System Generator, also referred to as SimGen, with a Fast Models `.sgproj` project file.

You can use SimGen in the following ways:

- By building the project in System Canvas, which invokes SimGen.
- By invoking simgen on the command line.

SimGen requires you to have installed GCC or Visual Studio C++ compiler. On Windows, if SimGen cannot find `devenv.exe` for Visual Studio, the build fails. You can specify the path to `devenv.exe` in the System Canvas **Preferences** dialog or using the `--devenv-path` command-line option.

To use SimGen to build a Fast Models project, a typical command line is:

```
simgen -p <projectfile>.sgproj --configuration <configuration_name> -b
```

where `configuration_name` identifies the host OS, the build mode (debug or release), and the toolchain, for example, `Linux64-Release-GCC-7.3` or `Win64-Debug-VC2019`.

To see all the available SimGen options, type `simgen --help`.

SimGen supports the following build targets, which you specify either in the `.sgproj` project file or in the System Canvas **Project Settings** dialog:

EVS (Exported Virtual Subsystem) library

A LISA+ component or subsystem that SimGen exports as a SystemC module. You can then integrate this module into a SystemC simulation.

ISIM (Integrated SIMulator)

An executable platform model that SimGen creates by building an EVS library and statically linking it with the SystemC framework.

Related information

System Canvas Tutorial on page 52
SimGen command-line options on page 41

4.2 SimGen command-line options

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

Table 4-1: SimGen command-line options

Option	Short form	Description
`--bridge-conf-file` *FILENAME*	-	Set auto-bridging JSON configuration file *FILENAME*.
`--build`	`-b`	Build the targets.
`--build-directory` *DIR*	-	Set build directory *DIR*.
`--clean`	`-C`	Clean the targets.
`--config` *FILENAME*	-	Set SimGen configuration file *FILENAME*. By default, `simgen.conf`.
`--configuration` *NAME*	-	The name of the configuration, for example `Linux64-Release-GCC-7.3`.
`--cpp-flags-start`	-	Ignore all parameters between this and `--cpp-flags-end`, except `-D` and `-I`.
`--cpp-flags-end`	-	See `--cpp-flags-start`.
`--cxx-flags-start`	-	Ignore all parameters between this and `--cxx-flags-end`, except `-D`.
`--cxx-flags-end`	-	See `--cxx-flags-start`.
`--debug`	`-d`	Enable debug mode.
`--define` *SYMBOL*	`-D`	Define preprocessor *SYMBOL*. You can also use *SYMBOL=DEF*.
`--devenv-path` *ARG*	-	Path to Visual Studio development environment, `devenv`.
`--disable-warning` *NUM*	-	Disable warning number *NUM*. This overrides the `--warning-level` *LEVEL* option.
`--dumb-term`	-	The terminal in which SimGen is running is dumb, so instead of fancy progress indicators, use simpler ones.
`--enable-warning` *NUM*	-	Enable warning number *NUM*. This overrides the `--warning-level` *LEVEL* option.
`--gcc-path` *PATH*	-	Under Linux, the GCC C++ compiler that builds the model. Passes the full path of the chosen g++ executable to SimGen. Match this GCC version to the GCC version in the model configuration. By default, SimGen uses the g++ in the search path.
`--gen-sysgen-lib`	-	Generate system library.
`--help`	`-h`	Print help message with a list of command-line options then exit.
`--ignore-compiler-version`	-	Do not stop on a compiler version mismatch. Try to build anyway.

Option	Short form	Description
`--include` *`INC_PATH`*	`-I`	Add include path *`INC_PATH`*.
`--indir_tpl` *`DIR`*	-	Set directory *`DIR`* where SimGen finds its template data files.
`--link-against` *`LIBS`*	-	Final executable will be linked against debug or release libraries. *`LIBS`* can be debug or release, does certain consistency checks.
`--MSVC-debuginfo-type` *`ARG`*	-	Set the debug info type for MSVC projects. *`ARG`* can be one of: • `none`: No debug info. • `/Zi`: Program Database. • `/Zd`: Line numbers only.
`--no-deploy`	-	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.
`--no-lineinfo`	`-c`	Do not generate line number redirection in generated source and header files.
`--num-build-cpus` *`NUM`*	-	The number of processors used during the build.
`--num-comps-file` *`NUM`*	-	The number of components generated into one file.
`--outdir_arch` *`DIR`*	-	Set output directory *`DIR`* for file with variable filenames.
`--outdir_fixed` *`DIR`*	-	Set output directory *`DIR`* for file with constant filenames.
`--override-config-parameter` *`PARAM=VALUE`*	`-P`	Override the configuration parameter from the `*.sgproj` file.
`--print-config`	-	Print out configuration parameters in file `.ConfigurationParameters.txt`.
`--print-preprocessor-output`	`-E`	Print preprocessor output, then exit.
`--print-resource-mapping`	-	Print flat resource mapping when generating a simulator.
`--project-file` *`FILENAME`*	`-p`	Set SimGen project file *`FILENAME`*.
`--replace-strings`	-	Replace strings in files, then exit. Ignore binary files. Usage: `simgen --replace-strings FOO BAR [FOO2 BAR2]… -- FILES…`
`--replace-strings-bin`	-	Replace strings in files, then exit. Do not ignore binary files. Usage: `simgen --replace-strings-bin FOO BAR [FOO2 BAR2]… -- FILES…`
`--top-component` *`COMP`*	-	Top level component (system).
`--user-MSVC-libs-start`	-	Set additional libraries for MSVC projects. The list is terminated by `--user-MSVC-libs-end`.
`--user-MSVC-libs-end`	-	See `--user-MSVC-libs-start`.
`--user-sourcefiles-start`	-	Add source files listed between this option and `--user-sourcefiles-end` to the executable.
`--user-sourcefiles-end`	-	See `--user-sourcefiles-start`.
`--verbose` *`ARG`*	`-v`	Verbosity. *`ARG`* can be: `on`, `sparse` (default), `off`.
`--version`	`-V`	Print the version and exit.

Option	Short form	Description
`--warning-level` *`LEVEL`*	`-w`	Warning level *`LEVEL`*.
`--warnings-as-errors`	-	Treat LISA parsing and compiler warnings as errors.

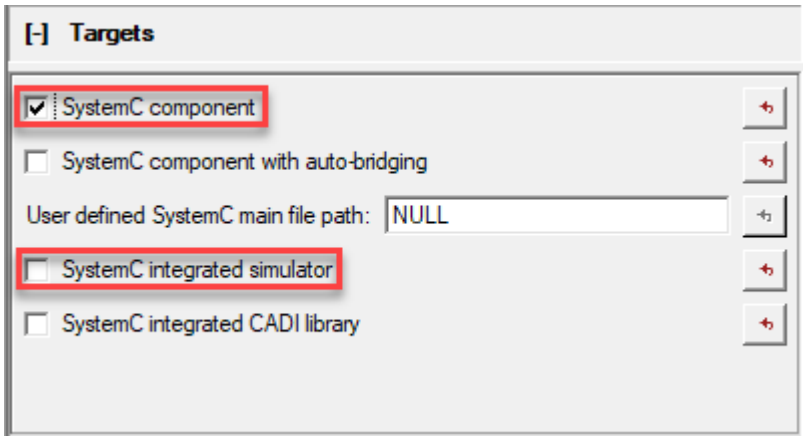
4.3 Select the build target

The first step in building an EVS library or an ISIM is to select the build target.

You can do this either:

- In the System Canvas **Project Settings** dialog, by selecting:
  - **SystemC component** for an EVS library.
  - **SystemC integrated simulator** for an ISIM.

  Figure 4-1: System Canvas Project Settings



- If you are building using `simgen` on the command line, the build target is specified in the project (`.sgproj`) file using one of the following statements. If no build target is specified, `simgen` returns an error.
  - `TARGET_SYSTEMC = "1";` for an EVS library.
  - `TARGET_SYSTEMC_ISIM = "1";` for an ISIM.

  You can find some example sgproj files under `$PVLIB_HOME/examples/`.


Note

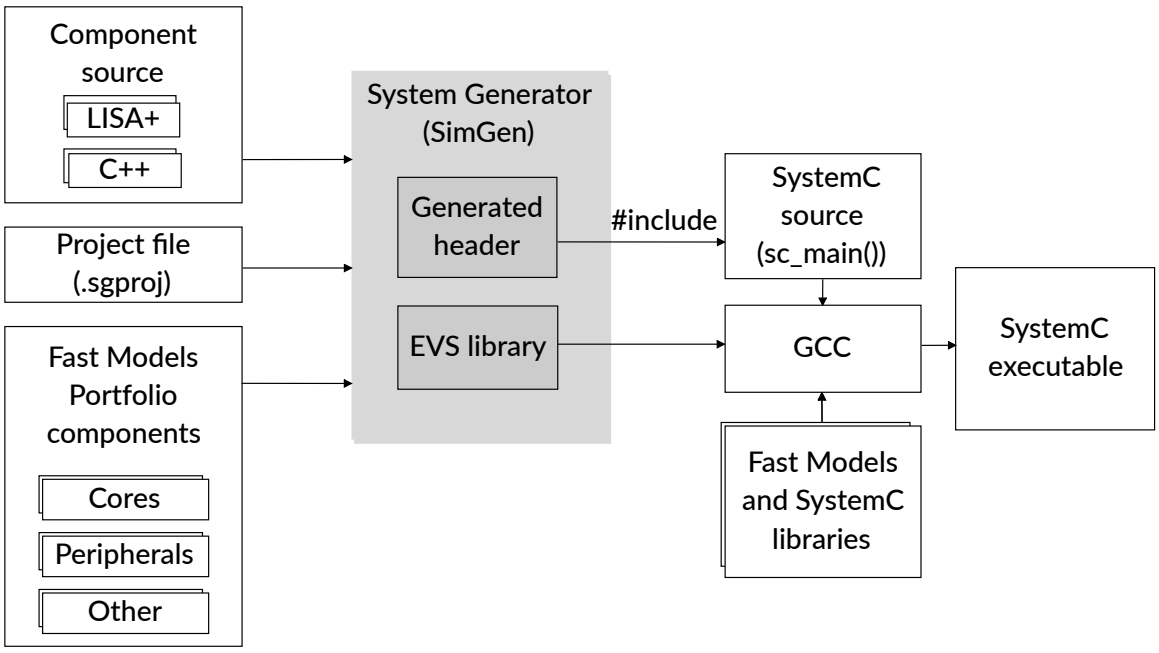
To build an ISIM or EVS, you must have installed SystemC 2.3.3 and set the `SYSTEMC_HOME` environment variable to the location of the SystemC installation. When you install Fast Models, you have the option of also installing Accellera SystemC. On Windows, the installer automatically sets `SYSTEMC_HOME`. On Linux, you need to run the appropriate setup script.

4.4 Building an EVS platform

An EVS platform is a SystemC simulation that includes an EVS library. Because you must provide an `sc_main()` function and integrate the EVS library into the simulation, you cannot build this type of platform entirely within System Canvas.

The following diagram shows the build process for an EVS platform. The shaded area represents SimGen and its output. The rest of the diagram is the responsibility of the user:

Figure 4-2: Build process for an EVS platform



The steps to build an EVS platform are:

1. Export the Fast Model as an EVS library. You can either use System Canvas or invoke SimGen directly to do this.
2. Define an `sc_main()` function that uses the SystemC Export API to initialize and configure the EVS. The file that defines `sc_main()` must `#include` the header file that SimGen generates in step 1, `scx_evs_<top_level_component>.h`.


Note

The top-level component is specified in the sgproj file entry `TOP_LEVEL_COMPONENT`.

3. Invoke the C++ compiler, specifying the required Fast Models and SystemC header files and libraries.

The EVS platform examples, located in `$PVLIB_HOME/examples/SystemCExport/EVS_Platforms/` use a Makefile to perform steps 1 and 3.

Related information

SystemC Export API on page 131

4.5 Steps for building an EVS platform

This section describes in more detail each step to build the EVS platform. It uses the Cortex-A65x1 EVS_Dhrystone platform as an example. This example platform is located in `$PVLIB_HOME/examples/SystemCExport/EVS_Platforms/EVS_Dhrystone/Build_Cortex-A65x1/`.

4.5.1 Export the Fast Model as an EVS library

To build an EVS library, invoke SimGen from the command line.

For example:

```
$MAXCORE_HOME/bin/simgen -p EVS_Dhrystone_Cortex-A65x1.sgproj --configuration Linux64-Release-GCC-7.3 -b
```

Where:

- `MAXCORE_HOME` is set by the Linux setup script or Windows installer to the installation directory of the Fast Models tools.
- The `-p` option specifies the project (`.sgproj`) file. To select EVS as the build target, the project file must contain the following statement:
  ```
  TARGET_SYSTEMC = "1";
  ```
  A build target must be specified because there is no default.
- The `--configuration` option selects the build configuration. This is also used as the name of the build directory and in some output filenames.
- `-b` performs the build.

This command generates:

- The EVS library, for example `./Linux64-Release-GCC-7.3/libscx-Dhrystone-Linux64-Release-GCC-7.3.a`
- The EVS header file, for example `./Linux64-Release-GCC-7.3/gen/scx_evs_Dhrystone.h`
- The EVS shared object, for example `./Linux64-Release-GCC-7.3/libDhrystone-Linux64-Release-GCC-7.3.so` which is required when launching the platform. See 4.7 Libraries required to run the platform on page 49.

4.5.2 Initialize and configure the simulation

For an example source file that demonstrates these steps, see `$PVLIB_HOME/examples/SystemCExport/EVS_Platforms/EVS_Dhrystone/Source/main.cpp`.

Procedure

1. Include the EVS header file that was generated by SimGen:
   ```
   #include <scx_evs_Dhrystone.h>
   ```
2. In `sc_main()`, initialize the simulation, giving it a name:
   ```
   scx::scx_initialize("Dhrystone");
   ```
3. Instantiate the generated SystemC component:
   ```
   scx_evs_Dhrystone dhrystone("Dhrystone");
   ```
4. Configure the simulation using command-line arguments set by the user, for example to load an application or to set parameters:
   ```
   scx::scx_parse_and_configure(argc, argv, help_quantum);
   ```

   See 7.4.26 scx::scx_parse_and_configure on page 140 for the supported arguments.

   EVS_Dhrystone loads the application using this function, but you could use `scx::scx_load_application()` instead.
5. Optionally, set parameters for Fast Models components within the simulation:
   ```
   scx::scx_set_parameter("*.Core.cpu0.semihosting-enable", true);
   ```

   where `semihosting-enable` is the parameter and `*.Core.cpu0` is the instance name (* means all EVSs in the platform).

   The Dhrystone example uses semihosting to read user input (the number of runs through the benchmark) and to print statistics to the console at the end of the simulation.
6. Bind the ports of the generated SystemC component to the ports of the other components in the SystemC simulation. This step is described in 4.6 Bridge between LISA+ and SystemC on page 49.
7. Start the simulation:
   ```
   sc_core::sc_start();
   ```

Related information

SystemC Export API on page 131
Bridge between LISA+ and SystemC on page 49

4.5.3 Required header files and libraries

Some Fast Models and SystemC header files and libraries are required when building an EVS platform.

$PVLIB_HOME/include/fmruntime/
Standard Fast Models utility code.

$PVLIB_HOME/include/fmruntime/eslapi/
CADI-related header files.

$PVLIB_HOME/Iris/include/
Iris debug and trace header files.

$SYSTEMC_HOME/include/
SystemC header files.

$MAXCORE_HOME/AMBA-PV/include/
AMBA-PV header files. Only required if AMBA-PV protocols are used.

./$CONFIG/gen/
Contains the generated EVS header file, `scx_evs_<top_level_component>.h`, which defines the SystemC wrapper class.

The following static libraries are required by all EVS platforms:

./Linux64-Release-GCC-7.3/libscx-<*top_level_component*>-Linux64-Release-GCC-7.3.a
Generated EVS platform library.

./Linux64-Release-GCC-7.3/libscx.a
Generated library containing the default implementations of the SystemC report handler, simulation controller, and scheduler.

$PVLIB_HOME/lib/Linux64_GCC-7.3/libcomponents.a
Fast Models components library.

$PVLIB_HOME/lib/Linux64_GCC-7.3/libpvbus.a
PVBus components library.

$PVLIB_HOME/lib/Linux64_GCC-7.3/libarmctmodel.a
Core and cluster models library.

$PVLIB_HOME/lib/Linux64_GCC-7.3/libfmruntime.a
Standard Fast Models utility code.

$PVLIB_HOME/Iris/Linux64_GCC-7.3/libIrisSupport.a
Iris support library.

$SYSTEMC_HOME/lib/Linux64_GCC-7.3/libsystemc.a
SystemC library.

Related information

IrisSupportLib Reference Guide

4.5.4 Building an EVS on Windows

When building an EVS on Windows, there are a few extra considerations to be aware of.

- On Windows, SimGen generates a solution containing several project files. For example, the following command outputs `Dhrystone.sln`, where `Dhrystone` is the name of the top-level component, and three project files, which must be built in the order in which they are listed below:

```
"%MAXCORE_HOME%"\bin\simgen -p EVS_Dhrystone_Cortex-A65x1.sgproj --configuration Win64-Release-VC2019 … -b
```


Note

When invoking SimGen on Windows, the `--devenv-path` option might be required to specify the path to `devenv`.

1. `scx.vcxproj`. This project builds the static library `scx.lib` containing default implementations of the SystemC report handler, simulation controller, and scheduler.
2. `scx_Dhrystone_Win64-Release-VC2019.vcxproj`, where `Win64-Release-VC2019` is the chosen configuration. This project builds the static EVS library `scx-Dhrystone-Win64-Release-VC2019.lib`.
3. `Dhrystone_sc_sg_wrapper_Win64-Release-VC2019.vcxproj`. This project builds the dynamic library `Dhrystone-Win64-Release-VC2019.dll`, which is required when you launch the platform.
4. For an ISIM, an extra project is created, called `scx_isim_<`*`top-component`*`>_<`*`config`*`>.vcxproj`. This file is the SystemC project that creates the executable, `isim_system_<`*`config`*`>.exe`.

- On Windows, any SystemC code that instantiates an exported Fast Model must include `$PVLIB_HOME\include\fmruntime\sg\IncludeMeFirst.h` before any other include files. If not, the compiler throws an error when compiling the SystemC code.

  This file, which is used to check the underlying Windows API version, is automatically included in EVSs generated by SimGen, but SystemC models that use Fast Models libraries can be built without using SimGen.
- On Windows, Fast Models libraries are built with one of the following MSVC compiler options:
  - `/MD` for release builds.
  - `/MDd` for debug builds.

  Any objects or libraries that link against the Fast Models libraries must also be built with the same `/MD` or `/MDd` option.
- The following additional libraries are needed when building the executable: `user32.lib`, `ws2_32.lib`, `imagehlp.lib`, `advapi32.lib`, `shlwapi.lib`, `Iphlpapi.lib`, and `zlib.lib`.

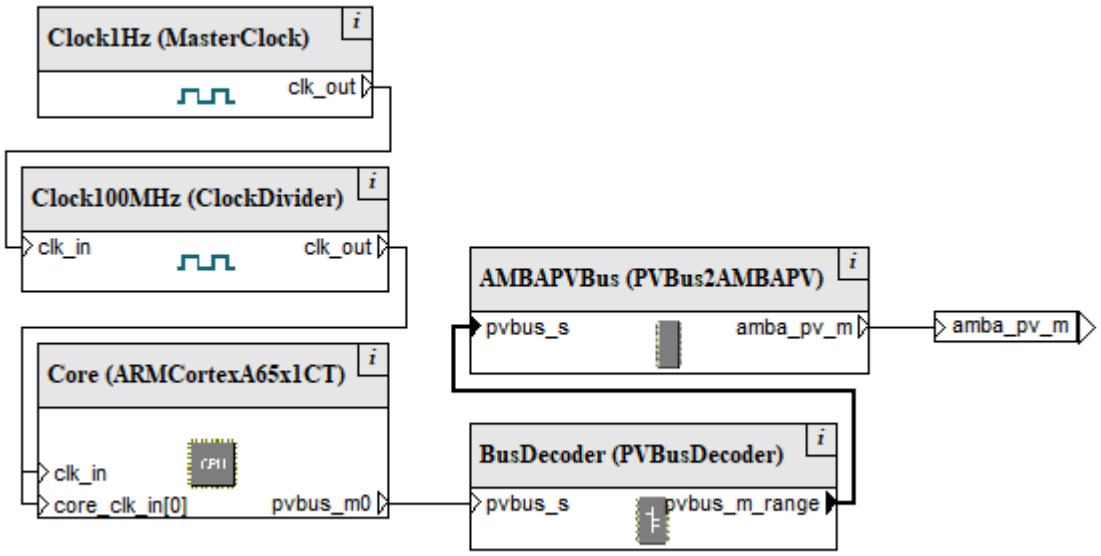
- On Windows, use the `/vmg` compiler option to correctly compile source code for use with SystemC.

4.6 Bridge between LISA+ and SystemC

The EVS_Dhrystone examples consist of an Arm core and some simple peripherals that are written in LISA+, some memory that is defined in SystemC, and a bridge between the exported LISA+ subsystem and the SystemC code.

The following block diagram from System Canvas shows the Fast Models LISA+ components that are defined in the top-level `Dhrystone` component and the connections between them. The `amba_pv_m` port at the right-hand side will be used to connect the `Dhrystone` component to the memory, which is defined in SystemC:

Figure 4-3: System Canvas block diagram for EVS_Dhrystone



The `PVBus2AMBAPV` component is a bridge that converts signals from the `PVBus` protocol to the `AMBA-PV` protocol. After exporting the `Dhrystone` component as an EVS, the `amba_pv_m` port can be connected to a SystemC component, in this example, to the slave port of the memory model, in `main.cpp`, as follows:

```
amba_pv::amba_pv_memory<64> memory("Memory", 0x34000100);
scx_evs_Dhrystone dhrystone("Dhrystone");
…
dhrystone.amba_pv_m(memory.amba_pv_s);
```

4.7 Libraries required to run the platform

When you run the executable platform model, some of the following libraries must be present in the same directory.

You can copy all except the first one from `$PVLIB_HOME/lib/Linux64_GCC-7.3/`:

lib<*top_level_component*>-Linux64-Release-GCC-7.3.so
: Required for an EVS. This shared library is created by SimGen in the build directory.

libMAXCOREInitSimulationEngine.3.so
: Required for initializing the platform.

libarmctmodel.so
: Required if your platform contains any core or cluster models.

libSDL2-2.0.so.0.4.0
: Required if your platform uses the PL041 AACI component or any visualization components.

arm_singleton_registry.so
: Singleton registry library that enables multiple simultaneous simulations on the same host platform. It should be located either in the same directory as the executable, or the `FASTSIM_SINGLETON_REGISTRY` environment variable should be set to the full path of the library. If the library is not found, a warning is reported. In this case, a single simulation will still run, but multiple simultaneous simulations might lead to a crash.

4.8 Building an SVP

An SVP (SystemC Virtual Platform) is a platform model that consists of LISA+ components or subsystems that are individually exported to SystemC as multiple EVSs, using the Multiple Instantiation (MI) feature.

The build process for an SVP is the same as for an EVS platform, except you must build and link multiple EVS libraries.

SVPs can provide more flexibility than EVS platforms because components in an SVP can be replaced without the need to modify any LISA+ code.

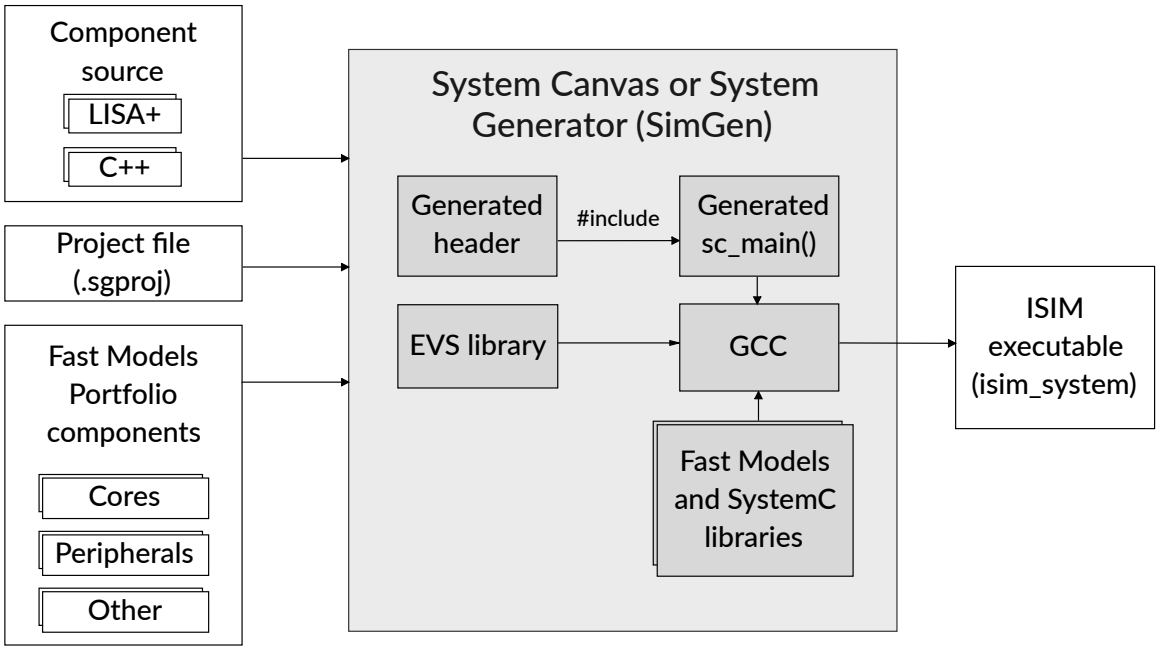
For more information, see the SVP examples under `$PVLIB_HOME/examples/SystemCExport/SVP_Platforms/`.

4.9 Building an ISIM

Building an ISIM is more straightforward than an EVS and can be done either entirely within System Canvas or by calling SimGen from the command line.

The following diagram shows the process. The shaded area represents the work that SimGen does for you:

Figure 4-4: Build process for an ISIM



SimGen takes as input the LISA+ or C++ source code for the platform and its components, and a `.sgproj` project file that contains the statement `TARGET_SYSTEMC_ISIM = "1";`.

It generates:

- An EVS library
- A header file, `./Linux64-Release-GCC-7.3/gen/scx_evs_<top_level_component>.h`, which defines the SystemC wrapper class
- A SystemC source file, `./Linux64-Release-GCC-7.3/gen/scx_main_system.cpp` which defines a default `sc_main()` function. This function is the entry point for the simulation. It initializes the simulation, constructs the SystemC wrapper, parses the command-line options, and starts the simulation.

SimGen then links the EVS library with the required Fast Models and SystemC libraries, and outputs the ISIM executable called `isim_system`.

5 System Canvas Tutorial

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

5.1 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.

5.2 Starting System Canvas

This section describes how to start the application.

About this task

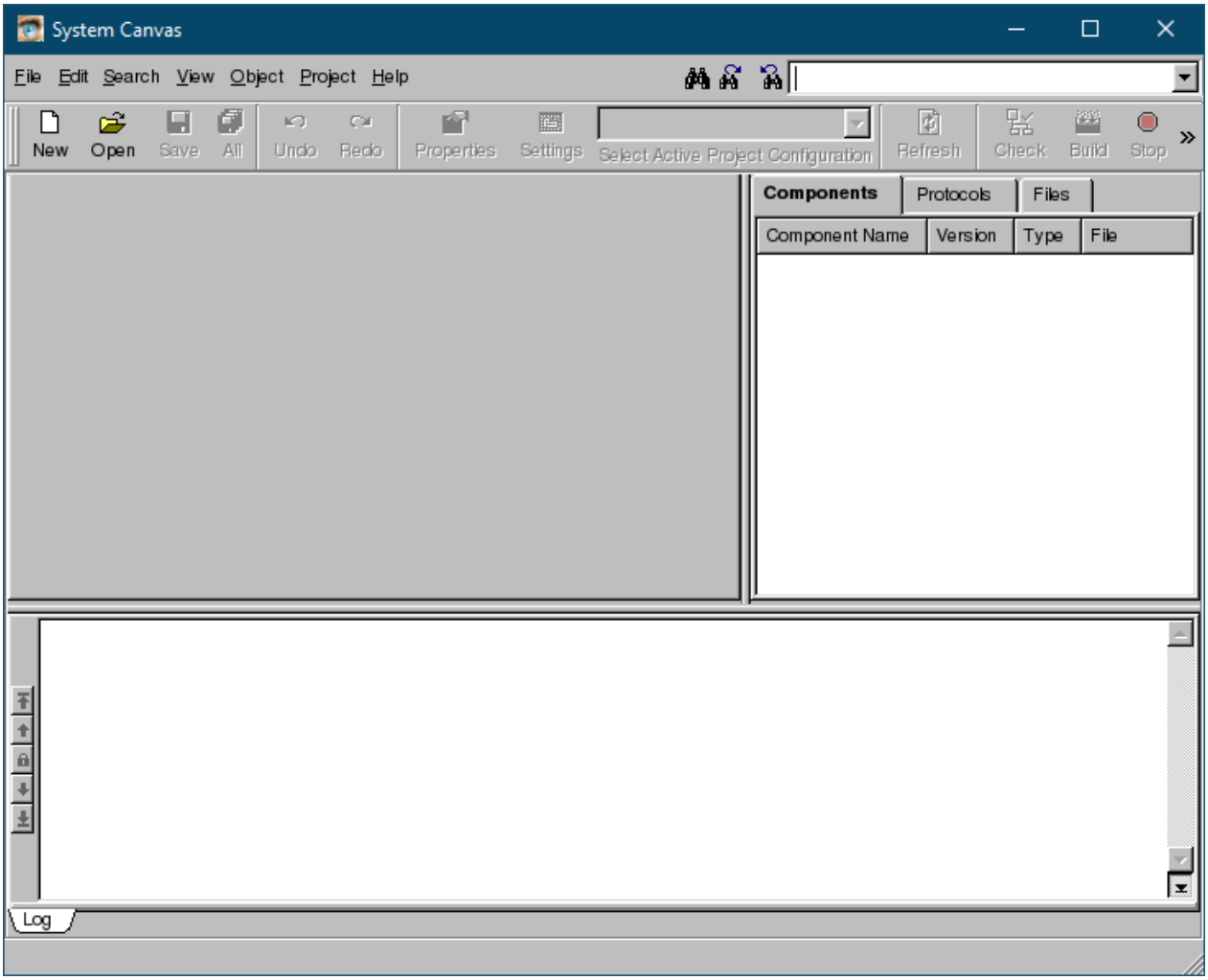
To start System Canvas:

- 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.

Figure 5-1: System Canvas at startup



Related information

Preferences - Applications group on page 105
System Canvas Reference on page 72

5.3 Creating a new project

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

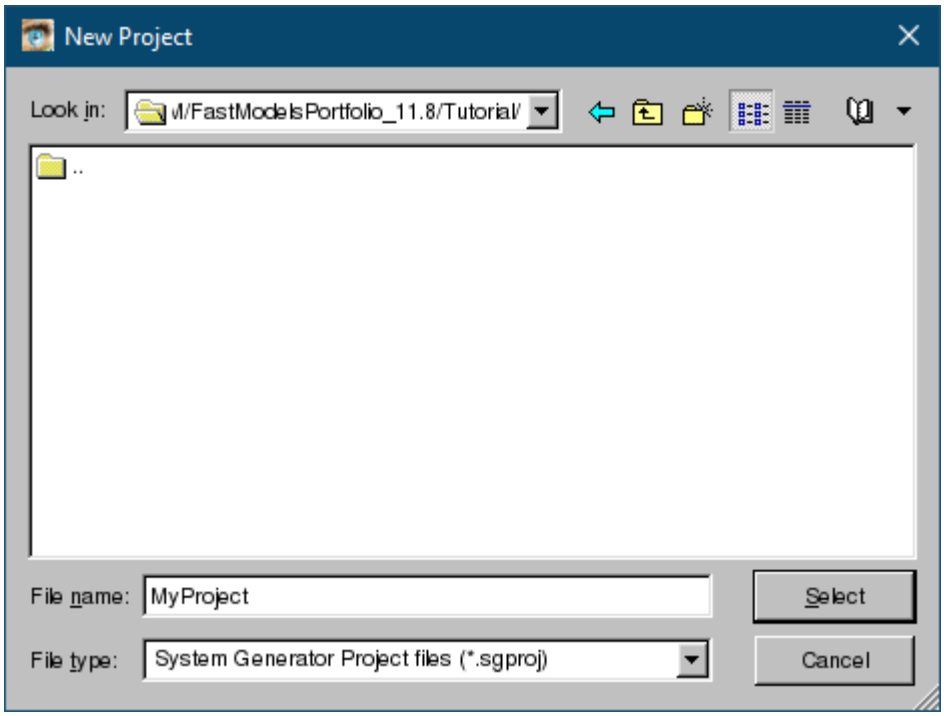
Before you begin

Make sure you have write permission for the directory in which you will create the project.

Procedure

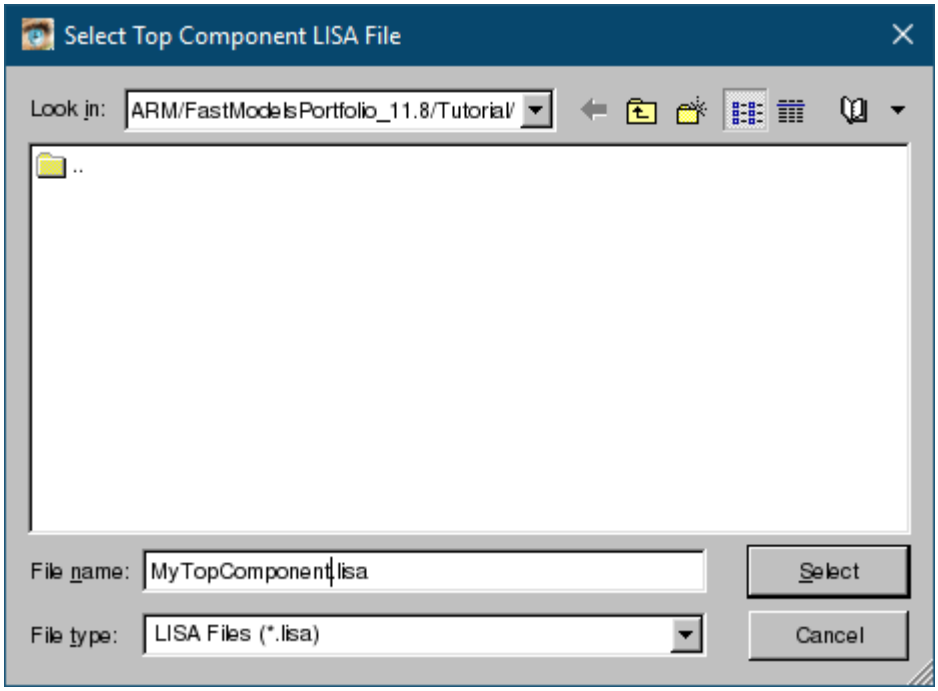
1. Select **New Project** from the **File** menu. Alternatively, click the **New** button on the toolbar. The New Project dialog appears.

Figure 5-2: New Project dialog



2. Navigate to the directory to use for your project. Enter `MyProject` in the filename box and click the **Select** button.
   A dialog appears for you to enter the name and location of the LISA+ file that represents your new system.

   Figure 5-3: Select Top Component LISA File dialog



3. Enter `MyTopComponent.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.

   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.

Results

These steps create a project file, `MyProject.sgproj` and a LISA+ source file, `MyTopComponent.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 `MyTopComponent.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**.

5.4 Add and configure components

This section describes how to add and configure the components required for the example system.

5.4.1 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.
   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.

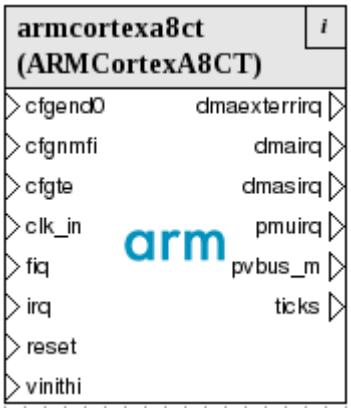
Note

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.
   The system receives the component.

Figure 5-4: ARMCortexA8CT processor component in the Block Diagram window



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.

Results

These steps create a System Canvas file, `MyTopComponent.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.

5.4.2 Naming components

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

About this task

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`.

5.4.3 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.

5.4.4 Hiding ports

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

About this task

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.

Results

Figure 5-5: Processor component after changes

Related information

Using port arrays on page 59

5.4.5 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.

5.4.6 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`.

5.4.7 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.
   - a. Display the **Component Instance Properties** dialog.
     b. Select the **Ports** tab.
     c. Click the + next to the port array to expand the port tree view.
     d. Select the **Show as Expanded** radio button.

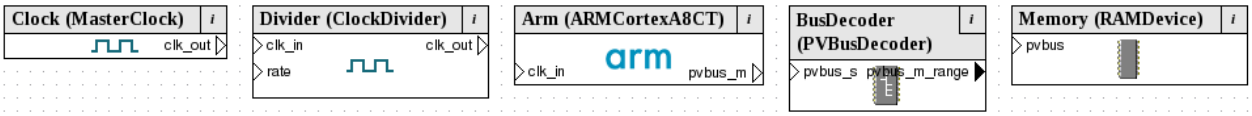
   Only the `axi_if_in[0]` and `axi_if_in[1]` ports are shown.

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 5-6: Example system with added components



5.5 Connecting components

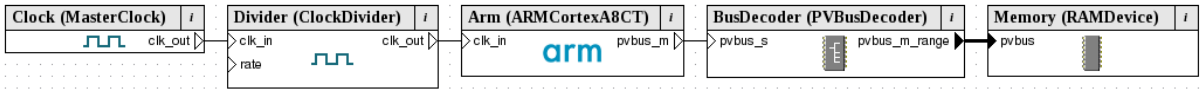
This section describes how to connect components.

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:

Option	Description
Not over an object	The pointer changes to the invalid pointer, a circle with a diagonal line through it.
Over an object	The pointer changes to the start connection pointer and the closest valid port is highlighted.

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.

   Figure 5-7: Connected components



   Connections between the addressable bus ports have bold lines.

5.6 View project properties and settings

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

5.6.1 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.

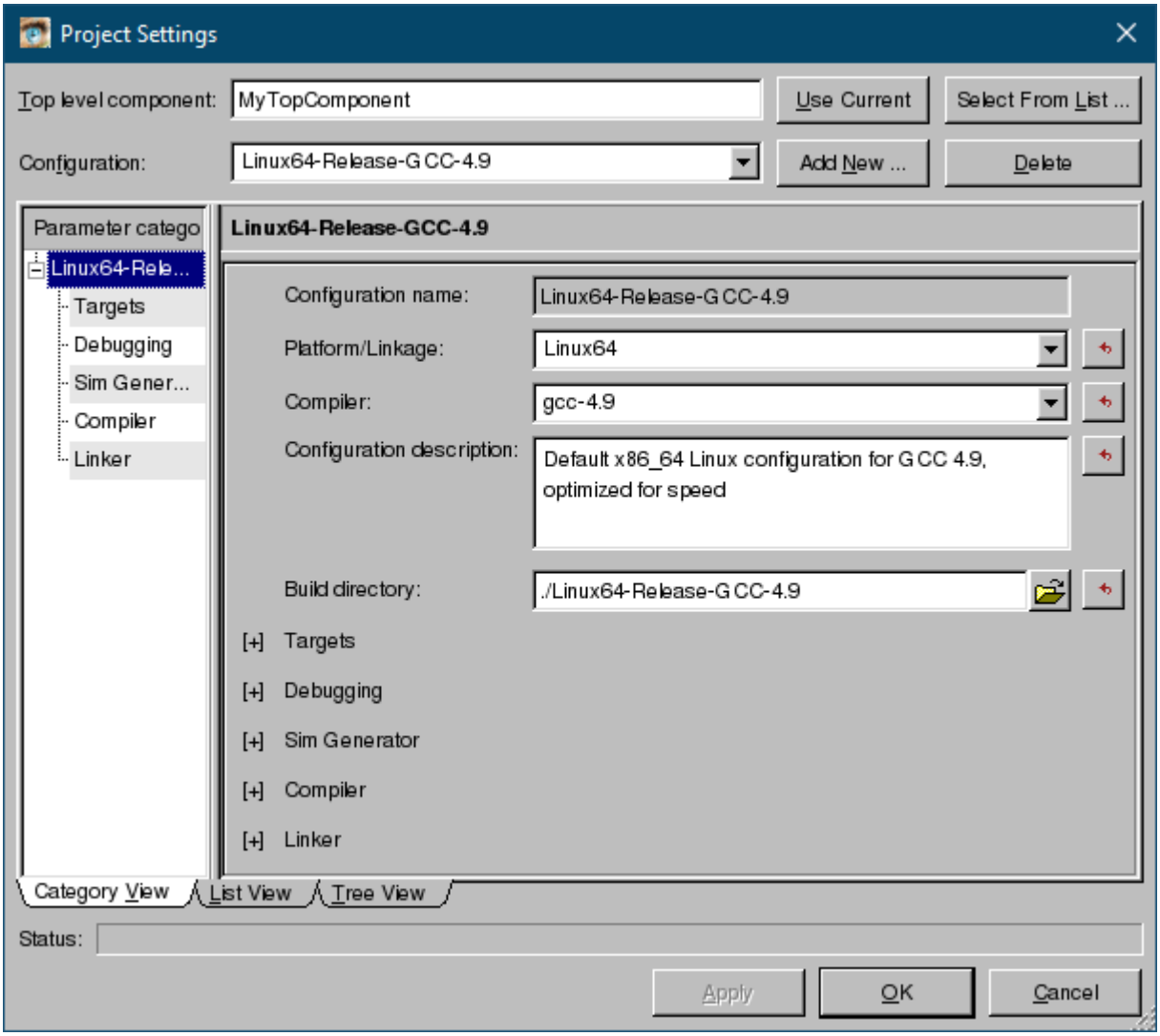
Procedure

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.

The Project Settings dialog appears:

Figure 5-8: Project settings for the example



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

5.6.2 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.

About this task

You can choose to build:

- Models with debug support.
- 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.

5.6.3 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.

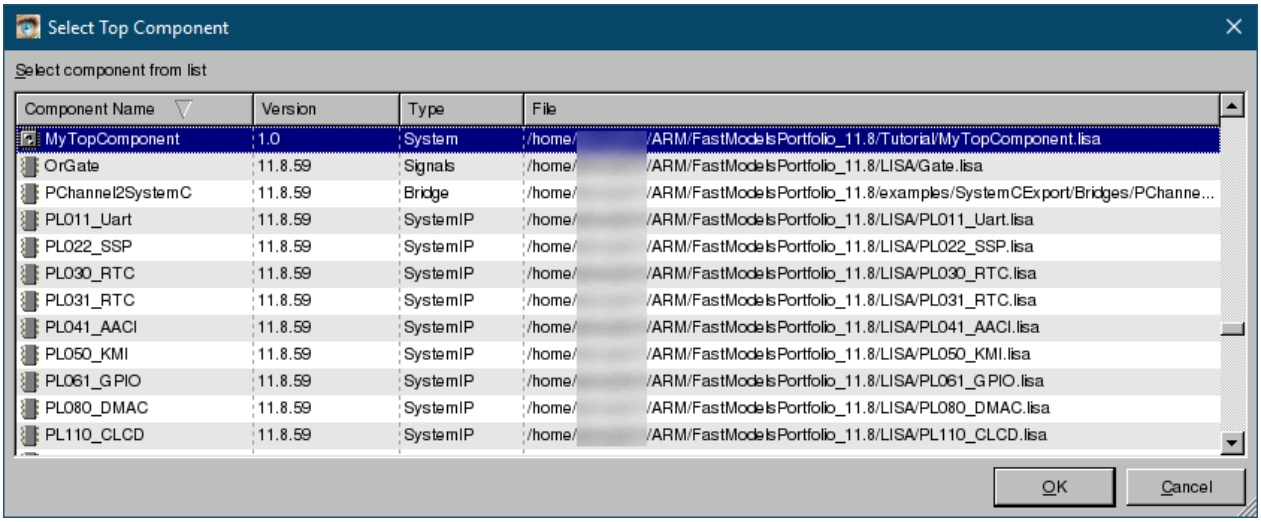
About this task

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 5-9: Select Top Component dialog showing available components



5.7 Changing the address mapping

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

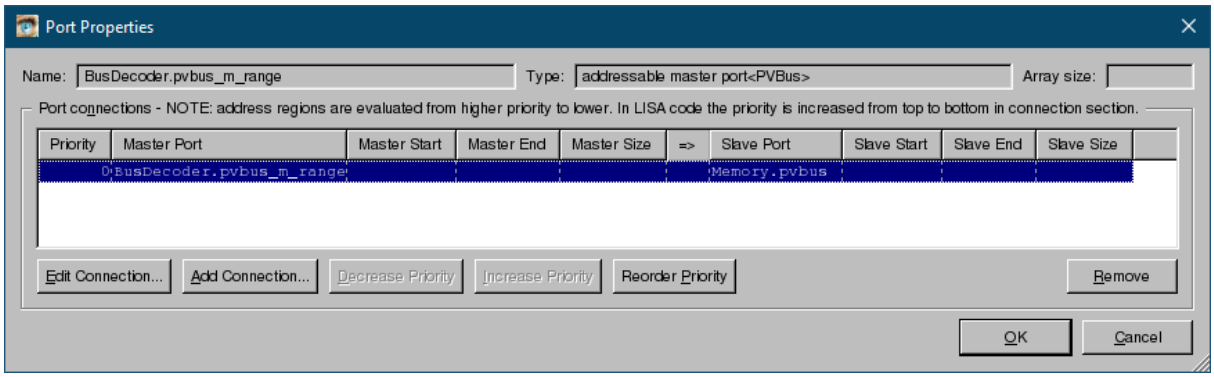
About this task

Follow this procedure to change the address mapping.

Procedure

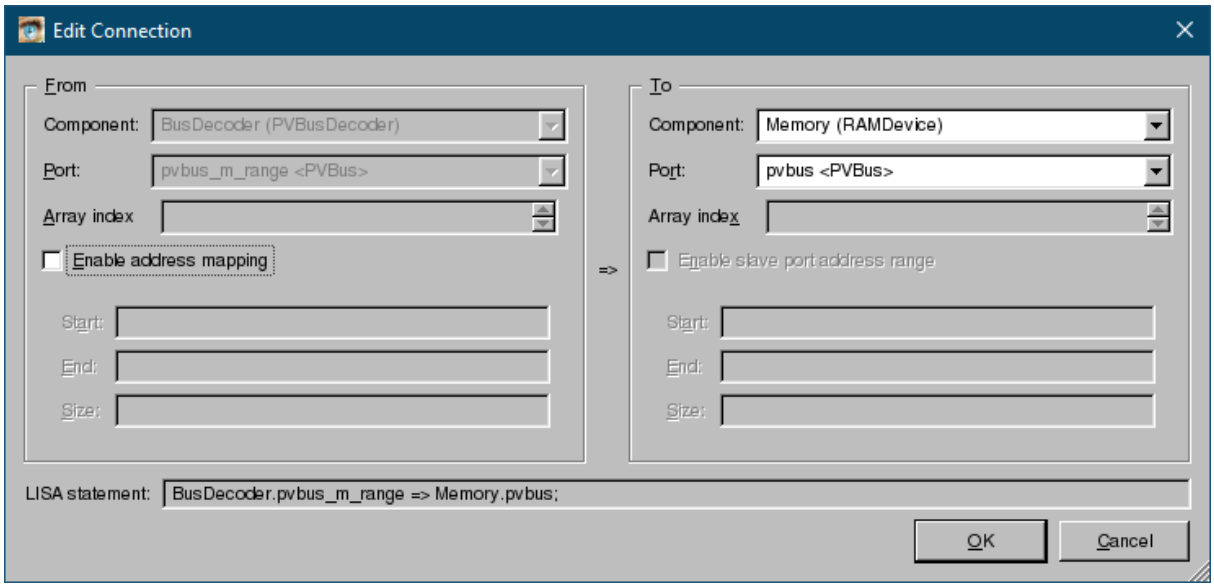
1. Double-click the `pvbus_m_range` port of the BusDecoder component to open the Port Properties dialog.

   Figure 5-10: Viewing the address mapping from the Port Properties dialog



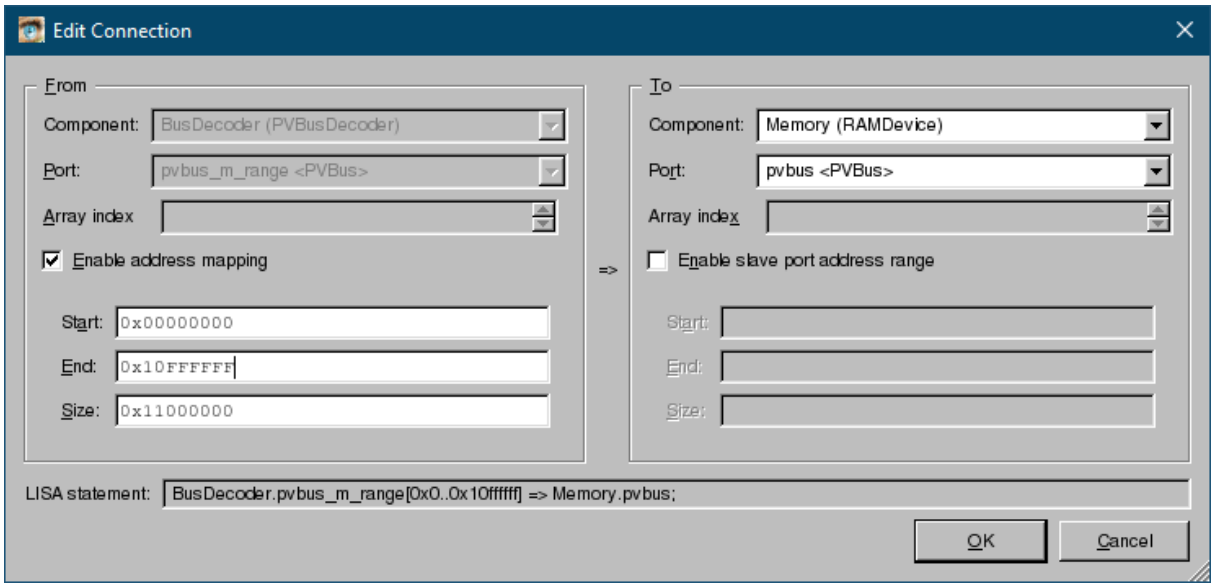
2. Open the Edit Connection dialog by doing either of the following:
   - Select the `Memory.pvbus` Slave Port line, and click **Edit Connection...**.
   - Double click on the entry.

   Figure 5-11: 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.

Figure 5-12: Edit address map for master port



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.

5.8 Building the system

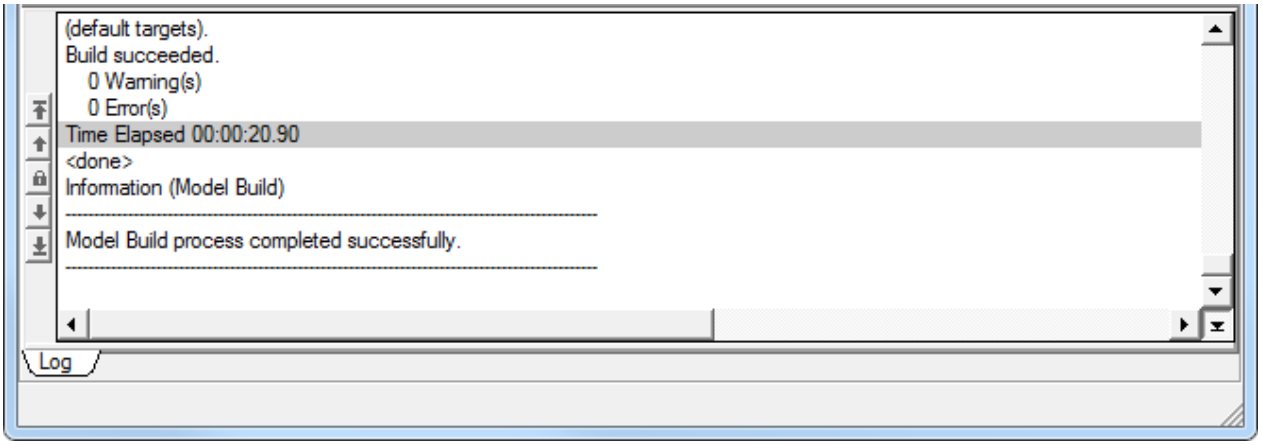
This section describes how to build the model as an `.so` or `.dll` library.

Procedure

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.

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

Figure 5-13: Build process output



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 information

Building a SystemC ISIM target on page 70

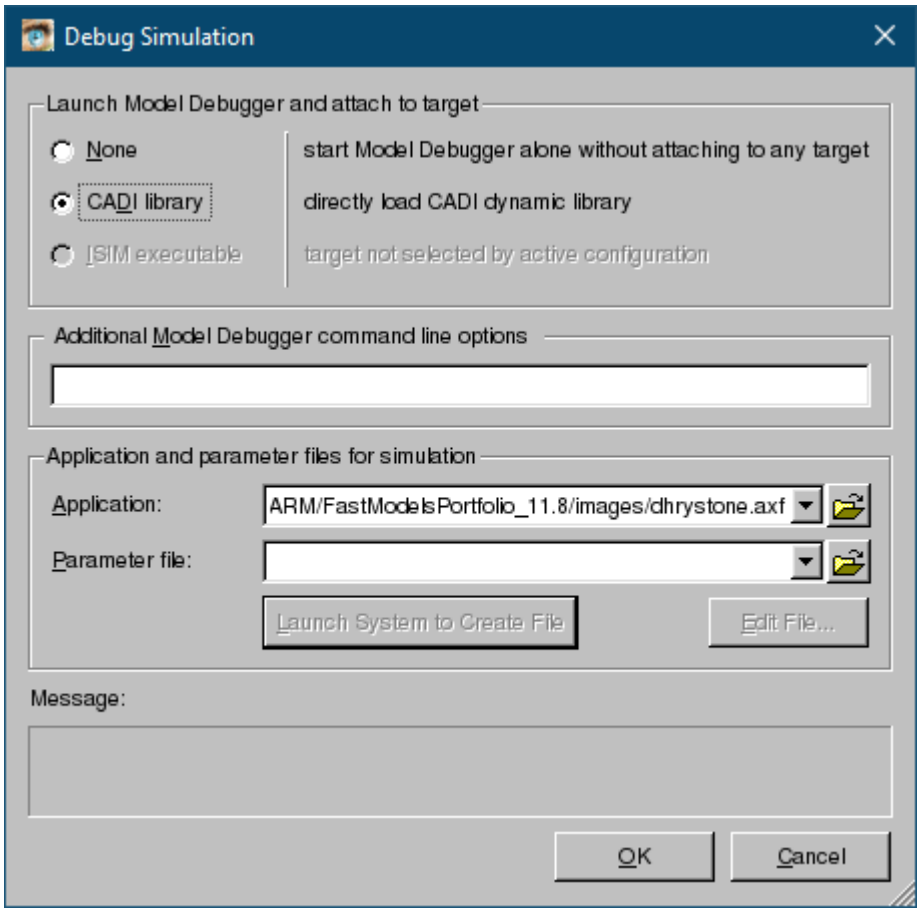
5.9 Debugging with Model Debugger

This section describes how to use Model Debugger to debug the model.

Procedure

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

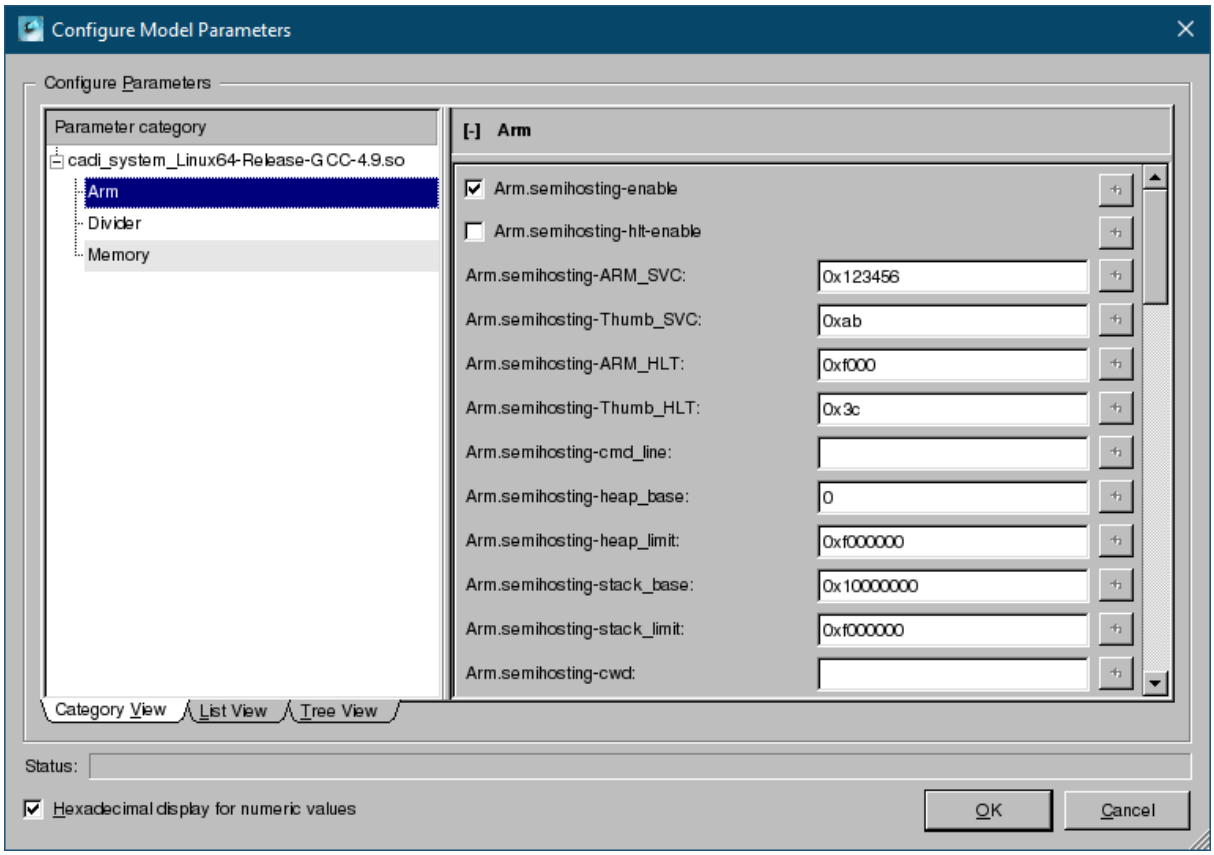
   Figure 5-14: Debug Simulation dialog



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 in the **Application** field.
   This example uses `dhrystone.axf`, which is part of the Third-Party IP add-on package for the Fast Models Portfolio.
4. Click **OK** to start Model Debugger.
   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:

Figure 5-15: Configure Model Parameters dialog



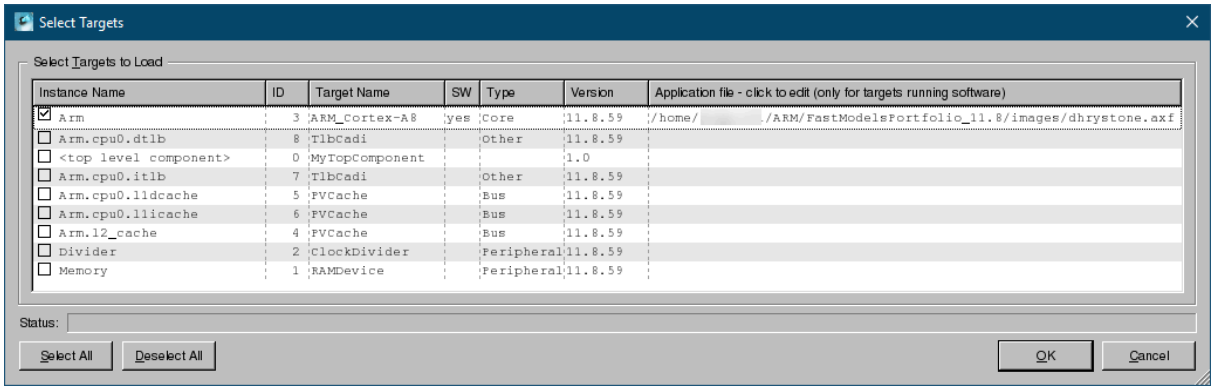
To display parameter sets:

- 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.

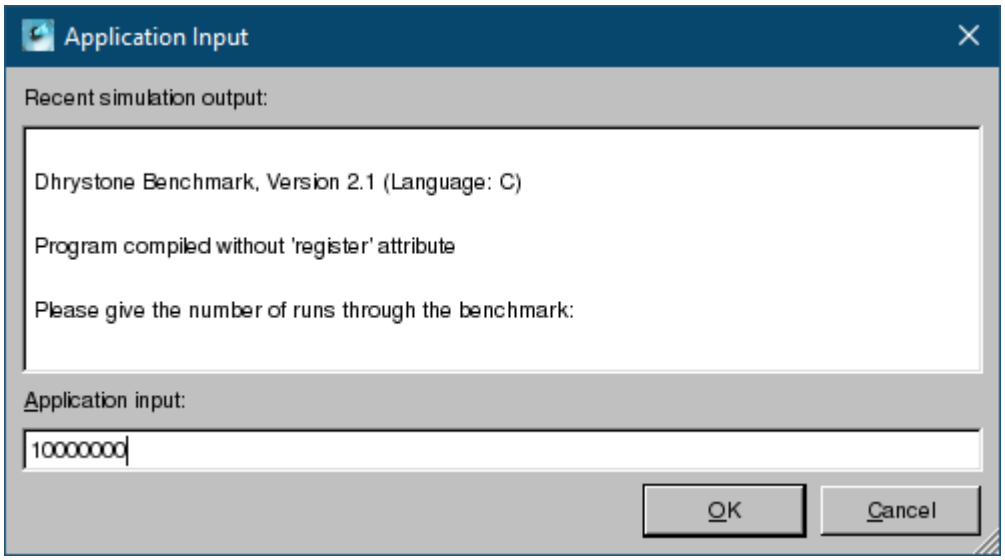
Figure 5-16: Select Targets dialog



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

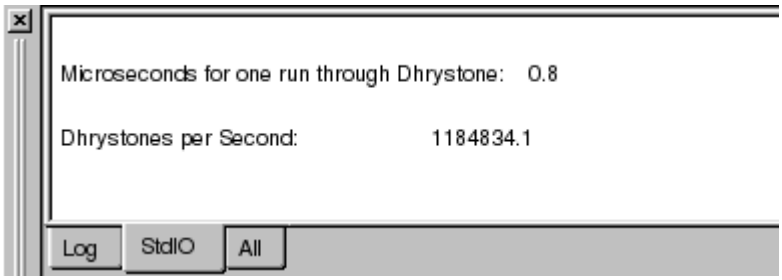
6. Click **OK** to close the dialog.
7. Click **Run** to start the simulation.
   The **Application Input** window appears:

Figure 5-17: Model Debugger Application Input window



8. Enter the required number of runs through the benchmark in the **Application input** field and click **OK**.
   After a short pause, the benchmark results are shown in the **StdIO** window.

Figure 5-18: Model Debugger StdIO window



Related information

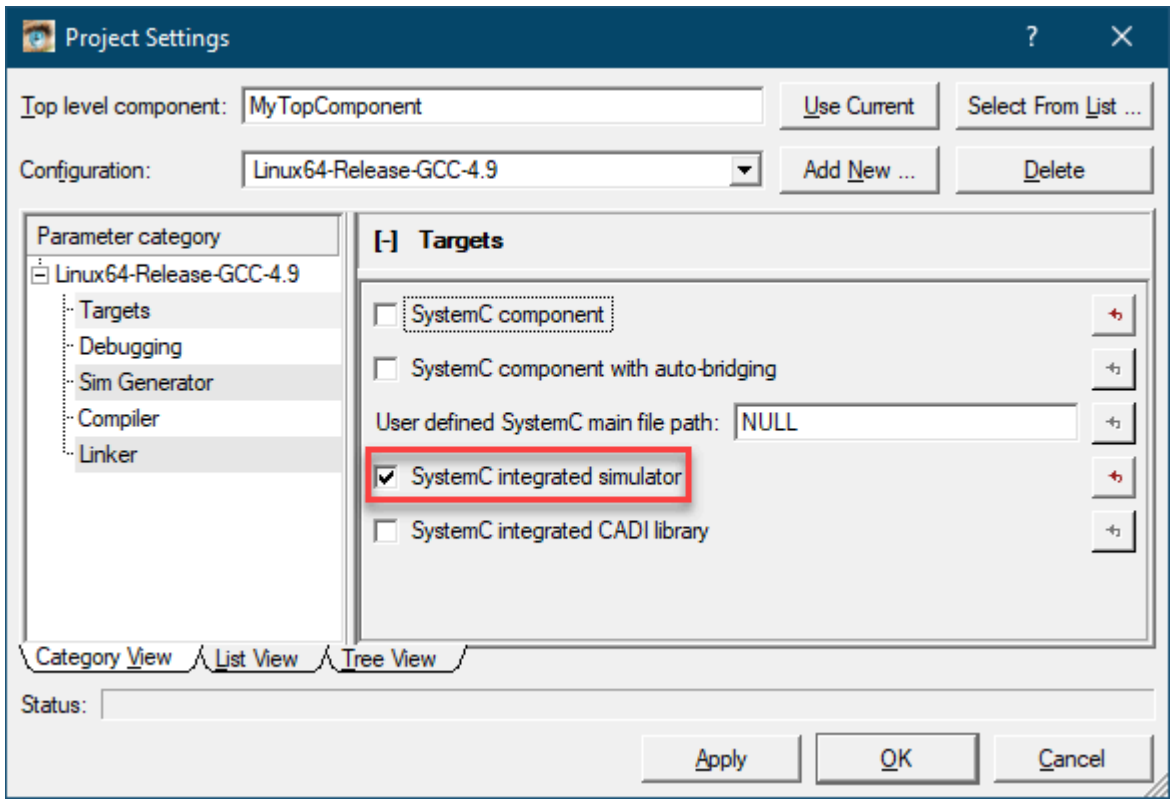
Model Debugger for Fast Models User Guide

5.10 Building a SystemC ISIM target

To build the platform as a standalone executable SystemC Integrated SIMulator (ISIM), tick the **SystemC integrated simulator** checkbox in the **Targets** category in the Project Settings dialog.

About this task

Figure 5-19: Building a SystemC integrated simulator target



This option is selected by default for new projects.

System Canvas generates a SystemC ISIM target by statically linking the model with the SystemC framework.

The output executable is called `isim_system`, and is generated in the build directory.

Related information

Building Fast Models on page 40

6 System Canvas Reference

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

6.1 Launching System Canvas

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

Procedure

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

The sgcanvas command has the following options:

Table 6-1: System Canvas command line options

Short form	Long form	Description
`-h`	`--help`	Print help text and exit.
`-v`	`--version`	Print version and exit.

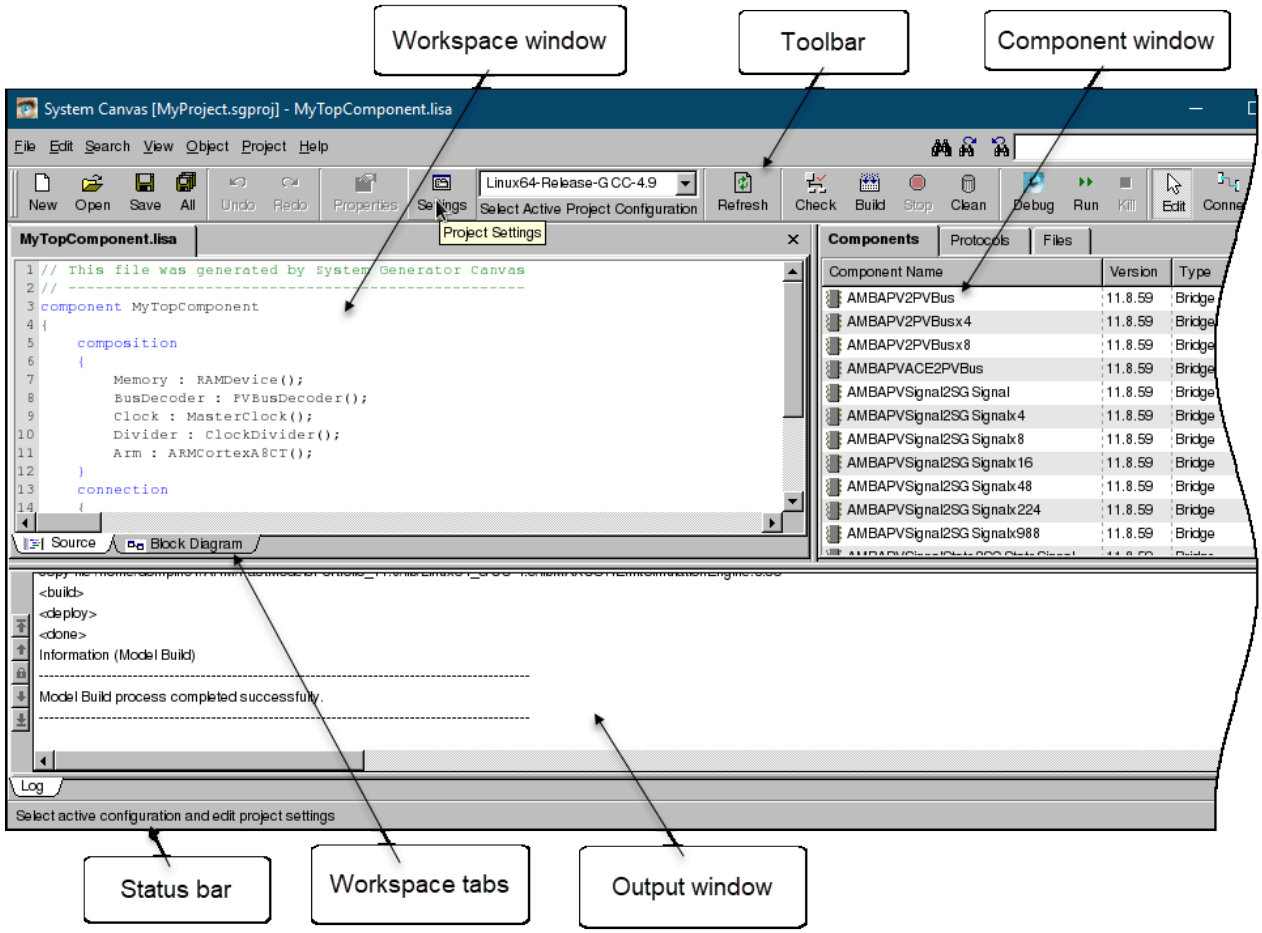
6.2 System Canvas GUI

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.

6.2.1 Application window

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

Figure 6-1: Layout of System Canvas



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.

6.2.2 Menu bar

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

6.2.2.1 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 information

New project dialogs on page 102
Preferences - Suppressed messages group on page 109

6.2.2.2 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.

6.2.2.3 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 information

Find and Replace dialogs on page 100

6.2.2.4 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.

6.2.2.5 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.

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.

6.2.2.6 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.

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.

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 information
Preferences - Applications group on page 105

6.2.2.7 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 Guide
Display the Model Shell for Fast Models Reference Guide.

LISA+ Language Reference Guide
Display the LISA+ Language for Fast Models Reference Guide.

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

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

Release Notes
Display this document.

Documents in $PVLIB_HOME/Docs
List the PDF files in the directory `$PVLIB_HOME/Docs`.

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.

6.2.3 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.

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 information

Viewing the project settings on page 61
Edit menu on page 75
Component Instance Properties dialog on page 91
Component Model Properties dialog for the system on page 93
Connection Properties dialog on page 96
Label Properties dialog on page 100
New File dialog (File menu) on page 101
Open File dialog on page 103
Port Properties dialog on page 103
Self Port dialog on page 122

6.2.4 Workspace window

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

Related information

Open File dialog on page 103
Preferences dialog on page 104

6.2.4.1 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.

6.2.4.2 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.

6.2.4.3 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.

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.

6.2.4.4 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.

6.2.5 Component window

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

6.2.5.1 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.

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.

6.2.5.2 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.

6.2.6 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.

6.3 System Canvas dialogs

This section describes the dialog boxes of System Canvas.

6.3.1 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.

Related information

Add Files dialog (Project menu) on page 90
New File dialog (File menu) on page 101

6.3.1.1 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

Display a dialog by right-clicking in the Component window and selecting from the context menu:

- **Add**.
- **Add New**.

6.3.1.2 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.

Option	Description
Add	In the Add Existing Files dialog, go to the file and select it.
Add New	In the Add New File dialog, go to the directory to contain the file and enter the name.

   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 information

File/Path Properties dialog on page 97

6.3.1.3 Using environment variables in filepaths

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

About this task

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 information

File/Path Properties dialog on page 97

6.3.1.4 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

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 information

File/Path Properties dialog on page 97
Project Settings dialog on page 109

6.3.2 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 information

Add Existing Files and Add New File dialogs (Component window) on page 88
File/Path Properties dialog on page 97

6.3.3 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 information

Edit Connection dialog on page 97

6.3.4 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 6-2: Suffixes for parameter values

Suffix	Name	Multiplier
K	Kilo	2^{10}
M	Mega	2^{20}
G	Giga	2^{30}
T	Tera	2^{40}
P	Peta	2^{50}

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.

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.

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 information

Component Model Properties dialog for the system on page 93
Component Properties dialog for a library component on page 95
Label Properties dialog on page 100
Port Properties dialog on page 103
Protocol Properties dialog on page 121
Self Port dialog on page 122

6.3.5 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 6-3: Component properties

Property	ID	Default	Description
Component name	`component_name`	""	A string containing the name for the component.
Component category	`component_type`	""	A string describing the type of component. This can be `"Processor"`, `"Bus"`, `"Memory"`, `"System"`, or any free-form category text.
Component description	`description`	""	A textual component description.
Component documentation	`documentation_file`	""	A filepath or an HTTP link to documentation. Supported file formats are `PDF`, `TXT`, and `HTML`.
Executes software	`executes_software`	0	The component executes software and can load application files. `1` for processor-like components, `0` for other components.
Hidden	`hidden`	0	`1` for components hidden from the Component window. Otherwise, hidden components behave exactly as normal components, and they do appear in the Workspace window.
Has CADI interface	`has_cadi`	1	`1` for components with a CADI interface, permitting connection to the target with a CADI-compliant debugger. `0` for components with no CADI interface.
Icon pixmap file	`icon_file`	""	The `XPM` file that contains the system icon.
License feature	`license_feature`	""	The license feature string required to run this system model.
Load file extension	`loadfile_extension`	""	The application filename extension for this target. Example: `".elf"` or `".hex"`.
Small icon pixmap file	`small_icon_file`	""	The `XPM` file that contains the 12x12 pixel system icon.
Component version	`version`	"1.0"	The version of the component.

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 6-4: Suffixes for parameter values

Suffix	Name	Multiplier
K	Kilo	2^{10}
M	Mega	2^{20}
G	Giga	2^{30}
T	Tera	2^{40}
P	Peta	2^{50}

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.

6.3.6 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`.

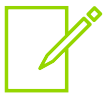
Version
The revision number for the component.

File
The file that defines the component.

Repository
The repository that contains the component.

Description
Information about the component.

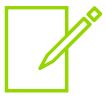
Properties (read only)
All the usable properties of the component.


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.

6.3.7 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 information

Port Properties dialog on page 103

6.3.8 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.

6.3.9 File/Path Properties dialog

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

- 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 information

Project parameter IDs on page 115

6.3.10 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.

6.3.11 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.

6.3.12 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.

Related information

Select Top Component LISA File dialog on page 103

6.3.13 New project dialogs

This section describes the dialogs that create new projects.

6.3.13.1 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.

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

Related information

Preferences - Default Model Repository group on page 108

6.3.13.2 Select Top Component LISA File dialog

This dialog controls the name of the top-level LISA file for a project.

After clicking **Select** in the New Project dialog, this dialog appears. By default, the filename for the top-level LISA file is the same as the project name. You can, however, specify a different name in this dialog.

6.3.14 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.

- 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.

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.

6.3.15 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 information

Connection Properties dialog on page 96

6.3.16 Preferences dialog

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

6.3.16.1 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.

6.3.16.2 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.

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:

`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:

```
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 information

SimGen command-line options on page 41
Model Debugger for Fast Models User Guide
Model Shell for Fast Models Reference Guide
Project Settings dialog on page 109

6.3.16.3 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.

6.3.16.4 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.

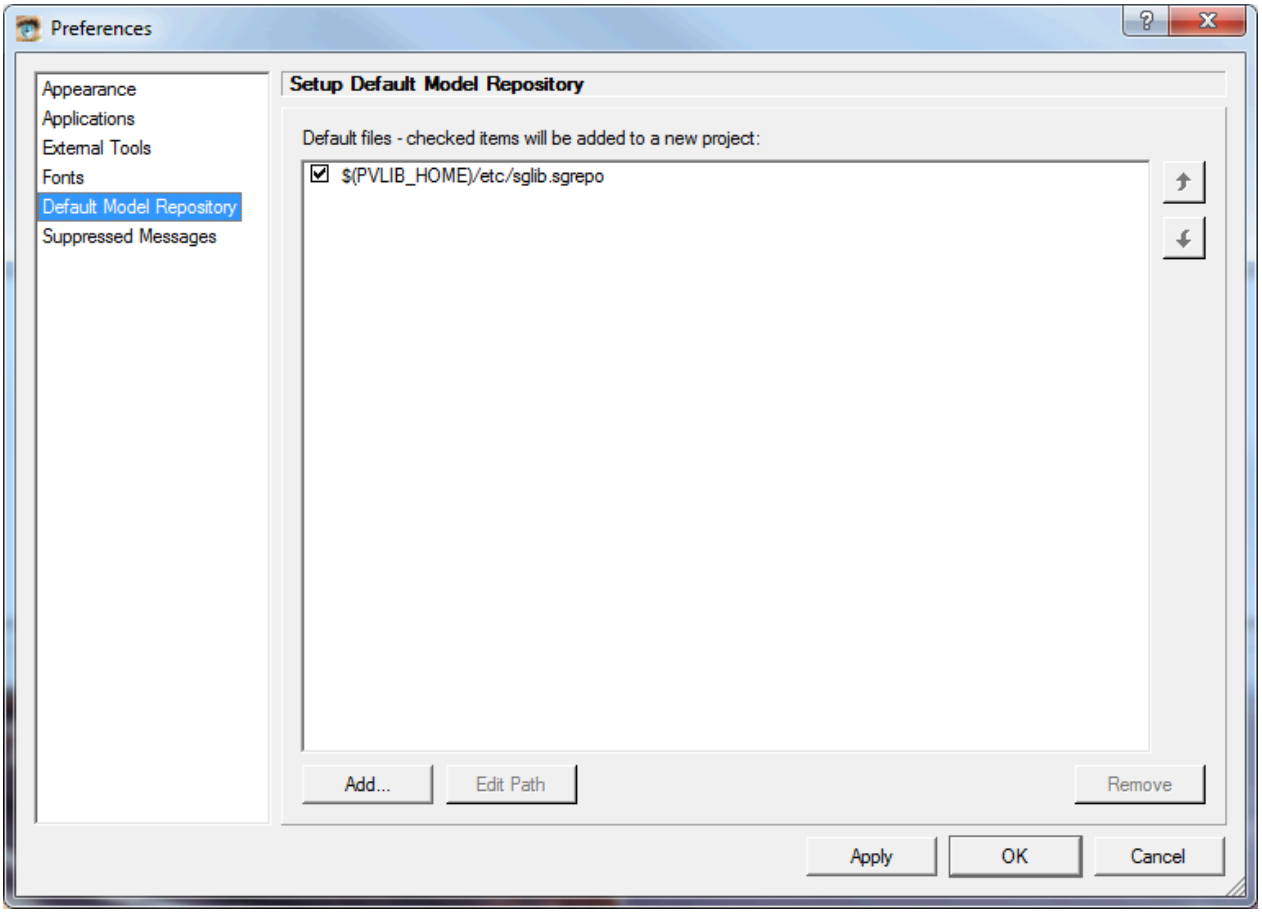
Note

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.

6.3.16.5 Preferences - Default Model Repository group

This group sets the default model repositories for new projects.

Figure 6-2: Preferences dialog, Setup Default Model Repository



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 information

Repository files on page 29

6.3.16.6 Preferences - Suppressed messages group

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

Enable selected messages

Click to enable selected suppressed messages.

6.3.17 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.

6.3.17.1 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.

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.

6.3.17.2 Parameter category panel

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

6.3.17.2.1 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 6-5: Configuration parameters in the Category View

Control name	Parameter
Configuration name	CONFIG_NAME
Platform/Linkage	PLATFORM
Compiler	COMPILER
Configuration description	CONFIG_DESCRIPTION
Build directory	BUILD_DIR

Targets

Select the **Targets** item to configure the build target parameters.

Table 6-6: Target parameters in the Category View

Control name	Parameter
SystemC component	TARGET_SYSTEMC
SystemC component with auto-bridging	TARGET_SYSTEMC_AUTO
SystemC integrated simulator	TARGET_SYSTEMC_ISIM
SystemC integrated CADI library	TARGET_SYSTEMC_MAXVIEW

Debugging

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

Table 6-7: Debugging parameters in the Category View

Control name	Parameter
Enable model debugging	ENABLE_DEBUG_SUPPORT
Source reference	GENERATE_LINEINFO
Verbosity	VERBOSITY
Model Debugger	MODEL_DEBUGGER_COMMAND_LINE
Model Shell and ISIM	MODEL_SHELL_COMMAND_LINE
SystemC executable	SYSTEMC_EXE
SystemC arguments	SYSTEMC_COMMAND_LINE

Sim Generator

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

Table 6-8: Simulation Generator parameters in the Category View

Control name	Parameter
Simgen options	SIMGEN_COMMAND_LINE
Warnings as errors	SIMGEN_WARNINGS_AS_ERRORS
Using namespace std	ENABLE_NAMESPACE_STD
Make options	MAKE_OPTIONS

Compiler

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

Table 6-9: Compiler parameters in the Category View

Control name	Parameter
Pre-Compile Actions	PRE_COMPILE_EVENT
Include Directories	INCLUDE_DIRS
Preprocessor Defines	PREPROCESSOR_DEFINES
Compiler Settings	ADDITIONAL_COMPILER_SETTINGS
SCX Library Settings	ADDITIONAL_SCX_LIB_SETTINGS
Enable pre-compiling	ENBALE_PRECOMPILE_HEADER

Linker

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

Table 6-10: Linker parameters in the Category View tab

Control name	Parameter
Pre-Link Actions	PRE_LINK_EVENT
Linker Settings	ADDITIONAL_LINKER_SETTINGS
Post-Build Actions	POST_BUILD_EVENT
Post-Clean Actions	POST_CLEAN_EVENT
Disable suppression of symbols	DISABLE_SYMBOL_SUPRESSION

6.3.17.2.2 Parameters - List View

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

6.3.17.2.3 Parameters - Tree View

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

6.3.17.2.4 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.

Option	Description
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.

6.3.17.2.5 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()`.

About this task

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

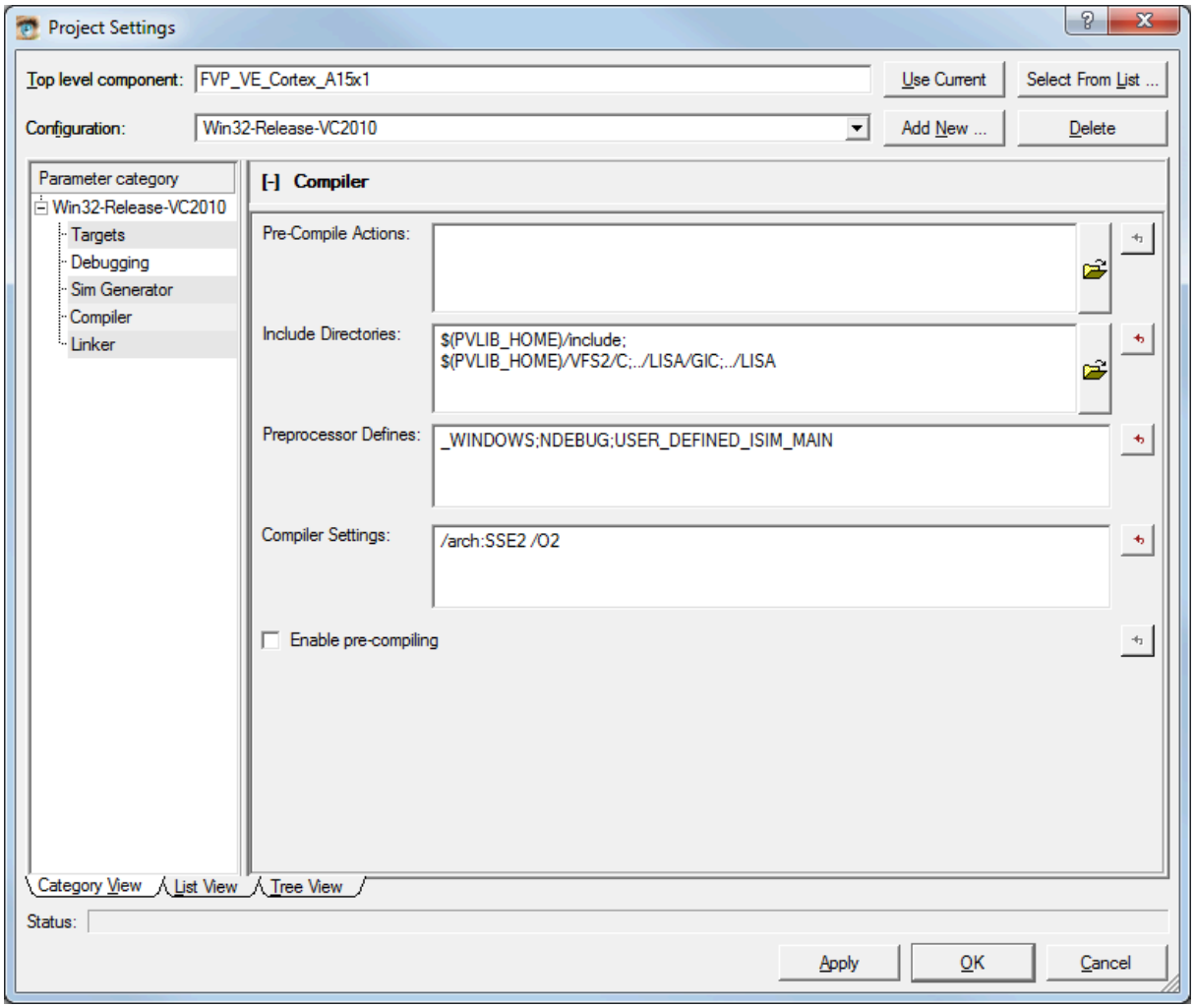
If a CADI shared library is required:

- 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.

   Figure 6-3: Specifying user-defined main() option



2. Supply a C++ file or a library with a user-defined `main()` function.
   A fragment of the standard `IsimMain.cpp` file:

```
#ifdef USER_DEFINED_ISIM_MAIN    // opposite logic to standard IsimMain.cpp
#include "SimGenTplMacros.h"
// function that performs command line parsing
// 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 // #ifdef  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()`.

3. Add the new source file containing the custom `main()` to the project.

Related information

Add Existing Files and Add New File dialogs (Component window) on page 88

6.3.17.3 Project parameter IDs

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

Table 6-11: Full list of parameters shown in List View

Parameter ID	Parameter name	Default	Description
`ADDITIONAL_COMPILER_SETTINGS`	Compiler settings	`""`	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.
`ADDITIONAL_LINKER_SETTINGS`	Linker settings	`""`	Linker settings. For Microsoft Windows, consult the Visual Studio documentation.
`BUILD_DIR`	Build directory	`""`	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`.
`COMPILER`	Compiler	–	**`VC2019`** Microsoft Visual Studio 2019. **`gcc`** The first gcc version in the Linux search path. **`gcc-7.3`** GCC 7.3. **`gcc-9.3`** GCC 9.3.
`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.

Parameter ID	Parameter name	Default	Description
ENABLE_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.
ENABLE_PRECOMPILE_HEADER	Enable precompiling	0	Precompile headers if true/1.
GENERATE_LINEINFO	Source reference	"LISA Code (incl headers)"	Control line redirection in the generated model source code: **"LISA Code"** Source code. **"LISA Code (incl. headers)"** Source and header. **"Generated Code"** No line redirection at all.
INCLUDE_DIRS	Include directories	""	Include directories. Separate multiple entries with semicolons.
MODEL_DEBUGGER_COMMAND_LINE	Model Debugger	""	Options to pass on the command line.
MODEL_SHELL_COMMAND_LINE	Model Shell	""	Options to pass on the command line.
PLATFORM	Platform/ linkage	-	**"Linux64"** 64-bit Linux. **"Win64"** 64-bit Microsoft Windows release. **"Win64D"** 64-bit Microsoft Windows debug.
POST_BUILD_EVENT	Postbuild actions	""	Commands to execute after building the model. Separate multiple entries with semicolons.
PRE_COMPILE_EVENT	Precompile actions	""	Commands to execute before starting compilation. Applies to Microsoft Windows only. Separate multiple entries with semicolons.
PREPROCESSOR_DEFINES	Preprocessor defines	""	Preprocessor defines. Separate multiple entries with semicolons.
PRE_LINK_EVENT	Prelink actions	""	Commands to execute before starting linking. Applies to Microsoft Windows only. Separate multiple entries with semicolons.
SIMGEN_COMMAND_LINE	SimGen options	""	Options to pass on the command line.
SIMGEN_WARNINGS_AS_ERRORS	Warnings as errors	"0"	If 1 (true), treat LISA parsing and compiler warnings as errors.
SYSTEMC_COMMAND_LINE	SystemC arguments	""	Command-line arguments for SystemC executable.
SYSTEMC_EXE	SystemC executable	""	Name of final SystemC executable. Call the file with 'Run SystemC executable'.

Parameter ID	Parameter name	Default	Description
TARGET_SYSTEMC	SystemC component	0	If 1 (true), build an EVS library.
TARGET_SYSTEMC_AUTO	SystemC component with auto-bridging	0	If 1 (true), build an EVS library with auto-bridging.
TARGET_SYSTEMC_ISIM	SystemC integrated simulator	0	If 1 (true), build a SystemC ISIM executable. **Note:** You cannot select both TARGET_SYSTEMC_MAXVIEW and TARGET_SYSTEMC_ISIM.
TARGET_SYSTEMC_MAXVIEW	SystemC integrated CADI library	0	If 1 (true), build a CADI system dynamic library with the SystemC scheduler for running from Model Debugger or Model Shell. **Note:** You cannot select both TARGET_SYSTEMC_MAXVIEW and TARGET_SYSTEMC_ISIM.
VERBOSITY	Verbosity	"Off"	Verbosity level: "Sparse", "On", or "Off".

Related information

Auto-bridging on page 125

6.3.17.4 Project file contents

Project (.sgproj) files describe the build settings and the files required to build a platform.

File and directory names can be either absolute or relative to the project file location. You can use environment variables in filenames.

Platform

"Linux64"
64-bit Linux.

"Win64"
64-bit Microsoft Windows release.

"Win64D"
64-bit Microsoft Windows debug.

Compiler

"VC2019"
Microsoft Visual Studio 2019.

"gcc"
The first gcc version in the Linux search path.

"gcc-7.3"
GCC 7.3.

"gcc-9.3"
GCC 9.3.

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 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 case sensitive.

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

```
path = my_file.lib, platform="WIN64|Win64D", compiler="VC2019", action="link|
deploy";
```

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

```
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:

```
path = ../lib/release_2019/my_lib.lib, platform = "win64", compiler="VC2019";
path = ../lib/my_lib.lib, platform = "win64", compiler="VC2019";
path = ../src/my_windows_code.cpp, platform = "win64"; // Not specifying the
 compiler allows
                                                        // more than one.
```

Related information

File processing order on page 30
File/Path Properties dialog on page 97

6.3.17.4.1 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 a file, System Generator applies them to the directory that contains the file. It forms the directory path by removing the filename from the full path. This path specification:

```
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.

6.3.17.4.2 Example project file

An example project file with different configuration sections for building an EVS library.

Example project file with Windows and Linux configurations

```
sgproject "exampleSystem.sgproj"
{
    TOP_LEVEL_COMPONENT = "exampleSystem";
    ACTIVE_CONFIG_LINUX  = "Linux64-Release-GCC-7.3";
    ACTIVE_CONFIG_WINDOWS  = "Win64-Release-VC2019";
    config "Linux64-Debug-GCC-7.3"
    {
        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-7.3";
        COMPILER = "gcc-7.3";
        CONFIG_DESCRIPTION = "Default x86_64 Linux configuration for GCC 7.3 with debug
 information";
        CONFIG_NAME = "Linux64-Debug-GCC-7.3";
        ENABLE_DEBUG_SUPPORT = "1";
        PLATFORM = "Linux64";
        SIMGEN_COMMAND_LINE = "--num-comps-file 10";
        TARGET_SYSTEMC = "1";
    }
    config "Linux64-Release-GCC-7.3"
    {
```

```
        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-7.3";
        COMPILER = "gcc-7.3";
        CONFIG_DESCRIPTION = "Default x86_64 Linux configuration for GCC 7.3, optimized for
 speed";
        CONFIG_NAME = "Linux64-Release-GCC-7.3";
        PLATFORM = "Linux64";
        PREPROCESSOR_DEFINES = "NDEBUG";
        SIMGEN_COMMAND_LINE = "--num-comps-file 50";
        TARGET_SYSTEMC = "1";
    }
    config "Linux64-Debug-GCC-9.3"
    {
        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-9.3";
        COMPILER = "gcc-9.3";
        CONFIG_DESCRIPTION = "Default x86_64 Linux configuration for GCC 9.3 with debug
 information";
        CONFIG_NAME = "Linux64-Debug-GCC-9.3";
        ENABLE_DEBUG_SUPPORT = "1";
        PLATFORM = "Linux64";
        SIMGEN_COMMAND_LINE = "--num-comps-file 10";
        TARGET_SYSTEMC = "1";
    }
    config "Linux64-Release-GCC-9.3"
    {
        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-9.3";
        COMPILER = "gcc-9.3";
        CONFIG_DESCRIPTION = "Default x86_64 Linux configuration for GCC 9.3, optimized for
 speed";
        CONFIG_NAME = "Linux64-Release-GCC-9.3";
        PLATFORM = "Linux64";
        PREPROCESSOR_DEFINES = "NDEBUG";
        SIMGEN_COMMAND_LINE = "--num-comps-file 50";
        TARGET_SYSTEMC = "1";
    }
    config "Win64-Debug-VC2019"
    {
        ADDITIONAL_COMPILER_SETTINGS = "/Od /RTCsu /Zi";
        ADDITIONAL_LINKER_SETTINGS = "/DEBUG";
        BUILD_DIR = "./Win64-Debug-VC2019";
        COMPILER = "VC2019";
        CONFIG_DESCRIPTION = "Default x86_64 Windows configuration for Visual Studio 2019 with
 debug information";
        CONFIG_NAME = "Win64-Debug-VC2019";
        ENABLE_DEBUG_SUPPORT = "1";
        PLATFORM = "Win64D";
        SIMGEN_COMMAND_LINE = "--num-comps-file 10";
        TARGET_SYSTEMC = "1";
    }
    config "Win64-Release-VC2019"
    {
        ADDITIONAL_COMPILER_SETTINGS = "/O2";
        BUILD_DIR = "./Win64-Release-VC2019";
        COMPILER = "VC2019";
        CONFIG_DESCRIPTION = "Default x86_64 Windows configuration for Visual Studio 2019,
 optimized for speed";
        CONFIG_NAME = "Win64-Release-VC2019";
        PLATFORM = "Win64";
        PREPROCESSOR_DEFINES = "NDEBUG";
        SIMGEN_COMMAND_LINE = "--num-comps-file 50";
        TARGET_SYSTEMC = "1";
    }
    files
```

```
    {
        path = "$(PVLIB_HOME)/etc/sglib.sgrepo";
        path = "../LISA/exampleSystem.lisa";
        path = "../LISA/exampleComponent.lisa";
    }
}
```

6.3.18 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.

6.3.19 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.

6.3.20 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.

7 SystemC Export with Multiple Instantiation

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

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. A platform that consists of multiple EVSs is called an SVP (SystemC Virtual 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 2.3.3, including integrated TLM 2.0.5. In this version, the TLM and SystemC headers are in the same place, and some filenames are different.

Before using SimGen to build a SystemC simulation, the environment variable `SYSTEMC_HOME` must be set to the directory containing the Accellera SystemC library installation.

When running a SystemC simulation, the following environment variables might be useful:

`SCX_EVS_VERBOSE`
Set to 1 to enable tracing of the default scheduler mapping implementation.

`FM_SCX_VERBOSITY_LEVEL`
Set to one of the following values to set the verbosity level for debug messages from the SystemC simulation:

0	None
100	Low
200	Medium
300	High
400	Full

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

Fast Models Reference Guide
Accellera Systems Initiative (ASI)
IEEE Std 1666-2005, SystemC Language Reference Manual, 31 March 2006
Accellera, TLM 2.0 Language Reference Manual, July 2009

7.2 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, `PVBus2AMBAPV` 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:

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

```
            "name" : "<bridge_name>"
        }
    },
```

- 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`
  - `ClockRateControl`
  - `ClockSignal`
  - `CounterInterface`
  - `GICv3Comms`
  - `InstructionCount`
  - `KeyboardStatus`
  - `LCD`
  - `MouseStatus`
  - `PChannel`
  - `SystemCoherencyInterface`
  - `VECBProtocol`
  - `VirtualEthernet`

  To access the generated TLM sockets from SystemC, you must `#include` the appropriate header files from under `$PVLIB_HOME/examples/SystemCExport/Common/Protocols/`.

Note

7.3 SystemC Export generated ports

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

Related information

About SystemC Export generated ports on page 251

7.3.1 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

LISA+ Language for Fast Models Reference Guide

7.3.2 TLM 1.0 protocol for an exported SystemC component

Here is an example of a TLM 1.0 signal protocol.

```
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);
}
```

7.3.3 TLM 2.0 bus protocol for an exported SystemC component

Here is an example of a TLM 2.0 bus protocol.

```
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);
  master behavior invalidate_dmi(uint32_t addr);
}
```

This protocol enables declaring ports that have `read()` and `write()` functions. This protocol can declare master and slave ports.

7.3.4 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:

```
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:

```
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:

```
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

```
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

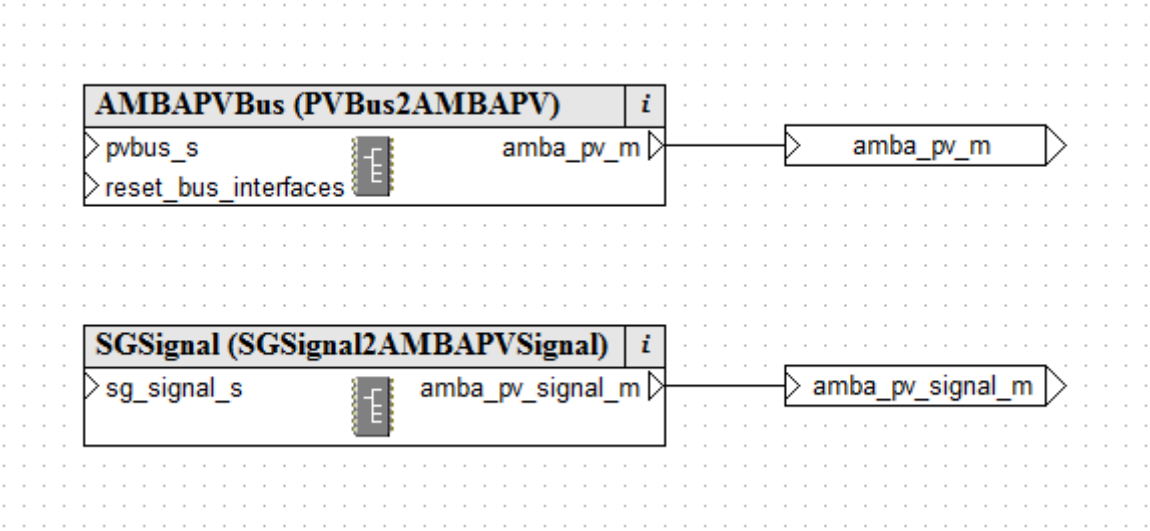
The SystemC module ports must use the corresponding names in the SystemC code.

```
class pv_dma: public sc_module,
              public amba_pv::signal_slave_base<bool> {

  /* Module ports */
  amba_pv::signal_master_port<bool> signal_out;
  amba_pv::signal_slave_export<bool> signal_in;
  ...
```

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.

Figure 7-1: SGSignal component in System Canvas



7.3.5 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:

```
my_master_base::invalidate_dmi(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:

```
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

```
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.

```
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;
    ...
}
```

7.4 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.

7.4.1 SystemC Export header file

To use the SystemC Export feature, an application must include the C++ header file `scx.h` 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.

```
#include "scx.h"
```

7.4.2 scx::scx_initialize

This function initializes the simulation.

Initialize the simulation before constructing any exported subsystem.

```
void scx_initialize(const std::string &id,
                    scx_simcontrol_if *ctrl = scx_get_default_simcontrol());
```

id

an identifier for this simulation.

ctrl
a pointer to the simulation controller implementation. It defaults to the one provided with Arm® models.


Note

Arm recommends specifying a unique identifier across all simulations running on the same host.

7.4.3 scx::scx_set_single_evs

Sets the simulation engine to accept a single EVS only.

```
void scx_set_single_evs();
```

The EVS name will be stripped from CADI parameters.

Call this function immediately after calling `scx_initialize()`.

7.4.4 scx::scx_load_application

This function loads an application in the memory of an instance.

```
void scx_load_application(const std::string &instance,
                          const std::string &application);
```

instance
the name of the instance to load into. The parameter `instance` must start with an EVS instance name, or with `"*"` to load the application into the instance on all EVSs in the platform. To load the same application on all cores of an SMP processor, specify `"*"` for the core instead of its index, in parameter `instance`.

application
the application to load.


Note

The loading of the application happens at `start_of_simulation()` call-back, at the earliest.

7.4.5 scx::scx_load_application_all

This function loads an application in the memory of instances that execute software, across all EVSs in the platform.

```
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.

7.4.6 scx::scx_load_data

This function loads binary data in the memory of an instance at a memory address.

```
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.

7.4.7 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.

```
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.

The loading of the binary data happens at `start_of_simulation()` call-back, at the earliest.

7.4.8 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.

- 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.

7.4.9 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.
  ```

7.4.10 scx::scx_get_parameter_list

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

```
std::map<std::string, std::string> scx_get_parameter_list();
```

The parameter names start with an EVS instance name for EVS parameters or with a plug-in prefix (defaults to `"TRACE"`) for plug-in parameters.

- 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()`.

7.4.11 scx::scx_get_parameter_infos

Retrieve a list of descriptions for all parameters within the simulation.

```
std::map<std::string, std::string> scx_get_parameter_infos();
```

The list includes parameters for all components present in all EVSs and for all plug-ins.

The names of EVS parameters start with an EVS instance name and the names of plug-in parameters start with a plug-in prefix, which defaults to `TRACE`.

Plug-ins must be specified before calling any of the platform parameter functions, otherwise plug-ins are not loaded and their parameters are not available. Any plug-in specified after the first call to any platform parameter function is ignored.

7.4.12 scx::scx_get_cadi_parameter_infos

Retrieve a vector of `CADIParameterInfo_t` objects for all the parameters in the simulation.

```
std::vector<eslapi::CADIParameterInfo_t> scx_get_cadi_parameter_infos();
```

Use this function to get CADI parameter objects with all the relevant fields present for all EVSs, external SystemC modules, and loaded plug-ins.


Note

Plug-ins must be specified before calling any of the platform parameter functions, otherwise plug-ins are not loaded and their parameters are not available. Any plug-in specified after the first call to any platform parameter function is ignored.

7.4.13 scx::scx_set_cpi_file

Sets the Cycles Per Instruction (CPI) file for CPI class functionality.

```
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.

7.4.14 scx::scx_cpulimit

Sets the maximum number of CPU (User + System) seconds to run, excluding startup and shutdown.

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

t
the number of seconds to run. Defaults to unlimited.

7.4.15 scx::scx_timelimit

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

```
void scx_timelimit(double t);
```

t
the number of seconds to run. Defaults to unlimited.

7.4.16 scx::scx_add_breakpoint

Set a breakpoint at a specific address.

```
void scx_add_breakpoint(std::string instance, uint64_t addr,
bool perthread, uint32_t threadid);
```

instance
Name of the core target instance.

addr
Address at which to set the breakpoint.

perthread
If true, the breakpoint only hits if `threadid` matches the current thread.

threadid
Thread ID for the breakpoint. Only used if `perthread` is true.

7.4.17 scx::scx_set_start_pc

Set the initial value of the PC register for a specific instance.

```
void scx_set_start_pc(std::string instance, uint64_t addr);
```

instance
Name of the core target instance.

addr
Start PC address.

7.4.18 scx::scx_dump

Set the details of a memory dump to be written to a file.

```
void scx_dump(std::string instance, std::string filename, std::string memSpace,
 uint64_t addr, uint64_t size);
```

instance
Name of the target instance to dump memory from.

filename
The path to the file to dump memory to.

memSpace
The name or ID of the memory space.

addr
Start address from which to dump.

size
Number of bytes of memory to dump.

7.4.19 scx::scx_load_params_file

Load parameter values from a configuration file.

```
void scx_load_params_file(const std::string& filename);
```

filename
The name of the configuration file to load.

Plug-ins must be specified before calling any of the platform parameter functions, otherwise these plug-ins will not be loaded and their parameters will not be available.

7.4.20 scx::scx_list_instances

List all instances in the simulation.

```
void scx_list_instances(const std::string& filename = std::string());
```

filename
The path to the file to hold the output. The default is an empty string, which sends output to `std::cout`.

7.4.21 scx::scx_list_registers

List all simulation registers.

```
void scx_list_registers(const std::string& filename = std::string());
```

filename
The path to the file to hold the output. The default is an empty string, which sends output to `std::cout`.

7.4.22 scx::scx_check_registers

List all simulation registers and perform extra consistency checks on the CADI register API.

```
void scx_check_registers(const std::string& filename = std::string());
```

filename
The path to the file to hold the output. The default is an empty string, which sends output to `std::cout`.

7.4.23 scx::scx_restore_checkpoint

Restore a checkpoint.

```
void scx_restore_checkpoint(const std::string& restoreCheckpointDirPath);
```

restoreCheckpointDirPath
Directory from which the checkpoint files will be restored.

7.4.24 scx::scx_save_checkpoint

Save a checkpoint.

```
void scx_save_checkpoint(const std::string& saveCheckpointDirPath);
```

saveCheckpointDirPath
Directory in which the checkpoint files will be stored.

7.4.25 scx::scx_list_memory

List all simulation memory.

```
void scx_list_memory(const std::string& filename = std::string());
```

filename
The path to the file to hold the output. The default is an empty string, which sends output to `std::cout`.

7.4.26 scx::scx_parse_and_configure

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

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

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

argv
command-line options.

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

--dump-params
dump the list of model parameters into a JSON file and exit.

-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.

--iris-port
set a specific port to use for the Iris server, format: `--iris-port PORT`

--iris-port-range
set the range of ports to scan when starting an Iris server. The first available port found is used, format: `--iris-port-range MIN:MAX`

-K, --keep-console
keep the console window open after completion. This option applies to Microsoft Windows only.

-l, --list-params
print the list of model 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.

7.4.27 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.

7.4.28 scx::scx_get_error_count

This function returns the number of errors recorded by the simulation engine.


Note

The count includes internal errors recorded by the simulation engine, some of which are not reported as errors by `scx_report_handler`.

```
size_t scx_get_error_count();
```

7.4.29 scx::scx_get_exitcode_list

This function returns the list of exit codes that were logged by the simulation engine.

The returned list is a `std::vector` that contains the logged exit codes in order. Each entry in the list is a struct of type 7.4.30 scx::scx_exitcode_entry on page 144. The last entry is the most recent.


Note

- If no exit code was logged, the returned list is empty.
- This function only produces valid output after `sc_start()` has returned. It must not be called beforehand.

```
const scx_exitcode_list_t & scx_get_exitcode_list();
```

7.4.30 scx::scx_exitcode_entry

Represents an entry in the exit code list.


Note

The exit code list is returned by 7.4.29 scx::scx_get_exitcode_list on page 144.

```
struct scx_exitcode_entry
{
    scx_exitcode_entry(int exitcode_, std::string component_name_, std::string
 kind_, std::string reason_)
    : exitcode(exitcode_)
    , component_name(component_name_)
    , kind(kind_)
    , reason(reason_)
    {}

    int exitcode;
    std::string component_name;
    std::string kind;
    std::string reason;
};
```

exitcode

The exit code that was logged.

component_name

The name of the component that generated the exit code. This name is auto-generated by the simulation engine at the time of logging.

kind

The type of component that generated the exit code.

reason

Optional field that provides a human-readable string explaining why the exit code was logged. If this field is empty, then no reason was given and this field can be ignored.

7.4.31 scx::scx_start_cadi_server

This function specifies whether to start a CADI server.

```
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 the 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 after 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()`.

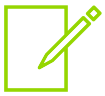
7.4.32 scx::scx_enable_cadi_log

This function specifies whether to log all CADI calls to XML files.

```
void scx_enable_cadi_log(bool log = true);
```

log

`true` to log CADI calls, `false` otherwise.


Note

You cannot enable logging once simulation starts.

7.4.33 scx::scx_print_port_number

This function specifies whether to enable printing of the TCP port number that the CADI server is listening to.

```
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.

7.4.34 scx::scx_print_statistics

This function specifies whether to enable printing of simulation statistics at the end of the simulation.

```
void scx_print_statistics(bool print = true);
```

print

`true` to enable printing of simulation statistics, `false` otherwise.


Note

- You cannot enable printing of statistics once simulation starts.
- The statistics include LISA `reset()` behavior run time and application load time. A long simulation run compensates for this.

7.4.35 scx::scx_register_cadi_target

Register a CADI target info and interface into the simulation.

```
void scx_register_cadi_target(eslapi::CADITargetInfo_t * info, eslapi::CAInterface *
 caif = NULL);
```

info

Points to an `eslapi::CADITargetInfo_t` structure describing this CADI target.

caif

Points to an `eslapi::CAInterface` of this CADI target.

Use this function to register a target into the simulation. The target is then accessible through a CADI debugger attached to the simulation.


Note

Registering a target must be perfomed before the end of elaboration.

7.4.36 scx::scx_unregister_cadi_target

Unregister a specific CADI target from the simulation.

```
void scx_unregister_cadi_target(const std::string &);
```

name
Instance name of this CADI target.

Use this function to unregister a target from the simulation. After calling this function, the target will not be accessible through a CADI debugger.

7.4.37 scx::scx_load_trace_plugin

Arm deprecates this function. Use `scx_load_plugin()` instead.

7.4.38 scx::scx_load_plugin

This function specifies a plug-in to load.

```
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.

7.4.39 scx::scx_get_global_interface

This function accesses the global interface.

```
eslapi::CAInterface *scx_get_global_interface();
```

The global interface allows access to all of the registered interfaces in the simulation.

7.4.40 scx::scx_enable_iris_server

Starts or stops the Iris server.

```
void scx_enable_iris_server(bool enable = true);
```

enable
`true` to start an Iris server (default), `false` to stop it.


Note

Starting the Iris server puts the simulation into a wait state, until a client connects to the server.

7.4.41 scx::scx_set_iris_server_port_range

Set the range of ports to scan. The Iris server uses the first available port found in the range.

```
void scx_set_iris_server_port_range(uint16_t port_min, uint16_t port_max);
```

port_min
the port number at the start of the range.

port_max
the port number at the end of the range.


Note

This function only takes effect if you call it before starting the Iris server.

Related information
scx::scx_enable_iris_server on page 148

7.4.42 scx::scx_get_iris_server_port

Return the Iris TCP port number that is assigned when the Iris server starts, or zero if the Iris server has not yet started.

```
uint16_t scx_get_iris_server_port();
```

7.4.43 scx::scx_set_iris_server_port

Set a specific port for the Iris server to listen on.

```
inline void scx_set_iris_server_port(uint16_t port)
```

port
The port number for the Iris server to listen on.


Note

This function only takes effect if you call it before starting the Iris server.

Related information

scx::scx_enable_iris_server on page 148

7.4.44 scx::scx_enable_iris_log

This function sets the Iris message log level.

```
void scx_enable_iris_log(unsigned level = 0);
```

`level`

the log level. The possible values are:

0

Logging is disabled. This is the default value.

1

Log messages use a compact, single-line format.

2

Log messages use a single-line, pseudo-JSON format.

3

Log messages use a more readable multi-line, pseudo-JSON format.

4

As 3 but also prints the U64JSON hex value of the message.


Note

An alternative way to set the Iris log level is to use the `IRIS_GLOBAL_INSTANCE_LOG_MESSAGES` environment variable.

7.4.45 scx::scx_get_iris_connection_interface

Return the `IrisConnectionInterface` for the simulation. This can be used to create and register `IrisInstanceS`.

```
iris::IrisConnectionInterface *scx_get_iris_connection_interface();
```

7.4.46 scx::scx_evs_base

This class is the base class for EVSs. EVSs are the principal subsystems of the Fast Models SystemC Export feature.

```
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();
};
```

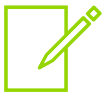
7.4.47 scx::load_application

This function loads an application in the memory of an instance.

```
void load_application(const std::string &instance, const std::string &application);
```

instance
: the name of the instance to load into.

application
: the application to load.


Note

The application loads at `start_of_simulation()`, at the earliest.

7.4.48 scx::load_data

This function loads raw data in the memory of an instance at a memory address.

```
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.

7.4.49 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.

7.4.50 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.

7.4.51 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();
```

7.4.52 scx::scx_evs_base 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.
```

7.4.53 scx::scx_evs_base destructor

This function destroys an EVS including the corresponding subsystem, and frees the associated resources.

```
~scx_evs_base();
```

7.4.54 scx::before_end_of_elaboration

This function calls the `instantiate()`, `configure()`, `init()`, `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.

7.4.55 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.

7.4.56 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.

7.4.57 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.

7.4.58 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.

7.4.59 scx::notify_running

This function notifies the simulation framework that the simulation is running.

```
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.

7.4.60 scx::notify_stopped

This function notifies the simulation framework that the simulation has stopped.

```
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.

7.4.61 scx::notify_debuggable

This function notifies the simulation framework that the simulation is debuggable.

```
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.

7.4.62 scx::notify_idle

This function notifies the simulation framework that the simulation is idle.

```
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.

7.4.63 scx::scx_simcallback_if destructor

Destructor.

```
~scx_simcallback_if();
```

This version of the function does not allow destruction of instances through the interface.

7.4.64 scx::scx_simcontrol_if

This is the simulation control interface.

```
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();
    virtual void notify_pending_debug();
    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}`.

7.4.65 scx::get_scheduler

This function returns a pointer to the implementation of the simulation scheduler.

```
eslapi::CAInterface *get_scheduler();
```

The simulation framework calls the `get_scheduler()` function to retrieve the scheduler implementation for the simulation at construction time.

Implementations of this function need not be thread-safe.

7.4.66 scx::get_report_handler

This function returns a pointer to the current implementation of the report handler.

```
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.

Implementations of this function need not be thread-safe.

7.4.67 scx::run

This function requests to run the simulation.

```
void run();
```

The simulation framework calls `run()` upon receipt of a CADI run request from a debugger.

7.4.68 scx::stop

This function requests to stop the simulation as soon as possible, that is at the next `wait()`.

```
void stop();
```

The simulation framework calls `stop()` upon receipt of a CADI stop request from a debugger, a component, or a breakpoint hit.

7.4.69 scx::is_running

This function returns whether the simulation is running.

```
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.

7.4.70 scx::stop_acknowledge

This function blocks the simulation while the simulation is stopped.

```
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`).

7.4.71 scx::process_debuggable

This function processes debug activity while the simulation is at a debuggable point.

```
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.

7.4.72 scx::notify_pending_debug

Notifies the simulation controller that debug requests are pending and need processing as soon as possible while the simulation is stopped.

```
virtual void notify_pending_debug()
```

An implementation of this behavior might simply call `scx_simcontrol::process_debuggable()` on all registered clients, while the simulation is stopped in `scx_simcontrol::stop_acknowledge()`.

7.4.73 scx::process_idle

This function processes idle activity while the simulation is stopped.

```
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.

7.4.74 scx::shutdown

This function requests to stop the simulation.

```
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()`.

7.4.75 scx::add_callback

This function registers call-backs with the simulation controller.

```
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.

7.4.76 scx::remove_callback

This function removes call-backs from the simulation controller.

```
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.

7.4.77 scx::scx_simcontrol_if destructor

Destructor.

```
~scx_simcontrol_if();
```

This version of the function does not allow destruction of instances through the interface.

7.4.78 scx::scx_get_default_simcontrol

This function returns a pointer to the default implementation of the simulation controller provided with Fast Models.

```
scx_simcontrol_if *scx_get_default_simcontrol();
```

7.4.79 scx::scx_get_curr_simcontrol

Return a pointer to the current simulation controller implementation.

```
extern scx_simcontrol_if * scx_get_curr_simcontrol();
```

7.4.80 scx::scx_report_handler_if

This interface is the report handler interface.

```
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

IEEE Std 1666-2005, SystemC Language Reference Manual, 31 March 2006

7.4.81 scx::scx_get_default_report_handler

This function returns a pointer to the default implementation of the report handler provided with Fast Models.

```
scx_report_handler_if *scx_get_default_report_handler();
```

7.4.82 scx::scx_get_curr_report_handler

This function returns a pointer to the current implementation of the report handler.

```
scx_report_handler_if *scx_get_curr_report_handler();
```

7.4.83 scx::scx_sync

This function adds a future synchronization point.

```
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.

Arm deprecates this function. Use IEEE 1666 SystemC 2011 `sc_core::sc_prim_channel::async_request_update()` instead.

7.4.84 scx::scx_set_min_sync_latency

This function sets the minimum synchronization latency for this scheduler.

```
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 information

scx::scx_get_min_sync_latency on page 163

7.4.85 scx::scx_get_min_sync_latency

This function returns the minimum synchronization latency, measured in seconds, for this scheduler.

```
double scx_get_min_sync_latency();
sg::ticks_t scx_get_min_sync_latency(sg::Tag<sg::ticks_t> *);
```

Related information

scx::scx_set_min_sync_latency on page 162

7.4.86 scx::scx_simlimit

This function sets the maximum number of seconds to simulate.

```
void scx_simlimit(double t);
```

t
the number of seconds to simulate. Defaults to unlimited.

7.4.87 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.

```
sg::SchedulerInterfaceForComponents *
 scx_create_default_scheduler_mapping(scx_simcontrol_if * simcontrol);
```

simcontrol
pointer to an existing simulation controller. When this is `NULL`, this function returns `NULL`.

7.4.88 scx::scx_get_curr_scheduler_mapping

This function returns a pointer to the current implementation of the scheduler mapping interface.

```
sg::SchedulerInterfaceForComponents * scx_get_curr_scheduler_mapping();
```

7.5 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.

7.5.1 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()`.

7.5.2 Scheduler API - use cases and implementation

This section describes uses of the Scheduler API.

7.5.2.1 Accessing the SchedulerInterfaceForComponents from within a modeling component

This section describes ways of accessing the global interfaces.

7.5.2.1.1 LISA component for accessing the SchedulerInterfaceForComponents

A way to access the global interfaces with `getGlobalInterface()`.

```
includes
{
 #include "sg/SGSchedulerInterfaceForComponents.h"
 #include "sg/SGComponentRegistry.h"
}

behavior init
{
 sg::SchedulerInterfaceForComponents *scheduler =
   sg::obtainComponentInterfacePointer<sg::SchedulerInterfaceForComponents>
     (getGlobalInterface(), "scheduler");
}
```

7.5.2.1.2 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.

```
#include "sg/SGSchedulerInterfaceForComponents.h"
#include "sg/SGComponentRegistry.h"

sg::SchedulerInterfaceForComponents *scheduler =
  sg::obtainComponentInterfacePointer<sg::SchedulerInterfaceForComponents>
    (simulationContext->getGlobalInterface(), "scheduler");
```

7.5.2.1.3 SystemC component for accessing the SchedulerInterfaceForComponents

A way to access the global interfaces with `scx::scx_get_global_interface()`.

```
#include "sg/SGSchedulerInterfaceForComponents.h"
#include "sg/SGComponentRegistry.h"

sg::SchedulerInterfaceForComponents *scheduler =
  sg::obtainComponentInterfacePointer<sg::SchedulerInterfaceForComponents>
    (scx::scx_get_global_interface(), "scheduler");
```

7.5.2.2 Using the default scheduler mapping in the SystemC export use case

Call the global function `scx_initialize()` to initialize the simulation infrastructure.

```
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.

7.5.2.3 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()`.

7.5.2.3.1 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.

7.5.2.3.2 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_event`s. 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).

7.5.3 sg::SchedulerInterfaceForComponents class

This section describes the main scheduler interface class.

7.5.3.1 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.

```
// 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 double getNextSyncPoint();
    virtual void getNextSyncRange(sg::ticks_t &, sg::ticks_t &);
    virtual void getNextSyncRange(double&, double&);
    virtual void addCallback(sg::SchedulerCallback *) = 0;
    virtual void removeCallback(sg::SchedulerCallback *) = 0;
    virtual sg::ticks_t getCurrentSimulatedTime(sg::Tag<sg::ticks_t> *);
    virtual double getCurrentSimulatedTime();
    virtual double getSimulatedTimeResolution();
    virtual void setSimulatedTimeResolution(double resolution);
    virtual void stopRequest() = 0;
    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 information

Accessing the SchedulerInterfaceForComponents from within a modeling component on page 165
Providing a custom mapping of the scheduler functionality onto SystemC on page 167

7.5.3.2 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.

`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`.

7.5.3.3 sg::SchedulerInterfaceForComponents::addCallback

This method schedules a callback in the simulation thread. `AsyncSignal` uses it.

```
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 information

sg::SchedulerInterfaceForComponents::removeCallback on page 174

7.5.3.4 sg::SchedulerInterfaceForComponents::addSynchronisationPoint

This method adds synchronization points.

```
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.

7.5.3.5 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`.)

```
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 information

sg::SchedulerInterfaceForComponents::currentThread on page 172
sg::SchedulerRunnable - about on page 178
sg::SchedulerThread - about on page 182
sg::SchedulerThread::destructor on page 182
sg::SchedulerThread::start on page 183

7.5.3.6 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.

```
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.

7.5.3.7 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).

```
virtual Timer* createTimer(const char* name, TimerCallback* callback)=0;
```

7.5.3.8 sg::SchedulerInterfaceForComponents::currentThread

This method returns the currently running scheduler thread, which `createThread()` created, or null if not in any `threadProc()` call.

```
virtual SchedulerThread* currentThread();
```

Related information

sg::SchedulerInterfaceForComponents::createThread on page 171

7.5.3.9 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.

```
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.

7.5.3.10 sg::SchedulerInterfaceForComponents::getGlobalQuantum

This method returns the global quantum in `ticks` relative to simulated time resolution.

```
virtual ticks_t getGlobalQuantum(Tag<ticks_t>*);
```

Related information

sg::SchedulerInterfaceForComponents::setGlobalQuantum on page 174

7.5.3.11 sg::SchedulerInterfaceForComponents::getMinSyncLatency

This method returns the minimum synchronization latency in ticks relative to simulated time resolution.

```
virtual ticks_t getMinSyncLatency(Tag<ticks_t>*);
```

Related information

sg::SchedulerInterfaceForComponents::setMinSyncLatency on page 175

7.5.3.12 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.

```
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.

7.5.3.13 sg::SchedulerInterfaceForComponents::getSimulatedTimeResolution

This method returns the simulated time resolution in seconds.

```
virtual double getSimulatedTimeResolution();
```

7.5.3.14 sg::SchedulerInterfaceForComponents::removeCallback

This method removes all callbacks that are scheduled using `addCallback()` for this callback object. `AsyncSignal` uses it.

```
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 information

sg::SchedulerInterfaceForComponents::addCallback on page 170

7.5.3.15 sg::SchedulerInterfaceForComponents::setGlobalQuantum

This method sets the global quantum.

```
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 information

sg::SchedulerInterfaceForComponents::getGlobalQuantum on page 173

7.5.3.16 sg::SchedulerInterfaceForComponents::setMinSyncLatency

This method sets the minimum synchronization latency.

```
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 information

sg::SchedulerInterfaceForComponents::getMinSyncLatency on page 173

7.5.3.17 sg::SchedulerInterfaceForComponents::setSimulatedTimeResolution

This method sets the simulated time resolution in seconds.

```
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.

7.5.3.18 sg::SchedulerInterfaceForComponents::stopAcknowledge

This function blocks the simulation thread until being told to resume.

```
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.

7.5.3.19 sg::SchedulerInterfaceForComponents::stopRequest

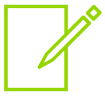
This function requests the simulation of the whole system to stop (pause).

```
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 `syncLevel`s 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 information

sg::SchedulerRunnable::stopRequest on page 181

7.5.3.20 sg::SchedulerInterfaceForComponents::wait(ThreadSignal)

This method waits on a thread signal.

```
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.

7.5.3.21 sg::SchedulerInterfaceForComponents::wait(ticks_t)

This method blocks the running thread and runs other threads for a specified time.

```
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".

7.5.4 sg::SchedulerRunnable class

This section describes the `SchedulerRunnable` class.

7.5.4.1 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 information
sg::SchedulerInterfaceForComponents::createThread on page 171

7.5.4.2 sg::SchedulerRunnable::breakQuantum

This function breaks the quantum. Arm deprecates this function.

7.5.4.3 sg::SchedulerRunnable::clearStopRequest

This function clears stop request flags.

```
void clearStopRequest();
```

Only `SchedulerInterfaceForComponents::stopAcknowledge()` calls this function, so calls are always from the simulation thread.

Related information
sg::SchedulerRunnable::stopRequest on page 181

7.5.4.4 sg::SchedulerRunnable::getName

This function returns the name of the instance that owns the object.

```
const char *getName() const;
```

By convention, this is the name that `createThread()` received. `SchedulerRunnable` inherits this function from `sg::SchedulerObject`.

7.5.4.5 sg::SchedulerRunnable::setThreadProperty, sg::SchedulerRunnable::getThreadProperty

These functions set and get thread properties.

```
bool setThreadProperty(ThreadProperty property, uint64_t value);
bool getThreadProperty(ThreadProperty property, uint64_t &valueOut);
```

Scheduler-configures-runnable properties

`TP_BREAK_QUANTUM`

Arm deprecates this property.
`SchedulerInterfaceForComponents::getNextSyncPoint()` gives the next quantum size.

`TP_DEFAULT_QUANTUM_SIZE`

Arm deprecates this property. Use `SchedulerInterfaceForComponents::set/getGlobalQuantum()`.

`TP_COMPILER_LATENCY`

`set`

Compiler latency, the maximum interval in which generated straight-line code checks for signals and the end of the quantum.

`get`

Compiler latency.

`default`

1024 instructions.

`TP_MIN_SYNC_LATENCY`

`set`

Synchronization latency, the minimum interval in which generated straight-line code inserts synchronization points.

`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 temporally 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 information

sg::SchedulerRunnable::breakQuantum on page 178

7.5.4.6 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.

```
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 information

sg::SchedulerRunnable::clearStopRequest on page 178

7.5.4.7 sg::SchedulerRunnable::threadProc

This is the main thread function, the thread entry point.

```
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.

7.5.5 sg::SchedulerThread class

This section describes the `SchedulerThread` class.

7.5.5.1 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 information

sg::SchedulerInterfaceForComponents::createThread on page 171

7.5.5.2 sg::SchedulerThread::destructor

This method destroys `SchedulerThread` objects.

```
~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 information

sg::SchedulerInterfaceForComponents::createThread on page 171

7.5.5.3 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`.

7.5.5.4 sg::SchedulerThread::setFrequency

This method sets the frequency source to be the parent clock for the thread. Arm deprecates this function.

7.5.5.5 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 information

sg::SchedulerInterfaceForComponents::createThread on page 171

7.5.6 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.

7.5.6.1 sg::ThreadSignal::destructor

This method destroys `ThreadSignal` objects, thread signals.

```
~ThreadSignal();
```

Destroying these objects while threads are waiting for them leaves the threads unscheduled.

7.5.6.2 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.

7.5.6.3 sg::ThreadSignal::getName

This method returns the name of the instance that owns the object.

```
const char *getName() const;
```

This is the name that `createThreadSignal()` received.

`ThreadSignal` inherits this method from `sg::SchedulerObject`.

7.5.7 sg::Timer class

This section describes the `Timer` interface class. The `SchedulerInterfaceForComponents::createTimer()` method creates `Timer` objects.

7.5.7.1 sg::Timer::cancel

This method unsets the timer so that it does not fire.

```
void cancel();
```

If the timer is not set, this method has no effect.

7.5.7.2 sg::Timer::destructor

This method destroys `Timer` objects.

```
~Timer();
```

The timer must not call `TimerCallback::timerCallback()` after the destruction of this object.

7.5.7.3 sg::Timer::getName

This method returns the name of the instance that owns the object.

```
const char *getName() const;
```

This is the name that `createTimer()` received.

`Timer` inherits this method from `sg::SchedulerObject`.

7.5.7.4 sg::Timer::isSet

This method returns `true` if the timer is set and queued for call-back, otherwise `false`.

```
bool isSet();
```

This method has no side effects.

7.5.7.5 sg::Timer::remaining

This method requests the remaining number of ticks relative to simulated time resolution until a timer makes a signal.

```
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.

7.5.7.6 sg::Timer::set

This method sets a timer to make a signal.

```
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 `set()` overrides the previous one.

This method returns `false` if `ticks` is too big to schedule the timer.

7.5.7.7 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 `SchedulerInterfaceForComponents::getSimulatedTimeResolution()` and `SchedulerInterfaceForComponents::setSimulatedTimeResolution()`.

7.5.8 sg::TimerCallback class

This section describes the `TimerCallback` base class. This interface does not allow object destruction.

7.5.8.1 sg::TimerCallback::getName

This method returns the name of the instance that owns the object.

```
const char *getName() const;
```

Conventionally, this is the name that `createTimer()` received.

`TimerCallback` inherits this method from `sg::SchedulerObject`.

7.5.8.2 sg::TimerCallback::timerCallback

The `createTimer()` method receives a `timerCallback` instance. This `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 `timerCallback()` again after *`t`* ticks.

```
ticks_t timerCallback();
```

7.5.9 sg::FrequencySource class

`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 `SchedulerInterfaceForComponents::getSimulatedTimeResolution()` and `SchedulerInterfaceForComponents::setSimulatedTimeResolution()`.

7.5.10 sg::FrequencyObserver class

`FrequencySource` instances notify `FrequencyObserver` instances of `FrequencySource` instance changes. This interface does not allow object destruction. Arm deprecates this class. Simulated time is relative to global time resolution. See `SchedulerInterfaceForComponents::getSimulatedTimeResolution()` and `SchedulerInterfaceForComponents::setSimulatedTimeResolution()`.

7.5.11 sg::SchedulerObject class

This section describes the `SchedulerObject` class. It is the base class for scheduler objects and interfaces. This interface does not allow object destruction.

7.5.11.1 sg::SchedulerObject::getName

This method returns the name of the instance that implements the object or interface. The intended use is debugging.

```
const char *getName() const;
```

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.

7.5.12 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.

```
sg::SchedulerInterfaceForComponents
 *scx_create_default_scheduler_mapping(scx_simcontrol_if *simcontrol);
```

`simcontrol`
a pointer to an existing simulation controller. If this is `NULL`, this function returns `NULL`.

7.5.13 sg::scx_get_curr_scheduler_mapping

This function returns a pointer to the scheduler mapping interface.

```
sg::SchedulerInterfaceForComponents *scx_get_curr_scheduler_mapping();
```

7.6 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.

7.6.1 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.

7.6.2 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()`.

7.6.3 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.

7.6.4 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.

8 Graphics Acceleration in Fast Models

Generic Graphics Acceleration (GGA) is a Fast Models framework for using host resources to perform graphics rendering on behalf of a GPU model. This chapter gives an introduction to GGA, describes how to enable it on a target platform model, and describes the main use cases.

8.1 Introduction to GGA

Fast Models provides various models of Mali™ GPUs, including Mali™-G51, Mali™-G72, and Mali™-G76.

You can run one of the OpenGL ES or Vulkan demo applications that are provided at Software Development Kits on a Fast Models platform that contains one of these Mali™ GPU models, a compatible Mali™ driver, and an Android or Linux distribution, but no pixels are rendered to the screen. This is because the GPU models are *register models*, which means they avoid the costly performance overhead of directly modeling the shader cores which perform the rendering operations in real hardware.

However, there are many use cases for Fast Models for which you might want to compare and validate actual rendering output from a simulated GPU. For example:

- Debugging a graphical application on a target platform.
- Validating graphical applications for target hardware in a continuous integration environment.
- Debugging or validating graphics driver integration.
- Booting Android OS version 8.0 or later on a target platform. These versions require hardware acceleration for graphics.

To enable these use cases, Fast Models provides a framework called Generic Graphics Acceleration (GGA) for using host resources to perform the rendering that is requested of the GPU model.

GGA enables Fast Models to:

1. Intercept graphics APIs within the model system
2. Mirror the graphics APIs on the host
3. Pass the results that are rendered by the host resources back into the modeled system

The GGA framework can also be configured to replace the Mali™ driver, by acting as a generic implementation of a graphics driver.

Related information

Media components

8.2 GGA modes

GGA can be used with or without a GPU Register Model (GRM).

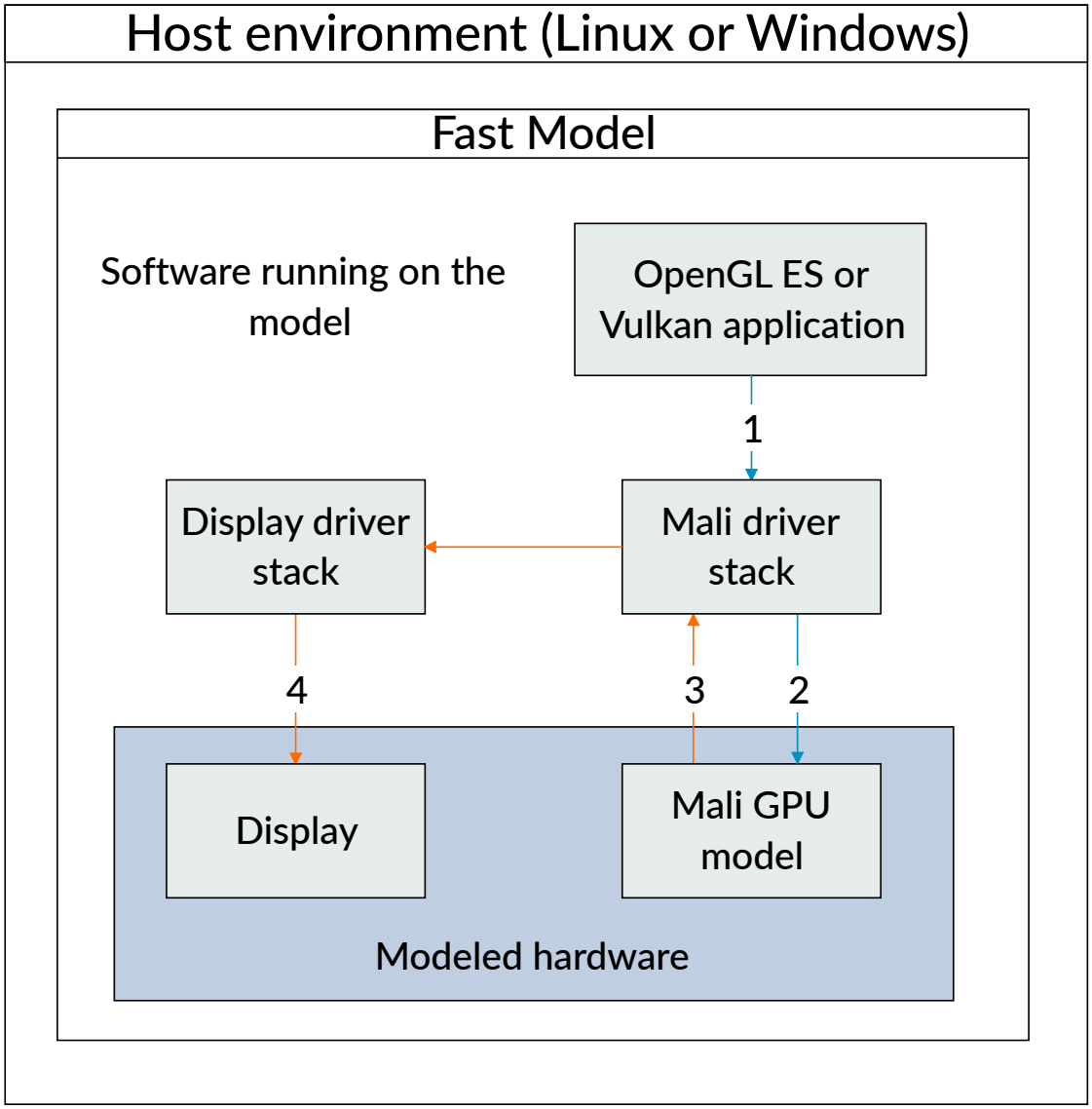
- If you use GGA with a GRM, this is referred to as GGA+GRM mode.
- If you use GGA without a GRM, this is referred to as GGA-only mode. In this mode, GGA acts as a generic graphics driver.

You can use also use a GRM without GGA, although in this mode, no pixels are displayed.

8.2.1 Using a GPU register model without GGA

The following figure shows a simplified view of a graphics and display driver stack running inside a Fast Model, without using GGA.

Figure 8-1: Simplified view of a graphics stack without GGA




Note

In this figure, blue arrows represent the data path of API calls, or data traveling to the GPU and orange arrows represent uninitialized data moving to the display.

The workflow shown in this figure is:

1. An application makes OpenGL ES or Vulkan API calls to a Mali™ driver.
2. The Mali™ driver stack issues rendering jobs to a GPU.
3. In hardware, the GPU would return the pixels, rendered through the shader cores, to the Mali™ driver. The model GPU returns uninitialized data.
4. The Mali™ driver, working together with the display driver stack, writes uninitialized data back to the display, when prompted by the application.

8.2.2 Using GGA with a GPU register model

The GGA framework consists of the following components:

Shim library
: A user space library within the model that intercepts graphics API calls before they reach the driver.

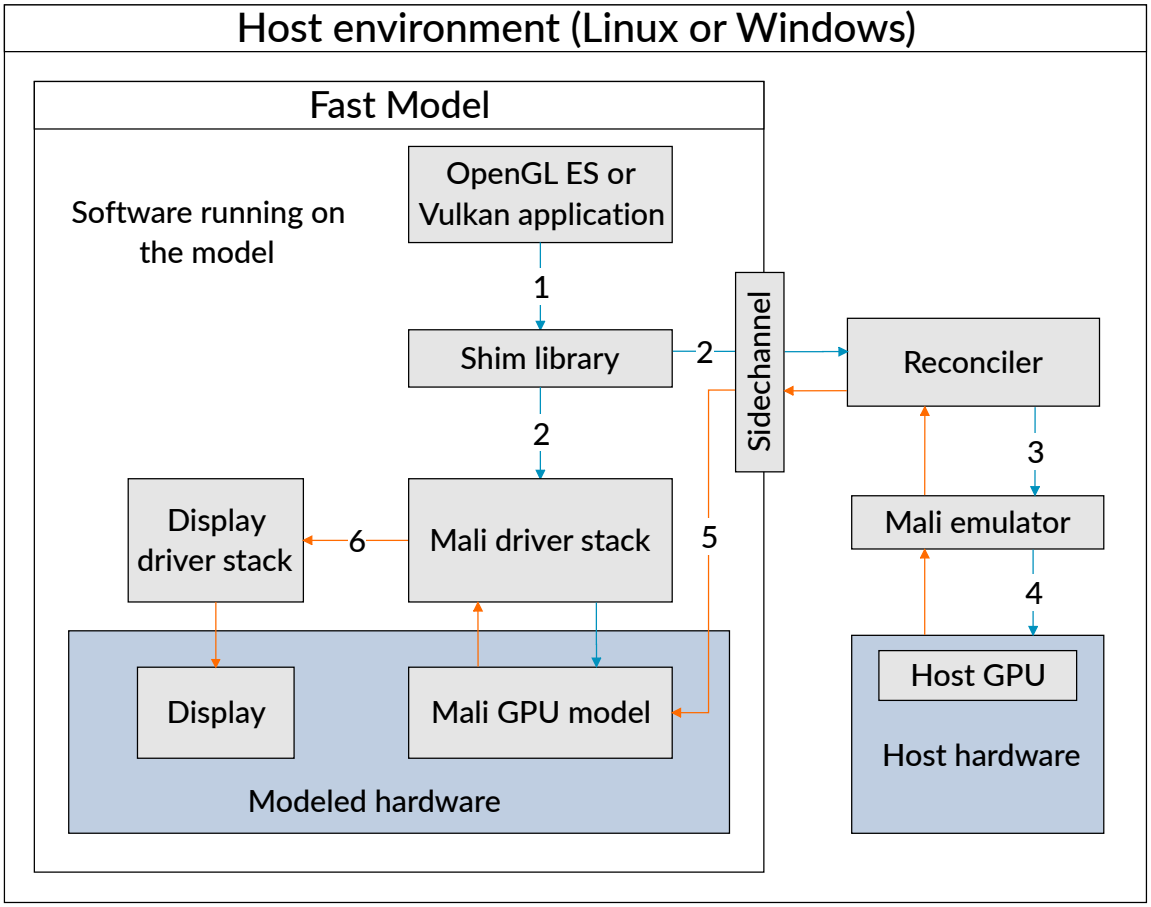
Reconciler
: A host library that receives the intercepted calls and forwards them to the graphics driver on the host.

Mali™ emulator
: A freely available Arm product that translates the OpenGL ES calls for the Mali™ driver into OpenGL calls for the host driver.

Sidechannel
: A Fast Models plug-in that enables communication between the Shim library and the Reconciler.

The following figure shows a Fast Models platform that contains a GPU register model, a graphics and display driver stack, and uses the GGA framework:

Figure 8-2: Using GGA with a GPU register model



Note

In this figure, blue arrows represent the data path of API calls, or data traveling to the GPU and orange arrows represent rendered pixels moving to the display.

The workflow shown in this figure is:

1. A graphical application calls the Shim library's implementation of the required OpenGL ES or Vulkan function.
2. The Shim library calls the Mali™ driver within the model, and also sends the call to the Reconciler on the host.
3. The Reconciler makes the OpenGL ES call on the Mali™ emulator.

4. The Mali™ emulator converts the OpenGL ES call to OpenGL and makes the OpenGL call on the host driver.

For Vulkan applications, the Mali™ emulator is not required. The Reconciler calls the host GPU directly because no translation is needed.

5. The rendered pixels are returned from the host GPU to the Reconciler, which inserts the pixels into the GPU model's memory.
6. When requested, the Mali™ driver passes the rendered pixels through the display stack.

8.2.3 Using GGA without a GPU register model

Some use cases for GGA do not require a GPU register model or simulation of the execution of Mali™ driver code.

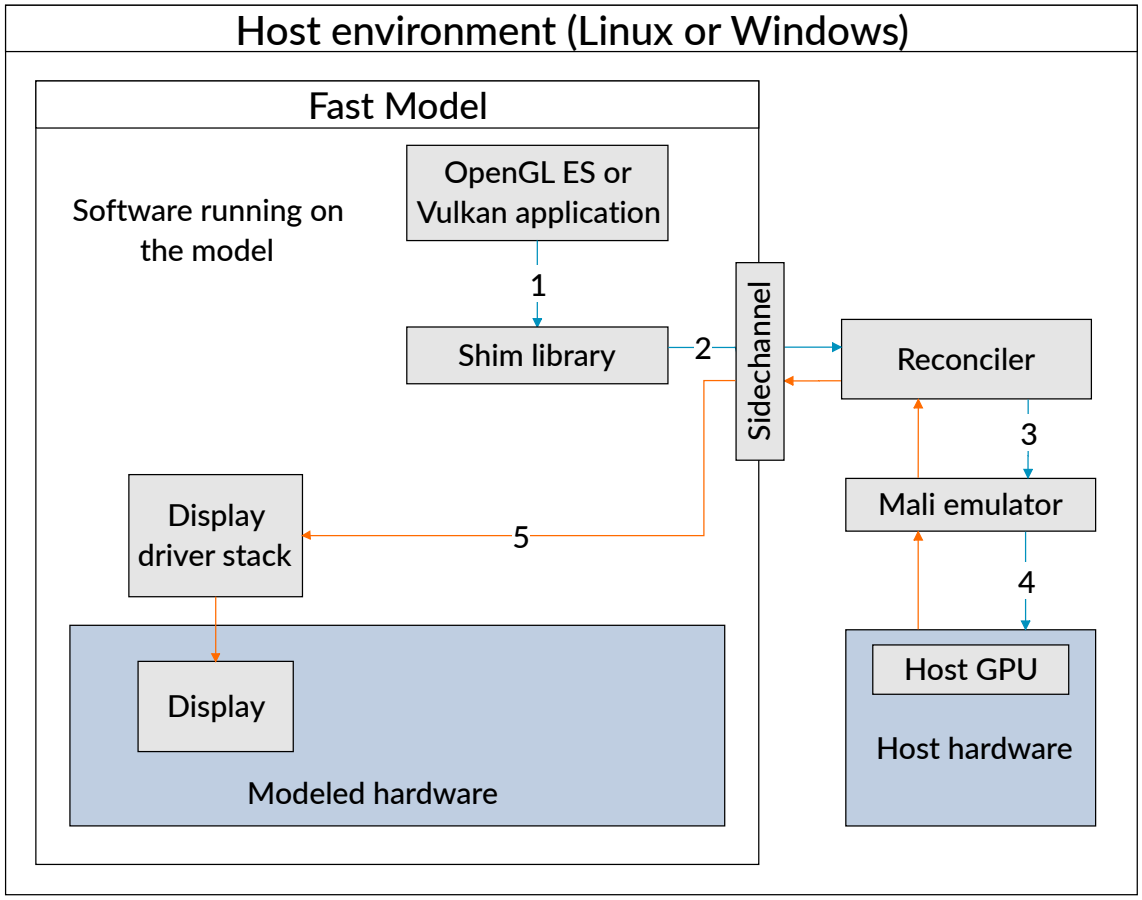
For example:

- To bring up Android v8.0 or a later distribution on a platform model. These versions require hardware acceleration.
- To simulate a system that is under development. You might not have access to the final Mali™ driver configuration and only need a basic OpenGL ES or Vulkan implementation to perform initial validation of applications.

These use cases only require a generic implementation of OpenGL ES or Vulkan, to satisfy target software dependencies. By omitting the GPU and driver, you can reduce the amount of simulated software in the model, and improve model performance.

You can configure the Shim library to pass graphics API calls directly to the Reconciler, bypassing the Mali™ driver and GPU model. The Reconciler then passes the rendering results directly to the display driver stack of the model, as shown here:

Figure 8-3: Using GGA without a GPU register model



Note

- In this figure, blue arrows represent the data path of API calls, or data traveling to the GPU and orange arrows represent rendered pixels moving to the display.
- For a description of the GGA framework components shown in this figure, see 8.2.2 Using GGA with a GPU register model on page 193

8.3 Prerequisites

GGA is available for both Windows and Linux hosts.

Host requirements:

- On a Windows host, GGA requires the Arm® Mali™ OpenGL ES Emulator version 3.0.5 or later. See 8.7.1 Install the Arm Mali OpenGL ES Emulator on page 203 for instructions.

The emulator is not required for Vulkan applications.

- On a Linux host, we recommend that you use the Mesa 20.3.4 or later OpenGL ES driver instead of the Mali™ Emulator. Using Mesa on Linux is necessary to run Android S or above. A reference Mesa build can be found in the Fast Models Third Party IP package. See 8.7.2 Install Mesa on page 204 for instructions.
- Nvidia host graphics implementations are supported, provided the driver implements OpenGL 4.3 or greater. GGA has been validated on NVIDIA GTX 1050 graphics cards, with driver versions 390.77 or later.

Target requirements:

- GGA supports the following APIs in both GGA-only and GGA+GRM modes:
  - EGL 1.4
  - OpenGL ES 2.0, 3.0, 3.1
  - Vulkan 1.0 (Android targets only)
- Target OS support is dependent on several factors, including the GPU model type and the operating system of the host machine.

Table 8-1: Target OS support for GGA

Target simulated OS	Supported GPU models	Host OS
Linux (Fbdev)	All including GGA-only[1]	Linux, Windows
Linux (Wayland)	All including GGA-only.[1]	Linux only
Android 8	All including GGA-only.[1]	Linux, Windows
Android 9	All including GGA-only.[1]	Linux, Windows
Android 10	All GPUs. Not supported in GGA-only mode.[1]	Linux, Windows
Android 11	Mali™-G710 and later.	Linux only
Android 12 (S)	Mali™-G710 and later.	Linux only

[1] GGA-only mode is a generic implementation that does not integrate with the Mali™ driver stack, so it does not require a Mali™ driver or a GPU model in the target platform. To simplify the table, it is listed with the GPU models.

- For details about the available GPU models, see Media components in the *Fast Models Reference Guide*. The Mali™ driver must have the following characteristics:
  - For Bifrost series GPUs, the driver version must be greater than r11p0. Additionally, the Mali™ driver must be built from the Mali™ Driver Development Kit with the **Mali Descriptor Tag** option enabled. You can enable this option in the following ways:
    - For scons-based builds, by adding `mali_descriptor_tag=1` to the build arguments.
    - For Blueprint-based builds, by setting `DESCRIPTOR_TAG=y` in the Blueprint build options.
  - For Valhal series GPUs, the driver version must be greater than r32p0.

Related information

Requirements for Fast Models on page 34

8.4 GGA contents

The components that you need to run Fast Models with host-accelerated graphics are installed in the `$PVLIB_HOME/GGA/` directory.

The contents of the `GGA` directory are shown here:

Figure 8-4: Locations of GGA components

```
$PVLIB_HOME/GGA/
  └── shim/
      └── <target_OS>/
          └── rel/
              └── libGLES.so
  └── reconciler/
      └── <host_OS>/
          └──<compiler>/
              └── rel/
                  ├── checkerrcode.ini
                  ├── libReconciler.so or Reconciler.dll
                  └── settings.ini
  └── examples/
      └── <target_OS>/
          └── Cube.apk
  └── HAL
      ├── readme.txt
      └──<gralloc-config>
          └── jni/
              ├── Android.mk
              ├── Application.mk
              └── src/
                  ├── shim_hal.cc
                  └── nw_hal.h
```

The following topics describe each of the subdirectories within the `GGA` directory.

8.4.1 Shim directory

The `shim` directory contains different versions of the Shim library, which intercepts graphics APIs for rendering on the host.

Each Shim library version is compiled for a specific target environment, for example:

android-armv7sfl
: For 32-bit Android targets with software-emulated floating point

android-armv8l_64
: For 64-bit Android targets

linux-armv7hf
: For Linux distributions with 32-bit hardware-enabled floating point

linux-armv8l_64

For Linux distributions with 64-bit Arm binaries


Note

- Shim libraries are not interchangeable between environments.
- The Shim library is named in the package as `libGLES.so`. Despite the name, the Android variants of the Shim library support Vulkan.

8.4.2 Reconciler directory

The `reconciler` directory contains the host library component of the GGA framework, which accepts incoming graphics APIs from the Shim library, and executes them on the host.

This directory also includes two example configuration files:

- `settings.ini`
- `checkerrcode.ini`

These files are used to set runtime configuration options, such as the GGA mode (GGA-only or GGA+GRM), and the verbosity level of log output.

Related information

Configuration on page 200

8.4.3 Examples directory

The `examples` directory contains a simple OpenGL ES spinning cube example. You can use it to verify the GGA framework installation on a model running Android.

Related information

Test the Android setup on page 208

8.4.4 HAL directory

The `HAL` directory contains example source files for building a `libnwhal.so` library.

On Android targets, `libnwhal.so` is required by the Shim library to translate from a particular version of the Android Gralloc module, available at Open Source Mali GPUs Android Gralloc Module on Arm Developer.

Related information

Generate libnwhal.so on page 205

8.5 Configuration

Use the configuration file `settings.ini` to choose between GGA+GRM mode or GGA-only mode, and to select which information is logged by GGA.

You must copy `settings.ini` from the following directory into the directory containing the model:

- On Linux, `$PVLIB_HOME/GGA/reconciler/linux-x86_64/gcc-x.x.x/rel/`, where *x.x.x* is the GCC version number.
- On Windows, `%PVLIB_HOME%\GGA\reconciler\win_32-x86_64\cl-19.xx.xxxxx\rel\`, where 19.xx.xxxxx is the MSVC compiler version number.

The configuration options are:

callOnTargetAPI

Specifies the mode in which GGA operates. The possible values are:

0 GGA-only mode. In other words, GGA acts as a generic OpenGL ES or Vulkan implementation.

2 GGA+GRM mode. In other words, GGA integrates with a Mali™ driver stack and Mali™ model in the target system.

LogLevel

Specifies the level of information to be logged to standard output by GGA. The possible values are:

0 or `LOG_LEVEL_OFF`

Disable logging.

1 or `LOG_LEVEL_FATAL`

Log the fatal issues from GGA. This is the default value.

2 or `LOG_LEVEL_ERROR`

Log the errors generated by GGA.

3 or `LOG_LEVEL_WARN`

Log the warnings generated by GGA.

6565 or `LOG_LEVEL_INFO`

Log the OpenGL ES API execution sequences.

6566 or `LOG_LEVEL_DEBUG`

Log the names of executed APIs and parameters.

6567 or `LOG_LEVEL_TRACE`

Log more detailed information generated by GGA.


Note

Each log level is a superset of all lower levels. For example, output for log level 6567 includes the output for all other levels.

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 only valid if `callOnTargetAPI` is set to 2. The possible values are:

0 Disable Error code check.
1 Enable Error code check.

enableErrorCheckWhiteList

Specifies whether the Error code check function should check errors from specific OpenGL ES APIs or from all of them. This option is only valid if `callOnTargetAPI` is set to 2 and `checkErrorCode` is set to 1. The possible values are:

0 Examine all APIs
1 Examine specific APIs

For more details about the Error code check function, see 8.8.2 Examine OpenGL ES execution in the graphics driver on page 211.

8.6 Feedback

To report issues or bugs in GGA, contact Arm Technical Support.

See https://developer.arm.com/support. Provide the following information for diagnostic purposes:

- The version of Fast Models
- The Fast Models virtual platform
- The host OS
- The OS of the target system, including the version number
- 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
- 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

8.7 Enabling GGA

This section describes how to enable GGA in your system. Follow these instructions for both GGA-only and GGA+GRM modes.

In summary, the goal of the GGA setup process is to:

- Ensure that client applications find the GGA implementation of `libGLES.so` before the Mali DDK implementation, if present
- Ensure that the GGA implementation can call the Mali DDK implementation after it has recorded the details of the API calls

- These instructions assume that you have a file system image of your target operating system.
- Before enabling GGA, ensure you can correctly boot the operating system on the Fast Models target:

Android

You can bring up Android using the Mali™ graphics driver together with the GPU models. See the Mali™ DDK documentation on how to install the driver in Android 8.

In this configuration, no graphics are visible. Follow the steps in this section to enable graphics rendering using GGA.

Linux

Linux can be brought up to command-line boot without the need of a graphics stack.

8.7.1 Install the Arm® Mali™ OpenGL ES Emulator

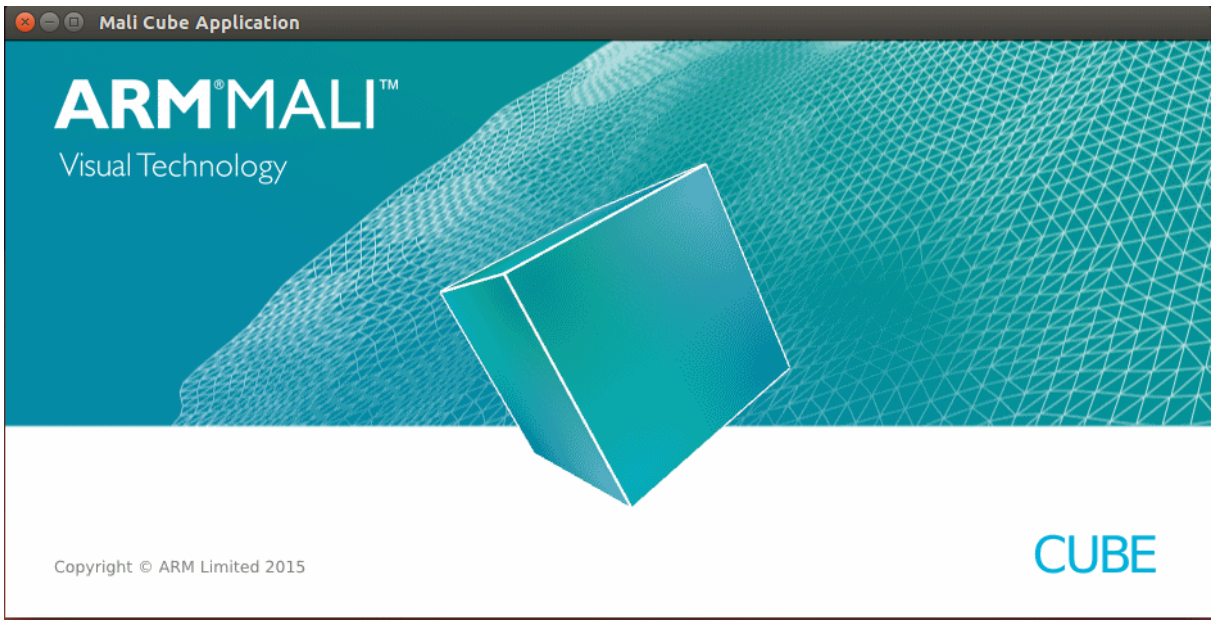
On Windows hosts, install the Mali™ OpenGL ES Emulator to translate OpenGL ES calls for the Mali™ driver into OpenGL calls for the host driver.

Procedure

1. Download the installation package for Windows from OpenGL ES Emulator.
2. Install and configure the emulator. For 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 user guide. If the installation is successful, you will see a spinning cube:

Figure 8-5: Mali™ Cube application



8.7.2 Install Mesa

Install Mesa3D on your Linux host to allow OpenGL calls in the model to be executed as OpenGL calls on the host. Mesa is only supported on Linux and is preferred to using the Arm® Mali™ OpenGL ES Emulator.

Procedure

1. Download and install the Fast Models Third-Party IP (TPIP) package from Product Download Hub.
2. To use the Mesa driver:
   a) Prepend the following directory to the `LD_LIBRARY_PATH` environment variable:
   ```
   <Install location>/FastModelsPortfolio_%(THIS-VERSION)/GGA/Mesa/
   Linux64_GCC-7.3/lib
   ```
   b) Set the following environment variable:
   ```
   LIBGL_DRIVERS_PATH=<Install location>/FastModelsPortfolio_%(THIS-VERSION)/GGA/
   Mesa/Linux64_GCC-7.3/lib/dri
   ```
3. To verify that Mesa was correctly installed, boot a GGA-enabled model. For instructions, see 8.7.7 Boot the model with the Android or Linux image on page 208. During initialization, the reconciler logs information about the identified OpenGL instance.

8.7.3 Preparing your image

Before enabling GGA in your target Android or Linux system, you must prepare your target file system image. For both Android and Linux, perform this step on the host machine.

Mount your Android or Linux file system on your host machine and make the necessary changes before booting the model. We recommend that you back up your file system before making any changes.

There are several ways to mount and modify your file system:

- On a Linux host, you can use the `mount` command as a root user
- On a Windows host, either:
  - Use one of the freely available utilities that are available for editing filesystem images
  - Edit the filesystem within a Linux virtual machine

8.7.4 Prepare an Android image

For Android, modify both the system and the vendor partitions.

- For details about Android partitions, see Partitions and Images in the official AOSP documentation.
- For information about the reasons behind the steps involved in the installation, see Implementing OpenGL ES and EGL in the official AOSP documentation.

Because the mounting points for system and vendor partitions can differ depending on your filesystem configuration and the Android version that you are running, the file paths provided in these instructions are relative to the mount point of the partition.

8.7.4.1 Mount and modify your system partition

The term *<system_mount_point>* in these instructions refers to the directory in which your system partition is mounted on the host, for example `/mnt/system/`.

Procedure

Append the following line to the end of the file *<system_mount_point>*`/build.prop`:

```
ro.hardware.egl=gga
```

This line specifies the string to be used by Android when identifying the OpenGL ES implementation. In this case, it will match the shim library.

8.7.4.2 Generate libnwhal.so

When running the shim on Android, an extra library is needed to understand the metadata that describes the windows created by Android. This library is referred to as the Native Window Hardware Abstraction Layer, or `libnwhal.so`.

Within Android, the Gralloc module is responsible for allocating and managing memory for the composition engine for graphics use. Vendors implementing the Android graphics stack write their own Gralloc module, often including their own metadata format that describes properties of the window. For Mali™ GPUs, the Android Gralloc is provided on developer.arm.com, and is matched with a particular version of the Mali™ DDK.

The purpose of `libnwhal.so` is to translate the version-specific metadata details for the shim, keeping the shim library independent of the Gralloc module used.

If you are using GGA-only mode, you can use the stock HALs that are shipped by Arm in:

- `$PVLIB_HOME/GGA/shim/android-armv8l_64/rel/stock/libnwhal.so`
- `$PVLIB_HOME/GGA/shim/android-armv7sfl/rel/stock/libnwhal.so`

These libraries work with the default Gralloc that is shipped in Android AOSP.

If you are using GGA+GRM mode, and integrating with the Mali™ driver, you must build your own `libnwhal.so`, built against the Mali™ DDK and AOSP source that you are using in your project. A `libnwhal.so` for your configuration can be autogenerated by following the instructions at `$PVLIB_HOME/GGA/HAL/bfst-custom/jni/README`.

8.7.4.3 Mount and modify your vendor partition

The term *<vendor_mount_point>* in these instructions refers to the directory in which your vendor partition is mounted on the host.

Procedure

1. Install the shim and HAL libraries to the 32-bit and 64-bit library locations:

```
cp $PVLIB_HOME/GGA/shim/android-armv7sfl/rel/libGLES.so <vendor_mount_point>/lib/egl/
libGLES_gga.so
cp $PVLIB_HOME/GGA/shim/android-armv8l_64/rel/libGLES.so <vendor_mount_point>/lib64/egl/
libGLES_gga.so
```

   If you are using GGA-only mode:

```
cp $PVLIB_HOME/GGA/shim/android-armv7sfl/rel/stock/libnwhal.so <vendor_mount_point>/lib/
cp $PVLIB_HOME/GGA/shim/android-armv8l_64/rel/stock/libnwhal.so <vendor_mount_point>/lib64/
```

   If you are using GGA+GRM mode, after building the custom HAL libraries, copy the built 32-bit and 64-bit libraries to the same locations specified for the stock HALs.

2. If you are using Vulkan:
   - Remove the files `vulkan.<ro.board.platform>.so` from the image. They are located in:
     - `<vendor_mount_point>/lib/hw/`
     - `<vendor_mount_point>/lib64/hw/`
   - Create two symbolic links that point to the Shim libraries:

```
cd <vendor_mount_point>/lib/hw/
ln -s ../egl/libGLES_gga.so vulkan.gga.so
cd <vendor_mount_point>/lib64/hw/
ln -s ../egl/libGLES_gga.so vulkan.gga.so
```

3. To test that GGA is enabled, copy the test application `$PVLIB_HOME/GGA/examples/linux-armv8l_64/Cube.apk` to your file system. You can install this application on the target after the model has booted. Note the directory where `Cube.apk` is copied to, because it will be needed after the model has booted.

8.7.5 Prepare a Linux image

Follow these steps to prepare a Linux file system image to use GGA.

Procedure

1. After mounting your Linux file system image, copy the Shim library `$PVLIB_HOME/GGA/shim/linux-armv8l_64/rel/libGLES.so` to `/home/<user>/libGLES.so`.
2. To enable GGA on your Linux target, ensure that all dynamic libraries for graphics APIs that are needed by a target application are symbolic links that point to the Shim library in `/home/<user>/libGLES.so`. Example commands:

```
ln -s libGLES.so libEGL.so
ln -s libGLES.so libEGL.so.1
ln -s libGLES.so libEGL.so.1.4.0
ln -s libGLES.so libGLESv1_CM.so
ln -s libGLES.so libGLESv1_CM.so.1
ln -s libGLES.so libGLESv1_CM.so.1.1.0
ln -s libGLES.so libGLESv2.so
ln -s libGLES.so libGLESv2.so.2
ln -s libGLES.so libGLESv2.so.2.1.0
ln -s libGLES.so libGLESv3.so
ln -s libGLES.so libGLESv3.so.3
ln -s libGLES.so libGLESv3.so.3.1.0
```

3. Before running a graphical application on the Linux target, add the directory that contains the Shim and the symlinks to the front of your `LD_LIBRARY_PATH` environment variable.

8.7.6 Choose the GGA mode

Use the `settings.ini` file to select the GGA mode.

Procedure

1. The `settings.ini` file can be found in `$PVLIB_HOME/GGA/reconciler/<OS>/<gcc-version>/rel/`. Copy it to the directory from which you will boot the model.

2. Before booting the model, select between GGA-only mode and GGA+GRM mode:
   - For GGA-only mode, set `callOnTargetAPI` to 0
   - For GGA+GRM mode, set `callOnTargetAPI` to 2

8.7.7 Boot the model with the Android or Linux image

Boot the target model, specifying some extra options to enable GGA.

Procedure

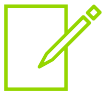
1. On a Windows host, before booting the model, add the Mali™ emulator to your host path.
2. In your boot command, specify the Reconciler as the interceptor, and load the Sidechannel as a plug-in:
   - To load the Sidechannel plug-in, add this option to the boot command:

     `--plugin $PVLIB_HOME/plugins/<compiler_version>/Sidechannel.so`
   - To load the interceptor, add this parameter to the boot command:

     `-C DEBUG.Sidechannel.interceptor=$PVLIB_HOME/GGA/reconciler/<OS>/<gcc-version>/libReconciler.so`

Results

One or two Fast Models CLCD displays appear on the screen, depending on the platform, along with one or two xterm consoles. The xterm console can be used to interact with the target OS.


Note

If SELinux is enabled on your Android target, you might observe in your xterm window that the system is stuck in a loop, repeatedly trying to restart several applications, including zygote, audioserver, and mediaserver. You can resolve this issue by switching to permissive mode to allow access to the Shim, in either of the following ways:

- Add the line `androidboot.selinux=permissive` to U-boot
- Press **ENTER** in the xterm window to check whether you have a command prompt. When you have a prompt, enter the following commands:

```
su root
setenforce 0
```

8.7.8 Test the Android setup

To test that GGA is enabled in your system, we provide an example graphical application, `Cube.apk`, that shows a spinning cube in a Fast Models window.

Procedure

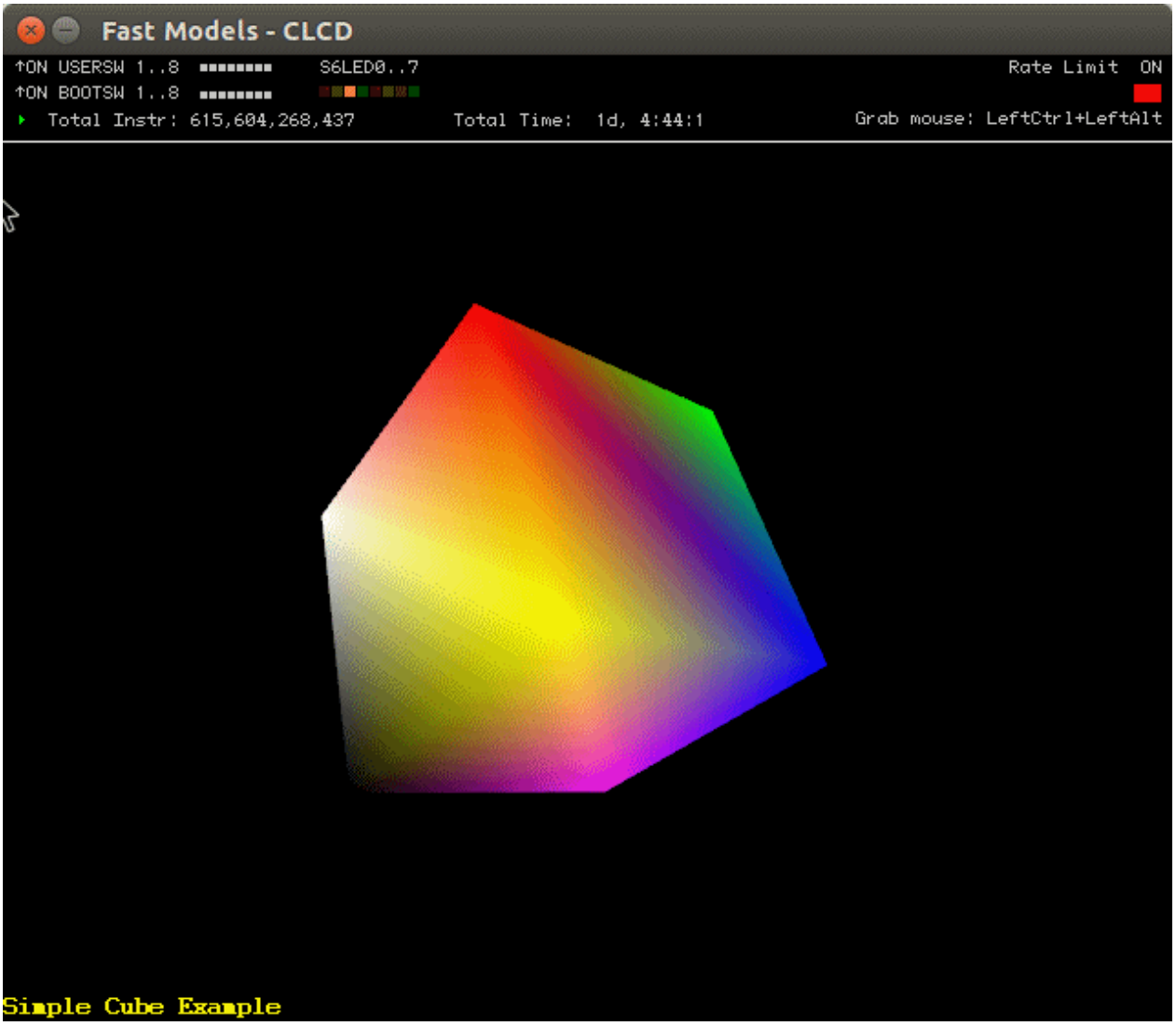
1. Install the example application in your target operating system, by running the following command in your xterm window:
   `pm install <Cube_install_dir>/Cube.apk`

   Cube_install_dir is the directory into which you previously copied the application.
2. Run the spinning cube application with the following command:
   ```
   am start -n com.arm.malideveloper.openglessdk.cube64/.Cube
   ```

Results

You should see a spinning cube in the CLCD window:

Figure 8-6: Spinning cube rendered using GGA



8.8 Using GGA

This section describes some useful features of GGA to help you view and debug the execution of graphics APIs in the model.

8.8.1 Log execution of graphics APIs

To log the execution of target graphics APIs, set the log level in the GGA configuration file, then reboot the target.

In `settings.ini`, 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 8.5 Configuration on page 200.


Note

If you find issues, try to reproduce them using a different platform model. Report bugs in GGA to the support team as described in 8.6 Feedback on page 202.

8.8.2 Examine OpenGL ES execution in the graphics driver

Use the Error code check function in GGA to report OpenGL ES APIs for which the host driver and the target driver return different error codes. Enable it using the `settings.ini` configuration file.

Procedure

1. The Error code check function works in GGA+GRM mode only, so `callOnTargetAPI` must be set to 2.
2. Assign a value to `LogLevel` other than `0` or `1`. For the allowed values, see 8.5 Configuration on page 200.
3. Set `checkErrorCode` to `1` to enable Error code check.
4. To examine all OpenGL ES APIs, set `enableErrorCheckWhiteList` to 0.
5. To only examine specific APIs:
   a) Set `enableErrorCheckWhiteList` to `1`.
   b) Set the APIs listed in `checkerrcode.ini` that you are interested in to `1`.

   
   Note

   `checkerrcode.ini` is located in the same directory as `settings.ini`.

6. Reboot the target to show the API execution in the driver.

Results

If abnormal APIs are detected, the host shows errors like this:

```
ERROR [RECONCILER] gles20_glCopyTexSubImage2D Inconsistent error code detected. host=0x0501,
 target=0x0502
```

For more details about this and other error messages, see 8.8.3 Error messages from Error code check on page 211.

8.8.3 Error messages from Error code check

The error messages show OpenGL ES APIs for which the host driver and the target driver return different error codes.

Errors can be generated by the target graphics driver, GGA, or the Mali™ OpenGL ES Emulator:

- Errors from the target graphics driver:

```
ERROR [RECONCILER] gles20_glCopyTexSubImage2D Inconsistent error code detected. host=0x0501,
 target=0x0502
```

  Here:

  - `gles20_glCopyTexSubImage2D` is the problematic API.
  - `0x0501` and `0x0502` are the error codes retrieved from the host driver and the target driver respectively. These error codes are defined in the OpenGL ES header file.

- Errors from GGA:

```
FATAL [RECONCILER] glProgramParameteri() Could not find program object descriptor for target-
side program id [0]
```

  Here, `glProgramParameteri()` is the problematic API. Report GGA bugs directly to Arm Technical Support. For more details, see 8.6 Feedback on page 202.

- 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.
```

  Report Mali™ emulator errors directly to Arm Technical Support.

8.8.4 Trace driver accesses to the GPU registers

Use the Trace and dump function provided by GGA to trace accesses by the graphics driver to the registers of the GPU register model.

Before you begin

- The Trace and dump function works in GGA+GRM mode only, so you must have integrated the graphics driver with the GPU register model in your target.
- Use the ListTraceSources plug-in to list the available trace sources and the GenericTrace plug-in to specify which events should be traced. They are located in `$PVLIB_HOME/plugins/<OS_compiler>/`.

Procedure

1. On the host, run the platform model with the ListTraceSources plug-in to list the trace sources that the model provides:

```
${PATH_Model} --plugin $PVLIB_HOME/plugins/Linux64_GCC-7.3/ListTraceSources.so
```

   The terminal shows:

   - The GPU model, for example:

     ```
     Component (292) providing trace: Kits3_Subsys.css.gpu
     ```

   - Trace sources provided by the GPU model, for example:

     INFO_ReadRegister
     Access time, addresses, data, and names of the registers that were read.

     INFO_Reset
     GPU reset data.

     INFO_WriteRegister
     Access time, addresses, and names of the registers that were updated, and the data before and after the update.

     INFO_IrqGpuControl
     ID, name, and state of the IRQ signal from the GPU. The state can be `Y` for Set, or `N` for Clear.

     INFO_IrqJobControl
     ID, name, and 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
     ID, name, and state of the IRQ signal from the MMU on the GPU. The state can be `Y` for Set, or `N` for Clear.

     WARN_ReadToWriteOnlyRegister
     Warning messages and addresses for the write-only registers that have been read by the graphics driver.

     WARN_WriteToReadOnlyRegister
     Warning messages and addresses for the read-only registers that have been written by the graphics driver.

     WARN_AccessToUnimplementedRegister
     Warning messages and addresses for the invalid registers that have been accessed by the graphics driver.

2. Boot the Android target with the following additional options to trace all events from the GPU model:

```
--plugin $PVLIB_HOME/plugins/Linux64_GCC-7.3/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 trace sources, add the suffix '`*`' to this GPU. For instance, `Kits3_Subsys.css.gpu.*`.
  - To output one GPU trace source only, add it as a suffix to the GPU. For instance, `Kits3_Subsys.css.gpu.INFO_ReadRegister`.
  - To output multiple trace sources, use a comma-separated list. For instance, `Kits3_Subsys.css.gpu.INFO_ReadRegister, Kits3_Subsys.css.gpu.INFO_WriteRegister`
- The `trace-file` option specifies the log file in which to save the trace output. If the `trace-file` option is not used, the trace results are shown on the host terminal.

For more details, see GenericTrace in *Fast Models Reference Guide*.

Results

The host terminal or the log file shows details about the driver-accessed registers, such as the register addresses, data, and the access time. For example:

```
HOST_TIME=1557460.545195s INFO_ReadRegister: REG_OFFSET=0x0000000000000000 VALUE=0x60000000
 REG_NAME="GPU_ID"
HOST_TIME=1557460.545266s INFO_ReadRegister: REG_OFFSET=0x0000000000000004 VALUE=0x07130206
 REG_NAME="L2_FEATURES"
HOST_TIME=1557460.545279s INFO_ReadRegister: REG_OFFSET=0x0000000000000008 VALUE=0x00000000
 REG_NAME="SUSPEND_SIZE"
HOST_TIME=1557460.545291s INFO_ReadRegister: REG_OFFSET=0x000000000000000c VALUE=0x00000809
 REG_NAME="TILER_FEATURES"
HOST_TIME=1557460.545303s INFO_ReadRegister: REG_OFFSET=0x0000000000000010 VALUE=0x00000001
 REG_NAME="MEM_FEATURES"
HOST_TIME=1557460.545316s INFO_ReadRegister: REG_OFFSET=0x0000000000000014 VALUE=0x00002830
 REG_NAME="MMU_FEATURES"
HOST_TIME=1557460.545325s INFO_ReadRegister: REG_OFFSET=0x0000000000000018 VALUE=0x000000ff
 REG_NAME="AS_PRESENT"
HOST_TIME=1557460.545334s INFO_ReadRegister: REG_OFFSET=0x000000000000001c VALUE=0x00000007
 REG_NAME="JS_PRESENT"
HOST_TIME=1557460.545345s INFO_ReadRegister: REG_OFFSET=0x00000000000000c0 VALUE=0x0000020e
 REG_NAME="JS0_FEATURES"
HOST_TIME=1557460.545362s INFO_ReadRegister: REG_OFFSET=0x00000000000000c4 VALUE=0x000001fe
 REG_NAME="JS1_FEATURES"
HOST_TIME=1557460.545364s INFO_ReadRegister: REG_OFFSET=0x00000000000000c8 VALUE=0x0000007e
 REG_NAME="JS2_FEATURES"
HOST_TIME=1515565849.690948s gpu.INFO_WriteRegister: REG_OFFSET=0x0000000000001870
 VALUE=0x00000000 UPDATED_VALUE=0x00000000 REG_NAME="JOB_SLOT0_JS_FLUSH_ID_NEXT"
HOST_TIME=1515565849.691304s gpu.INFO_WriteRegister: REG_OFFSET=0x0000000000001860
 VALUE=0x00000000 UPDATED_VALUE=0x00000001 REG_NAME="JOB_SLOT0_JS_COMMAND_NEXT"
HOST_TIME=1515565849.691322s gpu.INFO_IrqJobControl: IRQ_ID=0x01 IRQ_NAME="JOB Control"
 IRQ_STATE=Y
HOST_TIME=1515565849.691561s gpu.INFO_ReadRegister:  REG_OFFSET=0x000000000000100c
 VALUE=0x00000001 REG_NAME="JOB_IRQ_STATUS"
HOST_TIME=1515565849.691643s gpu.INFO_WriteRegister: REG_OFFSET=0x0000000000001004
 VALUE=0x00000000 UPDATED_VALUE=0x00000001 REG_NAME="JOB_IRQ_CLEAR"
HOST_TIME=1515565849.691647s gpu.INFO_IrqJobControl: IRQ_ID=0x01 IRQ_NAME="JOB Control"
 IRQ_STATE=N
```

9 Timing Annotation

This chapter describes timing annotation, which 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-based 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 Guide*
- Instruction and data prefetching.
- Cache and TLB latency.
- Latency caused by pipeline stalls. For details, see PipelineModel in the *Fast Models Reference Guide*.

Timing annotation is supported on all SystemC-based platforms. However, it is disabled by default on ISIMs. To enable timing annotation for an ISIM, set the environment variable `FASTSIM_DISABLE_TA` to 0.

9.1 Enabling and disabling timing annotation

Use the environment variable `FASTSIM_DISABLE_TA` to enable or disable timing annotation latency.

By default, timing annotation is disabled for ISIMs. To enable it, set `FASTSIM_DISABLE_TA` to 0. If it is disabled 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 one cycle per instruction.

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. However, the Fast Models simulation engine ignores any pipeline stall latencies or branch misprediction penalties that they calculate.

To print timing statistics on simulation exit, run the simulation with the `--stat` parameter.

9.2 CPI files

Cycles Per Instruction (CPI) files define classes of instructions and assign CPI values to them. CPI files give a more accurate estimate of the number of cycles required to run a program on the model.

Arm does not provide CPI files, only some pre-defined CPI instruction classes which can help you to create your own CPI files. To create a CPI file for a specific CPU:

1. Create a set of mappings between the instruction encodings for the instruction set and a set of instruction classes or groups of classes. Arm provides pre-defined instruction classes and groups for the A32, T32, and A64 instruction sets in `$PVLIB_HOME/etc/CPIPredefines/`. You can include these pre-defined instruction classes in your CPI files, or you can define your own classes.
2. Create a file to map these instruction classes to CPI values. This is the CPI file. Calculate the CPI values to use based on observations from a cycle accurate model, or see the Arm® Software Optimization Guides, which are available on Arm Developer.

- An alternative to using CPI files is to use the `cpi_mul` and `cpi_div` parameters on a core in the model. These parameters are integers that represent a CPI multiplication or division factor for all instructions. They can also be used together to represent non-integer values. For example, use `cpi_mul` = 5, `cpi_div` = 4 for a CPI of 1.25.
- To calculate values for `cpi_mul` and `cpi_div`, experiment with running a workload on a cycle accurate simulation to choose values that give the most accurate results.
- 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`. It can also support multiple processor types, including pre-defined and user-defined types.

Specify a CPI file when launching a platform model by using the `--cpi-file` command-line parameter, for example:

```
./EVS_Base_Cortex-A73x1 … --cpi-file /CPI_file.txt --stat
```


Note

The `--stat` parameter displays timing statistics on simulation exit.

Alternatively, specify a CPI file in your SystemC code by calling the function 7.4.13 scx::scx_set_cpi_file on page 137.

CPIValidator is a command-line tool provided in `$MAXCORE_HOME/bin/` to help you create valid CPI files. Use the `--help` 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`:

```
$MAXCORE_HOME/bin/CPIValidator --input-file ./CPI_file.txt --output-file ./CPIEvaluationTree.txt
```

Related information

CPI file syntax on page 217
BNF specification for CPI files on page 222

9.3 CPI file syntax

CPI files are plain text files that contain a series of statements, one per line. Lines that begin with a `#` character are ignored.

In the following syntax definitions, square brackets `[]` enclose optional attributes. An ellipsis … indicates attributes that can be repeated.

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-A5MP`
- `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 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:

```
DefineGroup Divide_instructions Classes=SDIV,UDIV CpuType=ARM_Cortex-A73
 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:

```
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`:

```
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
```

DefineCpuType

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 FVP_Base_AEMvA_AEMvA.cluster0.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_Mnemonics.txt
```

9.4 BNF specification for CPI files

CPI files have the following BNF specification:

```
                <CPIFile> ::= <Statements>
             <Statements> ::= <Statement> <Statements>
                            | <Statement>
              <Statement> ::= <Comment>
                            | <DefineCpiStatement>
                            | <DefaultsStatement>
                            | <DefineCpuTypeStatement>
                            | <MapCpuStatement>
                            | <DefineClassStatement>
                            | <DefineGroupStatement>
                            | <IncludeStatement>
                            | <DefineMixInStatement>
```

```
       <DefineCpiStatement ::= "DefineCpi" <InstructionClassOrGroup>
 <DefineCpiAttributes> <EOL>
      <DefaultsStatement> ::= "Defaults" <DefineCpiAttributes> <EOL>
   <DefineCpuTypeStatement ::= "DefineCpuType" <UserCpuType>
 <DefineCpuTypeAttributes> <EOL>
         <MapCpuStatement ::= "MapCpu" <CpuInstance> <MapCpuAttributes> <EOL>
    <DefineClassStatement ::= "DefineClass" <InstructionClass>
 <DefineClassAttributes> <EOL>
    <DefineGroupStatement ::= "DefineGroup" <InstructionGroup>
 <DefineGroupAttributes> <EOL>
       <IncludeStatement> ::= "Include" <IncludeAttributes> <EOL>
   <DefineMixInStatement> ::= "DefineMixIn" <MixInType> <DefineMixInAttributes>
 <EOL>
    <DefineCpiAttributes> ::= <DefineCpiAttribute> <DefineCpiAttributes>
                            | <DefineCpiAttribute>
     <DefineCpiAttribute> ::= <ISetAttribute>      { Mandatory }
                            | <CpuTypeAttribute>   { Optional }
                            | <CpiAttribute>       { Mandatory }
          <ISetAttribute> ::= "ISet" "=" <ISetOrStar>
             <ISetOrStar> ::= <ISet> | "*"
                   <ISet> ::= "A32" | "A64" | "Thumb" | "T2EE"
       <CpuTypeAttribute> ::= "CpuType" "=" <CpuType>
               <CpuType> ::= "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_CortexR5"
                            | "ARM_Cortex-A9MP" | "ARM_Cortex-A9UP"
                            | "ARM_Cortex-A8" | "ARM_Cortex-R4"
                            | "ARM_Cortex-M3" | "ARM_Cortex-M0+"
                            | "ARM_Cortex-M0" | <UserCpuType> | "*"
          <CpiAttribute> ::= "Cpi" "=" <Cpi>
 <DefineCpuTypeAttributes> ::= <ISetsAttribute>
         <ISetsAttribute> ::= "ISets" "=" <ISetsOrStar>
            <ISetsOrStar> ::= <ISets> | "*"
                  <ISets> ::= <ISet> "," <ISets> | <ISet>
       <MapCpuAttributes> ::= <ToCpuTypeAttribute>
     <ToCpuTypeAttribute> ::= "ToCpuType" "=" <CpuType>
   <DefineClassAttributes> ::= <DefineClassAttribute> <DefineClassAttributes>
                            | <DefineClassAttribute>
    <DefineClassAttribute> ::= <MaskAttribute>              { Mandatory }
                            | <ValueAttribute>              { Mandatory }
                            | <ProhibitedPairsAttribute>    { Optional }
                            | <ISetAttribute>               { Mandatory }
                            | <CpuTypeAttribute>            { Optional }
          <MaskAttribute> ::= "Mask" "=" <Mask>
         <ValueAttribute> ::= "Value" "=" <Value>
<ProhibitedPairsAttribute> ::= <ProhibitedPairAttribute> <ProhibitedPairsAttribute>
                            | <ProhibitedPairAttribute>
  <ProhibitedPairAttribute> ::= <ProhibitedMaskAttribute> <ProhibitedValueAttribute>
 <ProhibitedMaskAttribute> ::= "ProhibitedMask" "=" <Mask>
<ProhibitedValueAttribute> ::= "ProhibitedValue" "=" <Value>
   <DefineGroupAttributes> ::= <DefineGroupAttribute> <DefineGroupAttributes>
                            | <DefineGroupAttribute>
    <DefineGroupAttribute> ::= <ClassesAttribute>           { Mandatory }
                            | <ISetAttribute>               { Mandatory }
                            | <CpuTypeAttribute>            { Optional }
                            | <MixAttribute>                { Optional }
       <ClassesAttribute> ::= "Classes" "=" <InstructionClassOrGroups>
        <MixAttribute>    ::= "Mix" "=" <MixInTypes>
<InstructionClassOrGroups> ::= <InstructionClassOrGroup> ","
 <InstructionClassOrGroups>
     <instructionClasses> ::= <InstructionClass>
 <InstructionClassOrGroup> ::= <InstructionClass>
                            | <InstructionGroup>
             <MixInTypes> ::= <MixInType> "," <MixInTypes>
              <MixInType> ::= <Symbol>
      <IncludeAttributes> ::= <FilePathAttribute>
      <FilePathAttribute> ::= "FilePath" "=" <FilePath>
   <DefineMixInAttributes> ::= <DefineMixInAttribute> <DefineClassAttributes>
```

```
    <DefineMixInAttribute> ::= <MaskAttribute>
                             | <ValueAttribute>
                             | <SuffixAttribute>
         <SuffixAttribute> ::= "Suffix" "=" <String>
                <FilePath> ::= <String>
           <InstructionClass> ::= <Symbol>
           <InstructionGroup> ::= <Symbol>
                <UserCpuType> ::= <Symbol>
                <CpuInstance> ::= <QuotedString>       { Supports use of wild cards }
                        <Cpi> ::= <Double>
                       <Mask> ::= <UnsignedInteger>
                      <Value> ::= <UnsignedInteger>
```

9.5 Instruction and data prefetching

Arm® Cortex®-A series processors implement prefetching instructions and data into caches to improve the cache hit rate and improve performance. Fast Models supports prefetching instructions and data independently, by using model parameters.

9.5.1 Configuring instruction prefetching

Configure instruction cache prefetching by using the following cluster-level parameters.

`icache-prefetch_enabled`

`true` to enable simulation of instruction cache prefetching, `false` otherwise. Defaults to `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 `true`, the following extra parameters are available:

`icache-prefetch_level`

Specifies the zero-indexed cache level into which instructions are prefetched. Defaults to 0, which means L1.

`icache-nprefetch`

Specifies the number of additional, sequential instruction cache lines to prefetch. Defaults to 1.


Note

These parameters only have an effect when cache state modeling is enabled, which is controlled by the model parameter `icache-state_modelled` or `cache_state_modelled`.

Example

The following command line enables instruction cache prefetching and prints `WAYPOINT` trace events to the console. A `WAYPOINT` is a point at which instruction execution by the processor might change the program flow.

```
./FVP_Base_AEMvA …
```

```
-C cache_state_modelled=1 \
-C cluster0.icache-prefetch_enabled=1 \
--plugin $PVLIB_HOME/plugins/Linux64_GCC-7.3/GenericTrace.so \
-C TRACE.GenericTrace.trace-sources=WAYPOINT
```

Related information

Loading a plug-in

9.5.2 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 prefetcher is a basic stride-detecting prefetcher, but relatively configurable using the following parameters:

Table 9-1: Parameters for the example prefetcher

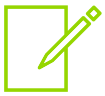
Parameter	Description
`history_length`	Length of history to maintain.
`history_threshold`	Number of misses to allow in history before issuing a prefetch.
`lfb_entries`	Number of access streams to track.
`mbs_expire`	Number of non-hitting loads to allow before the prefetcher stops tracking a potential access stream.
`pf_count`	Number of prefetch streams available.
`pf_tracker_count`	Number of prefetches tracked.
`pf_initial_number`	Initial number of prefetches to issue for a new stream.

Parameter	Description
`prefetch_all_levels`	Prefetch to all cache levels rather than just the lowest level.

An *access stream* is created whenever a load is made to an address that 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 `pf_initial_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 `history_threshold` cache misses among the last `history_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 `pf_tracker_count`, and if the prefetcher 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.

9.6 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 only take effect 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.

9.7 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.

9.7.1 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.

9.7.1.1 Prerequisites

To use timing annotation, you require the following:

- A SystemC virtual platform.
- An application that enables caches.
- A way of calculating the execution of 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.

9.7.1.2 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.

About this task

The platform is not provided pre-built in the Fast Models Portfolio installation, so you must first build it, for example:

```
cd $PVLIB_HOME/examples/SystemCExport/EVS_Platforms/EVS_Base/Build_Cortex-A73x1/
make rel_gcc64_64
```

9.7.1.3 Calculating the execution time of an instruction

The `INST` MTI trace source displays every instruction that is executed while running a program. When timing annotation is enabled, it also displays the current simulation time after an instruction has completed executing.

The number of ticks an instruction takes to execute is 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, it uses the `GenericTrace` plug-in. This plug-in 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:

```
--plugin=$PVLIB_HOME/plugins/Linux64_GCC-7.3/GenericTrace.so \
-C TRACE.GenericTrace.trace-sources=INST \
-C TRACE.GenericTrace.trace-file=/path/to/trace/file.txt
```

If timing annotation is enabled, 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=0x0000000080000000 NSDESC=0x00 PADDR2=0x0000000080000000 NSDESC2=0x00 NS=0x00
ITSTATE=0x00 INST_COUNT=0x0000000000000001 LOCAL_TIME=0x0000000000001388
CURRENT_TIME=0x0000000000001388 CORE_NUM=0x00 DISASS="LDR      x1,{pc}+0x248 ;
0x80000248"

INST: PC=0x0000000080000004 OPCODE=0xd518c001 SIZE=0x04 MODE=EL3h ISET=AArch64
PADDR=0x0000000080000004 NSDESC=0x00 PADDR2=0x0000000080000004 NSDESC2=0x00 NS=0x00
ITSTATE=0x00 INST_COUNT=0x0000000000000002 LOCAL_TIME=0x0000000000003a98
CURRENT_TIME=0x0000000000003a98 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 information

GenericTrace
Enabling and disabling timing annotation on page 215
MTI trace sources on page 239

9.7.1.4 Displaying the total execution time of the simulation

You can use MTI trace to calculate the execution time of individual instructions, but to determine the overall simulation time, use the command-line option `--stat` instead.

This option causes the model to print performance statistics 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                       : 0.00
Base.cluster0.cpu0                      : 0.42 MIPS (172289 Inst)
-------------------------------------------------------------------------------
```


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 in different timing annotation configurations.

9.7.1.5 Calculating the average CPI value

Calculate the average CPI value for the simulation by using the instruction count and the simulated time value, as displayed by the `--stat` option.

Use the following formula:

```
average_cpi = simulated_time_in_picoseconds / (10000 * instruction_count)
```

This example calculates an average CPI value of 0.69999:

```
average_cpi = (0.001206 * 10^12) / (10000 * 172289) = 0.69999
```

9.7.2 Modeling Cycles Per Instruction (CPI)

This section demonstrates how to precisely model the simulated time per instruction by using the CPI timing annotation feature.

9.7.2.1 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, 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 9.7.1.5 Calculating the average CPI value on page 229.

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.

9.7.2.2 Specifying a fixed CPI value

Specify a fixed CPI value by using the per-cluster model parameters `cpi_mul` and `cpi_div`.

These parameters are integers that represent a CPI multiplication or division factor that is applied to all instructions during execution within that cluster. They can be used together to represent non-integer values. For example, use `cpi_mul` = 5, `cpi_div` = 4 for a CPI of 1.25. 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. The fixed CPI value is used in a way that *core_clock_period * fixed_cpi_value* is rounded to the nearest picosecond.

Related information

Running the example with a fixed CPI value on page 236

9.7.2.3 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 for demonstration purposes only. They are arbitrary and are not representative of any Arm® processor.

```
# -------------------
# Instruction classes
# -------------------
## PC-relative addressing
DefineClass ADRP                    Mask=0x9F000000 Value=0x90000000 ISet=A64
DefineClass ADR                     Mask=0x9F000000 Value=0x10000000 ISet=A64
## Arithmetic
DefineClass ADD_ext_reg             Mask=0x7FE00000 Value=0x0B200000 ISet=A64
DefineClass ADD_sft_reg             Mask=0x7F200000 Value=0x0B000000 ISet=A64
DefineClass ADD_imm                 Mask=0x7F000000 Value=0x11000000 ISet=A64
DefineClass CMP_ext_reg             Mask=0x7FE0001F Value=0x6B20001F ISet=A64
DefineClass CMP_sft_reg             Mask=0x7F20001F Value=0x6B00001F ISet=A64
DefineClass CMP_imm                 Mask=0x7F00001F Value=0x7100001F ISet=A64
## Logical
DefineClass ORR_sft_reg             Mask=0x7F200000 Value=0x2A000000 ISet=A64
DefineClass ORR_imm                 Mask=0x7F800000 Value=0x32000000 ISet=A64
## Branches, exception generating and system instructions
DefineClass B_gen_except_sys        Mask=0x1C000000 Value=0x14000000 ISet=A64
## Load register pair
DefineClass LDP_post_idx            Mask=0x7FC00000 Value=0x28C00000 ISet=A64
DefineClass LDP_pre_idx             Mask=0x7FC00000 Value=0x29C00000 ISet=A64
DefineClass LDP_sgn_off             Mask=0x7FC00000 Value=0x29400000 ISet=A64
## Store register
DefineClass STR_reg                 Mask=0xBFE00C00 Value=0xB8200000 ISet=A64
DefineClass STR_imm_post_idx        Mask=0xBFE00C00 Value=0xB8000400 ISet=A64
DefineClass STR_imm_pre_idx         Mask=0xBFE00C00 Value=0xB8000C00 ISet=A64
DefineClass STR_imm_usg_off         Mask=0xBFC00000 Value=0xB9000000 ISet=A64
# -------------------
# Instruction groups
# -------------------
DefineGroup PC_rel_addr_instr       Classes=ADRP,ADR                                ISet=A64
DefineGroup ADD_instr               Classes=ADD_ext_reg,ADD_sft_reg,ADD_imm         ISet=A64
DefineGroup CMP_instr               Classes=CMP_ext_reg,CMP_sft_reg,CMP_imm         ISet=A64
DefineGroup ORR_instr               Classes=ORR_sft_reg,ORR_imm                     ISet=A64
DefineGroup B_gen_except_sys_instr Classes=B_gen_except_sys                         ISet=A64
DefineGroup LDP_instr               Classes=LDP_post_idx,LDP_pre_idx,LDP_sgn_off ISet=A64
DefineGroup STR_instr
 Classes=STR_reg,STR_imm_post_idx,STR_imm_pre_idx,STR_imm_usg_off ISet=A64
# ----------
# CPI values
# ----------
DefineCpi   PC_rel_addr_instr       ISet=A64 Cpi=0.25
DefineCpi   ADD_instr               ISet=A64 Cpi=0.50
DefineCpi   CMP_instr               ISet=A64 Cpi=0.75
DefineCpi   ORR_instr               ISet=A64 Cpi=0.50
DefineCpi   B_gen_except_sys_instr ISet=A64 Cpi=1.00
DefineCpi   LDP_instr               ISet=A64 Cpi=2.00
DefineCpi   STR_instr               ISet=A64 Cpi=1.00
# --------
# Defaults
```

```
# --------
Defaults ISet=* Cpi=0.75
```

Related information

CPI file syntax on page 217

9.7.2.4 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.

Results

The encodings for each instruction in the A64 instruction set are provided by the Arm®v8-A Architecture Reference Manual, chapter 4. It also describes groups of instructions that share encodings. 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, chapter 4 of the Arm®v8-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 `0b11111`. As shown in the reference manual, a value of `0b10000` for bits [28:24] identifies the instruction as being `ADR` or `ADRP`. Therefore, both `Value` fields set bits [28:24] to `0b10000`. 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 information

CPI file syntax on page 217
Armv8-A Architecture Reference Manual

9.7.2.5 Validating a CPI file

To validate CPI files, use the CPIValidator tool. You can find this tool in a Fast Models Tools installation 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
    (0x1c000000|0x14000000) Cpi:1 Name:B_gen_except_sys
    (0x7f000000|0x11000000) Cpi:0.5 Name:ADD_imm
    (0x7f00001f|0x7100001f) Cpi:0.75 Name:CMP_imm
    (0x7f200000|0x0b000000) Cpi:0.5 Name:ADD_sft_reg
    (0x7f200000|0x2a000000) Cpi:0.5 Name:ORR_sft_reg
    (0x7f20001f|0x6b00001f) Cpi:0.75 Name:CMP_sft_reg
    (0x7f800000|0x32000000) Cpi:0.5 Name:ORR_imm
    (0x7fc00000|0x28c00000) Cpi:2 Name:LDP_post_idx
    (0x7fc00000|0x29400000) Cpi:2 Name:LDP_sgn_off
    (0x7fc00000|0x29c00000) Cpi:2 Name:LDP_pre_idx
    (0x7fe00000|0x0b200000) Cpi:0.5 Name:ADD_ext_reg
    (0x7fe0001f|0x6b20001f) Cpi:0.75 Name:CMP_ext_reg
    (0x9f000000|0x10000000) Cpi:0.25 Name:ADR
    (0x9f000000|0x90000000) Cpi:0.25 Name:ADRP
    (0xbfc00000|0xb9000000) Cpi:1 Name:STR_imm_usg_off
    (0xbfe00c00|0xb8000400) Cpi:1 Name:STR_imm_post_idx
    (0xbfe00c00|0xb8000c00) Cpi:1 Name:STR_imm_pre_idx
    (0xbfe00c00|0xb8200000) Cpi:1 Name:STR_reg
Instruction Set: Thumb Default Cpi:0.75
Instruction Set: T2EE Default Cpi:0.75
```

9.7.2.6 CPI class example program

This example program is designed to show the effect of the CPI values specified in the example CPI file.

The example consists of two source files, `main.c` and `asm_func.s`.

`main.c` contains the following code:

```
#include <stdio.h>
#include <string.h>
```

```
extern void asm_cpi(volatile int *value0, volatile int *value2);

volatile int values[2] = {1, 2};

int main(void) {
 asm_cpi(&values[0], &values[1]);
 return 0;
}
```

`asm_func.s` defines an embedded assembly language function `asm_cpi()` which uses instructions with defined CPI values:

```
  .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 of instructions 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 this code on a 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 name of the executable used in these examples is `ta_cpi.axf` and the platform is `EVS_Base_Cortex-A73x1.x`.

9.7.2.7 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-7.3/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` plug-in produces, find the instruction at address `0x800005a4`. The trace for this instruction and the one before it is as follows:

```
INST: PC=0x00000000800005a0 OPCODE=0x910003fd SIZE=0x04 MODE=EL1h ISET=AArch64
PADDR=0x00000000800005a0 NSDESC=0x01 PADDR2=0x00000000800005a0 NSDESC2=0x01 NS=0x01
ITSTATE=0x00 INST_COUNT=0x000000000000b7bc LOCAL_TIME=0x0000000000007530
CURRENT_TIME=0x000000001c091fc0 CORE_NUM=0x00 DISASS="MOV      x29,sp"

INST: PC=0x00000000800005a4 OPCODE=0x90000020 SIZE=0x04 MODE=EL1h ISET=AArch64
PADDR=0x00000000800005a4 NSDESC=0x01 PADDR2=0x00000000800005a4 NSDESC2=0x01 NS=0x01
ITSTATE=0x00 INST_COUNT=0x000000000000b7bd LOCAL_TIME=0x0000000000009c40
CURRENT_TIME=0x000000001c0946d0 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.

9.7.2.8 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-7.3/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` plug-in 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 9-2: CPI values for embedded assembly instructions

Address	Instruction	Simulated time (ps)	CPI value observed
0x800005a4	ADRP x0,{pc}+0x4000	2500	0.25
0x800005a8	ADD x0,x0,#0x9f0	5000	0.50
0x800005ac	ADD x1,x0,#4	5000	0.50
0x800005b0	BL {pc}+0x4294	10000	1.00
0x80004844	LDP w1,w2,[x0,#0]	20000	2.00
0x80004848	CMP w1,w2	7500	0.75
0x8000484c	B.GT {pc}+0xc	10000	1.00
0x80004850	ORR w1,w1,w2	5000	0.50
0x80004854	STR w1,[x0,#0]	10000	1.00
0x80004858	RET	10000	1.00

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 9-3: Statistics for the whole simulation

Total number of instructions	Overall simulated time in seconds	Average CPI value
47701	0.000362	0.75889


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.

9.7.2.9 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-7.3/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 9-4: CPI values for simulation with branch prediction latency

Branch prediction example CPI configuration	Overall simulated time in seconds	Average CPI value
Using the average CPI value that was observed in the CPI class example program.	0.001726	1.00754
Using the custom CPI file.	0.001945	1.13538

Related information

Branch prediction example program on page 241

9.7.3 Modeling branch prediction

This section demonstrates various techniques for measuring the effectiveness of different branch prediction algorithms.

9.7.3.1 Branch predictor types and parameters

The `BranchPrediction` plug-in allows you to select the branch prediction algorithm to use, the type of statistics to collect, and the misprediction latency.

The plug-in parameters that are used in this tutorial are as follows:

Table 9-5: `BranchPrediction` plug-in parameters

Plug-in parameter	Purpose in this example	Values that are used in this example
`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

9.7.3.2 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` plug-in to get an overall picture, without context about the execution order.
- Load the `BranchPrediction` plug-in and use the MTI trace sources `INST`, `BRANCH_MISPREDICT`, and `WAYPOINT` to see branch misprediction details for individual instructions in execution order.

9.7.3.2.1 BranchPrediction plug-in statistics

The statistics feature of the `BranchPrediction` plug-in 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 9-6: Overall statistics

Statistic	Description	Example
`Processor Core`	Name of the core to which the branch prediction plug-in was connected.	`ARM_Cortex-A73`
`Cluster instance`	The cluster number in the processor.	0
`Core instance`	The core number in the cluster.	0
`Mispredict Latency`	The branch misprediction latency as specified using the `mispredict-latency` parameter.	11
`Image executed`	The name of the application file that was executed.	`ta_brpred.axf`
`PredictorType`	The branch prediction algorithm as specified using the `predictor-type` parameter.	`FixedDirectionPredictor`
`Total branch calls`	The total number of times all branch instructions were executed.	37434
`Total Mispredictions`	The total number of mispredictions for all executed branch instructions.	5106

Statistic	Description	Example
`Average prediction accuracy`	The fraction of all branch instructions that were correctly predicted.	0.8636
`Conditional Branches`	The total number of unique conditional branch instructions. This figure does not include the instructions `CBZ` and `CBNZ`.	123
`Total unique branch instructions`	The total number of unique conditional and unconditional branch instructions.	300

The following table shows the `BranchPrediction` plug-in 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 9-7: Per-branch statistics

Statistic	Description	Example
`PC Addr`	The address of the branch instruction.	`0x8000062c`
`Calls`	The total number of times the branch was called.	2100
`Mispredict`	The total number of times the branch was mispredicted.	260
`Accuracy`	The fraction of calls to the branch instruction that were correctly predicted.	0.87619

Related information

Branch prediction example program on page 241
Branch predictor types and parameters on page 237

9.7.3.2.2 MTI trace sources

`INST`, `BRANCH_MISPREDICT`, and `WAYPOINT` are trace sources that can be used in combination to get useful information about branch mispredictions.

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` plug-ins. For example:

```
$PVLIB_HOME/examples/SystemCExport/EVS_Platforms/EVS_Base/Build_Cortex-A73x1/EVS_Base_Cortex-
A73x1.x \
-C Base.bp.secure_memory=0 \
-C Base.cache_state_modelled=1 \
-C Base.cluster0.icache-prefetch_enabled=1 \
--plugin=$PVLIB_HOME/plugins/Linux64_GCC-7.3/BranchPrediction.so \
-C BranchPrediction.BranchPrediction.predictor-type=FixedDirectionPredictor \
-C BranchPrediction.BranchPrediction.mispredict-latency=11 \
-C BranchPrediction.BranchPrediction.bpstat-pathfilename=stats.txt \
--plugin=$PVLIB_HOME/plugins/Linux64_GCC-7.3/GenericTrace.so \
-C TRACE.GenericTrace.trace-sources=INST,BRANCH_MISPREDICT,WAYPOINT \
-C TRACE.GenericTrace.trace-file=trace.txt \
-a $PVLIB_HOME/images/ta_brpred.axf \
--stat
```

Related information

Calculating the execution time of an instruction on page 228

9.7.3.2.3 Example trace for a branch misprediction

The following example trace is for a branch misprediction with a misprediction latency of 11 ticks:

```
INST: PC=0x0000000080000628 OPCODE=0x7100655f SIZE=0x04 MODE=EL1h ISET=AArch64
PADDR=0x0000000080000628 NSDESC=0x01 PADDR2=0x0000000080000628 NSDESC2=0x01 NS=0x01
ITSTATE=0x00 INST_COUNT=0x000000000001080b LOCAL_TIME=0x000000000003f7a0
CURRENT_TIME=0x000000002eab53a0 CORE_NUM=0x00 DISASS="CMP      w10,#0x19"

INST: PC=0x000000008000062c OPCODE=0x54000168 SIZE=0x04 MODE=EL1h ISET=AArch64
PADDR=0x000000008000062c NSDESC=0x01 PADDR2=0x000000008000062c NSDESC2=0x01 NS=0x01
ITSTATE=0x00 INST_COUNT=0x000000000001080c LOCAL_TIME=0x0000000000041eb0
CURRENT_TIME=0x000000002eab7ab0 CORE_NUM=0x00 DISASS="B.HI     {pc}+0x2c ;
 0x80000658"

WAYPOINT: PC=0x000000008000062c ISET=AArch64 TARGET=0x0000000080000658
TARGET_ISET=AArch64 TAKEN=N IS_COND=Y CORE_NUM=0x00

BRANCH_MISPREDICT: PC=0x000000008000062c

INST: PC=0x0000000080000630 OPCODE=0x7100151f SIZE=0x04 MODE=EL1h ISET=AArch64
PADDR=0x0000000080000630 NSDESC=0x01 PADDR2=0x0000000080000630 NSDESC2=0x01 NS=0x01
ITSTATE=0x00 INST_COUNT=0x000000000001080d LOCAL_TIME=0x000000000005f370
CURRENT_TIME=0x000000002ead4f70 CORE_NUM=0x00 DISASS="CMP      w8,#5"
```

The following information can be gathered from this trace:

- The branch instruction at address `0x8000062c` 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.

9.7.3.3 Branch prediction example program

This 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 within the following constant string, and loops over this operation a set number of times:

```
Timing annotation can be used with an SVP, Split Virtual Platform, or an EVS,
 Exported Virtual Subsystem.
```

The following code shows the branch operations of interest:

```
#define MAX_LENGTH 5
#define LOOP_COUNT 20
…
// 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 branch prediction statistics and trace sources.

The conditions are described in the following table. The branch behavior column describes the relationship between the condition and the associated branch instruction.

Table 9-8: Branch behavior for each condition

Condition	Description	Compiled instruction	Branch behavior
A	Outer loop for processing string `LOOP_COUNT` times. Loop not entered `1/LOOP_COUNT` times.	`B.NE 0x800005f4` at address `0x80000698`.	Backwards branch. Taken to start of loop if more iterations remain.
B	Inner loop for iterating through characters in the string.	`B.NE 0x80000618` at address `0x8000068c`.	Backwards branch. Taken to start of loop if more iterations remain.
C	Condition true if the character being processed is upper case.	`B.HI 0x80000658` at address `0x8000062c`.	Forwards branch. Taken if the condition is false. Skips code that handles upper case characters.
D	Condition true up to `MAX_LENGTH` times consecutively.	`B.GE 0x80000644` at address `0x80000634`.	Forwards branch. Taken if the condition is false. Skips code that appends a letter to an acronym.
E	Condition true up to `MAX_LENGTH+1` times consecutively.	`B.GT 0x80000684` at address `0x80000648`.	Forwards branch. Taken if the condition is false. Skips code that increments the acronym length.

Condition	Description	Compiled instruction	Branch behavior
F	Condition true if E was true, after which C was false.	`B.HI 0x80000674` at address `0x80000660`.	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.
G	Condition true if E was false, after which C was false.	`CBZ w8,0x80000684` at address `0x80000674`.	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.

9.7.3.4 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, where `ta_brpred.axf` is the name of the executable and EVS_Base_Cortex-A73x1.x is the platform:

```
$PVLIB_HOME/examples/SystemCExport/EVS_Platforms/EVS_Base/Build_Cortex-A73x1/EVS_Base_Cortex-
A73x1.x \
-C Base.bp.secure_memory=0 \
-C Base.cache_state_modelled=1 \
-C Base.cluster0.icache-prefetch_enabled=1 \
--plugin=$PVLIB_HOME/plugins/Linux64_GCC-7.3/BranchPrediction.so \
-C BranchPrediction.BranchPrediction.predictor-type=FixedDirectionPredictor \
-C BranchPrediction.BranchPrediction.mispredict-latency=11 \
-C BranchPrediction.BranchPrediction.bpstat-pathfilename=stats.txt \
--plugin=$PVLIB_HOME/plugins/Linux64_GCC-7.3/GenericTrace.so \
-C TRACE.GenericTrace.trace-sources=INST,BRANCH_MISPREDICT,WAYPOINT \
-C TRACE.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                               : 0.642064s
Performance index                       : 0.00
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 information

Branch predictor types and parameters on page 237

9.7.3.5 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 9-9: Comparison of branch predictor accuracy

		Branch instruction						
Branch predictor	**Statistic**	**A**	**B**	**C**	**D**	**E**	**F**	**G**
All	Calls	20	2100	2100	260	260	1840	1800
	`TAKEN`	19	2080	1840	0	0	1800	1800
	`NOT_TAKEN`	1	20	260	260	260	40	0
`FixedDirectionPredictor`	Mispredictions	1	20	260	260	260	40	0
	Mispredicted as `TAKEN`	1	20	280	260	260	40	0
	Mispredicted as `NOT_TAKEN`	0	0	0	0	0	0	0
	Accuracy (%)	95*	99*	88*	0	0	98*	100*
`BiModalPredictor`	Mispredictions	1	20	341	1	1	40	0
	Mispredicted as `TAKEN`	1	20	220	1	1	40	0
	Mispredicted as `NOT_TAKEN`	0	0	121	0	0	0	0
	Accuracy (%)	95*	99*	84	100*	100*	98*	100*
`GSharePredictor`	Mispredictions	1	20	279	241	241	40	0
	Mispredicted as `TAKEN`	1	20	260	241	241	40	0
	Mispredicted as `NOT_TAKEN`	0	0	19	0	0	0	0
	Accuracy (%)	95*	99*	87	7	7	98*	100*
`CortexA53Predictor`	Mispredictions	1	23	324	2	1	49	0
	Mispredicted as `TAKEN`	1	20	221	2	1	40	0
	Mispredicted as `NOT_TAKEN`	0	3	103	0	0	9	0
	Accuracy (%)	95*	99*	85	99	100*	97	100*

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 with an asterisk. 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 9-10: Overall branch predictor accuracy

Predictor type	Overall accuracy (%)
`FixedDirectionPredictor`	86
`BiModalPredictor`	98
`GSharePredictor`	86
`CortexA53Predictor`	98

9.7.3.6 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 9-11: Overall simulation time for each predictor type

Predictor type	Simulation time with `mispredict-latency=11`	Simulation time with `mispredict-latency=0`
`FixedDirectionPredictor`	0.002275s	0.001713s
`BiModalPredictor`	0.001805s	0.001713s
`GSharePredictor`	0.002289s	0.001713s

Predictor type	Simulation time with `mispredict-latency=11`	Simulation time with `mispredict-latency=0`
`CortexA53Predictor`	0.001806s	0.001713s

10 FastRAM

FastRAM is a bus optimization for Fast Models that can bring significant speed improvements to platform models.

10.1 Introducing FastRAM, a bus optimization for Fast Models

FastRAM is a fast interface to simulated RAM which allows platform models to avoid using bus models for most transactions.

FastRAM uses a cache of DMI pointers, each of which points to 64MB. This memory is tightly coupled to the Fast Models bus masters and models of IP that are bus masters. When FastRAM is enabled, accesses by Fast Models bus masters to platform RAM components do not use the PVBus or TLM bus models. Accesses to other platform components and areas of RAM for which FastRAM has not been enabled work as normal.

FastRAM can give significant speed improvements to large and complex platform models which can spend a lot of time in the bus models. It can particularly benefit SystemC platforms that use TLM, and multi-threaded platforms.

Most, but not all, platform models can safely use FastRAM. For conditions that can prevent its use, see 10.5 FastRAM limitations on page 250.

The behavior of platform models is functionally equivalent whether FastRAM is enabled or disabled. However, modeling bus transactions in a platform can lead to scheduling changes, so the overall flow of execution by components in a platform might not be identical.

10.2 How to enable FastRAM

Enable FastRAM by launching the platform model with the command-line parameter `--fast-ram <config_file>`.

You must provide a configuration file in the current working directory of the simulation. This is an ASCII file that specifies:

- Whether the FastRAM implementation allocates the RAM itself or uses memory allocated by the platform RAM model.
- One or more physical address ranges to enable for FastRAM.
- Details of any address aliasing for the enabled ranges.
- Which bus masters to enable to use FastRAM.

10.3 FastRAM configuration file syntax

Each line in the configuration file starts with a single character option followed by the required arguments, separated with whitespace.

The following options are available:

T

Enable FastRAM trace on stdout from this point in the file.

Q

Disable FastRAM trace on stdout from this point in the file.


Note

The position of the `T` and `Q` options in the file is significant:

- To enable FastRAM trace during the entire initialization and runtime, start the file with `T` and do not use `Q`.
- To enable FastRAM trace during runtime but not initialization, end the file with `T` and do not use `Q`.
- To enable FastRAM trace during the initialization only, start the file with `T` and end the file with `Q`.
- To enable FastRAM trace during specific parts of the initialization, use one or more pairs of `T` and `Q` within the file.

A | D

The mode for obtaining the DMI pointers, either:

A Allocate FastRAM memory in the masters.
D Use platform DMI for FastRAM memory.

S

Optimize FastRAM for single-threaded simulations.

M <*string*> | ALL

Identify the bus masters to use FastRAM. You can select either masters whose id contains <*string*> or all masters. This option can be specified multiple times. For example, to enable FastRAM use by all masters with `A57` or `R52` in their id, specify:

```
M A57
M R52
```


Note

If the argument to `M` is not `ALL` and trace is enabled, then the ids of all masters are shown on the console with a message stating whether the master is enabled for FastRAM or not. To find the list of masters, use `M foo` then use the list to select the masters required.

+ <*base*> <*size*>

Add the physical address range <*base*> to <*base*>+<*size*>.

- <*base*> <*size*>

Remove the physical address range <*base*> to <*base*>+<*size*>.

= <*base-a*> <*base-b*> <*size*>

Alias a physical address range.

 # <text>

Comment.


Note

All addresses and sizes must be 64MB-aligned (`0x4000000`) hexadecimal.

10.4 FastRAM configuration file example

This example FastRAM configuration file is written for a Base Platform FVP.

It does the following:

- Uses the `T` option at the start of the file to enable FastRAM trace output from the start of the FastRAM initialization.
- Enables FastRAM for the address range `0x08_00000000-0xff_ffffffff`.
- Defines `0x00_80000000-0x00_ffffffff` as an alias for the range `0x08_00000000-0x08_7fffffff`.
- Uses the `Q` option at the end of the file to disable FastRAM trace output at the end of the FastRAM initialization.

```
# FastRAM config file for FVP Base
T
A
M ALL
+ 800000000 F800000000
= 80000000 800000000 80000000
Q
```

If FastRAM has been successfully enabled, it prints the following output:

```
FastRAM: CONSTRUCTED
FastRAM: Address space size = 40 bits
FastRAM: Slab size = 64 Mb
FastRAM: Page size = 4 kb
FastRAM: Singleton size = 147 kb
FastRAM: Number of monitors = 16
FastRAM: Allocate RAM
FastRAM: Enable ALL masters
FastRAM: Add range 0x08_00000000...ff_ffffffff
FastRAM: Add range 0x00_80000000...00_ffffffff
```

```
FastRAM: Alias range 0x00_80000000...00_ffffffff <=> 0x08_00000000...08_7fffffff
```

10.5 FastRAM limitations

FastRAM can be used with most, but not all, platform models.

It can be used in a platform in which all of the following conditions are true:

- The platform contains one or more very frequently accessed RAM components that are a whole multiple of 64MB in size.
- These RAM components are always mapped to the same static physical range as seen by the bus masters that frequently access the RAM.
- The physical ranges used to access the RAM components by the bus masters can include aliased regions.
- The RAM components and the buses to them always give back DMI and for a given physical address always give back exactly the same DMI pointer and never invalidate DMI.
- Either all the bus masters in the platform use FastRAM for the configured physical ranges or you can identify the subset of masters that can use it by name. See 10.3 FastRAM configuration file syntax on page 247 for how to find the list of bus masters.
- All the bus masters that use FastRAM use the same physical address map to access the RAM components.
- If the RAM components internally allocate memory that is a whole multiple of 64MB, then FastRAM can be used with RAM instances that are accessed by:
  - Bus masters that are enabled to use FastRAM.
  - Bus masters that are not, or cannot, be enabled to use FastRAM.
- If the RAM components internally allocate memory that is not a whole multiple of 64MB, for example the RAMDevice LISA component, then FastRAM can only be used with RAM instances that are accessed by masters that are enabled to use FastRAM.

It cannot be used in a platform if any of the following conditions are true:

- Cache state modeling is enabled.
- The physical address map used by the bus masters to access the RAM is dynamic and can change at run time.
- The set of bus masters that will use FastRAM cannot be identified. See 10.3 FastRAM configuration file syntax on page 247 for how to find the list of bus masters.
- There is System IP between the bus masters and the RAM that needs to provide functionality other than a global monitor. However, a CCI or CCN with cache state modeling disabled is allowed.
- The platform RAM is mapped to an address greater than or equal to `0x100_0000_0000`.
- The expected functionality of the platform depends on being able to invalidate DMI. FastRAM ignores DMI invalidations other than what is required internally to support exclusives and RevokeReadOnWrite behavior.

Appendix A SystemC Export generated ports

This appendix describes Fast Models SystemC Export generated ports.

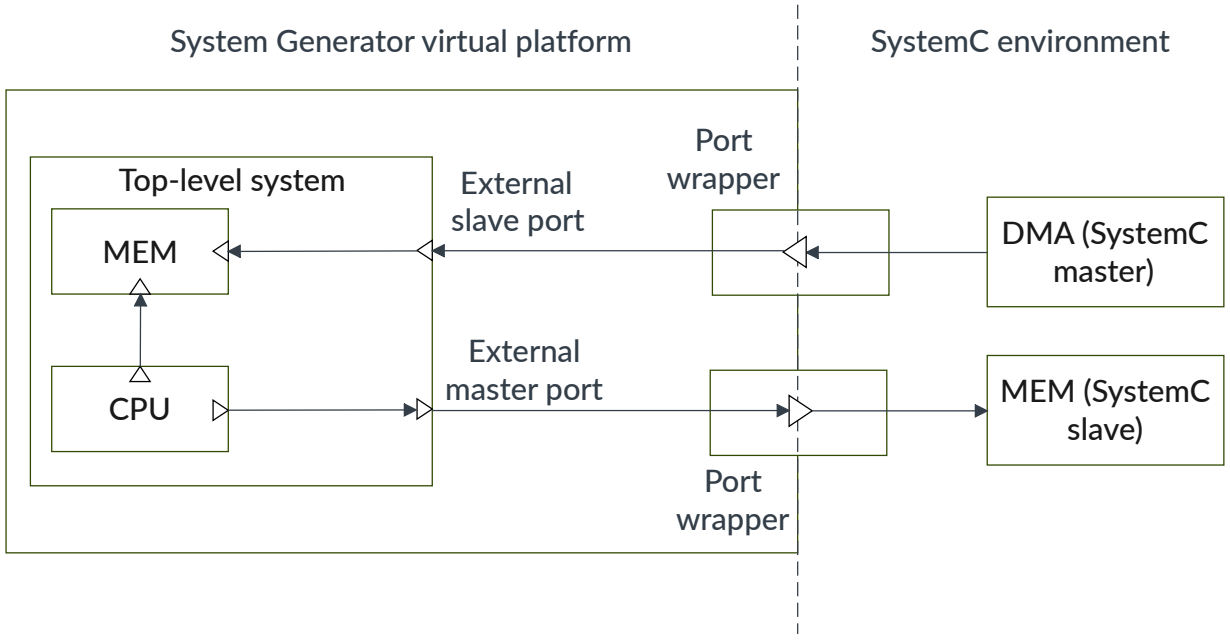
A.1 About SystemC Export generated ports

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.

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



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

Fast Models Reference Guide