Fast Models

Version 8.1

User Guide

ARM®

ARM DUI 0370P (ID060613)

Fast Models
User Guide

Release Information

Change History

Date	Issue	Confidentiality	Change
August 2007	A	Confidential	New document.
December 2007	B	Confidential	Release v3.1.
February 2008	C	Confidential	Release v3.2.
June 2008	D	Confidential	Update for System Generator v4.0.
August 2008	E	Confidential	Update for System Generator v4.0 SP1.
December 2008	F	Confidential	Update for Fast Models v4.1.
March 2009	G	Non-Confidential	Update for Fast Models v4.2.
April 2009	H	Non-Confidential	Update for Fast Models v5.0.
September 2009	I	Non-Confidential	Update for Fast Models v5.1.
February 2010	J	Non-Confidential	Update for Fast Models v5.2.
October 2010	K	Non-Confidential	Update for Fast Models v6.0.
May 2011	L	Non-Confidential	Update for Fast Models v6.1.
November 2011	M	Non-Confidential	Update for Fast Models v7.0.
May 2012	N	Non-Confidential	Update for Fast Models v7.1.
December 2012	O	Non-Confidential	Update for Fast Models v8.0.
May 2013	P	Non-Confidential	Update for Fast Models v8.1.

Proprietary Notice

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.

Product Status

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

Web Address

`http://www.arm.com`

Contents
Fast Models User Guide

Preface

This preface introduces the *Fast Models User Guide*. It contains the following sections:

About this book

This book is for ARM Fast Models and describes how to set up and use System Canvas and related Fast Models Tools.

Intended audience

This book has been written for experienced hardware and software developers to aid the development of ARM-based products using Fast Models as part of a development process.

Using this book

This book is organized into the following chapters:

Chapter 1 *Introduction*

Read this chapter for an introduction to Fast Models.

Chapter 2 *Getting Started*

Read this chapter for a step by step tutorial on using Fast Models.

Chapter 3 *Debugging*

This chapter describes how to debug component and system designs.

Chapter 4 *System Canvas Reference*

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

Chapter 5 *SystemC Export*

This chapter describes how to export a Fast Models system as a SystemC component.

Chapter 6 *Creating a New Component*

This chapter describes how to use the LISA+ language to create a new component.

Appendix A *Building and Running the EB FVP Example System*

This appendix describes how to build and run the supplied Emulation Baseboard FVP model.

Appendix B *Red Hat Linux Dependencies*

This appendix describes the dependencies for Fast Models installations on Red Hat Linux.

Appendix C *Building System Models in Batch Mode*

This appendix lists the command-line options for simgen, the simulation generator.

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 *ARM Glossary*, `http://infocenter.arm.com/help/topic/com.arm.doc.aeg0014-/index.html`.

Conventions

Conventions that this book can use are described in:

- *Typographical conventions*
- *Signals*.

Typographical conventions

The typographical conventions are:

italic	Highlights important notes, introduces special terminology, denotes internal cross-references, and citations.
bold	Highlights interface elements, such as menu names. Denotes signal names. Also used for terms in descriptive lists, where appropriate.
`monospace`	Denotes text that you can enter at the keyboard, such as commands, file and program names, and source code.
`mono`space	Denotes a permitted abbreviation for a command or option. You can enter the underlined text instead of the full command or option name.
`monospace italic`	Denotes arguments to monospace text where the argument is to be replaced by a specific value.
`monospace bold`	Denotes language keywords when used outside example code.
< and >	Enclose replaceable terms for assembler syntax where they appear in code or code fragments. For example: `MRC p15, 0 <Rd>, <CRn>, <CRm>, <Opcode_2>`
SMALL CAPITALS	Used in body text for a few terms that have specific technical meanings, that are defined in the *ARM Glossary*. For example, IMPLEMENTATION DEFINED, IMPLEMENTATION SPECIFIC, UNKNOWN, and UNPREDICTABLE.

Signals

The signal conventions are:

Signal level	The level of an asserted signal depends on whether the signal is active-HIGH or active-LOW. Asserted means: • HIGH for active-HIGH signals • LOW for active-LOW signals.
Lower-case n	At the start or end of a signal name denotes an active-LOW signal.

Terminology

In this book the following terms of the LISA language are used and have the following meaning:

LISA terminology

Term	Definition
Component	An individual sub-system element such as a processor, memory, bus, or peripheral.
Internal port	Internal ports communicate with subcomponents and are not visible if the component is used in a higher-level system. Unlike hidden external ports, they are permanently hidden.
External port	A port that is used to connect the subsystem to other components within a higher-level system.
Connection	A link between two components. The connection is made between a master port on one component and a slave port on the second component.
Label	An annotation in a diagram. There is no processing or connections associated with a label.
Object	One of the following: • component • connection • external port • label.
Component	The basic element in LISA which describes a single component, a subsystem or a complete system. Components are hierarchical. Components can represent concrete pieces of hardware or more abstract entities.
Protocol	A protocol is an interface signature declaration used by the ports of components.
Behavior	Each LISA component can have multiple behavior sections. These sections describe the behavior code in C/C++. A behavior can be thought of like a C++ member function.

For a complete reference of the LISA language and its terms, see the *LISA+ Language for Fast Models Reference Manual*, http://infocenter.arm.com/help/topic/com.arm.doc.dui0372-/index.html.

Additional reading

This section lists publications by ARM and by third parties.

See Infocenter, http://infocenter.arm.com for access to ARM documentation.

ARM publications

This book contains information that is specific to this product. See the following documents for other relevant information:

- *ARM Architecture Reference Manuals*, http://infocenter.arm.com/help/topic/com.arm.doc.set.architecture/index.html
- *Model Debugger for Fast Models User Guide* (ARM DUI 0314)
- *LISA+ Language for Fast Models Reference Manual* (ARM DUI 0372)
- *Component Architecture Debug Interface Developer Guide* (ARM DUI 0444)
- *MxScript for Fast Models Reference Manual* (ARM DUI 0371)
- *Model Shell for Fast Models Reference Manual* (ARM DUI 0457)

- *RealView® Debugger User Guide* (ARM DUI 0153).

Feedback

ARM welcomes feedback on this product and its documentation.

Feedback on this product

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

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

Feedback on content

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

- the title
- the number, ARM DUI 0370P
- the page numbers to which your comments apply
- a concise explanation of your comments.

ARM also welcomes general suggestions for additions and improvements.

Chapter 1
Introduction

This chapter provides a general introduction to Fast Models. It contains the following sections:

1.1 About Fast Models

Fast Models is an environment for easy creation of virtual platform models that execute with high simulation speeds. Fast Models uses *Code Translation* (CT) models, so the simulation speed of generated platforms ranges between 100-500 MIPS on a typical workstation.

Generated platforms are equipped with the *Component Architecture Debug Interface* (CADI) and can run stand-alone or from a suitable debugger. Fast Models automatically generate the required interfaces for both standalone and integrated platforms.

System Canvas provides the Graphical User Interface (GUI) to the Fast Models tools and enables you to assemble systems by using graphical representations of:

- components in a system
- component ports
- external ports, if the system is itself a component in a larger system
- connections between ports.

System Canvas has a block diagram editor for creating the graphical representation of a system. It provides features similar to vector oriented drawing tools such as flow charting programs. The graphical nature of System Canvas provides a rapid way to create and configure components or systems consisting of multiple components.

New components can be added to a single project or added to a component repository for use in multiple projects. Components are defined by LISA+ code and can be written using the text editor that is part of System Canvas.

1.1.1 Software requirements

This section lists the software components that must be available to run Fast Models.

Linux

The following software is required for Linux:

Operating system

Red Hat Enterprise Linux version 4, 5 or 6 (on either 32-bit or 64-bit architectures).

Note

Security Enhanced Linux is enabled by default. Fast Models only supports SELinux for 64-bit models, so you must disable SELinux before you startup 32-bit models.

To do this:

1. Access `/etc/sysconfig/selinux`
2. Change SELINUX=enforcing to SELINUX=permissive.

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

Compiler

gcc version 4.1.2 (32-bit or 64-bit) or gcc version 4.4.4 (32-bit or 64-bit).

Adobe Acrobat reader

Version 8 or higher.

License management utilities

If you are using floating licenses, FLEX*lm* version 9.2 or higher is required. Newer versions are called FLEX*net*.

Use the highest version of the license management utilities provided with any ARM tools you are using. If you do not have a copy of these utilities, contact ARM License Support at: `license.support@arm.com`.

Microsoft Windows

The following software is required for Microsoft Windows:

Operating system

Microsoft Windows XP with Service Pack 2 or higher (32-bit) or Microsoft Windows 7 with Service Pack 1 (32-bit or 64-bit).

Compiler

Fast Models supports Microsoft Visual Studio 2008 and Microsoft Visual Studio 2010 if certain updates are applied. All updates are available from the Microsoft Download Center.

To compile models using Microsoft Visual Studio 2008 you must install:

- Microsoft Visual Studio 2008 Service Pack 1 (KB945140)
- Microsoft Visual Studio 2008 Service Pack 1 ATL Security Update (KB971092)

To compile models using Microsoft Visual Studio 2010 you must install:

- Microsoft Visual Studio 2010 Service Pack 1 (KB983509).

To run models that were compiled using Microsoft Visual Studio 2008 on a different machine you must install:

- Microsoft Visual C++ 2008 Redistributable Package ATL Security Update (KB973551).

To run models that were compiled using Microsoft Visual Studio 2010 on a different machine you must install:

- Microsoft Visual C++ 2010 Redistributable Package (KB2019667).

Note

Fast Models does not support Express editions of Microsoft Visual Studio.

Adobe Reader

Version 8 or higher.

License management utilities

If you are using floating licenses, FLEX*lm* version 9.2 or higher is required. Newer versions are called FLEX*net*.

Use the highest version of the license management utilities provided with any ARM tools you are using. If you do not have a copy of these utilities, contact ARM License Support by emailing `license.support@arm.com`.

Note

If you are using Microsoft Windows *Remote Desktop* (RDP) to access System Canvas or a simulation generated by System Canvas, the type of license you have for Fast Models can restrict use of the product:

- Floating licenses require a license server and do not have any special restrictions relating to use of Remote Desktop. Such licenses are normally issued to Fast Models users who have purchased the tool.
- Node-locked licenses are tied to a specific workstation. These licenses do not work over Remote Desktop connections. Node-locked licenses are typically issued to users who are evaluating Fast Models. If this limitation prevents your evaluation of the product, contact `license.support@arm.com` and request a floating evaluation license.

1.1.2 Installation and uninstallation

Run the installation executable to install Fast Models. If an existing installation is found, a dialog is displayed to enable you to select re-installation or to uninstall the existing installation.

To uninstall Fast Models from a Microsoft Windows workstation, click the **Start** button and select **All Programs → ARM → Fast Models 8.0 → Uninstall Fast Models Tools**.

1.2 Fast Models design flow

The basic design flow for Fast Models is:

1. Create or buy standard component models.
2. Use System Canvas to interconnect components and set parameters.
3. Generate a new model from System Canvas.
4. Use the new model as input to a more complex system or distribute it as a standalone simulation environment.

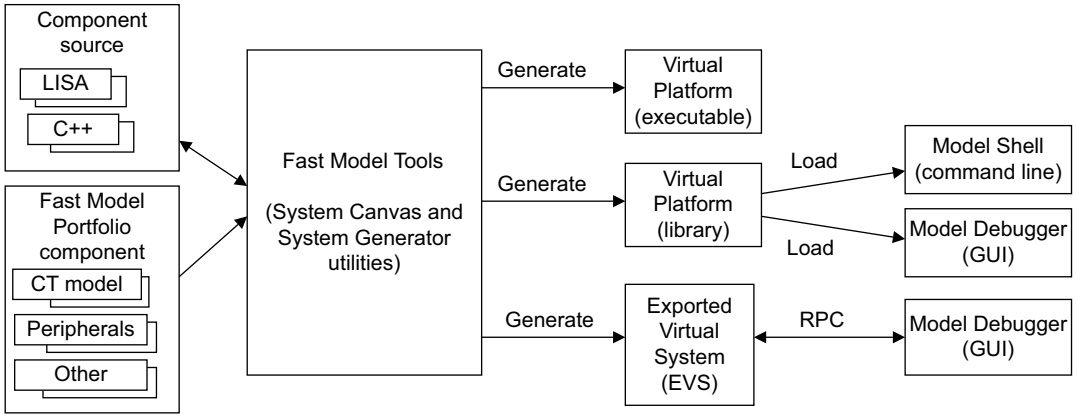

Figure 1-1 Fast Models design flow

The input to System Canvas 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. These can be existing files in the Fast Model Portfolio or new LISA+ files that were created with System Canvas. The LISA+ descriptions can be located in any directory. One LISA+ file can contain one or more component descriptions.

After the required components have been added and connected, System Canvas produces the output object as one or more C++ library objects or SystemC objects, or an executable in the case of an isim. The output can be either a single component or a system that contains many different components.

1.2.1 Project and repository files

Fast Models simplifies projects by using a single project file (`.sgproj`) to drive the complete design process of system creation, build configuration, and building the model:

- There is no requirement to provide a makefile and a set of configuration files for each new project.
- Each project is self contained with respect to the components being used. The project file references all files required to build and run a simulation, including LISA, C and C++ sources, libraries including directories, files to deploy to the simulation directory, and nested repository files.

Repository files have the same format as project files and group references to commonly-used files shared across different projects.

You can add single files or a complete repository, such as the Fast Model Portfolio, to the project file. For more information on project file contents, see *Project Settings dialog* on page 4-60.

Figure 1-2 shows the organization of directories and files for a typical project on Microsoft Windows:

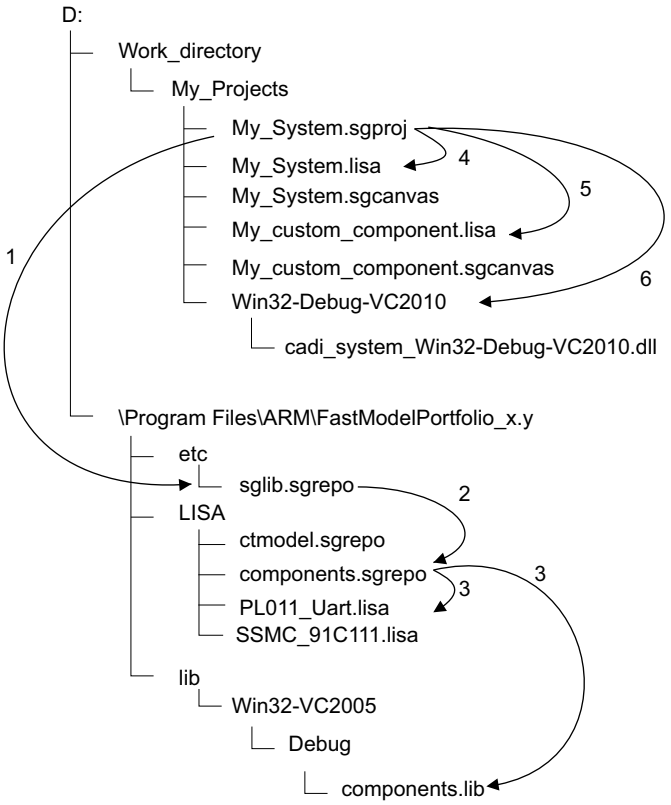

Figure 1-2 Project organization

For Figure 1-2, the `Build` subdirectory of the `My_System` directory contains the `My_System.sgproj` project file:

1. `My_System.sgproj` points to the standard Fast Model Portfolio repository file `sglib.sgrepo`.
2. The `sglib.sgrepo` file contains a list of repository locations such as `components.sgrepo`. Repository files eliminate the requirement to individually specify the path and library for each component. The repository can be hierarchical, as this one is, or can list individual component details.
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 that are used in the system and how the components are interconnected.
5. This project uses a custom component in addition to the standard Fast Model Portfolio components. Custom components can be located anywhere in the directory structure. For Figure 1-2, the custom component is only used with the `My_System` component, so the `My_custom_component.lisa` file is in the same directory.

The `My_System.sgcanvas` and `My_custom_component.sgcanvas` files are generated when a component is edited in the System Canvas 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
- whether ports have been moved or hidden
- grid spacing
- `.sgcanvas` files are not used by the build process in any way. They are only used by the block diagram editor of scganvas.

6. `My_System.sgproj` lists `Win32-Debug-VC2010` as the build directory for the selected platform. The build options in the project file determine:
   - the libraries that are used
   - the location
   - the format of the generated system.

     For example, the extension for the model is `.so` for Linux or `.dll` for Microsoft Windows.

File processing order

Sub-repositories are expanded as soon as they are encountered. The File List view in the right pane corresponds to the processing order.

Example 1-1 and Example 1-2 show an example project and a repository file:

Example 1-1 Source project file

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

Example 1-2 Files in repository

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

The files listed in Example 1-1 on page 1-7 and Example 1-2 on page 1-7 are processed in the following order:

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

This processing order enables a custom implementation of a component to be used in place of standard component implementations. If `MySubComponent1.lisa` and `../LISA/ASubComponent1.lisa` both list a component with the same name, only the first definition is used.

The order of occurrence of files is significant. Use the System Canvas controls to specify the order of files and repositories:

- Use **Up** and **Down** context menu entries in the File List view of the Component window to change the file order within a repository. 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.
- Use the **Up** and **Down** buttons on the **Default Model Repository** tab in the Properties dialog to specify the order of repositories in new projects.

1.2.2 Hierarchical systems

The term 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.

Figure 1-3 on page 1-9 shows the advantage of using a hierarchical system with a complex model. This system includes a Cortex-A15 CoreTile and a VE motherboard and daughterboard.

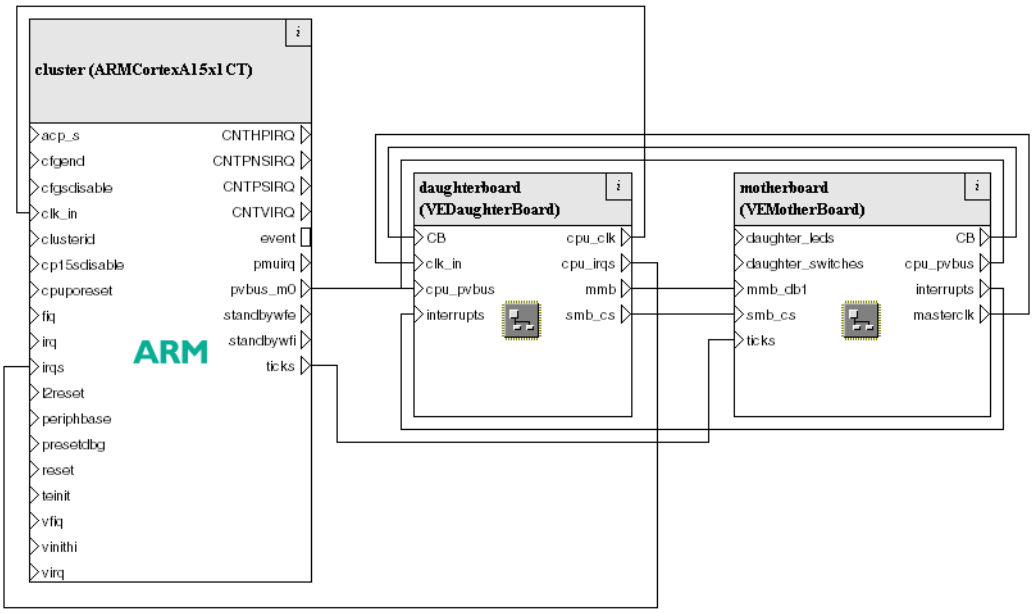

Figure 1-3 Block diagram of top-level VE model in System Canvas

The central component in the system is a VE motherboard component. To open this item, select it and select **Open Component** from the **Object** menu. Figure 1-4 on page 1-10 shows that it is a complex object with many subcomponents:

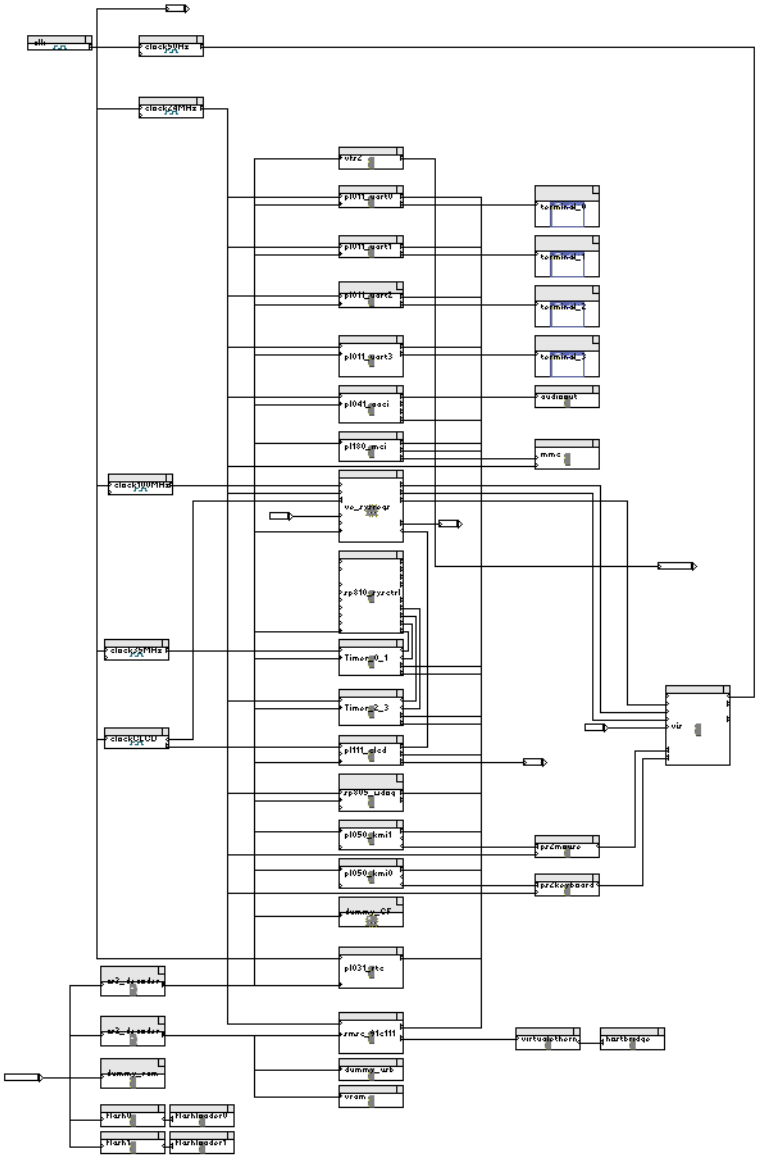

Figure 1-4 Contents of VE motherboard component

Hiding the complexity of the VE motherboard in a component simplifies the drawing shown in Figure 1-3 on page 1-9 and enables the VE motherboard component to be shared between different FVP models.

For example, a clockdivider component located at the top-left of Figure 1-4 is shown in more detail in Figure 1-5 on page 1-11, where there is a connection to an external port called `masterclk`.

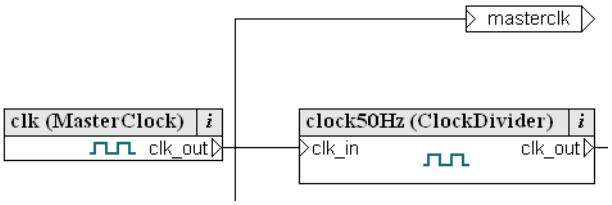

Figure 1-5 Self port detail

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. See Figure 1-6.

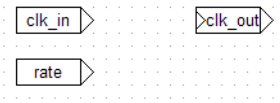

Figure 1-6 Clock divider component external ports

The clock divider component contains only external ports, and it has no subcomponents. The behavior for this component is determined by the LISA code and, if present, the C++ code. Only two ports are shown for the clock divider in Figure 1-5 because the `rate` port is hidden. To display this port, right-click the component and select **Show all ports** from the context menu. There is no connection to the `rate` port.

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 either:

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;
```

Chapter 2
Getting Started

This chapter is a tutorial that covers the steps to create a new project and create a system model. It contains the following sections:

Note

See also Appendix A *Building and Running the EB FVP Example System* for the steps to build an EB FVP model.

2.1 About the tutorial

This tutorial covers the basic operations that are required to build a typical system in System Canvas.

The goal of this tutorial is to use the System Canvas Block Diagram view to build a simple standalone system that can run a simple application image. LISA source is not edited directly.

Note

- The tutorial relies on an executable image that is included in the Third Party IP add-on for the Fast Model Portfolio. You must download this package and install it before you can run the dhrystone image.
- In this tutorial, path names and environment variables for a Microsoft Windows environment are used. If you are using Linux, reverse the direction of the slash character in path names and substitute `%PVLIB_HOME%` for `$PVLIB_HOME`. Linux models have the extension `.so` instead of `.dll`.

2.2 Starting System Canvas

To start System Canvas:

- on Linux, enter `sgcanvas` in a terminal window and press return
- on Microsoft Windows, select **Start → All Programs → ARM → Fast Models 8.0 → System Canvas**

Additional startup options can be specified directly on the command line or by adding them to the **Additional Command Line** options in the Preferences dialog as described elsewhere. See *Applications* on page 4-52.

System Canvas starts as shown in Figure 2-1. 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.

The System Canvas window is described in more detail in Chapter 4 *System Canvas Reference*.

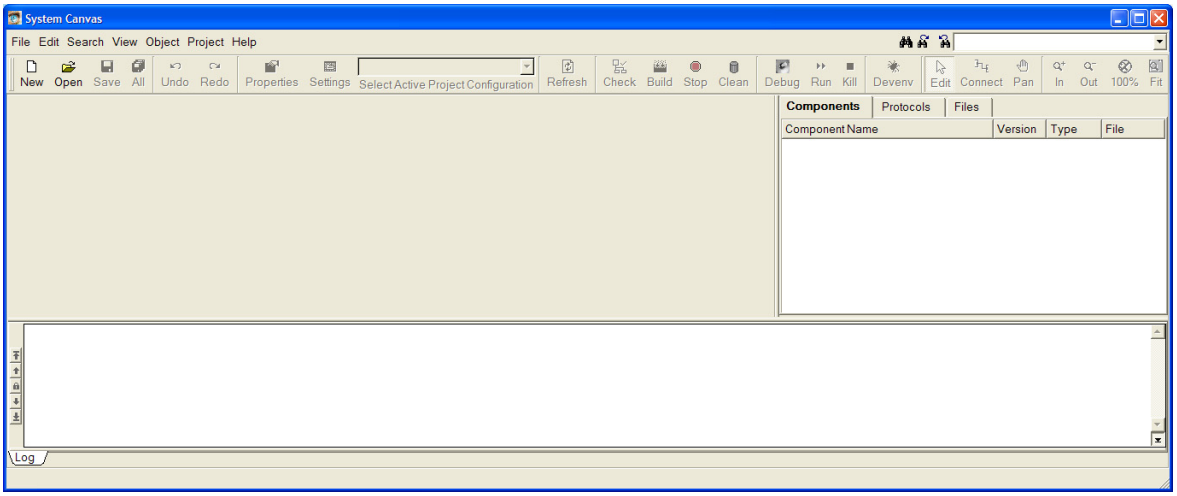

Figure 2-1 System Canvas at startup

2.2.1 Setting up the Fast Model Portfolio

Note

The Fast Model Portfolio installation typically updates the system variables and paths to use the standard libraries. This section describes how to manually configure the libraries. This step is only necessary if you are using non-standard libraries or if your system has been modified.

To incorporate existing components in a system, System Canvas requires information about these components such as their ports, protocols and library dependencies. For convenience, multiple components are grouped together in model repositories such as the Fast Model Portfolio (`sglib.sgrepo`).

Repositories contain lists of components with details of all the LISA files and Linux or Microsoft Windows libraries required to build systems. System Canvas can be configured to use default repositories if a new project is started. The project files store the references to the repositories that are used for the assembled system.

In addition to using the default repository, you can include more components. These are typically newly created custom components and previously-assembled hierarchical systems.

Use the Preferences dialog to select the required components:

1. Select **Setup Default Repository...** from the **Project** menu. The Preferences dialog is displayed as shown in Figure 2-2:

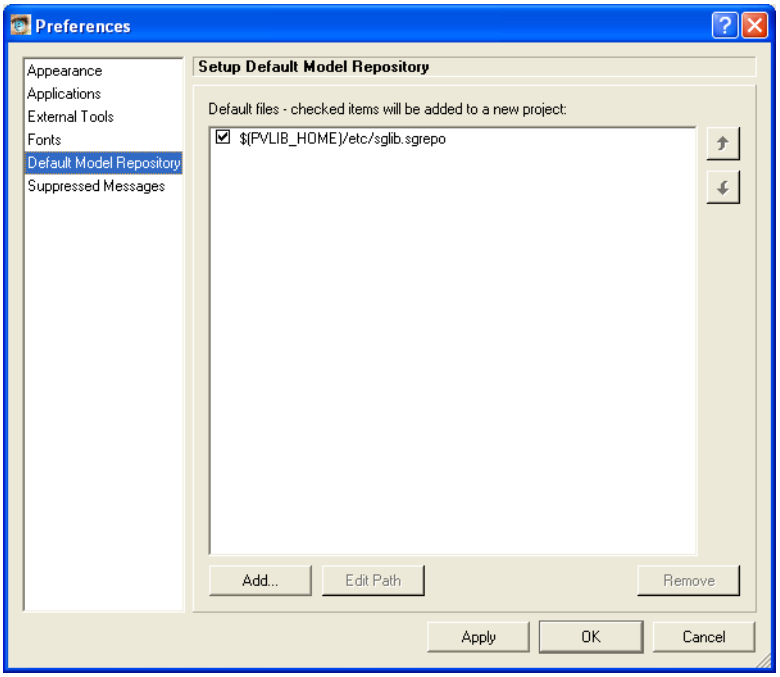

Figure 2-2 Preferences dialog, Setup Default Model Repository

2. The default model repository at `%PVLIB_HOME%\etc\sglib.sgrepo` is always present in the list of available default files. It is the only library required by the example covered in this tutorial.

   Note

   The default model repository does not influence the repositories used by the currently loaded project. This dialog only determines the repositories added by default to new projects. To add a new repository to an existing project, you add a file using the Component window context menu. See *Add Existing Files and Add New File dialogs (Component window)* on page 4-18.

   Although the `%PVLIB_HOME%\etc\sglib.sgrepo` entry cannot be edited or deleted, if the repository is not used by your new system, you can uncheck the entry to exclude the repository from your new project file.

   You can also add new repositories for use with your project, though this is not required for the tutorial:

   - Click the **Add** button, in the Default Model Repository view of the Preferences dialog, to display the Add Model Repository File dialog shown in Figure 2-3 on page 2-6.

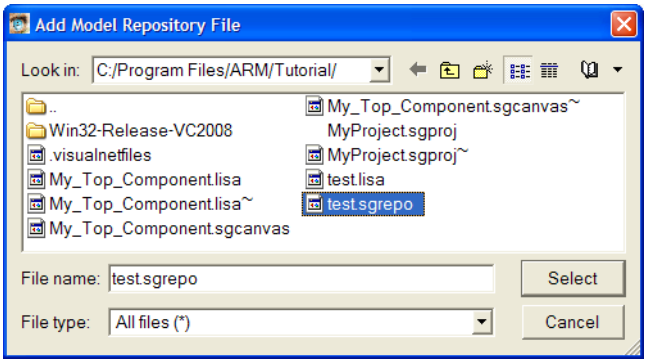

Figure 2-3 Add Model Repository File dialog

- Browse to the required model repository file.
- Click **Select**.
- The Preferences window is updated to show the added repository.

When a new project is created, the component window is filled with the collection of peripherals, processor components, and memories that are present in the repository list.

2.3 Creating a sample system

New projects are started by creating a project file.

1. Select **New Project** from the **File** menu.

   You can also create a new system by clicking the **New** button on the toolbar.

2. The New Project dialog is displayed, shown in Figure 2-4. Navigate to the directory to use for your project.

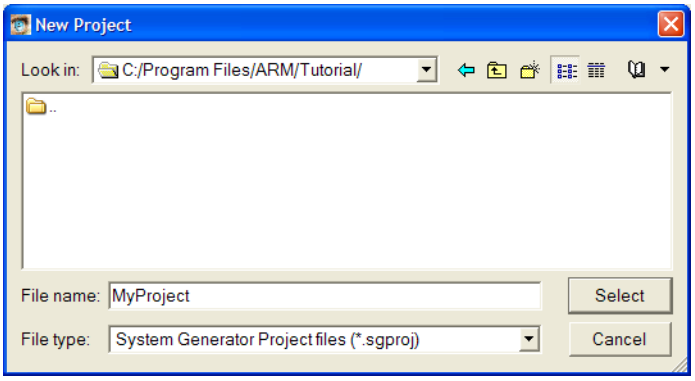

Figure 2-4 New project dialog

3. Enter `MyProject` in the filename box and click the **Select** button. The dialog in Figure 2-5 is displayed for you to enter the name and location of the LISA file that represents your new system.

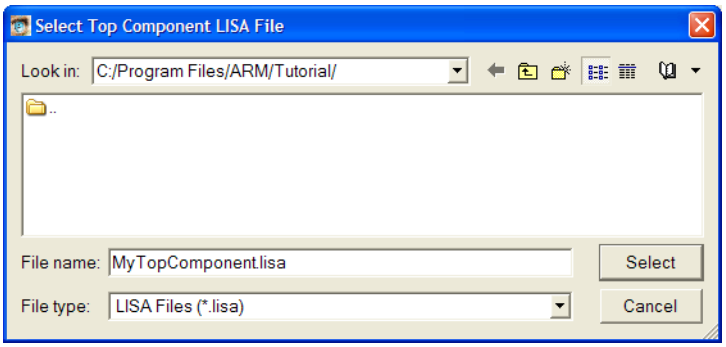

Figure 2-5 Select Top Component LISA File dialog

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

   Note

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

5. The Workspace area now contains a blank block diagram with scroll bars.

6. The Component window, to the right of the Workspace area, lists all of the components that are contained in the specified default repositories. See *Setting up the Fast Model Portfolio* on page 2-4.

2.4 Adding and configuring components

This section describes how to add and configure the components required for the example system. Some key features of System Canvas are introduced in this way.

2.4.1 Adding the ARM processor

Drag-and-drop the ARM processor component from the Component window onto the Workspace window:

1. If it is not already visible, click the **Block Diagram** tab in the Workspace window. This displays a blank window with grid points.
2. Select the **Components** tab in the Components window to display the components that are contained in the Fast Model Repository.
3. Move the mouse pointer over the `ARMCortexA8CT` processor component in the Component window and press and hold the left mouse button (if your mouse is configured for right-handed users).
4. Drag the component over to the middle of the Workspace window.

   Note

   If you move the component in 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 component shown in Figure 2-6 is added to the current system.

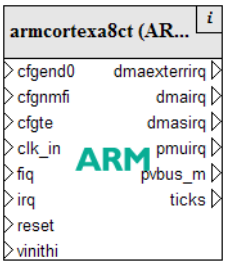

Figure 2-6 ARMCortexA8CT processor component in the Block Diagram window

6. The system name in the title bar now contains an asterisk (*) next to the name to indicate that the LISA file has been modified but not yet saved. You can save the file at any time by selecting **File → Save File** or using **Ctrl+S**.

2.4.2 Naming components

You can give your component a custom name. The following steps describe how to change the name of the processor from the default to `ARM`.

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.

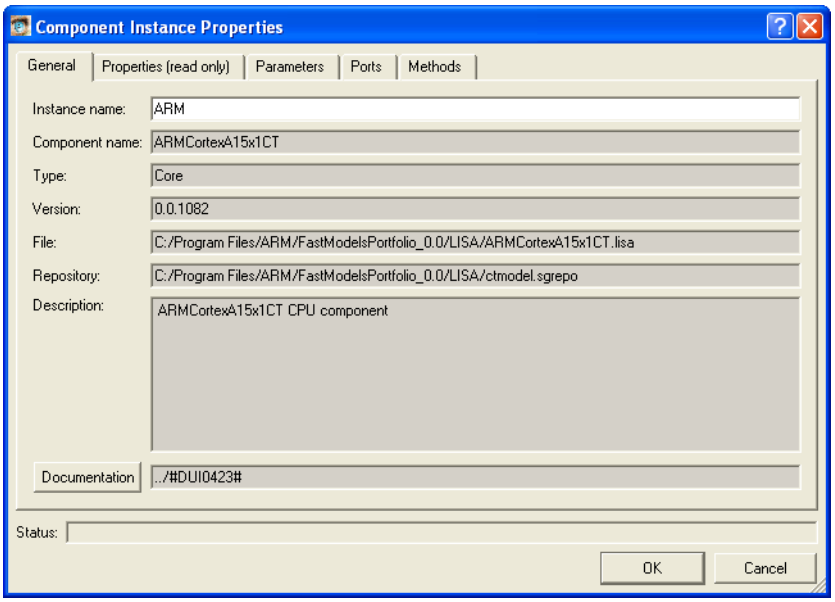

Figure 2-7 Rename component

4. Click **OK** to accept the change. The instance name of the component, that is the name displayed in processor component title, is now `ARM`.

Note

You can edit every part of the system using the block diagram editor or the text editor. To do this, click the Block Diagram tab or the Source tab, respectively, below the System Canvas output window. You can change between the two views at any time. Changes made in one view are reflected in the other view.

2.4.3 Hiding ports

You can hide some of the component ports. This is useful if they are not connected to anything. To hide multiple ports:

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.

If there is only a small number of ports to hide, use the port context menu shown in Figure 2-9 on page 2-11. Right click on the port and select **Hide Port**.

2.4.4 Moving ports

Move the remaining visible ports in the tutorial system to improve readability:

1. Move the mouse pointer so that it is over the port that is to be moved. 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.
3. While holding down the mouse button, drag the port to the location you want. 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.
4. When the port is in the position you want, release the mouse button.
5. Move the other port so that there is one port on each side of the processor component. The `clk_in` port must be on the left side.

2.4.5 Resizing components

You can resize components for aesthetic purposes or to make labels easier to read.

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. Your processor component might look similar to that shown in Figure 2-8.

   To vertically resize the component title bar to avoid truncating text, click the component and drag the lower handle of the shaded title bar.

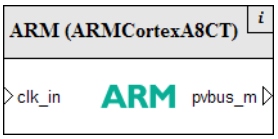

Figure 2-8 Processor component after changes

2.4.6 Adding additional components

Add more components to the project:

1. Use drag and drop to place one of each of the following components onto the Block Diagram window:
   - `ClockDivider`
   - `MasterClock`
   - `PL340_DMC`

     Note

     The `PL340_DMC` component is included to demonstrate some features of System Canvas and is not part of the final example system.

   - `PVBusDecoder`
   - `RAMDevice`.
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.

Note

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

2.4.7 Using port arrays

This section demonstrates how to expand and collapse port arrays using the PL340 component.

1. Right click on one of the axi_if_in ports in the PL340 component. This opens a context menu as shown in Figure 2-9. Select **Collapse Port** to reduce the port array to a single visible item in the component.

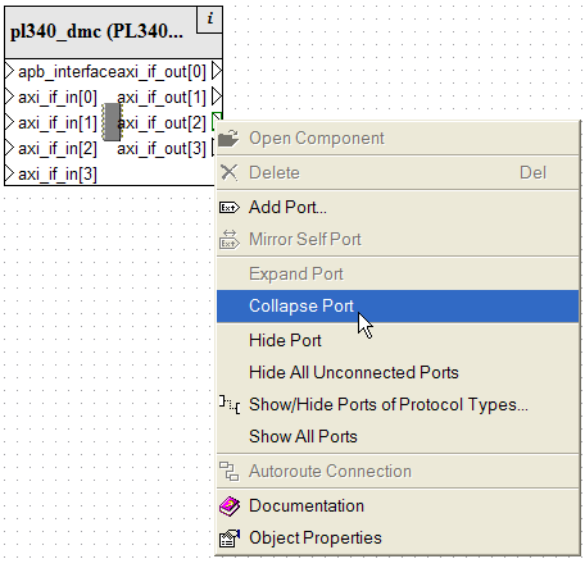

Figure 2-9 Port context menu

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.
4. The axi_if_in port is a port array as indicated by the + beside the port name. Click on the + to expand the port tree view.
5. Deselect the checkboxes beside axi_if_in[2] and axi_if_in[3]. This hides 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**.
6. Right click on the axi_if_in port in the PL340 component and this time select **Expand Port** from the port context menu. Only the axi_if_in[0] and axi_if_in[1] ports are shown.
7. To redisplay the axi_if_in[2] and axi_if_in[3] ports, you can either:
   - use the port context menu shown in Figure 2-9 and select **Show All Ports**, or
   - repeat step 5 and select the checkboxes next to the hidden ports.

Note

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

An alternative sequence of steps to expand the port is:

a. Display the Component Instance Properties dialog

b. Select the **Ports** tab

c. Click on the + next to the port array to expand the port tree view

d. Select the **Show as Expanded** radio button.

8. After you have completed this section of the tutorial, select the `PL340` component and delete it from your system. It is not required for the remainder of the tutorial.

9. Rearrange the components so that your system looks similar to the one shown in Figure 2-10.

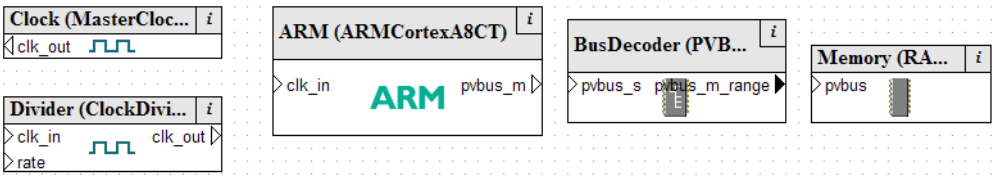

Figure 2-10 Example system with added components

2.5 Connecting components

Connect the components:

1. Switch connection mode by clicking the **Connect** button. You can also enter the connect mode by selecting **Connect Ports Mode** from the **Edit** menu.
2. Move the mouse pointer around in the Block Diagram window:
   - If the mouse pointer is not over an object, the pointer changes to the invalid pointer. This is a circle with a diagonal line through it.
   - If the mouse pointer is over an object, the mouse 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. Ensure that the `clk_out` port is highlighted and 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. A connection is made between the two ports.

   Note

   The application remains in connect mode after the connection is made.

7. Continue connecting components until the system looks like the one shown in Figure 2-11. Some ports have been moved to non-default locations to improve clarity.

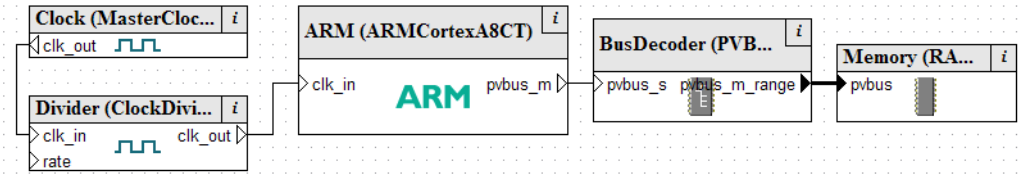

Figure 2-11 Connected components

Note

Connections between the addressable bus ports are displayed with bold lines.

2.6 Fast Models files

This section describes the content and purpose of the Fast Models files that have been created by this stage of the tutorial. All files are located in your project directory.

2.6.1 LISA source

You can use the LISA language directly to create or modify your system. The Block Diagram view of your system is a graphical representation of the LISA source displayed on the **Source** tab of the workspace window. Click on the **Source** tab to display the contents of the `My_Top_Component.lisa` file. 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 view the source, double-click a component in the Block Diagram. Alternatively you can right click on a component in the Components window and select **Open**.

For more information about the LISA language, see the *LISA+ Language for Fast Models Reference Manual*, `http://infocenter.arm.com/help/topic/com.arm.doc.dui0372-/index.html`.

2.6.2 System Canvas file

The System Canvas file contains the Block Diagram layout information for your system. In this tutorial, your system layout file is called `My_Top_Component.sgcanvas`. Do not edit this file yourself.

2.6.3 Fast Models project file

When you created the tutorial project, a project file was also created. Project files have the extension `.sgproj` and contain:

- components used in the system
- connections between system components
- references to the component repositories used
- settings for model generation and compilation are stored in project files.

In this tutorial, the project file is called `MyProject.sgproj`. Do not edit this file yourself. Instead let System Canvas modify it for you automatically if you change project settings. See *Viewing the project settings* on page 2-15.

2.7 Viewing the project settings

System Canvas displays the content of the project file. All LISA files, model repositories, and libraries are shown in the Components list.

The configuration settings can be viewed and edited through the Project Settings dialog. No changes are required for this tutorial. This section, however, demonstrates the steps to use if changes were necessary.

1. Click the **Settings** button, or select **Project Settings** from the **Project** menu, to inspect the project settings for the tutorial system.
2. The Project Settings dialog is displayed as shown in Figure 2-12. The figure shows the Compiler options panel.

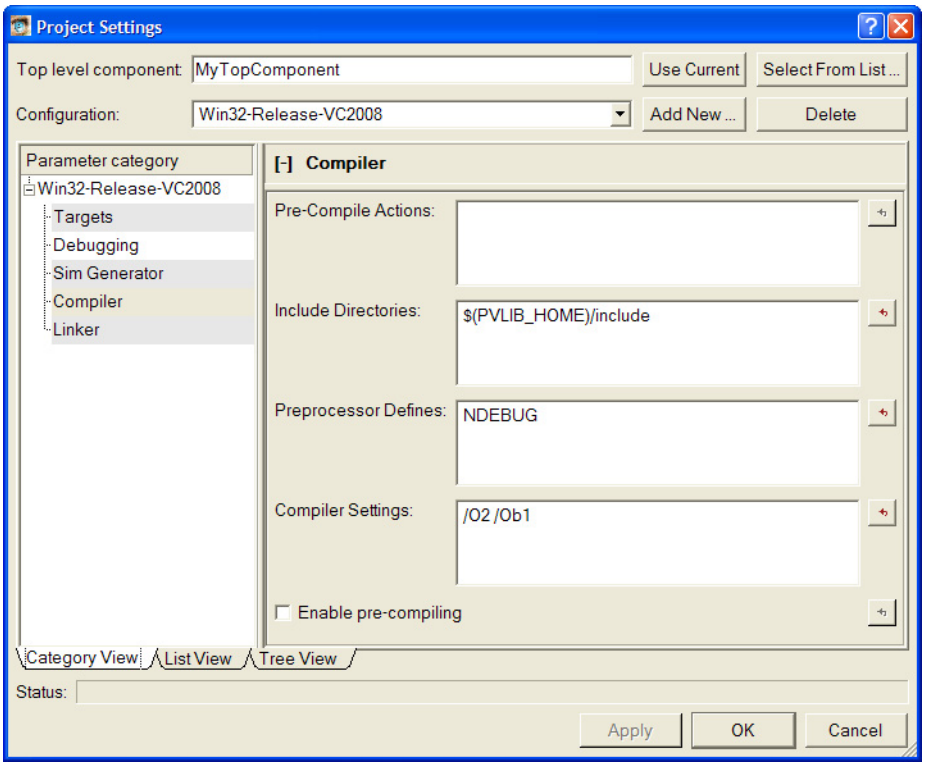

Figure 2-12 Project settings for the example

Note

For Microsoft Visual Studio 2008 and Visual Studio 2010, you must have Service Pack 1 (SP1) installed.

3. Click the **Category View**, **List View**, and **Tree View** tabs and observe how the project parameters are displayed in different groupings.

2.7.1 Specifying the Project Active Configuration

The **Select Active Project Configuration** drop-down menu box on the main tool bar indicates the target for which your system is built. The project file for the system contains one or more sets of configuration values that control how the target is generated.

The **Configuration** drop-down in the Project Settings dialog has configurations to:

- build models with debug support

- build release models optimized for speed.

The full list of project settings can be displayed and edited by selecting **Project Settings** from the **Project** menu. You can inspect and modify a configuration for your operating system by selecting it from the **Configuration** combo box and clicking the different hierarchy elements to view the settings.

Note

- The configuration options available, including compilers and platforms, might be different depending on your operating system.
- Projects created with earlier versions of System Generator might not have the compiler version specified in the Project Settings dialog. You can update an earlier project to specify the compiler option.

2.7.2 Viewing the top component properties

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

In the Project Settings dialog shown in Figure 2-12 on page 2-15, click the **Select From List...** button to view the top components in the system.

The Select Top Component dialog opens and lists all possible top components. See Figure 2-13.

Note

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

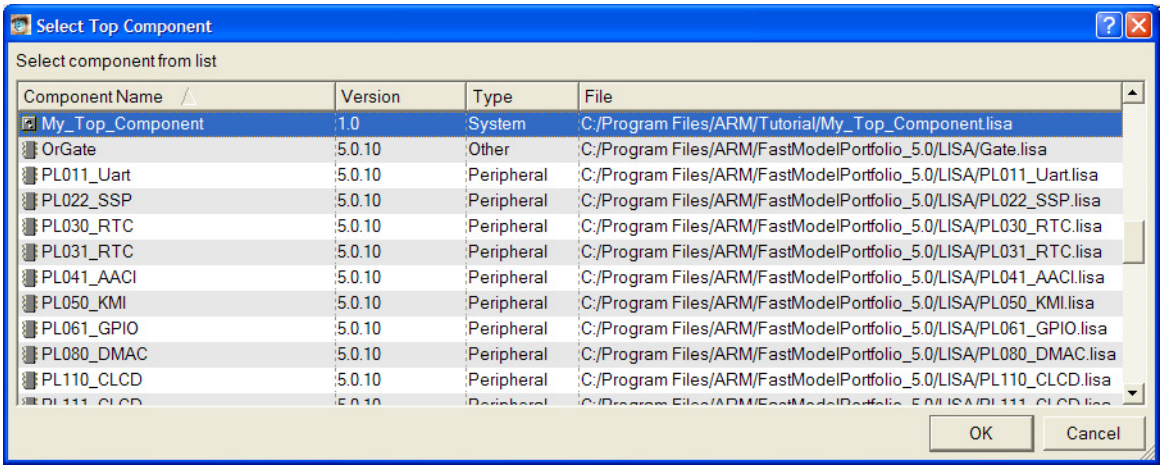

Figure 2-13 Select Top Component dialog showing available components

`My_Top_Component` is the top component in the tutorial project.

2.8 Changing the address mapping

All addressable bus mappings, that is connections that have bold lines, are parameterized by editable address maps.

1. Double-click the `pvbus_m_range` port of the BusDecoder component to open the Port Properties dialog for the port, shown in Figure 2-14.

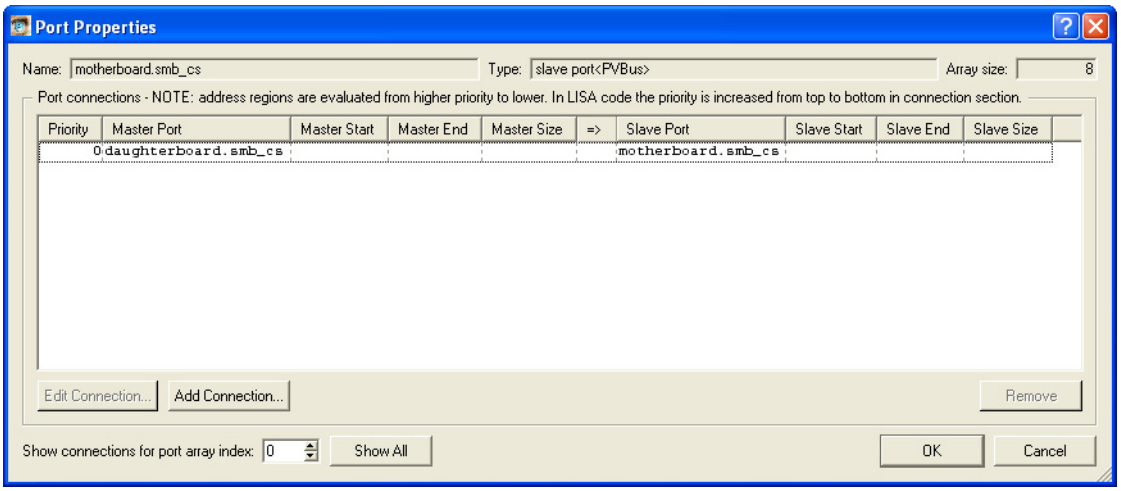

Figure 2-14 Viewing the address Mapping from the Port Properties dialog

2. Select the `Memory.pvbus` Slave Port line, and click **Edit Connection...** to open the Edit Connection dialog shown in Figure 2-15.

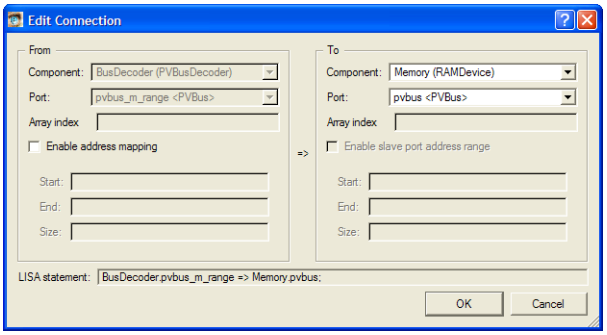

Figure 2-15 Edit Connection dialog

You can also double-click on the entry to open the 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.

   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 maps the master port to the selected address range.

4. If mapping the master port to a different address range on the slave port is required, select **Enable slave port address range**. If **Enable slave port address range** is checked, the parameters for the slave port can be edited as shown in Figure 2-16 on page 2-18. The default values are the same as for the master port when the slave address range is enabled.

Note

When the slave address range is disabled, this is equivalent to specifying the address range 0...size-1, and not the master address range.

In this tutorial, a slave port address range is not required, so deselect the **Enable slave port address range** checkbox.

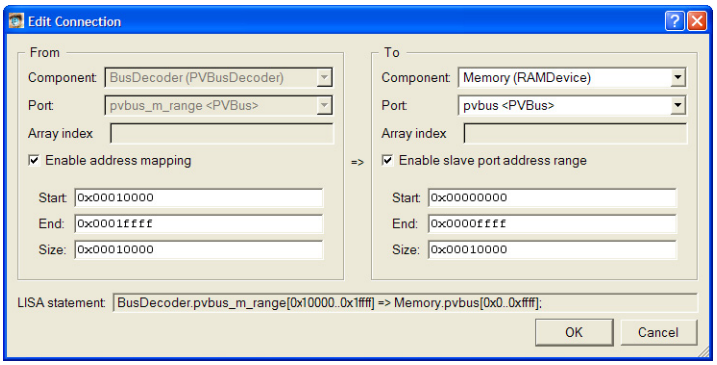

Figure 2-16 Edit address map for slave port

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.

2.9 Building the system

Click the **Build** icon on the System Canvas toolbar to build the model as a `.so` or `.dll` library. You can also build an executable model as described in *Building an ISIM target* on page 2-23.

Note

Depending on your preference setting, a system check might be performed and a window might open if warnings or errors occur. Click **Proceed** to start the build.

The progress of the build is displayed in the Log window as shown in Figure 2-17. Depending on the speed of your computer and the type of build selected, this process might take several minutes.

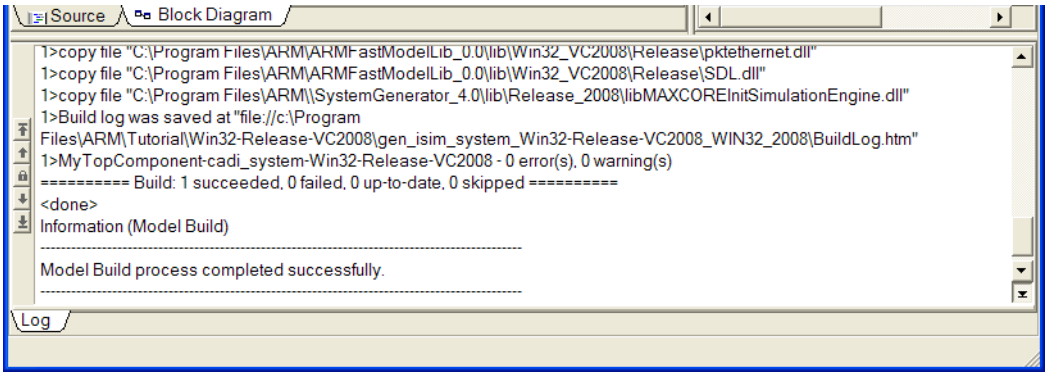

Figure 2-17 Build process output

Note

- Using the `num-comps-file` and `num-build-cpus` build options can considerably reduce compilation time.
- For more information about the simulation generation options, see *Simgen command-line options* on page C-3.

2.10 Debugging the system

After the model build is successful, use Model Debugger to execute and debug it:

1. Select **Project → Project Settings**:

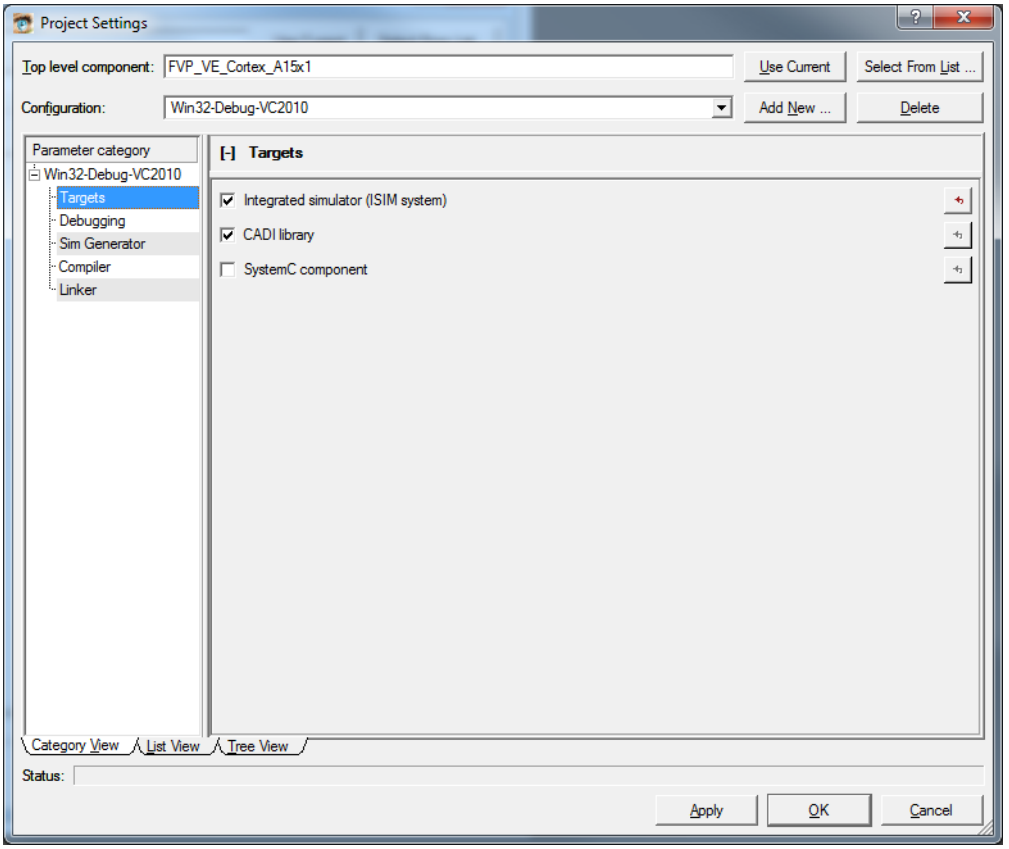

Figure 2-18 Project Settings dialog

2. In the Project Settings dialog, select the required checkboxes from the **Targets** option and click **OK**.

3. Click the **Debug** button in the System Canvas toolbar. See Figure 2-19 on page 2-21:

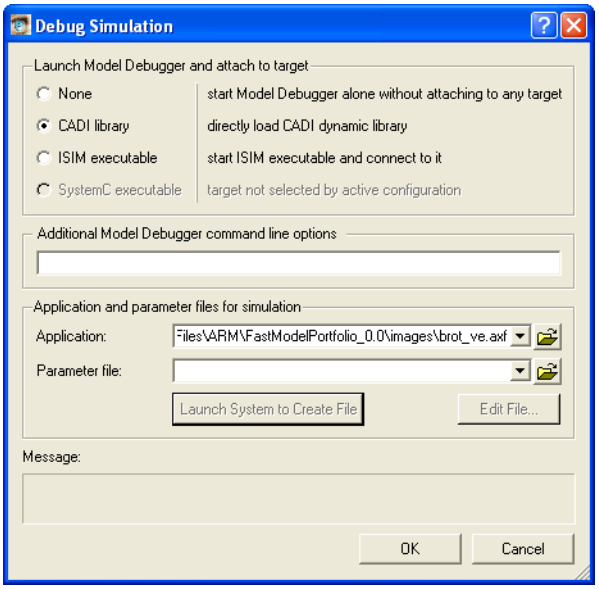

Figure 2-19 Debug Simulation dialog

4. In the **Debug Simulation** dialog, select the required radio buttons to launch Model Debugger and attach to your target. The radio buttons that are available depend on the target settings that you made above.

   If required, enter additional command line options by using the **Additional Model Debugger command line options** field. You can also set the application and parameter files to be simulated by using the **Application** and **Parameter file** drop-down menus.

5. Click **OK**. See Figure 2-20.

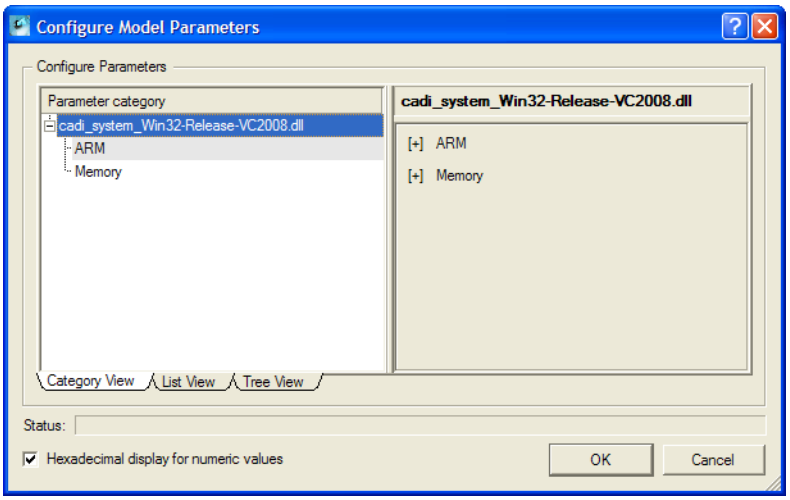

Figure 2-20 Configuring Model Parameters dialog

   The Configure Model Parameters dialog displays the instantiation parameters for the top-level components in the model. To display parameter sets, either:

   - select a Parameter category in the left side of the dialog
   - click on a + next to a component name in the right-hand side.

   Select the **List View** or **Tree View** tabs for different views of the system parameters.

   No changes are required in this tutorial, so click **OK** to close the dialog.

6. The Select Target dialog opens. See Figure 2-21 on page 2-22.

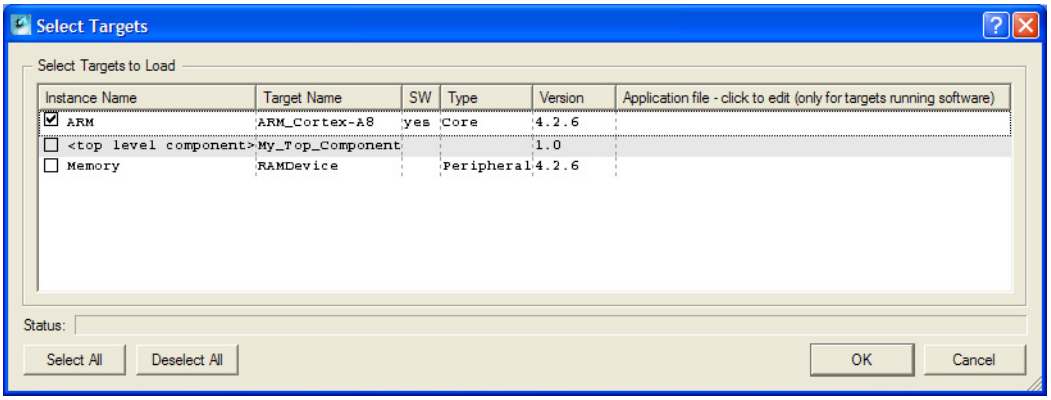

Figure 2-21 Select Targets dialog

The Select Target dialog displays the components to use in Model Debugger. The ARM processor component is selected by default.

You can select additional targets to load in their own instances of Model Debugger. This is not required for the tutorial.

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

   The debugger loads the model library from the `build` directory of the active configuration. One instance of Model Debugger starts.

   If the Load Application Code dialog is not automatically displayed, select **Load Application Code** from the **File** menu.

   Select `%PVLIB_HOME%\images\dhrystone.axf` as the application.

   Note

   You must have installed the Third Party IP add-on for the Fast Model Portfolio.

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

9. A Model Debugger Application Input pop-up window appears. Type `1000000` into the **Application input** field. See Figure 2-22.

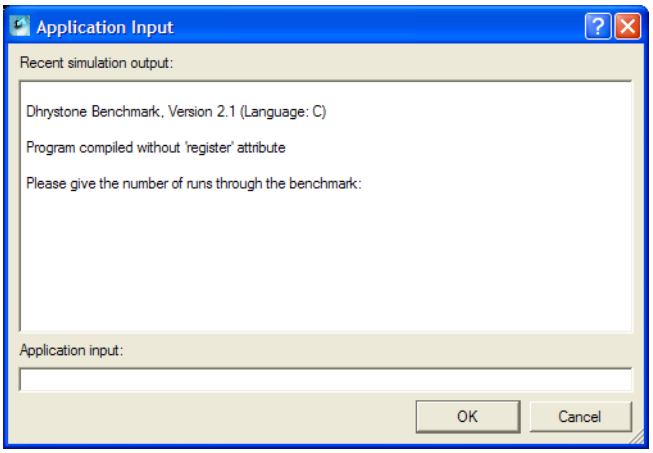

Figure 2-22 Model Debugger Application Input window

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

This tutorial used existing library components to create a system. Creating new components is described in Chapter 6 *Creating a New Component*.

2.11 Building an ISIM target

You can build an *Integrated Simulator* (isim) target by checking the **Integrated simulator** checkbox under the **Targets** option in the Project Settings dialog.

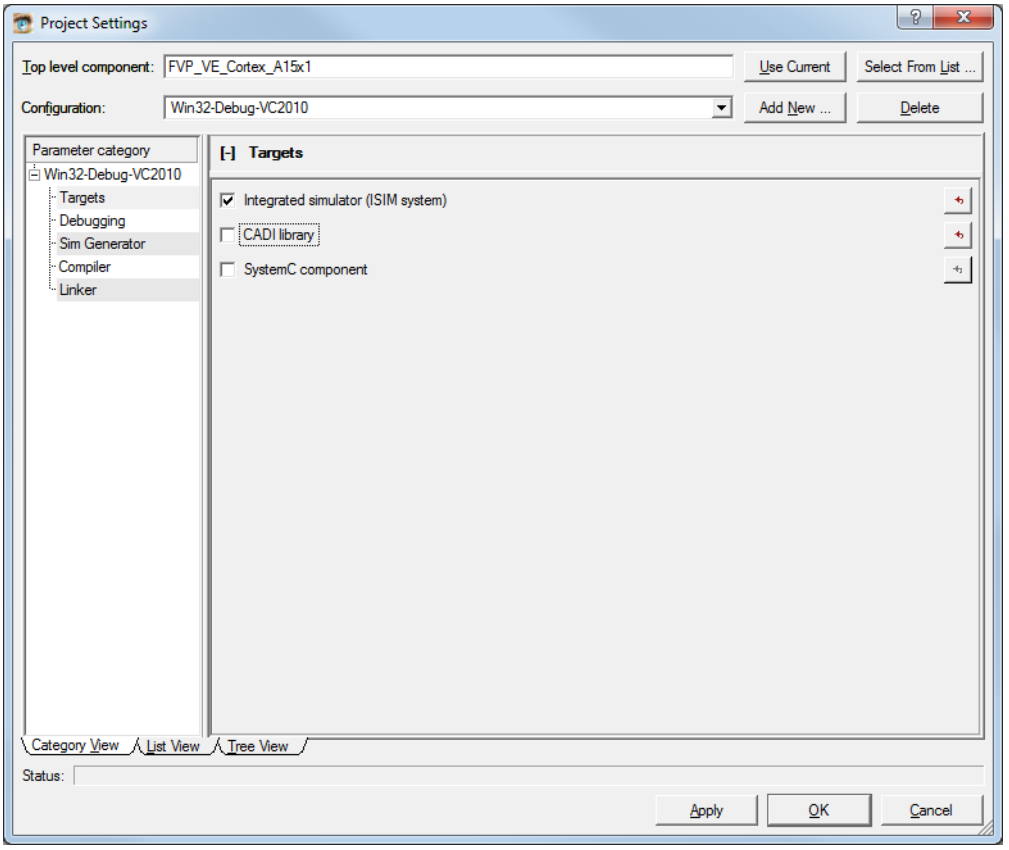

Figure 2-23 Build Integrated Simulator target

An isim target is generated by statically linking the Model Shell executable with a CADI simulation library. Integration of the executable and library simplifies host-level debugging and profiling.

2.11.1 Overloading the main() function in the target

The generated isim executable target has its own `main()` function. To overload the default `main()` with a user-supplied `main()`:

- use the Project Settings dialog to define the `USER_DEFINED_ISIM_MAIN` preprocessor option for compiler.

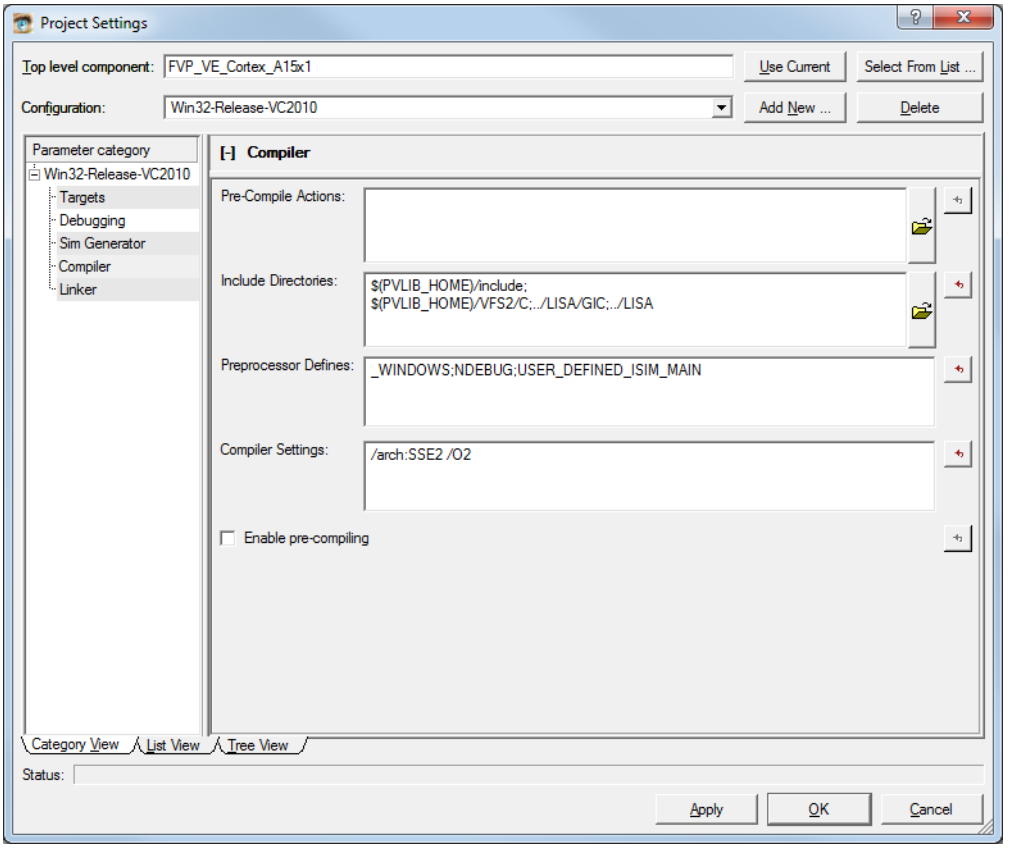

Figure 2-24 Specifying user-defined main() option

- supply a C++ file or a library with user-defined `main()` function. A fragment of the standard `IsimMain.cpp` file is shown in Example 2-1 for comparison:

Example 2-1 Standard IsimMain()

```
#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()`.

- Add the new source file containing the custom `main()` to the project. See *Add Existing Files and Add New File dialogs (Component window)* on page 4-18.

  Caution

  If the option `USER_DEFINED_ISIM_MAIN` and a user-supplied `main()` are used, 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.

2.11.2 Command line switches for the executable target

An isim target includes the Model Shell executable, so all Model Shell command line options apply to isim executable except `--model`. The model is integrated into the executable so there is no requirement to specify a model for the isim executable.

2.12 Setting-up a TAP network connection and configuring the networking environment for Microsoft Windows

This section describes how to set up a network connection for your PC, and then to configure the networking environment for use by Fast Models on a Microsoft Windows platform.

Note

You must ensure that you first install the third-party IP package in the Fast Models networking environment.

2.12.1 Setting-up a network connection

To set up the network connection:

1. Close all non-essential applications.
2. Install the TAP driver and configure it:
   - Access the base location of Fast Model Portfolio, default `C:\Program Files\ARM\FastModelPortfolio_X.Y`
   - Run either:
     - `ModelNetworking\add_adapter_32.bat` for 32-bit Microsoft Windows
     - `ModelNetworking\add_adapter_64.bat` for 64-bit Microsoft Windows.
3. Select **Start → Control Panel → Network Connections** and locate the newly-installed TAP device.
4. Press the **Ctrl** key to multi-select at least one real Ethernet adapter connection.
5. Right-click and select **Bridge Connections**.

Note

You only have to perform this procedure once, at the beginning.

2.12.2 Configuring the networking environment

Use System Generator to load the required project and build a model, or use Model Shell or Model Debugger to start a pre-built model.

Set the following parameters on the **HostBridge** and **SMSC_91C111** components. Enter:

```
hostbridge.interfacename="ARM<x>"
smsc_91c111.enabled=1
```

where **ARM<x>** is the adapter built into the network bridge, and is 0 by default. If ARM0 already exists, then use the next available integer, that is, 1, 2, 3...

Note

This procedure has not been tested with wireless network adapters.

Note

If you have to enable promiscuous mode for the TAP device, you must set an additional parameter on the device. To do this:

1. Select **Start → Run → cmd.exe**

2. Enter:

```
netsh bridge show adapter
netsh bridge set adapter <N> forcecompatmode=enable
```

where **<N>** is the id number of the TAP adapter listed by the first command.

Note

Firewall software might block network traffic in the network bridge, and might result in a networking failure in the model. If the model does not work after configuration, check the firewall settings.

2.12.3 Usage of tap_setup_32.exe or tap_setup_64.exe

The following are example commands for use with `tap_setup_32.exe <commands> <params>...`:

help Help information.

set_media Configure the TAP devices to `Always Connected`.

set_perm Configure the TAP devices to `Non-admin Accessible`.

restart Restart the TAP devices.

list_dev List available TAP devices and output to a file.

rename Rename the device to **ARMx**.

install <inffile> <id> Install a TAP Win32 adapter.

remove <dev> Remove TAP Win32 adapters. Set **<dev>** to `All` to remove all tap devices.

setup <inffile> <id> Automated setup process.

2.12.4 Uninstalling

To uninstall all TAP adapters, run either:

- `ModelNetworking\remove_all_adapters_32.bat` for 32-bit Microsoft Windows
- `ModelNetworking\remove_all_adapters_64.bat` for 64-bit Microsoft Windows.

Do this from the base location of Fast Model Portfolio, default `C:\Program Files\ARM\FastModelPortfolio_X.Y`.

If the uninstallation does not delete the bridge, you can delete it manually:

1. Close all non-essential applications.
2. Select **Network Connections**.
3. Right-click on the bridge and select **Delete**.

2.13 Setting-up a network connection and configuring the networking environment for Linux

This section describes how to set up a network connection, and then to configure the networking environment for use by Fast Models on a Linux platform.

The following instructions assume that your network provides IP addresses by DHCP. If this is not the case, consult your network administrator.

2.13.1 Setting-up a network connection

To set up the network connection:

1. Ensure that you have installed the `brctl` utility on your system.

   Note

   This utility is included in the installation package, but you are recommended to use the standard Linux bridge utilities, which are included in the Linux distribution.

2. In a shell, change to the `FastModelPortfolio` directory of your Fast Models installation. For example, `cd /FastModelPortfolio_X.Y/ModelNetworking`
3. You must run the following scripts from the directory in which they are installed, because they do not work correctly if run from any other location:
   - for a 32-bit operating system, run `add_adapter_32.sh` as root. For example, `sudo ./add_adapter_32.sh`
   - for a 64-bit operating system, run `add_adapter_64.sh` as root. For example, `sudo ./add_adapter_64.sh`
4. At the prompt **Please specify the TAP device prefix:(ARM)**, select **Enter** to accept the default.
5. At the prompt **Please specify the user list**, type a space-separated list of all users who are to use the model on the network, then select **Enter**. All entries in the list must be the names of existing user accounts on the host.
6. At the prompt **Please enter the network adapter which connect to the network:(eth0)**, select **Enter** to accept the default, or input the name of a network adapter that connects to your network.
7. At the prompt **Please enter a name for the network bridge that will be created:(armbr0)**, select **Enter** to accept the default, or input a name for the network bridge. You must not have an existing network interface on your system with the selected name.
8. At the prompt **Please enter the location where the init script should be written:(/etc/init.d/FMNetwork)**, select **Enter** to accept the default, or input another path with a new filename in an existing directory.
9. At the prompt **WARNING: The script will create a bridge which includes the local network adapter and tap devices. You may suffer temporary network loss. Do you want to proceed? (Yes or No)**, check all values input so far, and type **Yes** if you wish to proceed. If you type **No**, no changes are made to your system.
10. At the prompt that informs you of the changes that the script is to make to your system, input **Yes** if you are happy to accept these changes, or input **No** to leave your system unchanged.

Note

After entering **Yes**, you might temporarily lose network connectivity. Also, the IP address of the system might change.

11. The network bridge is disabled after the host system is reset. If you want the host system to be configured to support FVP bridged networking, you might have to create links to the `init` script (FMNetwork) in appropriate directories on your system. The script suggests some appropriate links for Red Hat Enterprise Linux 5.

Note

You only have to perform this procedure once, at the beginning.

2.13.2 Configuring the networking environment

Using System Canvas or a related Fast Models tool, load the required project or model and select your component.

Set the following parameters on the **HostBridge** and **SMSC_91C111** components. Enter:

```
hostbridge.interfacename="ARM<username>"
smsc_91c111.enabled=1
```

where **ARM<username>** is the adapter built into the network bridge.

Note

Firewall software might block network traffic in the network bridge, and might result in a networking failure in the model. If the model does not work after configuration, check the firewall settings.

2.13.3 Disabling and re-enabling networking

You can disable FVP networking without having to uninstall it. Use the installed `init` script (by default, `/etc/init.d/FMNetwork`) for this purpose.

To disable FVP networking, invoke the `init` script as root with the parameter `stop`. For example, `sudo /etc/init.d/FMNetwork stop`. To re-enable FVP networking, invoke the `init` script as root with the parameter `start`. For example, `sudo /etc/init.d/FMNetwork start`.

Warning

These operations remove/restore TAP devices and the network bridge. There is a temporary loss of network connectivity and your IP address might change.

2.13.4 Uninstalling networking

To uninstall networking:

1. In a shell window, change to the `bin` directory of your Fast Models installation. For example, `cd /FastModelPortfolio_X.Y/ModelNetworking`
2. Run `uninstall.sh` as root, passing the location of the `init` script (FMNetwork) as an argument. For example, `sudo ./uninstall.sh /etc/init.d/FMNetwork`

   You must run this script from the directory in which it is installed, because it does not work correctly if run from any other location.

Warning

There is a temporary loss of network connectivity and your IP address might change.

The uninstall script removes everything that can be safely removed. It does not remove:

- symlinks to the `init` script
- `/sbin/brctl`

You must remove any symlinks that you have created. Removing `brctl` is optional.

2.14 User mode networking

User mode networking emulates a built-in IP router and DHCP server, and routes TCP and UDP traffic between the guest and host. It also uses the user mode socket layer of the host to communicate with other hosts. This allows the use of a significant number of IP network services without requiring administrative privileges, or the installation of a separate driver on the host on which the model is running.

To set up and use user mode networking, run the model with the following additional CADI parameters:

```
-C motherboard.hostbridge.userNetworking=true
-C motherboard.smsc_91c111.enabled=true
```

To map a host port to a model port, run the model with the following additional CADI parameters:

```
-C motherboard.hostbridge.userNetPorts="8022=22"
```

This example maps port 8022 on the host to port 22 on the model.

Note

- You can use only TCP and UDP over IP. ICMP (ping) is not supported.
- DHCP is only supported within the private network.
- You can only make inward connections by mapping ports on the host to the model. This is common to all implementations that provide host connectivity using NAT.
- Operations that require privileged source ports, for example NFS in its default configuration, do not work.
- If setup fails, or the parameter syntax is incorrect, there is no error reporting.

2.14.1 HostBridge component

The HostBridge component is a virtual programmer's view model, acting as a networking gateway to exchange Ethernet packets with the TAP device on the host, and to forward packets to NIC models. This TAP/TUN method is an alternative to user mode networking.

The HostBridge component has a number of configuration options:

userNetPorts

Used to specify an optional port number mapping between listening ports on the host and listening ports on the model. The syntax is a comma-separated list of items in the form `[host-ip:]hostport=[model-ip:]modelport`. For example, `5022=22,5080=80` causes sshd and httpd services on the model to listen on ports 5022 and 5080 on the host machine. You can use the optional host-ip or model-ip to select a specific interface on which the mapping is to occur, for example `127.0.0.1`. The default is to accept connections on any interface `(INADDR_ANY)`.

userNetSubnet

Used to specify a range of network addresses that are presented to the model as if it was connected to that subnet. This is in the `ip-address/significant-bits` syntax, for example `192.168.0.0/24` or `10.0.0.0/8`. You can omit trailing zero octets, so `192.168/16` for example is acceptable. The default subnet is `172.20.51.0/24`, taken from the RFC1918 private-use area.

userNetworking

If set to `true`, the outgoing Ethernet packets of the model are directed to a proxy router that translates them into host socket calls. If set to `false` (default) the model attempts to open a TAP/TUN device.

For more information on the HostBridge component, see the *Fast Models Reference Manual*, http://infocenter.arm.com/help/topic/com.arm.doc.dui0423-/index.html.

Chapter 3
Debugging

This chapter describes how to debug component and system designs. It contains the following sections:

3.1 About debugging

After a system has been generated, there are two ways to debug applications that are running on it:

- Start the Model Debugger GUI directly from System Canvas. This is the most convenient way to debug the system. See *Debugging from System Canvas* on page 3-3.
- Run the debugger from a debug server, using the Model Shell application. This method is useful for running long or repetitive tests on the same system. See *Batch mode debugging* on page 3-9.

You can debug the LISA+ code that defines the model itself. See *Using other debuggers to debug LISA source* on page 3-11.

3.2 Debugging from System Canvas

After you have built your model, it can be simulated and debugged with Model Debugger.

For more information on using Model Debugger, see the *Model Debugger for Fast Models User Guide*, http://infocenter.arm.com/help/topic/com.arm.doc.dui0314-/index.html.

Note

You can use RealView Debugger instead of Model Debugger. See *Using RealView Debugger* on page 3-7.

To use Model Debugger:

1. If it has not already been configured, ensure that System Canvas can locate the executable for Model Debugger. See *Configuring Model Debugger* on page 3-6.
2. Click the **Debug** button on the System Canvas toolbar to launch Model Debugger.
3. Click the **Debug** button in the System Canvas toolbar. See Figure 3-1:

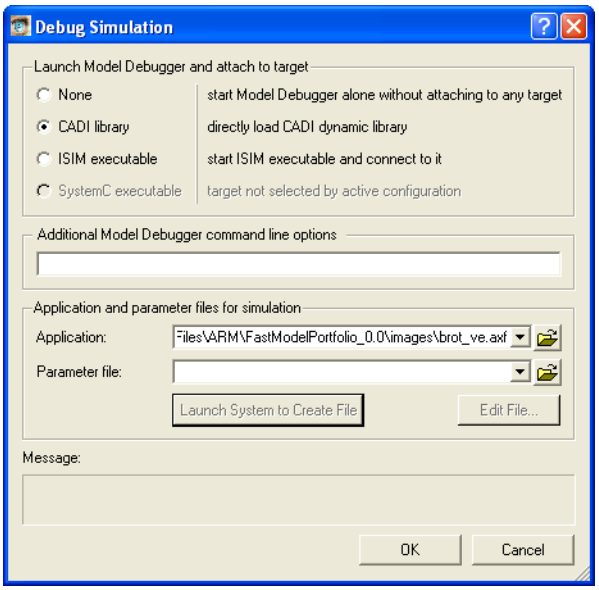

Figure 3-1 Debug Simulation dialog

4. In the **Debug Simulation** dialog, select the required radio buttons to launch Model Debugger and attach to your target. The radio buttons that are available depend on the settings that you made in the Project Settings dialog.

   If required, enter additional command line options by using the **Additional Model Debugger command line options** field. You can also set the application and parameter files to be simulated by using the **Application** and **Parameter file** drop-down menus.

5. Click **OK**. The Model Debugger Configure Model Parameters dialog opens to show the parameters that you can configure. See Figure 3-2 on page 3-4. Make any required changes, then click **OK**.

   Note

   The names of the panes in the dialog might be different to those shown in Figure 3-2 on page 3-4, depending on the model you are using, but the purpose and usage is the same in all cases.

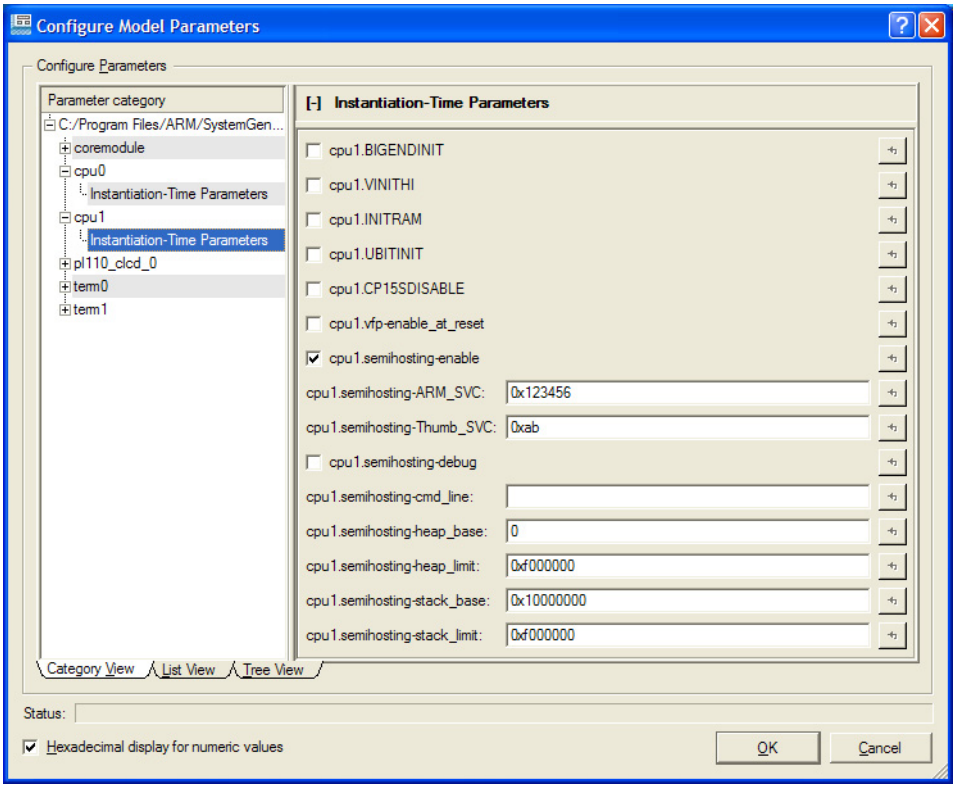

Figure 3-2 Configure Model Parameters dialog

6. The Model Debugger Select Targets dialog opens and displays the debuggable components that are available for connection to Model Debugger. See Figure 3-3 on page 3-5.

   Select the targets to load by clicking the box next to it and clicking **OK**. One or more instances of Model Debugger are then created, depending on how many targets you selected to load.

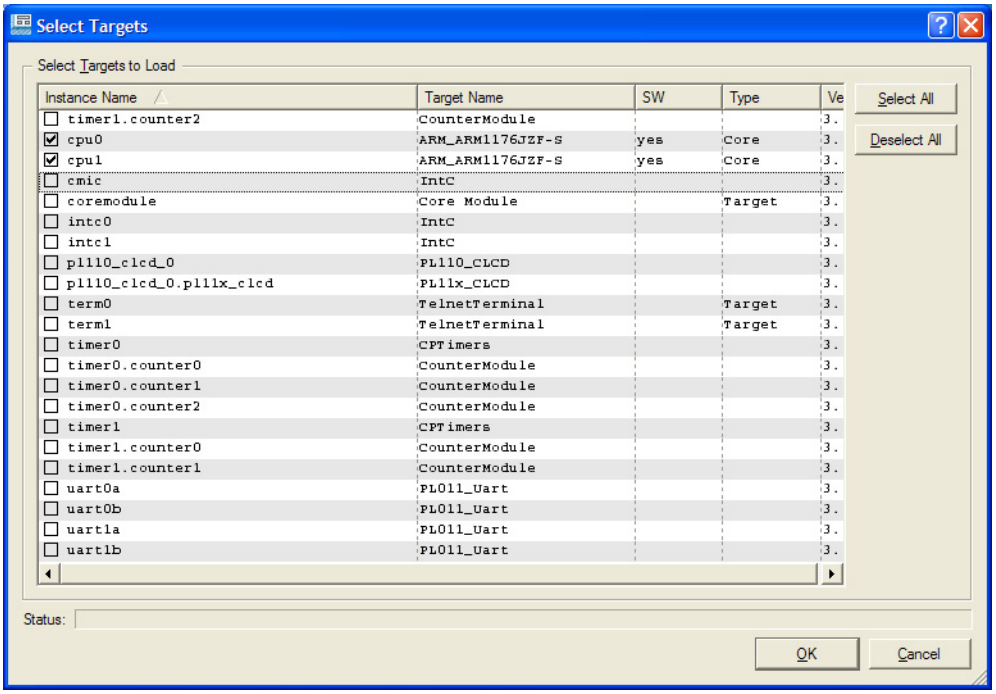

Figure 3-3 Select Targets dialog

7. For each instance of Model Debugger connected to a processor:
   a. Use the Load Application dialog shown in Figure 3-4 to specify the application to load for the processor.

      Note

      If there is only one processor in the model, the dialog is typically displayed after the target is selected. If the dialog is not automatically displayed, select **Load Application Code** from the **File** menu.

   b. Select the application code (`.axf`) to load for the processor.
   c. Click **Open** to select the file and close the dialog.

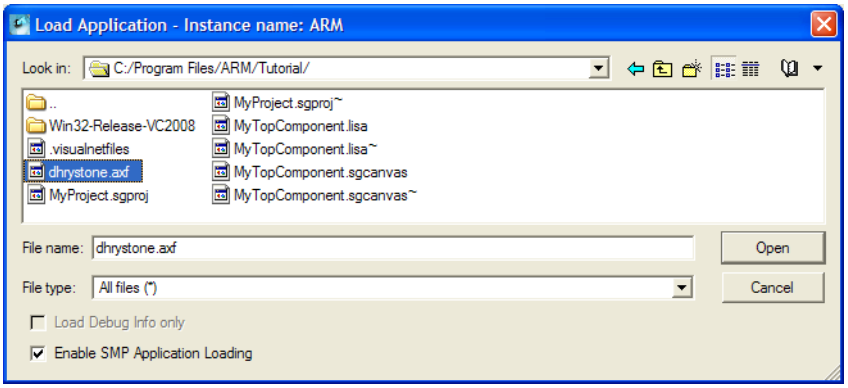

Figure 3-4 Load application for processor

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

An example of a dual processor system connected to two instances of Model Debugger is shown in Figure 3-5 on page 3-6. One processor is running ARM embedded Linux, and the other is running the `brot.axf` example.

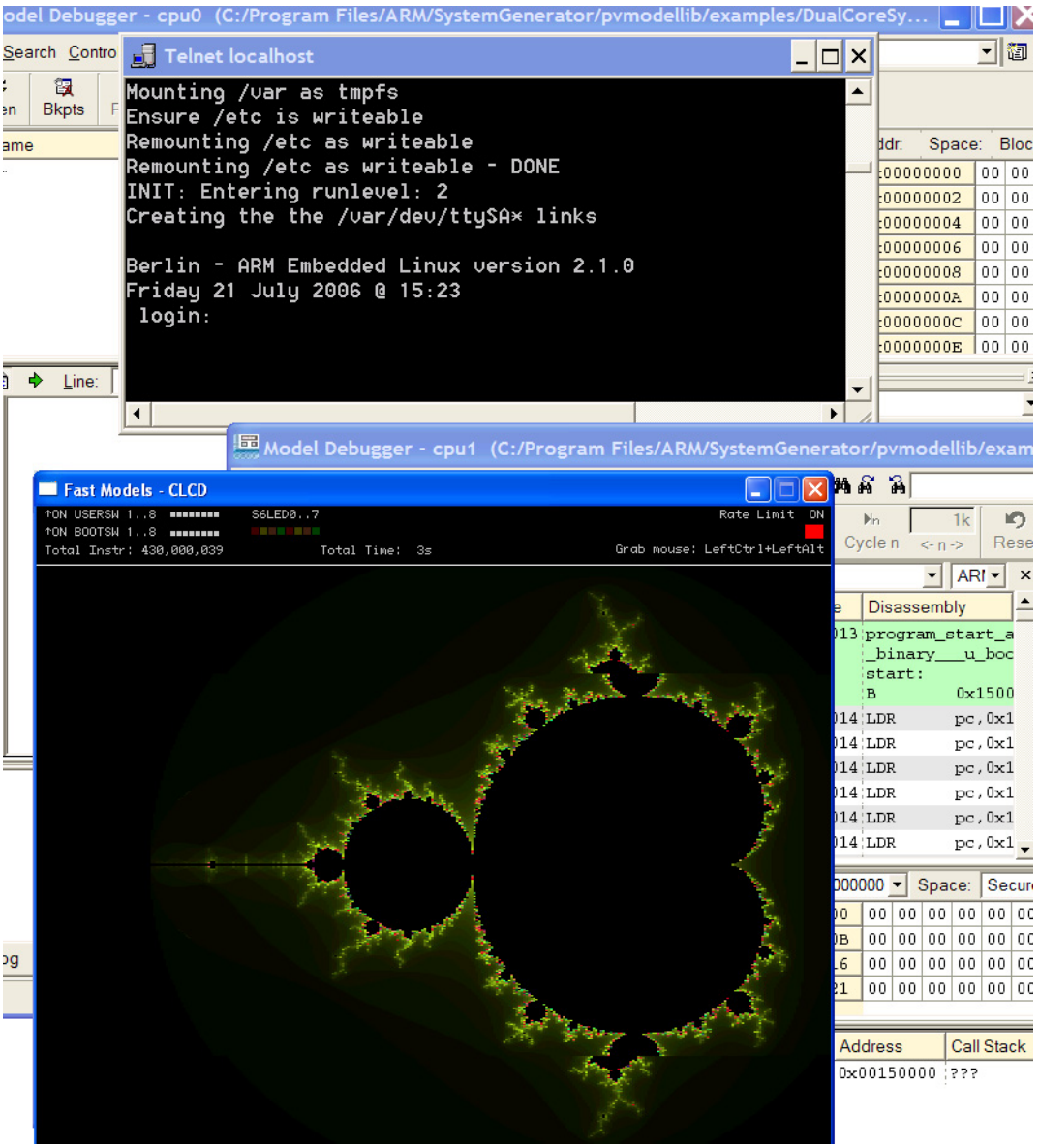

Figure 3-5 Dual processor system running in Model Debugger

3.2.1 Configuring Model Debugger

The Fast Models installation includes System Canvas and Model Debugger and sets the correct path.

If required, you can change the path by using the Preferences dialog:

- Click **File** → **Preferences** → **Applications** to open the dialog.

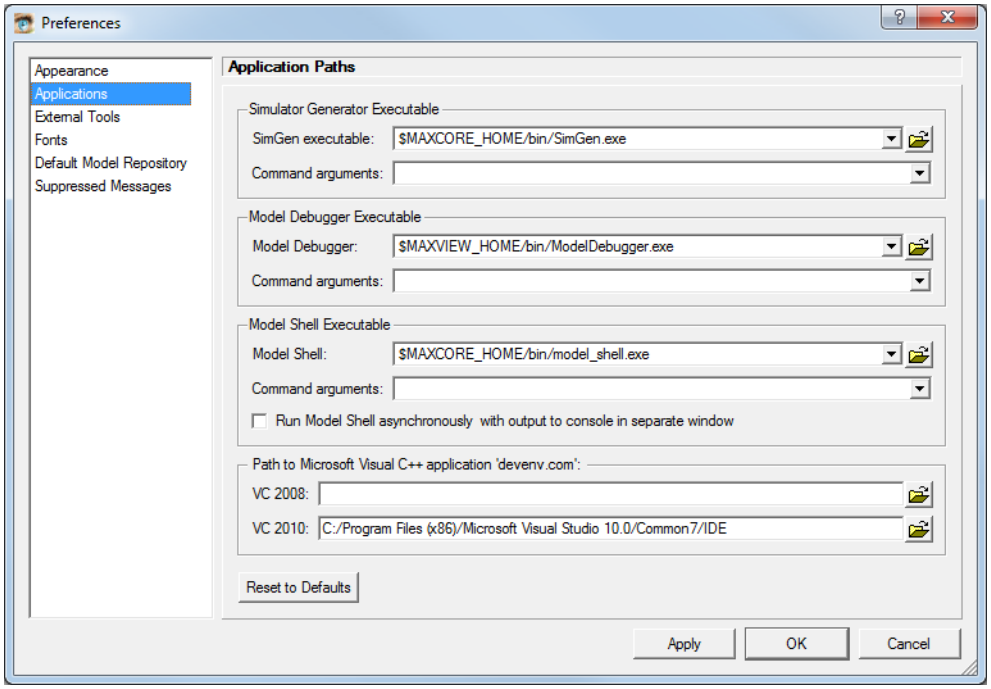

Figure 3-6 Model Debugger paths in the Preferences dialog

- Click on the **Model Debugger** or **Model Shell** drop-down lists to select the executable you use. You can also specify Model Shell or Model Debugger command line options here. See also the *Model Debugger for Fast Models User Guide*, `http://infocenter.arm.com/help/topic/com.arm.doc.dui0314-/index.html`.

3.2.2 Using RealView Debugger

As an alternative to Model Debugger, you can debug your model using RealView Debugger version 3.1 or later. See the following sections for more information on the ways that you can connect to your model:

- *Starting model as an FVP connection*
- *Connecting to a running model using RealView Debugger* on page 3-8.

More information on how to use RealView Debugger is provided elsewhere. See the *RealView Debugger User Guide*.

Note

You must not connect to more than one virtual platform model or FVP at any one time in RealView Debugger. If you do, you might experience unexpected behavior, including crashes.

Starting model as an FVP connection

You can add your model to the RealView Debugger Connect to Target window under the ModelLibrary debug interface configuration.

1. In RealView Debugger, click **Target** → **Connect to Target...** to open the Connect to Target window.

2. Click the **Add** button beside the ModelLibrary debug interface name. This opens the Model Configuration Utility window. Click the **Browse...** button to open a file browser for locating your model `.dll` or `.so` file.

3. Select the model to use in the Models pane on the left side of the Model Configuration Utility. Configure the device parameters if required by selecting the device in the upper right Devices pane, and setting the parameters in the lower right Parameters pane. When you have finished, or if no parameters require configuration, click **OK**.

4. In the RealView Debugger Connect to Target window, double click on your newly-created target to connect to it. If you are grouping targets by Configuration, expand the target connection tree view to see your target instance. Connecting to a target opens a CLCD window, where you can, for example, enable Tortoise before loading an image.

Connecting to a running model using RealView Debugger

You can use RealView Debugger to connect to an already running Model Shell instance. You can make multiple debugger connections to a single model instance.

1. Start Model Shell, if it is not already running. See *Batch mode debugging* on page 3-9.

2. In RealView Debugger, click **Connect to Target...** from the **Target** menu to open the Connect to Target window.

3. Click the **Add** button beside the required target name. The debugger detects any running CADI servers and displays them in a pop-up window. Click **OK**.

4. In the RealView Debugger Connect to Target window, double click on your newly-created Model Process target to connect to it.

   Note

   You cannot define instantiation-time parameters at connection time because the model is already running.

3.3 Batch mode debugging

For batch mode debugging, use Model Shell to start and simulate systems from the command line.

Model Shell is a debugging host that can be used with local or networked models. To start Model Shell from the System Canvas GUI, select **Run** from the **Project** menu or click the **Run** icon on the toolbar. For more information on Model Shell, see the *Model Shell for Fast Models Reference Manual*, `http://infocenter.arm.com/help/topic/com.arm.doc.dui0457-/index.html`.

3.3.1 Direct execution

Direct execution is used to start a standalone simulation directly from the command line, for example to perform regression tests that do not require interaction:

```
model_shell -m <model library >
```

This is illustrated in Figure 3-7. Simulation of the system model begins immediately.

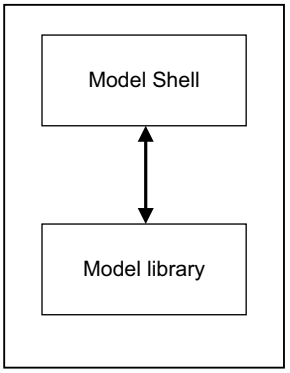

Figure 3-7 Direct execution use case

3.3.2 Debug server

Model Shell can be started with a debug server by using the `-S` command line option with a CADI server. This enables CADI-compliant debuggers to connect to the simulation and take control of it.

After starting, the debugger can connect to the simulation and debug the model as though the model had been directly loaded into the debugger. The connection can be either local or remote across a network. See Figure 3-8 on page 3-10. For more information on connecting to CADI and using a debugger, see the documentation supplied with your debugger.

Multiple debuggers can connect to a single model shell instance. This is useful for systems that consist of multiple processors, for example the dual processor system.

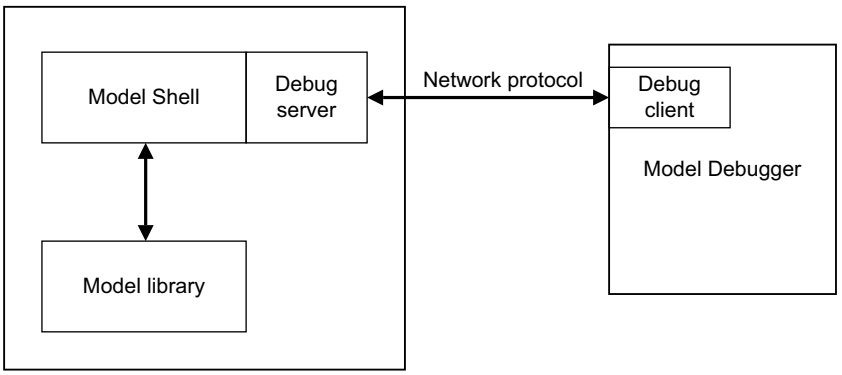

Figure 3-8 Debug server use case

3.4 Using other debuggers to debug LISA source

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

The generated simulation code is annotated with line redirections to the corresponding LISA+ code. This enables debugging a LISA+ model with all the debug features provided by GDB or Microsoft Visual Studio.

For both GDB and Microsoft Visual Studio, you can start model execution in one of the following ways:

- Directly start the model in the debugger

  This method is easier for an ISIM system. It is especially useful if the user provided a new implementation of `main()`.

- Dynamically connect to an already running simulation process.

3.4.1 Model generation requirements

To debug the model, build a system model library for CADI-compliant debuggers:

1. Start System Canvas and load the project for your model.
2. In the **Select Active Project Configuration** drop-down menu, select a valid debug configuration for your build environment. For example, on Microsoft Windows, if you have Microsoft Visual Studio 2010 installed, this would be `Win32-Debug-VC2010` or `Win64-Debug-VC2010`.
3. Click the **Build** button to build a debug version of your model.

3.4.2 Debugging with GDB

To debug ISIM targets, start a GDB session directly from Fast Models Canvas by selecting **Launch Host Debugger** from the **Project** menu. Command line arguments can be specified on the **Applications** page of the Preferences Dialog.

Note

You can also debug CADI shared library from either the Model Shell or Model Debugger executable. This method of debugging is not directly available from System Canvas, but you can use a Linux shell to start the executable.

To start the GDB session with Model Debugger as the executable:

1. Load Model Debugger into GDB by typing `gdb modeldebugger.exec` in the console.

   Note

   Ensure that you use GDB version 6.2 or higher.

   Starting `modeldebugger.exec` might require manually modifying the `LD_LIBRARY_PATH` environment variable to include a reference to the directory containing `libcadiclient.so`. This directory is `$MAXVIEW_HOME/lib`.

2. Type `run` at the GDB prompt to start Model Debugger.

   You can load the model in the same step by adding the name of model library, for example:

   ```
   run cadi_system_Linux-Debug-GCC-4.1.so
   ```

3. Model Debugger starts and the Configure Model Parameters dialog opens.
4. Configure any required parameters for the target, then click **OK**.
5. The Select Targets dialog opens. The ARM processor is selected by default. Select the additional targets, if any, to load into Model Debugger. A separate Model Debugger window is created for each target. Click **OK** to close the target selection dialog.
6. Load the applications to the targets that execute software:
   - If there is one target selected, a Model Debugger prompt for the application is displayed.
   - If more then one target was selected, load the application to the targets in each debug view separately by selecting **Load Application** from the **File** menu.

   Note

   Selecting the application can also be done when Model Debugger is started by adding command line option `-a` in line with the name of the target and application file:

   `run cadi_system_Linux-Debug-GCC-4.1.so -a targetName1=application1 -a targetName2=application2`
7. GDB is now ready to debug the model source code.

Example GDB LISA debug session

This section describes a typical example of debugging a system model on the source code level and setting and running to a breakpoint in the LISA source.

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

   ```
   break MyCode.lisa:123
   ```
3. In GDB, continue execution of Model Debugger by typing:

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

You can use GDB to perform any debug action on the LISA code such as printing values of variables or stepping through code.

3.4.3 Debugging with Microsoft Visual Studio

Use Microsoft Visual Studio to perform LISA+ source-level debugging on Microsoft Windows. ARM recommends that you generate the system and launch Microsoft Visual Studio from System Canvas:

- Start System Canvas and load the project for your model.
- If you have not already done so, build a Debug version of your system by selecting one of the Win32-Debug-VC20*xx* or Win64-Debug-VC20*xx* options in the **Select Active Project Configuration** menu and rebuilding your model.

  Note

  If you select **Generate**, you can use Microsoft Visual Studio later to build the model.

- Launch Microsoft Visual Studio from System Canvas by clicking the **Devenv** button or pressing **Alt**+**F5**.
- From Microsoft Visual Studio, select **StartUp project** to select the target to debug. The target file can be either an ISIM or CADI dll.
- Use the Project Properties dialog to configure any required parameters for the target.

  Note

  For ISIM systems, all Model Shell command lines options are available. Specifying the model on the command line is not required.

  For a CADI dll, you must specify either Model Debugger or Model Shell as the executable to run.

- Start debugging by selecting **Run** in Microsoft Visual Studio.

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

1. Start System Canvas and load the project for your model.
2. If you have not already done so, build a Debug version of your system by selecting one of the Win32-Debug-VC20*xx* or Win64-Debug-VC20*xx* options in the **Select Active Project Configuration** menu and rebuilding your model.
3. Click the **Debug** button or pressing **F5** to launch Model Debugger from System Canvas.
4. Model Debugger starts and the Configure Model Parameters dialog is displayed.
5. Configure any parameters required for the target and click **OK**.
6. The Select Targets dialog opens. The ARM processor is selected by default. Select the additional targets, if any, to load into Model Debugger. A separate Model Debugger window is created for each target. Click **OK** to close the target selection dialog.
7. Load the applications to the targets that execute software:
   - If there is one target selected, Model Debugger displays a dialog prompting for the application.
   - If more then one target was selected, load the application to the targets in each debug view separately by selecting **Load Application** from the **File** menu.
8. Start Microsoft Visual Studio.
9. In Microsoft Visual Studio, select **Tools → Attach to Process...**. Select the `ModelDebugger.exe` process in the Attach to Process dialog. Click the **Attach** button. It might be necessary to close additional dialogs before you can proceed.
10. Microsoft Visual Studio is now attached to Model Debugger and can control the entire host level simulation.

Example Microsoft Visual Studio LISA debug session

This section describes a typical example of debugging a system model on the source code level and setting and running to a breakpoint in the LISA source.

1. In Microsoft Visual Studio, select **File → Open → File...** to open the LISA source file to debug.
2. Set a breakpoint in the LISA source by double clicking on a source line.

3. In Model Debugger, click the **Run** button to start execution. When the breakpoint is reached, use Microsoft Visual Studio to perform host-level debugging of the LISA+ source code.

3.5 Working with the ARM Profiler

Virtual platforms produce with Fast Models generate profiling data that can be used by ARM Profiler. This data can be visualized using ARM Workbench IDE, which is part of *RealView Development Suite* (RVDS). This chapter explains how to set up the virtual platform with ARM Profiler and how the data is visualized in the ARM Workbench IDE. For more information, consult the ARM Profiler documentation.

To extract data from the model a utility, `armprocap`, is required.

3.5.1 ARM Profiler on Microsoft Windows workstation

ARM Profiler is installed as part of the RealView Developers Studio 4.0sp2 or later. On Microsoft Windows the ARM Profiler is typically installed into the standard Program Files directory. Depending on your local settings the executable (`armprocap.exe`) is installed in:

`<RVDS_Installation_Folder>\Profiler\tools\<version>\windows\armprocap.exe`

This document refers to this executable as `armprocap`.

3.5.2 ARM Profiler on Linux Workstation

ARM Profiler is installed as part of the RealView Developers Studio 4.0sp2 or later. On Linux workstations the location of `armprocap` is in:

`<RVDS_Installation_Folder>/Profiler/tools/<version>/linux/armprocap`.

This document refers to this executable as `armprocap`

3.5.3 Running ARM Profiler

`armprocap` must be called with a set of options to indicate that the ARM Profiler is executing an virtual platform or EVS. The syntax is:

```
armprocap ---create_connection_script <CON_SCRIPT> --models --output <PROFILING_OUTPUT>
--load <APPLICATION>
```

where:

`PROFILING_OUTPUT`	defines the filename where the profiling data is stored
CON_SCRIPT	points to the location of a script file that `armprocap` creates
`APPLICATION`	points to the location of the application which must be loaded onto the virtual platform or EVS.

3.5.4 Running the ARM profiler with Dhrystone example

The following command executes the dhrystone EVS with the ARM Profiler. To run this example, make sure you have built the Dhrystone EVS. Replace `armprocap` with the fully qualified path to `armprocap` in your installation.

Example 3-1 Running ARM Profiler with Dhrystone EVS

```
armprocap --models --output test --create_connection_script myscript.sh --load
$PVLIB_HOME/dhrystone.axf

source ./myscript.sh
```

```
./Dhrystone.x $PVLIB_HOME/dhrystone.axf
```

While running the simulation `armprocap` prints messages similar to the following:

```
Loading symbols from dhrystone.axf...
Finished loading symbols.
Transferring target image
Enabled streaming trace.

ARM Profiler (Build: Jun 24 2009, 01:00:06)
Writing ARM Profiler analysis file to 'test.apd'.
Please wait.
Done. (0.2 seconds)


Disconnect...
```

This generates a folder called `test.apd` which can be loaded into the ARM Profiler as follows:

1. Open ARM Workbench IDE 4.0
2. If the **ARM Profiler Data** tab is not already displayed, go to: **Window → Show View → Other → ARM Profiler → ARM Profiler Data**
3. In the **ARM Profiler Data** tab, click on the Folder icon at the top right-hand side. It has the tooltip **Edit locations**. Add the directory containing your Profiling directory.
4. Your Profiling directory displays in the list.

Chapter 4
System Canvas Reference

This chapter describes the windows, menus, dialogs, and controls present in System Canvas. It contains the following sections:

4.1 Launching System Canvas

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

To start System Canvas from the command line, at the prompt type `sgcanvas`. You can optionally define additional parameters as listed in Table 4-1.

Table 4-1 System Canvas command line options

Short form	Long form	Description
`-h`	`--help`	Prints help text and exits.
`-v`	`--version`	Prints version and exits.

4.2 Overview of System Canvas

System Canvas consists of the following windows and graphical elements:

Main menu The main menu contains the available options with their corresponding keyboard short cuts. See *Menu bar*.

Toolbar The tool bar contains buttons for frequently-used features. See *Toolbar* on page 4-11.

Workspace The workspace panel has tabs to select the views:

- the **Block Diagram** panel contains the components, ports, and connections
- the **Source** panel is used to edit the LISA code of the component.

See *Workspace windows* on page 4-13.

Component list

The component list contains a list of all components and their protocols and libraries available in the current project. See *Component window* on page 4-15.

Output window

The output window contains status messages that are output from the build process.

Status bar The status bar gives information about menu items, commands, buttons, and component information.

The main elements of the System Canvas window are shown in Figure 4-1.

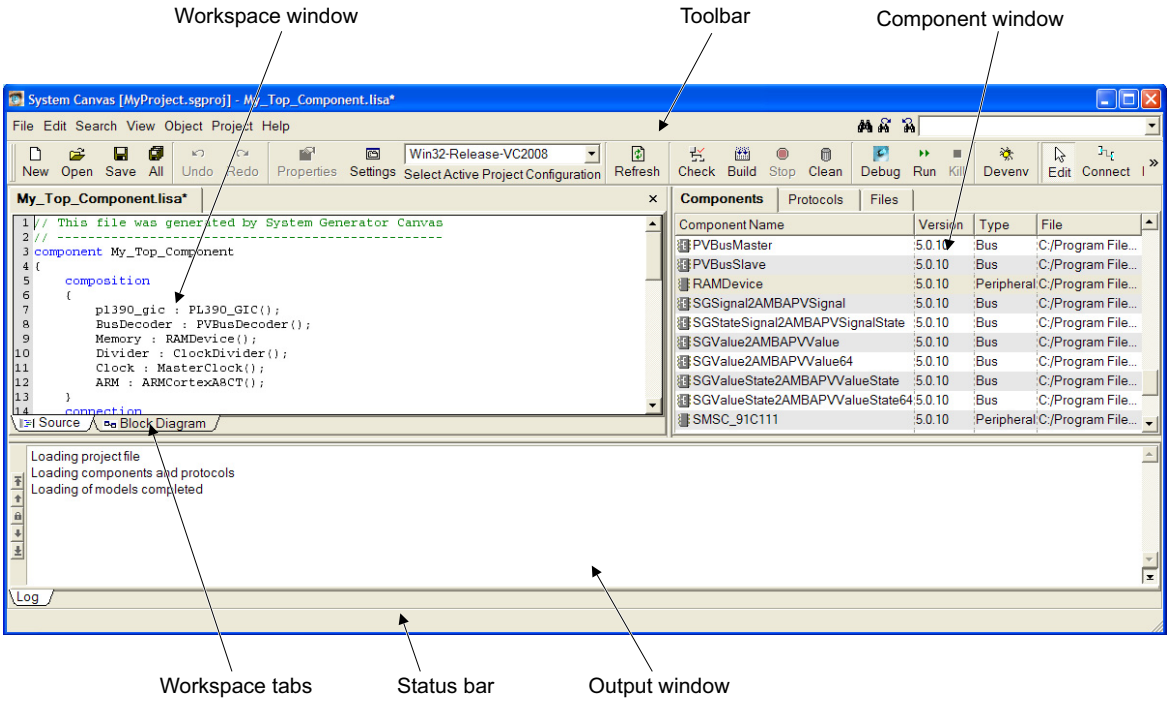

Figure 4-1 Layout of System Canvas

4.2.1 Menu bar

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

File menu

The **File** menu has the following options:

New Project Use the New Project entry to create a new model project. See *New Project dialogs on page 4-44*.

Load Project

This option opens an existing project.

Close Project

This operation closes the currently-opened project. If there are pending changes, the Save changes dialog is displayed.

Save Project

This command saves the changes made to the current project.

Save Project As

This command saves the current project to a new location and name.

New File This displays the New File dialog to create a new file. Select the type from the **File type** drop-down list.

Open File This displays the Open File dialog to create a new file. Filter the types to display by selecting the type from the **File type** drop-down list. If the file type is not LISA, you can open the file as text in the source editor.

Close File This operation closes the currently-open LISA file. If there are pending changes, the Save changes dialog is displayed.

Save File This command saves the changes made to the current LISA file.

Save File As

This command saves the current LISA file to a new location and name.

Save All This command saves the changes made to the current project and any modified LISA files.

Print This command prints the contents of the Block Diagram window. A dialog displays to enable you to select the destination printer.

Preferences

Use this option to modify the user preferences.

Recently Opened Files

A list of 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

A list of 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 End System Canvas. If you have modified files or projects, a dialog prompts you to save your changes. You can choose to not have the dialog presented in future by selecting **Do not show this message again**. To re-enable the dialog, edit the Preferences. See *Suppressed Messages* on page 4-58.

Edit menu

The **Edit** menu has the following options:

Undo Undo the last change to a file in the Source window or to the layout in the Block Diagram view. Up to 42 actions can be undone. The following actions can be undone:

- adding an object such as a component, label, or connection
- pasting or duplicating objects
- cutting or deleting objects
- editing object properties
- moving objects
- resizing objects.

Note

Undo and **Redo** operations can affect block diagram view zoom and scroll actions.

Undo and Redo typically work in the usual intuitive way, but the following special case requires explanation:

1. Change the system in the Block Diagram view by, for example, 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();` from the Source view.
4. Change the code by adding, for example, the line `PVS : PVBusSlave();` to the Source view.
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. This Block Diagram view shows the system as it was before step 2. `RAM` is present but `PVS` is not.
7. Select **Redo** from the **Edit** menu. This Block Diagram view shows the system as it was before step 6. `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. See the description for **Undo** for more information.

Cut Cut the marked element and place it into the copy buffer.

Copy Place 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. Components can be selected with the cursor.

Connect Ports Mode

Select connection mode. The cursor is used to connect components.

Pan Mode Select movement mode. The cursor is used to move the entire system in the Workspace window.

Search menu

The **Search** menu has the following options:

Find Selecting **Find** opens a dialog that permits searching for a string in a currently-active window. The Active window is highlighted by a thick black frame around it. See *Find and Replace dialogs* on page 4-39.

Find Next The last defined search is repeated to find the next occurrence.

Find Previous

The last defined search is repeated, but the search direction is backwards in the document.

Replace Selecting **Replace** opens the Find dialog with replace controls displayed. This lets you replace strings in a text document. The **Replace** feature is only available if the active window is a source editor. See *Find and Replace dialogs* on page 4-39.

Go To Line Selecting this option displays a dialog that enables you to specify a line number in the currently-open LISA file. This option is only enabled in the **Source** view.

Note

You can also use the search icons displayed at the top right of the application window to search for text. Enter the text to search for in the text box to the right of the search icons. As you enter text, an incremental search is performed in the active window.

View menu

The **View** menu has the following options:

Show Grid Display the grid in the Workspace window. Using the grid simplifies component alignment.

Zoom In Increase the magnification for the Workspace window to show part of the system in more detail.

Zoom Out Decrease the magnification for the Workspace window to show more of the system.

Zoom 100% Change the magnification for the Workspace window to the full workspace canvas area.

Zoom Fit Change the magnification for the Workspace window to fit the entire system into the workspace canvas area.

Zoom Fit Selection

Change the magnification for the Workspace window to fit the selected portion of the system into the workspace canvas area.

Object menu

Use the **Object** menu to modify the system or a component in the system. The **Object** menu has the following options:

Open Component

This option opens the source for the component that has been selected in the Workspace.

Add Component

This option displays a dialog that contains all the available components that can be added to the diagram.

Add Label This option changes the mouse cursor into a default label. To add a label, move the label to the required location and click the left mouse button. The label properties dialog is displayed to enable you to change the label text and properties.

Add Port This option displays the External Port dialog used to specify the type of port to add.

Mirror Self Port

This option switches the direction of drawing the external port. 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

If a port array is selected, this option is enabled. Select it to display all of the individual port elements. Port arrays with more than eight elements are collapsed by default.

Note

Ports with a large number of elements might expand so that all elements appear on top of one another. Click and drag to move the individual port elements to separate locations. Alternatively, collapse the port, increase the component size, then expand the port again.

Collapse Port

If a port array is selected, this option is enabled. Select it to hide the individual port elements and only display the top-level port name. Ports with eight or fewer elements are expanded by default.

Hide Port Use this option to disable the selected port and make it invisible in the Block Diagram window.

Hide All Unconnected Ports

This option hides all ports that are not connected to a component.

Show/Hide Ports of Protocol Types...

This option hides all ports that use a specified protocol. Select the protocols to filter from the displayed Show/Hide Connection Types dialog.

Show All Ports

This option shows all ports that have previously been hidden. This might result in overlapping port symbols if there is not enough space for displaying all of the hidden ports.

Autoroute Connection

This option removes any manually placed points or manually moved line segment end points from the currently-selected connection and reroutes the connection.

Autoroute All Connections

This option removes any manually placed points or manually moved line segment end points from all connections and reroutes the connections.

Documentation

This option opens the PDF file that contains the documentation for the selected component.

Object Properties

This option displays the Component Instance Properties dialog to view and edit the properties for the selected component.

Note

If a component is not selected, the Component Model Properties dialog is displayed.

Project menu

The **Project** menu has the following options:

Check System

Use this entry to check if the system being developed has any errors or missing information. This feature cannot check everything, but does give useful feedback.

Generate System

Generate a system. This action only generates the C++ source code, and does not compile it. This action is only useful in rare circumstances. After generation finishes, click **Build System** and **Debug** to run the model.

Build System

Compile the system files. This action generates and compiles the generated C++ source code, and produces a runnable model (or a library in the SystemC use case).

Stop Build Cancel the build process that is currently active.

Clean Delete all generated files.

Launch Model Debugger

Execute simulation under control of Model Debugger.

Run

Run... Open Run Dialog to specify command to run.

Run in Model Shell

Execute simulation under control of Model Shell with command line options taken from project settings and user preferences. For ow to configure the latter options see *Applications* on page 4-52.

Run ISIM system

Execute simulation as ISIM executable with Model Shell command line options taken from project settings and user preferences.

Run SystemC executable

Run SystemC simulation application specified in project settings (see *Building a SystemC component from System Canvas* on page 5-4).

Clear History

Clear all recent run command entries.

Recent Command Entries (up to 10)

Call recent command entries.

Kill Running Command

Stop currently-running synchronous command.

Launch Host Debugger

Launch the debugger. The behavior of this entry depends on the operating system:

- For Microsoft Windows, Microsoft Visual Studio is launched. A generated solution file is supplied as an argument that can generate any target. If the system has not already been built, that can be done in Microsoft Visual Studio.

  You can start a debug session from the Microsoft Visual Studio session. Command line arguments for isim systems or model_shell can be supplied by from Microsoft Visual Studio by selecting **Project → Properties → Configuration Properties → debugging**.
- For Linux, the executable or script defined in Application Preferences is launched (see *Preferences dialog* on page 4-50). Only an isim target can be specified as the argument. ARM recommends this as the preferred way to do source-level debugging.

Add Files Add an existing file 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

Displays the **Default Model Repository** section of the Preferences window. Use this dialog to select the repositories that are automatically included in the next new project.

Note

The currently-open project is not affected by changes to the default model repository list.

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, and you can set any component as the top 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 configuration to use to build the system from the list of project files.

Project Settings

Display the Project Settings dialog.

Help menu

The **Help** menu has the following options:

Fast Model Tools User Guide

This option opens this book in Adobe Acrobat Reader.

Model Shell Reference Manual

This option opens the PDF file that contains the Model Shell reference.

LISA+ Language Reference Manual

This option opens the PDF file that contains the language reference.

AMBA-PV Developer Guide

This option opens the PDF file that contains the AMBA-PV extensions guide.

CADI Developer Guide

This option opens the PDF file that contains the Component Architecture Debug Interface guide.

Release Notes

This option opens the text file that contains the release notes.

Documents in $PVLIB_HOME/Docs

This option lists the PDF files that are in the directory `$PVLIB_HOME/Docs`. The location syntax is the same on both Microsoft Windows and Linux. The `PVLIB_HOME` environment variable is set when the Fast Model Portfolio is installed.

End User License Agreement (EULA)

This option opens the HTML file that contains the license agreement.

About This displays the standard About dialog box displaying version and license information.

System Information

This displays a dialog that contains extended information about the tools and loaded models.

4.2.2 Toolbar

The Toolbar provides access to the most frequently-used functions that are also accessible through the Main Menu:

New Create a new project or LISA file. See *New File dialog (from File menu)* on page 4-42.

Open Open an existing project or file. See *Open File dialog* on page 4-46.

Save Save current changes to the file.

All Save project and all open files.

Undo Undoes the last change in the Source or Block Diagram view. See *Edit menu* on page 4-5.

Redo Redoes the operation that was cancelled by the last **Undo**.

Properties Display the Properties dialog for the selected component.

Note

The **Properties** button only displays properties for items in the Block Diagram:

- If nothing is selected, the Component Model Properties dialog opens. This gives you the properties for your top-level component. See *Component Model Properties dialog for system* on page 4-27.
- If a component is selected, the Component Instance Properties dialog opens. See *Component Instance Properties dialog* on page 4-23.
- If a connection is selected, the Connection Properties dialog opens. See *Connection Properties dialog* on page 4-34.
- If a port is selected, the Port Properties dialog opens. See *Port Properties dialog* on page 4-48.
- If a self port is selected, the Self Port Properties dialog opens. See *Self Port dialog* on page 4-80.
- If a label is selected, the Label Properties dialog opens. See *Label Properties dialog* on page 4-40.

Settings Display the project settings. See *Viewing the project settings* on page 2-15.

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 settings from the Project Properties.

Stop Stop the current generation process.

Clean Delete all generated files.

Debug Start Model Debugger to debug the generated simulator.

Run Clicking the button executes the most recent run command. The down arrow next to the button opens the Run Dialog.

Kill Stop Model Shell and end the simulation.

Devenv Open the project in the compiler. For Microsoft Windows, Microsoft Visual Studio is opened. For Linux, gdb is opened.

Note

The project and debugger that is opened is determined by the value of the **Select Active Project Configuration** drop-down menu. The project solution must be generated before using this button. If the solution is not found, the compiler opens without a project.

Edit Select edit mode. The cursor is used to select and move components.

Connect Select connection mode. The cursor is used to connect components.

Pan Select movement mode. The cursor is used to move the entire system in the Workspace window.

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

4.2.3 Workspace windows

The workspace window is the large pane on the left side of the application window.

Use the workspace views to display and edit the graphical representation of a system the LISA code for the components.

Components can be hierarchical system components or single components that do not have any sub components or internal connections.

The workspace window has two tabs at the bottom of the window:

Source Displays the LISA source code of the component.

The workspace can also display other text files. See *Open File dialog* on page 4-46.

The Source view is a fully-featured source text editor that includes the following:

- Similar operation to common Microsoft Windows text editors.
- Standard copy and paste operations on selected text are supported. Text copied from the Source view can be pasted into another System Canvas text field or into an external text editor.
- Undo/redo operations are enabled. Text changes can be undone by using **Ctrl-Z** or **Edit → Undo**. Text changes can be repeated by using **Ctrl-Y** or **Edit → Redo**.
- Syntax highlighting is provided for LISA, C++, HTML, Makefiles, project (`*.sgproj`) and repository (`*.sgrepo`) files.
- Auto-indenting and brace matching is provided. Indenting is done by using four spaces rather than tab characters.
- Auto-completion for LISA source. If you type a delimiter such as "**.**" or "**:**", a list box with appropriate components, ports or behaviors for the context is shown. Master and slave ports are indicated by icons.
- Call hint functionality is included. If you type a delimiter such as "(", a tooltip is displayed with either a component constructor or behavior prototype, depending on the context. Call hints are enabled if tooltips are enabled in the Appearance section of the Preferences dialog. See *Preferences dialog* on page 4-50.

Note

Lexical information for auto-completion and call hint functionality is updated every time a LISA file is parsed. This occurs, for example, when switching between the Source and Block Diagram.

Block Diagram

Displays the graphical representation of the component. It enables adding components, connections between components, and ports to the displayed component. Add labels to the diagram to provide comments.

Displaying the block diagram fails if:

- the file is not written in LISA
- the syntax of the LISA file is incorrect
- the LISA file contains more than one component
- the LISA file contains a protocol.

The Block Diagram view supports copy and paste operations on selected components, connections, labels, and self ports. Either:

- use the cursor to draw a bounding rectangle around the box
- press and hold shift while clicking on the components to copy.

Component instance names are changed if the component is copied. Connections are only copied if both ends of the connection are copied.

Note

If the system is changed in the Source view, the change is immediately reflected in the Block Diagram view. If a change is made in the Block Diagram view, it is immediately reflected in the Source view.

For each open component, or other open file, a new workspace tab is created at the top of the Workspace window. The name of the file shown in the workspace tab. The status of the file is indicated by:

- an asterisk is displayed behind the name if the file is modified
- a question mark behind the filename if the file is not part of the project.

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

Context menu for the Source view

The context menu of the **Source** tab contains items for common text operations:

Undo Cancel last action.

Redo Cancel Undo. Repeat the action that was undone.

Cut Cut selected text.

Copy Copy selected text.

Paste Paste text from global clipboard.

Delete Delete selected text.

Select All Selects all text in the window.

Context menu for the Block Diagram view

The context menu for the **Block Diagram** tab has the following entries:

Open Component

This item opens a new workspace tab for the selected component.

Delete Deletes the component or connection, depending on the item the mouse pointer was positioned when activating the context menu

Add Port... Adds a port to the component in the active workspace.

Mirror Self Port

Mirrors the graphical object representing the port.

Expand Port

If a port array is selected, this option is enabled. Select it to display all of the individual port elements. Port arrays with eight or fewer elements are expanded by default.

Note

Ports with a large number of elements might expand so that all elements appear on top of one another. Click and drag to move the individual port elements to separate locations. Alternatively, collapse the port, increase the component size, then expand the port again.

Collapse Port

If a port array is selected, this option is enabled. Select it to hide the individual port elements and only display the top-level port name. Port arrays with more than eight elements are collapsed by default.

Hide Port

If a port is selected, this option is enabled. Select it to hide only the selected port.

Hide All Unconnected Ports

If a port is selected, this option is enabled. Select it to hide ports that have not yet been connected to other ports.

Show/Hide Ports of Protocol Types...

This option hides all ports that use a specified protocol. Select the protocols to filter from the displayed Show/Hide Connection Types dialog.

Show All Ports

Shows all ports of the component.

Autoroute connection

If a connection is selected, this option is enabled. If selected, the connection line is redrawn.

Documentation

This option opens the PDF file that contains the documentation for the selected component.

Object Properties

Opens a dialog that lists the object properties.

4.2.4 Component window

The component window is located to the right of the workspace window. The component window contains a list of all components and their protocols and libraries available in the current project.

Note

This view is empty if no project is loaded.

The component list consists of the following tabs:

Components Lists all components along with their version number, type and file location. The components can be dragged and dropped into the block diagram. Double clicking on a component opens it in the workspace.

Protocols Lists all protocols and their file locations that are used by the listed components. Double clicking on a protocol opens it in the workspace.

Files All files that belong to the current project are displayed hierarchically. The root of this hierarchy is always the project file describing the loaded project. The project file can reference LISA files or component repositories. A repository can itself contain a repository. Double clicking on a file opens it in the workspace.

Note

- The view shows the fully expanded file tree. The order of file processing is from top to bottom of the list.
- To move files within the project and change the processing order from this view, either:
  - Select a repository and use the **Up** and **Down** entries in the context menu

    Use the keyboard shortcuts of **Alt + Arrow Up** or **Alt + Arrow Down**, to change the order of files inside the repository.
  - Drag a file from one repository to a new location in the same repository or to a different repository.

Component window context menu

The context menu of the component list contains the following items:

Open Opens the file associated to the selected item.

Add... Opens a dialog for adding an existing file containing a repository, component or protocol. It also permits adding existing libraries.

Add New... Opens a dialog for adding a new file to the project.

Add Directory...

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

Remove Removes the selected item.

Up (**Files** tab only) Moves the selected file higher in the file list. The file order determines the processing order for the file contents.

Down (**Files** tab only) Moves the selected file higher in the file list. The file order determines the processing order for the file contents.

Reload Reloads the selected component or protocol.

Refresh Component List

Refreshes the entire component list.

Documentation

This option opens the PDF file that contains the documentation for the selected component.

Properties A dialog opens showing the properties of the selected item.

4.2.5 Output window

The Output window displays the output resulting from build or script commands.

The left side of the window has the following controls:

First click the icon to navigate to the first error message in the text.

Previous click the icon to navigate to the previous error message in the text.

Lock click the icon to stop scrolling if an error occurs.

Next click the icon to navigate to the next error message in the text.

Last click the icon to navigate to the last error message in the text.

The right side of the window has the following controls:

Scroll bar If the output window contains more text that fits in the window, use the scroll bar to view the text.

Lock Click the icon to force the output window to include the bottom line in the text output and ensure the most recent text is viewable in the window.

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

Note

- Do not use Japanese or Korean characters in project filename paths, because they can lead to failure to find the specified libraries.
- On Microsoft Windows, the direction of the slash characters separating directories appears to be reversed. You can enter slash characters in either direction but are displayed as / in all cases. This does not affect operation.

Use these dialogs to add a component, protocol, library, repository, or additional source code to a project:

- The Add New File dialog in Figure 4-2 is displayed by right-clicking in the Component window and selecting **Add New** from the context menu.

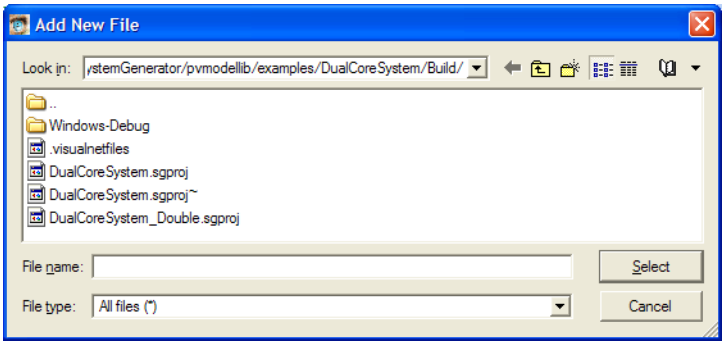

Figure 4-2 Add New File dialog

- The Add Existing Files dialog in Figure 4-3 is displayed by right-clicking in the Component window and selecting **Add** from the context menu.

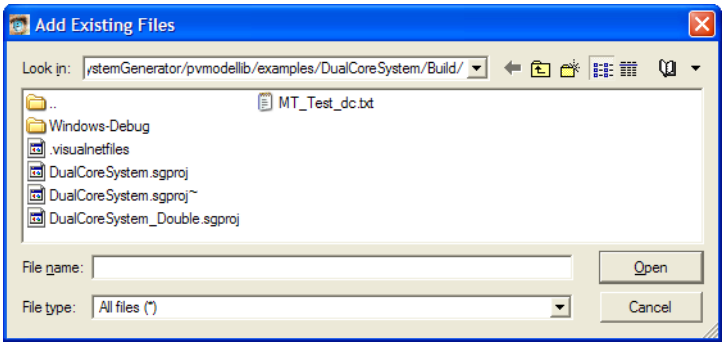

Figure 4-3 Add Existing Files dialog

- For more information about the dialog used for creating a new project file, see *New File dialog (from File menu)* on page 4-42.
- For more information about the dialog used to add files from the Project menu, see *Add Files (Project menu)* on page 4-21.

Note

Repository and LISA files are always processed in the order they occur in the project file (`*.sgproj`) and sub-repository files (`*.sgrepo`). You can change the order of files and repositories in the File List view in the Component window by clicking and using the context menu to move the item up or down, or clicking and dragging the item to its new position. See *File processing order* on page 1-7.

To add a file using the Component window context menu:

1. Select **Components**, **Protocols**, or **Files** tab in the Component window to select the type of file to add.

   Note

   - If you selected the **Files** tab and you are adding a file at the top level, select the top entry in the list.
   - If you are adding a file to an existing repository, select the repository file in the list.

2. Right-click in the Component window and select **Add** or **Add New** from the context menu:

   Add The Add Existing Files dialog is displayed.
   Navigate to the location of the file and select it.

   Add New The Add New File dialog is displayed.
   Navigate to the directory to contain the new file and enter a name for the file.

3. Click **Open** to add the file and close the dialog.

If you have added a library file, that is, a file with a `.lib` or `.a` extension, you must also specify the build actions and platform for the library. See *File/Path Properties dialog* on page 4-36.

Note

You can use the **Recently selected files** dropdown list on the dialog to display files that have been added to other projects. 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.

4.3.1 Using environment variables in filenames

Using environment variables in filenames enables switching to new repository versions without the requirement to modify the project. For example, use `$(PVLIB_HOME)/etc/sglib.sgrepo` as the reference to the components of the Fast Model Portfolio to enable migrating to future versions of this library by modifying environment variable `PVLIB_HOME`.

The path of a file can be edited through the File Properties dialog as listed in *File/Path Properties dialog* on page 4-36. Select **Properties** from the context menu activated while the file is selected. The file path can be modified by editing the **File** entry.

Note

On Microsoft Windows, you can use Unix syntax to define environment variables and paths. This means `$PVLIB_HOME/etc/my.sgrepo` is valid on Microsoft Windows.

4.3.2 Assigning libraries and compiler option for platforms

You must assign the operating system that a library is built for and specify the compilers that can be used to build the library. For Microsoft Windows libraries it might also necessary to distinguish between Debug and Release versions.

Use the File Properties dialog to specify the operating system and compilers by checking the appropriate boxes in the **Supported platforms** panel as described in *File/Path Properties dialog* on page 4-36.

See *Project Settings dialog* on page 4-60 for information on specifying compiler and build options for the project.

4.4 Add Files (Project menu)

Note

Japanese or Korean characters must not be used in project filename paths, as they can lead to failure to find the specified libraries.

Select **Add File** from the **Project** menu to add a new file to the project. See Figure 4-4.

Note

- Repository and LISA files are always processed in the order they occur in the project file (`*.sgproj`) and sub-repository files (`*.sgrepo`). You can change the order of files and repositories in the File List view in the Component window by clicking and using the context menu to move the item up or down, or clicking and dragging the item to its new position. See *File processing order* on page 1-7.
- On Microsoft Windows, the direction of the slash characters separating directories appears to be reversed. You can enter slash characters in either direction but they are displayed as forward slash (/) in all cases. This does not affect operation.

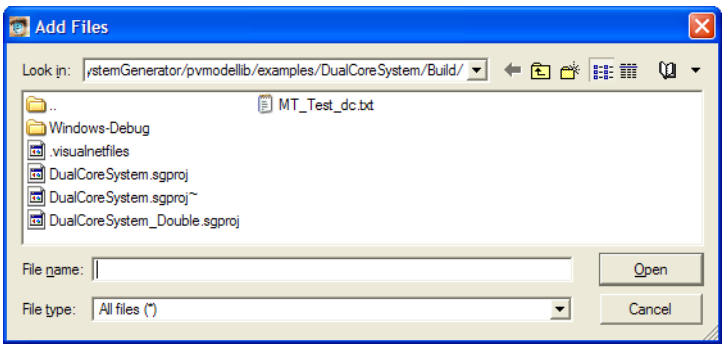

Figure 4-4 Add Files dialog

The behavior of this dialog is identical to the Add Existing Files dialog described in *Add Existing Files and Add New File dialogs (Component window)* on page 4-18.

If a new file is currently being edited in the **Source** tab, select **Add Current File** from the **Projects** menu to immediately add the file to the project. No dialog is displayed.

Note

You can use the **Recently selected files** dropdown list on the dialog to display files that have been added to other projects. 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.

4.5 Add Connection dialog

The Add Connection dialog enables adding a connection to a component port. To display the dialog in Figure 4-5:

- Select a component port.
- Display the Port Properties dialog by selecting **Object Properties** from the context menu or by selecting **Object Properties** from the **Object** menu.
- Click the **Add Connection** button on the Port Properties dialog.

The enabled fields for the dialog depend on whether a slave or master was displayed in the Port Properties dialog. The controls are the same as the Edit Connection dialog but this dialog adds a new connection instead of modifying an existing connection. See *Edit Connection dialog* on page 4-35.

Note

This dialog is also displayed if the cursor is used in connect mode to connect two ports in the Block Diagram area and one or more of the ports is a port array.

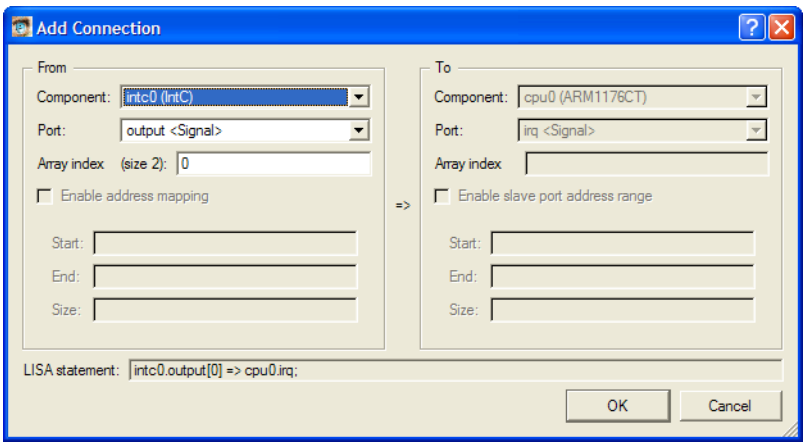

Figure 4-5 Add Connection dialog

4.6 Component Instance Properties dialog

To display the properties for a component you can either:

- select a component in the Block Diagram and then click on the **Properties** button in the toolbar
- select a component in the Block Diagram and select **Object Properties** from the **Object** menu.

Note

The following additional properties dialogs are also available:

Component Model Properties

The properties for the top-level component. See *Component Model Properties dialog for system* on page 4-27.

Repository Component Properties

The properties for component in the Components list. See *Component Properties dialog for a library component* on page 4-31.

Self Port Properties

The properties for a self port in the top-level component. See *Self Port dialog* on page 4-80.

Port Properties

The properties for a self port in an individual component. See *Port Properties dialog* on page 4-48.

Label Properties

The properties for a label. See *Label Properties dialog* on page 4-40.

Protocol Properties

The properties for a protocol. See *Protocol Properties dialog* on page 4-77.

The Component Instance Properties dialog has the following tabs:

General tab

This tab displays the component name, instance name, filename and path, and repository. See Figure 4-6 on page 4-24.

The **Instance name** field is editable. To change the name of a component instance, enter the new name in the field.

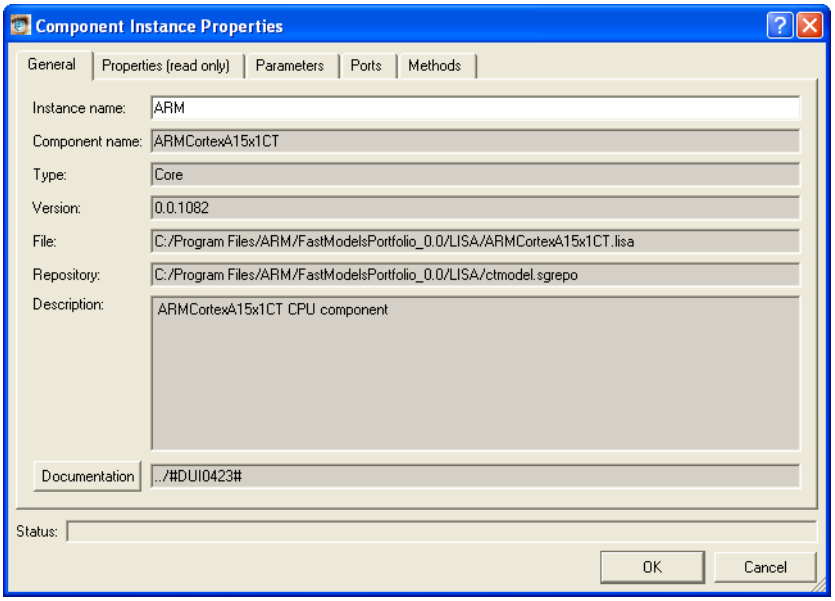

Figure 4-6 Component Instance Properties dialog, General tab

Note

- To view the properties of the top-level component, double-click in an area of the Workspace window that does not contain a component.
- On Microsoft Windows, the direction of the slash characters separating directories appears to be reversed. You can enter slash characters in either direction but they get displayed as / in all cases. This does not affect operation.

Properties tab

This tab displays all properties for the selected component. See Figure 4-7. If the properties for the instance of the component in the workspace are not editable, the tab is labeled **Properties (read only)**.

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

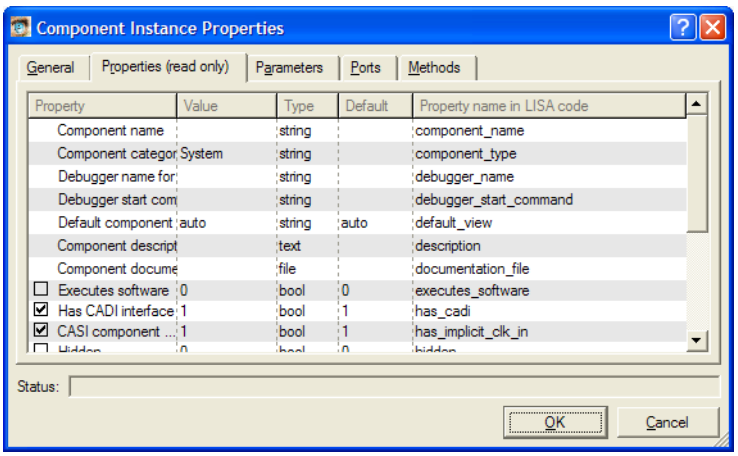

Figure 4-7 Component Instance Properties dialog, Properties tab

Parameters tab

This tab displays all editable parameters for this component. See Figure 4-8. To modify the parameter, enter a new value in the **Value** edit box.

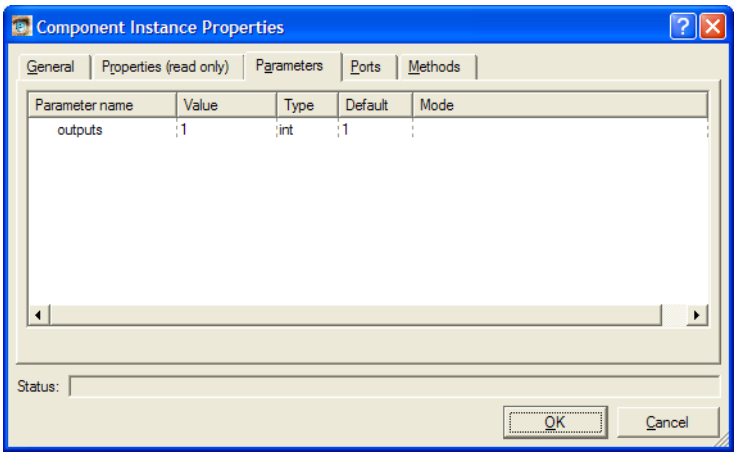

Figure 4-8 Component Instance Properties dialog, Parameters tab

The following controls are present on the Parameters tab:

Parameter name

This list contains 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 component parameter.

Integer parameters in decimal format can contain the binary multiplication suffixes listed in Table 4-2. These left-shift the bits in parameter value by the corresponding power of two.

Table 4-2 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 tab

This tab displays all ports present in this component. See Figure 4-9 on page 4-26.

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

The properties of individual ports can be edited:

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

Note

If you click **OK**, the changes are applied immediately.

Individual ports can be enabled or disabled with the checkboxes:

- Click **Show selected ports** to display the ports that have a check mark.
- Click **Hide selected ports** to hide ports that have a check mark.

Note

If the top level of a port array is hidden, all of the individual ports are hidden but they retain their check mark setting. If the port array is expanded and not hidden, individual ports can be either shown or hidden.

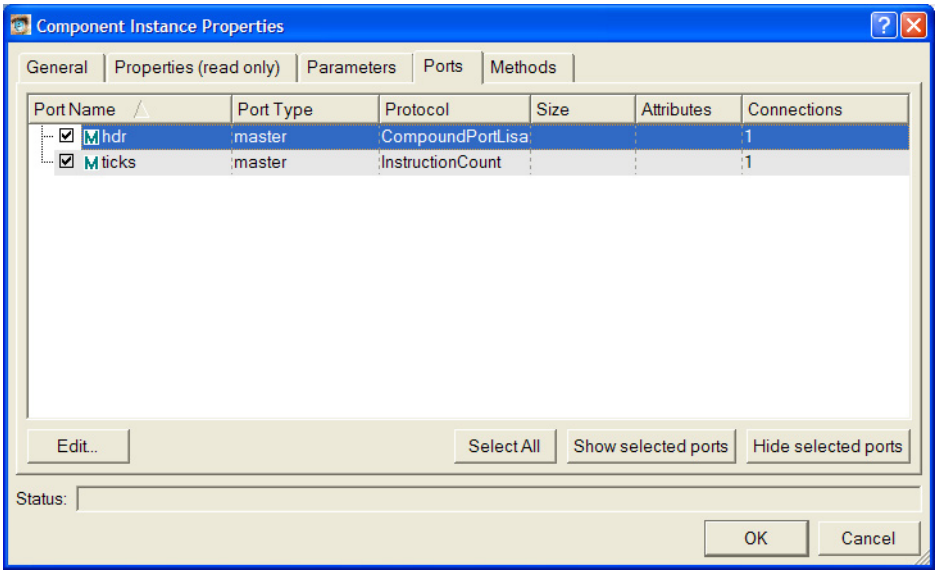

Figure 4-9 Component Instance Properties dialog, Ports tab

Methods tab

This tab displays all the behaviors, that is, component functions, that are implemented by the selected component. See Figure 4-10.

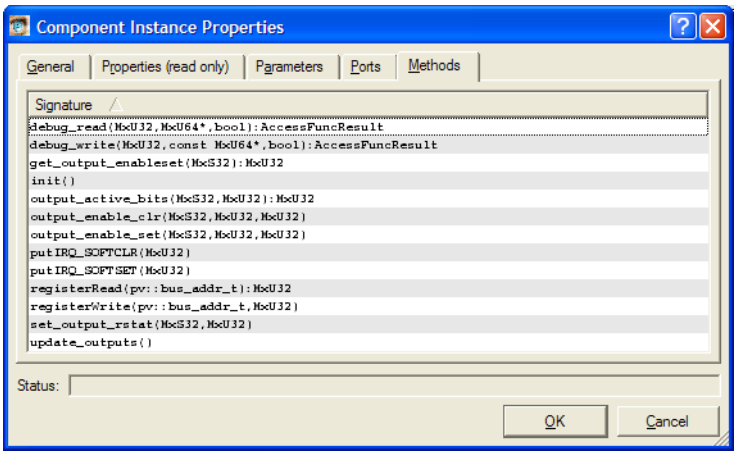

Figure 4-10 Component Instance Properties dialog, Methods tab

4.6.1 Component Model Properties dialog for system

Select a blank area in the Block Diagram view and right-click and select **Object Properties** from the context menu to display the properties for the system. If no component is selected, you can select **Object Properties** from the **Object** menu.

The dialog has the following tabs:

General tab

This tab displays the system name, filename and path, and repository as shown in Figure 4-11.

The **Component name** field is editable. To change the name of the system, enter the new name in the field.

Note

On Microsoft Windows, the direction of the slash characters separating directories appears to be reversed. You can enter slash characters in either direction but are displayed as / in all cases. This does not affect operation.

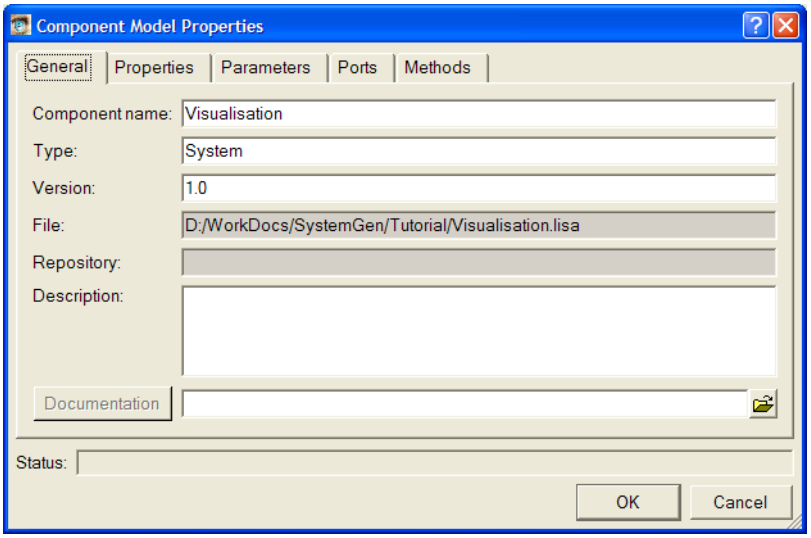

Figure 4-11 Component Model Properties dialog

Properties tab

This tab displays all properties for the system as shown in Figure 4-12 on page 4-28. If a component can have a specific property, but it is not used in this component instance, the box next to the property is unchecked.

Changes made in these dialogs result in changes to the LISA code in the model.

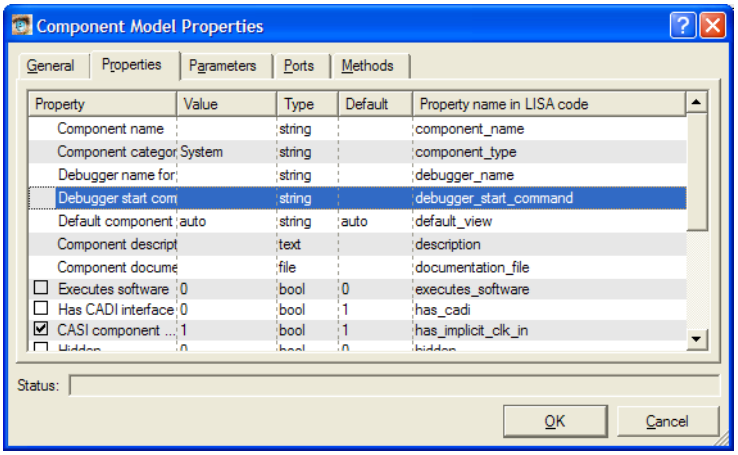

Figure 4-12 Component Model Properties dialog, Properties tab

Double-click in the **Value** column to change the property.

The available properties are listed in Table 4-3:

Table 4-3 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 description of the component.
Component documentation	`documentation_file`	`""`	Path to a file or http link containing documentation on the component. A path to a file can be absolute or relative to the directory containing the LISA file for the component. Supported file formats are `pdf`, `txt`, and `html`.
Executes software	`executes_software`	`0`	This property indicates that the component executes software and that application files can be loaded into this type of component. This typically has to be set to 1 for processor-like components and to 0 for all other components.
Hidden	`hidden`	`0`	If set to 1, the component is not displayed in the component window in the Components tab of System Canvas. The component is, however, displayed in the workspace window itself. Hidden components behave exactly as normal components.
Has CADI interface	`has_cadi`	`1`	If set to 1 a CADI interface is generated for this component, permitting connection to the target with a CADI compliant debugger. If set to 0 no CADI interface is generated for this component.
Icon pixmap file	`icon_file`	`""`	The name of the file that contains the icon, in xpm format, for this system. This icon can be used in hierarchical systems.
License feature	`license_feature`	`""`	The license feature string required to run this system model.

Table 4-3 Component properties (continued)

Property	ID	Default	Description
Load file extension	`loadfile_extension`	`""`	Application filename extension for this target. Example: ".elf" or ".elf;*.hex"
Small icon pixmap file	`small_icon_file`	`""`	The name of the file that contains the 12x12 pixel icon, in xpm format, for this system.
Component version	`version`	`"1.0"`	Version of the component.

Parameters and properties that are integers displayed in decimal format can have the K, M, G, T, and P binary multiplier suffixes listed in Table 4-2 on page 4-25.

Parameters tab

This tab displays the parameters for the system. See Figure 4-13.

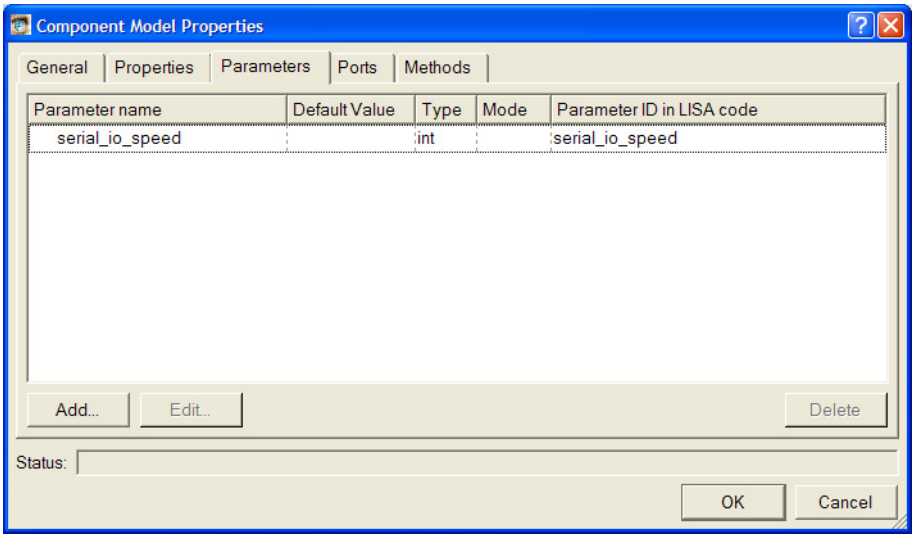

Figure 4-13 Component Model Properties dialog, Parameters tab

The **Parameters** tab contains the following controls:

Parameter name

This column contains 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 component parameter.

Integer parameters in decimal format can contain the binary multiplication suffixes listed in Table 4-2 on page 4-25. These left-shift the bits in parameter value by the corresponding power of two.

Parameter ID in LISA code

This column contains the LISA ID for the component parameters.

Add Click to add a new parameter to the selected component.

Edit Select a parameter and then click to change the name of an existing component parameter. See Figure 4-14 on page 4-30.

Delete Select a parameter and then click to delete the component parameter.

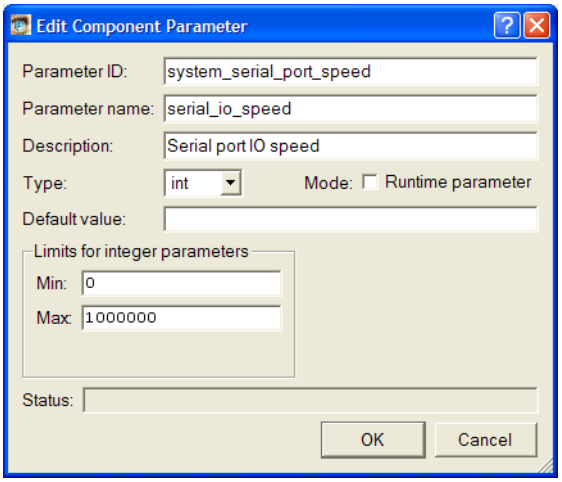

Figure 4-14 Add Component Parameter dialog

Ports tab

This tab displays all external ports present in the system. See Figure 4-15.

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

Individual ports can be enabled or disabled with the checkboxes:

- Click **Show selected ports** to display the ports that have a check mark.
- Click **Hide selected ports** to hide ports that have a check mark.

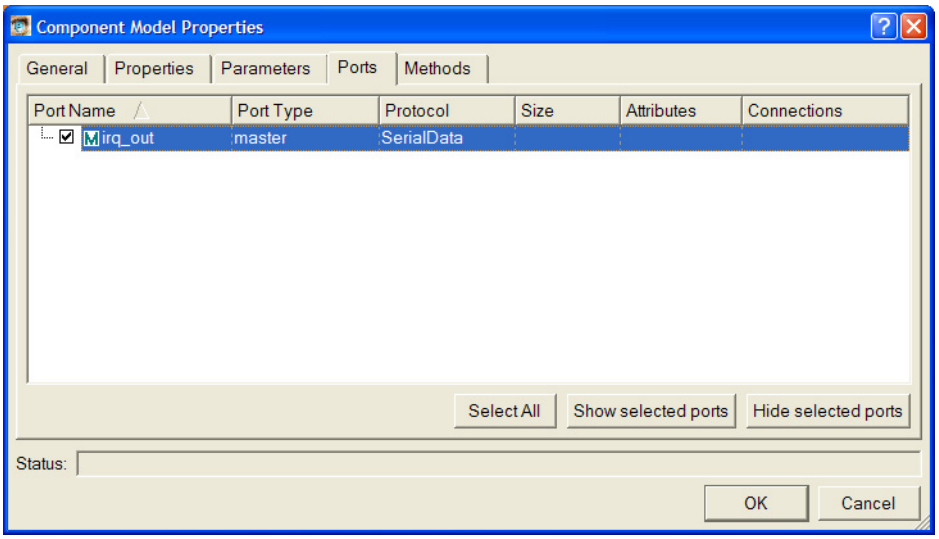

Figure 4-15 Component Model Properties dialog, Ports tab

Methods tab

This tab displays the LISA prototypes that are available for the selected component. The list is for reference only and cannot be edited.

4.7 Component Properties dialog for a library component

Select a component from the **Components** list and right-click and select **Properties** from the context menu to display the properties for the selected component. You can also select a component and then select **Object Properties** from the **Object** menu.

Note

If a component is not selected, selecting **Properties** from the context menu or **Object Properties** from the **Object** menu displays the properties for the entire model. See *Component Model Properties dialog for system* on page 4-27.

The **General** tab shown in Figure 4-16 displays the following:

Component name
: The name of the component.

Type
: The component category as, for example, `Core` or `Peripheral`.

Version
: The revision number for the component.

File
: The file that defines the component.

Note

On Microsoft Windows, the direction of the slash characters separating directories appears to be reversed. You can enter slash characters in either direction but they get displayed as / in all cases. This does not affect operation.

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

Description
: Descriptive information that was added when the component was added to the project.

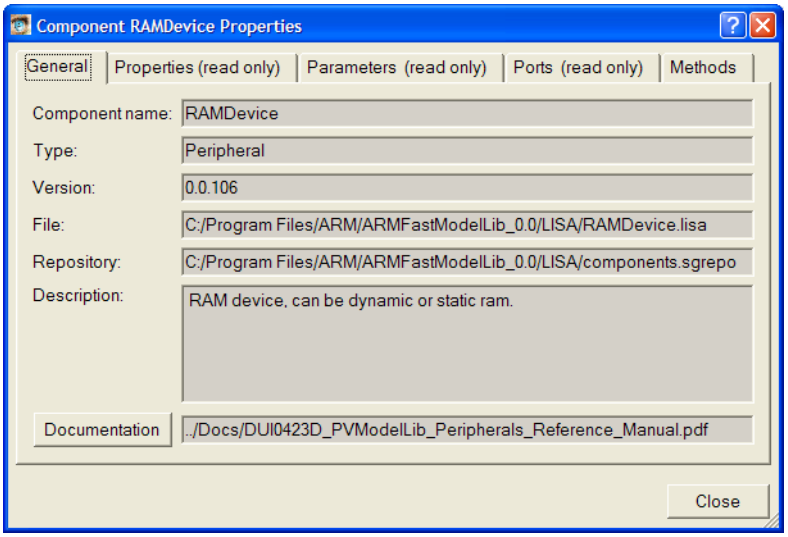

Figure 4-16 Component Properties dialog for SerialCharDoubler

Click the other tabs to display read-only information about the component:

Properties
: Any properties that can be used with the component. See Figure 4-17 on page 4-32.

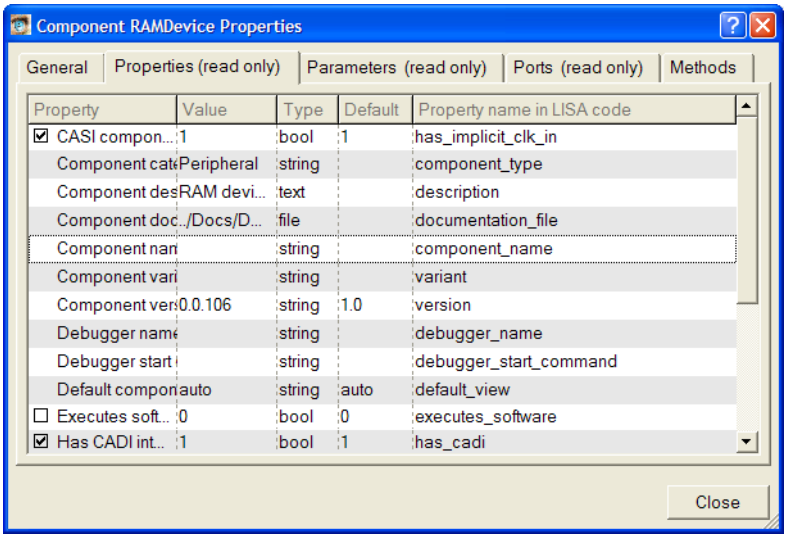

Figure 4-17 Component Properties dialog, Properties tab

Note

- The properties of a component in the Components list cannot be modified. To display the dialog, select **Object Properties** from the **Object** menu.
- The `license_feature` property identifies the license feature string that is required to use this component in a virtual platform model.

Parameters Any parameters that have been defined for the component. See Figure 4-18.

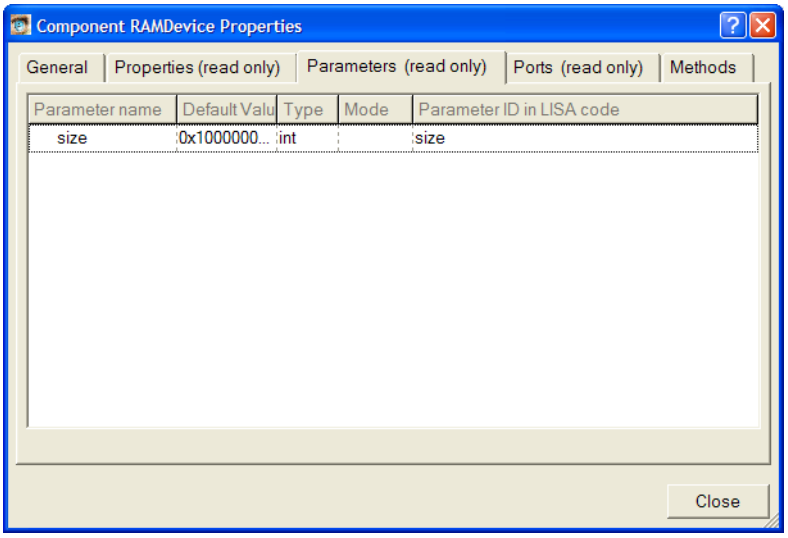

Figure 4-18 Component Properties dialog, Parameters tab

Ports Any ports that are present in the component. See Figure 4-19 on page 4-33.

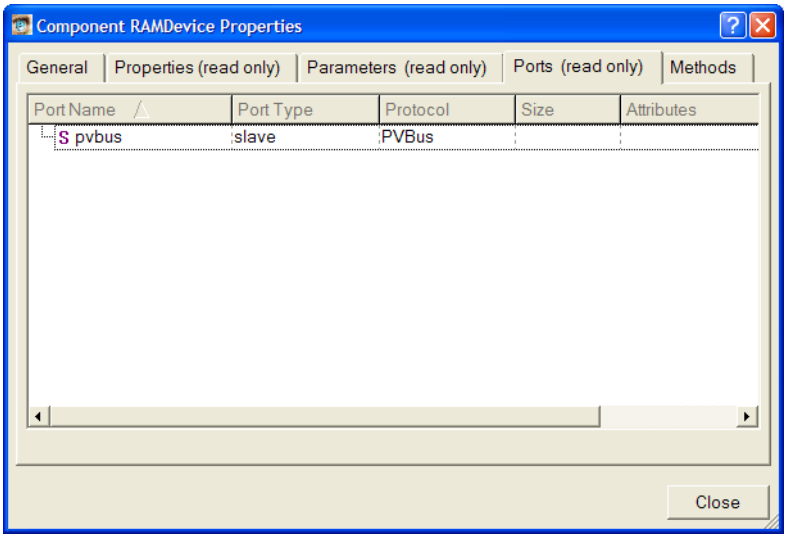

Figure 4-19 Component Properties dialog, Ports tab

Note

Port arrays are displayed as the array name only. Individual ports in the array cannot be expanded and viewed.

Methods This tab displays the LISA prototypes of methods, that is, behaviors, that are available for use by the selected component. The list is for reference only and cannot be edited.

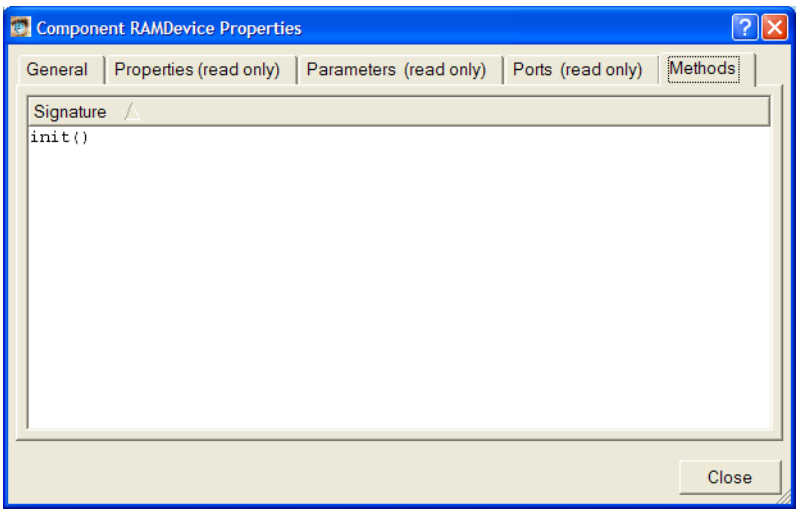

Figure 4-20 Component Properties dialog, Methods tab

4.8 Connection Properties dialog

To display connection properties, double click on a connection between components in the Workspace window. The dialog shown in Figure 4-21 has the following fields:

Name The name of the ports.

Type The type of port and the protocol used.

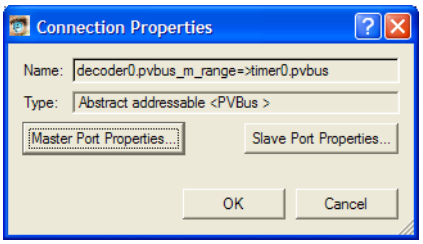

Figure 4-21 Connection Properties dialog

To change the address mapping, click either **Master Port Properties** or **Slave Port Properties**. The Port Properties dialog for the port is displayed as described in *Port Properties dialog* on page 4-48.

4.9 Edit Connection dialog

To change the connected port or the address mapping, select a connection from the Port Properties dialog and click **Edit Connection...**. The resulting dialog, shown in Figure 4-22, has the following controls:

Component The source component if a slave port was displayed in the Port Properties dialog or the destination component if a master port was displayed in Port Properties.

Port The master port for the connection if a slave port was displayed in the Port Properties dialog or the slave port for the connection if a master port was displayed in Port Properties.

Array index If the port selected in the **Port** field is a port array, enter an index value to specify which of the elements of the array is used for the connection.

Enable address mapping

If this box is checked, the port address range is determined by the values in the **Start** and **End** boxes.

Start The start address port the port.

End The end address for the port.

Size The size of the address region. If the **Start** and **End** values are entered, this value is calculated automatically. If the **Size** value is entered, the **End** value is calculated automatically.

OK/Cancel After specifying the port and address, click **OK** to modify the connection. Click **Cancel** to close the dialog without changing the connection.

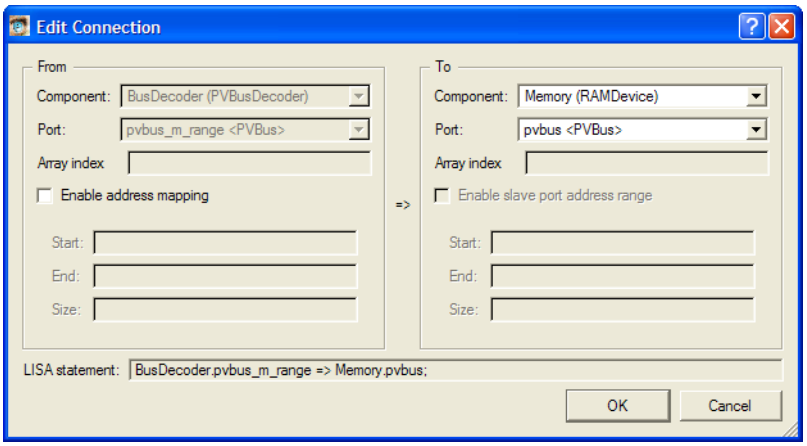

Figure 4-22 Add/Edit Connection Mapping dialog

The **LISA statement** field shows the code that matches the entered address ranges.

4.10 File/Path Properties dialog

The File/Path Properties dialog displays properties for the file and lets you specify build and compile options. For more information about how the settings made from this dialog affect the project file, see *Alphabetical list of project parameter IDs* on page 4-69.

Note

On Microsoft Windows, the direction of the slash characters separating directories appears to be reversed. You can enter slash characters in either direction but they are always displayed as forward slashes (/). This does not affect operation.

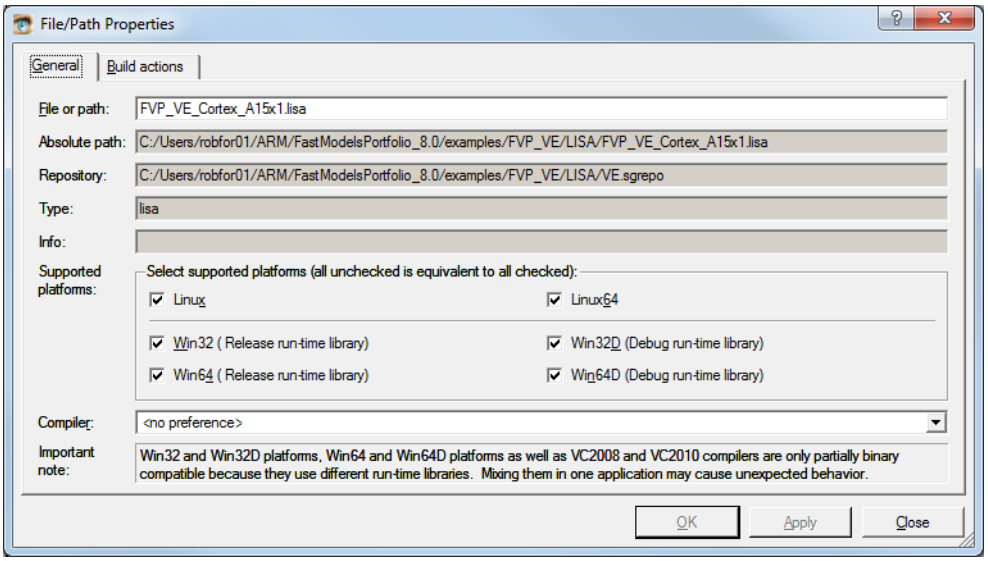

Figure 4-23 File/Path Properties dialog, General tab

Select a component from the Component window **Files** tab, right click to open the context menu, then click **Properties** to display the properties for the selected file.

The **General** tab contains:

File or path The name of the file.

Note

The File Properties dialog is modeless. A different file can be selected without closing the dialog. If, however, any of the properties for the previously-selected file have been modified, a warning message is displayed to prompt you to save the changed contents for the current file before displaying the properties for the selected file.

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:

- Linux
- Linux64
- Win32 (Release run-time library)
- Win32D (Debug run-time library)
- Win64 (Release run-time library)
- Win64D (Debug run-time library).

Compiler Select the compiler to use to compile this component. The Compiler dropdown list enables you to specify the preferred compiler as:

- no preference
- Microsoft Visual C++ 2005 SP1
- Microsoft Visual C++ 2008 SP1
- gcc version found in $PATH at compile time
- gcc-4.1 - gcc version 4.1.2

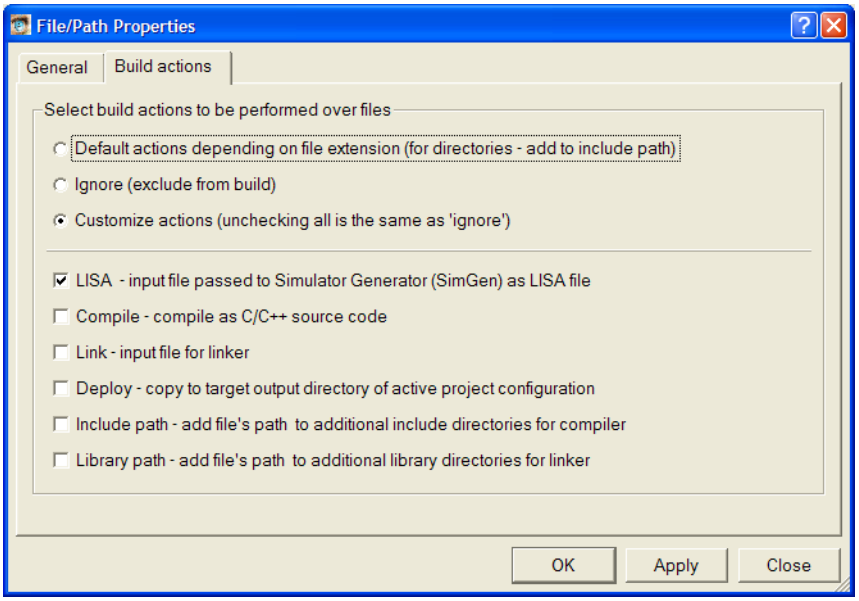

Figure 4-24 File/Path Properties dialog, Build actions tab

The **Build actions** tab enables assigning actions to selected files:

Default actions depending on file extension

The actions are determined by the extensions:

`.lisa` a LISA source file that is parsed by simgen.

`.c .cpp .cxx`
a C or C++ source file that is compiled.

`.a .o` an object file that is linked on Linux.

`.lib .obj` an object file that is linked on Microsoft Windows.

`.sgproj` a project file that is parsed by simgen.

`.sgrepo` a component repository file that is parsed by simgen.

directory_path/

is an include directory that is added to the search path used by the compiler. The trailing slash is important, and is used to identify this as an include path. For example, to add the directory that contains the `*.sgproj` file, you must specify `./` (dot slash), not only the dot.

All other files

a *deploy* file is copied to the build directory.

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

Customize actions

The file extension is ignored. Use the check boxes to specify the action as one or more of:

LISA - input file passed to Simulator Generator as LISA

The file is passed to the Simulator Generator as a LISA file. This option must not be selected if the file is not written in LISA.

Compile - compile as C/C++ source code

The file is added to the list of files that are compiled as C/C++ code during the build process.

Link - input file for linker

The file is linked with the object code during the build process.

Deploy - copy to build directory

The file is copied into the build directory. This option can be used, for example, to add dynamic link libraries that are required to run the generated system model.

Include path - add the file's path to additional include directories

The path of the parent directory that holds the file is added to the list of additional include directories for the compiler.

Library path - add the file's path to additional library directories

The path of the parent directory that holds the file is added to the list of additional library directories for the compiler.

4.11 Find and Replace dialogs

The Find and Replace dialog enables you search for, and optionally replace, text in an editor window. Essentially it is the same dialog in two modes, find only, and find and replace. You can switch modes by clicking the **Find mode** or **Find and replace mode** buttons in the dialog. By default, matches are case sensitive but matches can appear as part of longer words. You can change the default behavior by setting or clearing the relevant checkboxes in the dialog.

The Find dialog, shown in Figure 4-25, is opened 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.

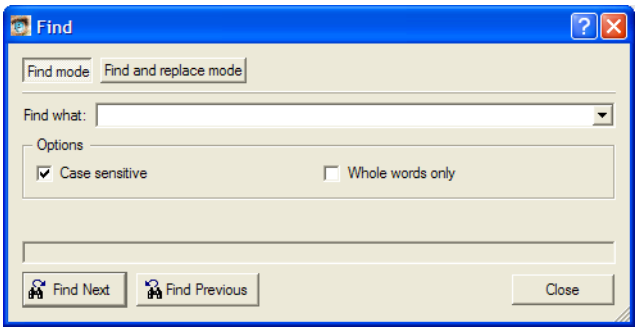

Figure 4-25 Find dialog

The Find and Replace dialog, shown in Figure 4-26, is opened by clicking **Search → Replace** in the main menu. You can 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.

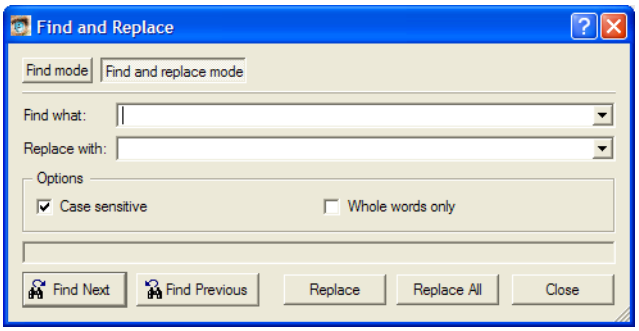

Figure 4-26 Find and Replace dialog

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

4.12 Label Properties dialog

Use the Label Properties dialog to change the text and display properties for a label. See Figure 4-27. Double-click on a label to display the dialog. Select **Add Label** from the **Object** menu to add a label to the component.

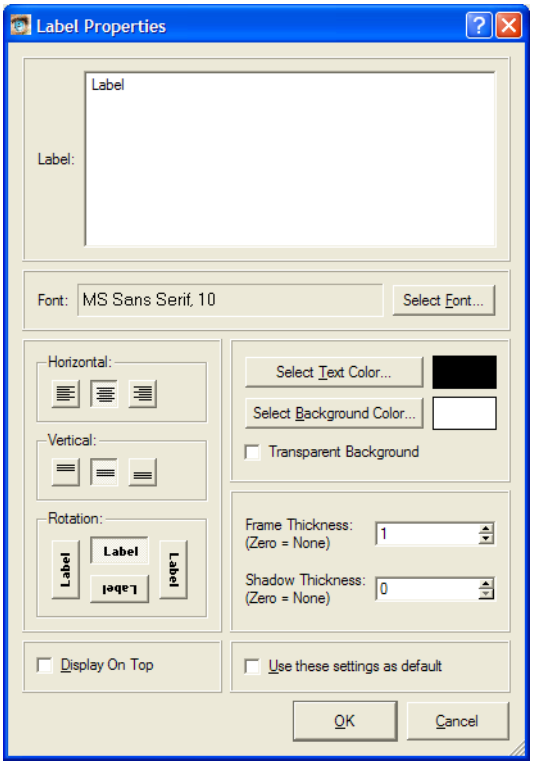

Figure 4-27 Label Properties dialog

The dialog has the following controls:

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 to enable objects behind the label to be visible. If this box is selected, the background color setting is ignored.

Horizontal Select the required horizontal justification for the label text.

Vertical Select the required vertical justification for the label text.

Rotation Select the required orientation for the label.

Frame Thickness

Select the required thickness of the label border.

Shadow Thickness

Select the required 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.

4.13 New File dialog (from File menu)

Use the New File dialog to create a new project or LISA source file. Select **New File** from the **File** menu or click the **New** button to display the dialog in Figure 4-28.

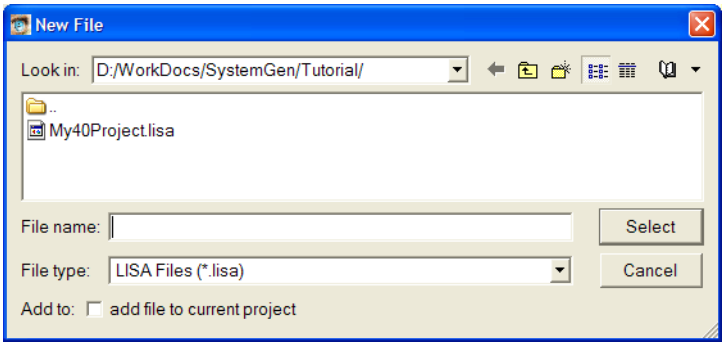

Figure 4-28 New File dialog

The dialog has the following controls:

Look in Specify the directory for the new file in this text box.

Note

- Japanese or Korean characters must not be used in project filename paths, as they can lead to failure to find the specified libraries.
- On Microsoft Windows, the direction of the slash characters separating directories appears to be reversed. You can enter slash characters in either direction but they get displayed as / in all cases. This does not affect operation.

File name Enter the name for the new file in this box.

File type The file type:

- If a project is not already open, this box displays `.sgproj` by default to enable creating a new project.
- If a project is open, this box displays `.lisa` by default to enable creating a new LISA source file.

Add to If the selected file type is not `.sgproj`, this control is enabled and the created file is added to the currently-open project.

Select Click **Select** to accept the name and path. Click **Cancel** to close the dialog without creating a new file.

If the new file is of type `.sgproj`, you are prompted for the top level LISA file. See *Select Top Component LISA File dialog* on page 4-44.

Note

- If an existing file is selected, you are prompted whether to replace the existing file with a new file of the same name.
- Attempting to add multiple files that are already part of the project results in a warning message and the existing project files are not replaced.

Note

You can use the **Recently selected files** dropdown list on the dialog to display files that have been added to other projects. 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.

4.14 New Project dialogs

Select **New Project** from the **File** menu to display the New Project dialog and create a new project. See Figure 4-29.

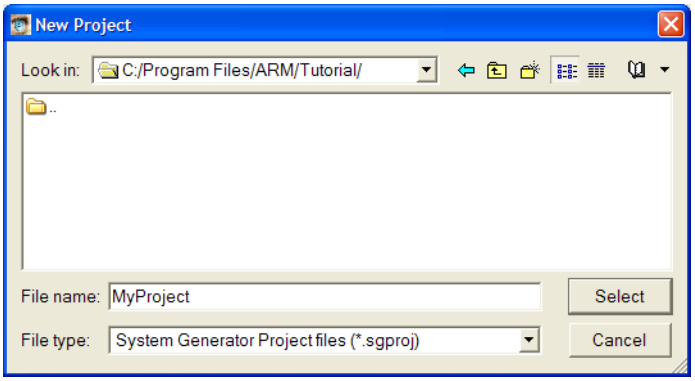

Figure 4-29 New Project dialog

The dialog has the following controls:

Look in Specify the directory to hold the new project file in this box.

Note

- Japanese or Korean characters must not be used in project filename paths, as they can lead to failure to find the specified libraries.
- On Microsoft Windows, the direction of the slash characters separating directories appears to be reversed. You can enter slash characters in either direction but they get displayed as / in all cases. This does not affect operation.

File name Enter the name for the new project in this box.

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

File type The drop down menu selects the files that are displayed in the file list. This is useful if it is necessary to replace an existing file or use a similar name for the project. The default type for Fast Models projects is `.sgproj`.

Select Click **Select** to accept the project name and path. Click **Cancel** to close the dialog without creating a new project.

If an existing project is selected, you are prompted to replace the existing project with a new project of the same name.

Note

The path to the model repositories that are currently specified by the Default Model Repositories section of the Preferences dialog is included in the project file. More information on specifying repositories is given later. See *Default Model Repository* on page 4-57.

4.14.1 Select Top Component LISA File dialog

After clicking **Select** in the New Project dialog, the Select Top Component LISA File dialog is displayed as shown in Figure 4-30 on page 4-45. 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.

Note

On Microsoft Windows, the direction of the slash characters separating directories appears to be reversed. You can enter slash characters in either direction but they are always displayed as forward slash (/). This does not affect operation.

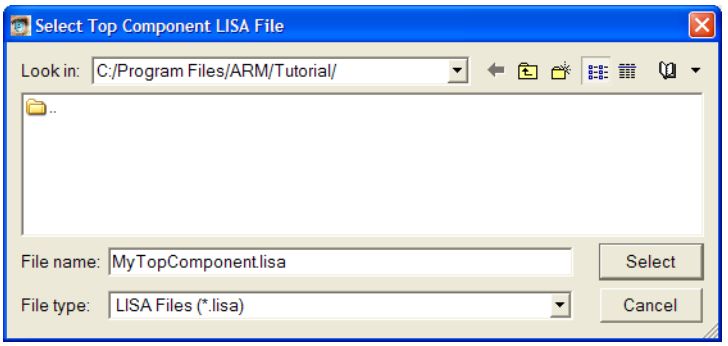

Figure 4-30 Select Top Component LISA File dialog

4.15 Open File dialog

Use the Open File dialog shown in Figure 4-31 to open a project file, LISA source file, or text document.

To display the dialog, either:

- select **Open File** from the **File** menu
- select a file in the Component window and select **Open** from the context menu.

Note

- Japanese or Korean characters must not be used in project filename paths, as they can lead to failure to find the specified libraries.
- On Microsoft Windows, the direction of the slash characters separating directories appears to be reversed. You can enter slash characters in either direction but they are always displayed as forward slashes (/). This does not affect operation.

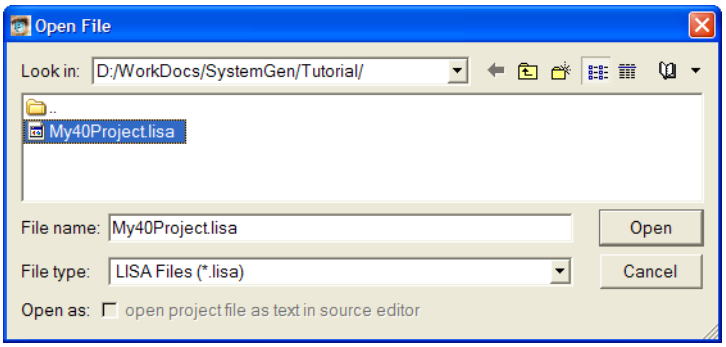

Figure 4-31 Open File dialog

The dialog has the following controls:

Look in Specify the directory that contains the file.

File name Enter the name of the file to open in this box.

File type Select the type of file to filter for.

Open Click **Open** to open the file. Click **Cancel** to close the dialog without opening a file.

Open project file as text in source editor

This box is enabled if the selected file is not of type `.lisa` or `.sgproj`. Check the box to open the file as a plain text file in the Source window.

Note

- If you select a `.sgproj` file and this box is not checked, the project is loaded. All changes to `.sgproj` files are normally done through the Project menu.
- You can use this option, for example, for a `.sgproj` file to manually edit the list of repositories that are used by the project. Changes that are manually made to `.sgproj` files do not take effect until the file has been closed and reopened.

Note

You can use the **Recently selected files** dropdown list on the dialog to display files that have been added to other projects. 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.

4.16 Port Properties dialog

The Port Properties dialog in Figure 4-32 can be displayed by selecting a port or a connection:

- Select a component port in the Block Diagram view and do one of the following:
  - 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 display the Connection Properties dialog. See *Connection Properties dialog* on page 4-34. To display the Port Properties dialog either:
  - 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.

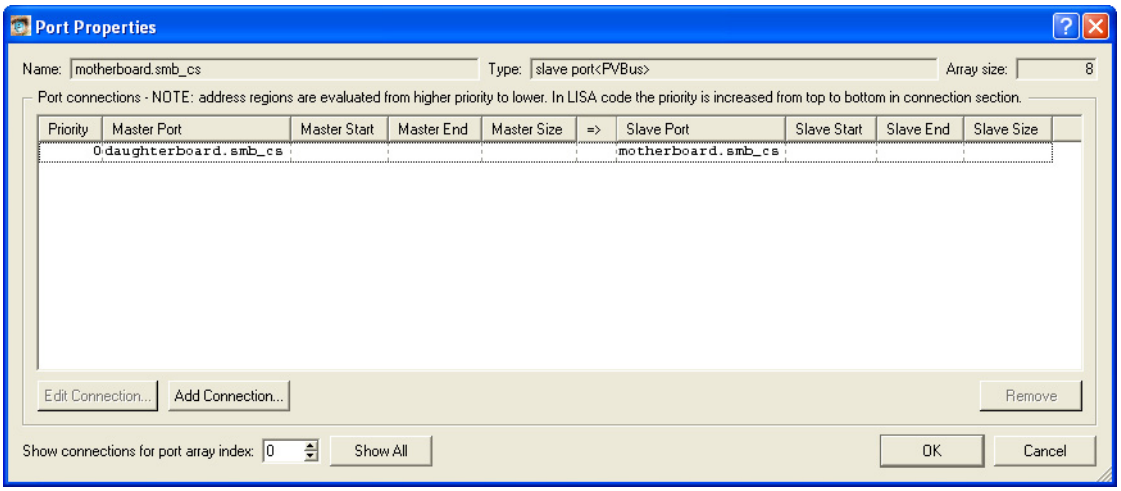

Figure 4-32 Port Properties dialog

The dialog contains the following controls:

Name The name of the port.

Type The type of port and the protocol used.

Array size If the port is a port array, the number of elements is displayed.

Show connections for port array index

If the port is a port array, enter a value in the text box to display only the specified element of the port array.

If the port array is expanded and an individual port is selected, this box displays the index for the selected port.

Port connections

This section displays the connections to the port:

- To sort the connections, click on one of the column headings.
- To change the connected port or address mapping, select a connection from the list and click **Edit Connection**.
- To add a connection, select a connection from the list and click the **Add Connection** button.
- To delete a connection, select the connection from the list and click the **Remove** button.

- To change the priority of a single connection, select the connection from the list and click either the **Increase Priority** or **Decrease Priority** buttons. The list is refreshed to show the new order.

4.17 Preferences dialog

Use the Preferences dialog to configure the working environment of System Canvas. Select **Preferences** from the **File** menu to display the dialog. See Figure 4-33.

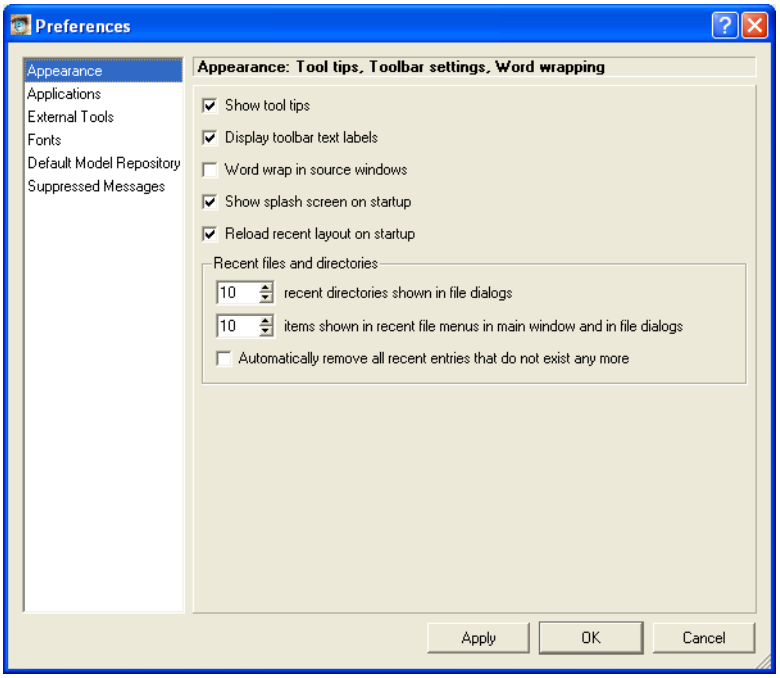

Figure 4-33 Preferences dialog

The left side of the dialog displays the preferences that can be set from this dialog. For more information about the options, see:

- *Appearance*
- *Applications* on page 4-52
- *Fonts* on page 4-56
- *Default Model Repository* on page 4-57
- *Suppressed Messages* on page 4-58.

4.17.1 Appearance

This group, shown by default when the Preferences dialog is opened, contains check boxes that enable you to modify the appearance of the application:

Show Tool Tips

Selecting this option turns on the display of all tool tips within the application. The default setting is on.

Display tool bar text labels

This option toggles the display of the status bar labels.

Word wrap in source windows

This option toggles wrapping long lines to display within the source window. If off, the scroll bar must be used to read or edit long lines.

Show splash screen on startup

This option enables you to toggle between showing the splash screen on startup. The default setting is on.

Reload recent layout on startup

This option reloads the layout settings that were used for the last modified project.

Recent files and directories

This option lets you set the maximum number of directories and files shown in System Canvas file dialogs and menus. You can display up to 32 directories and 16 files in your history.

4.17.2 Applications

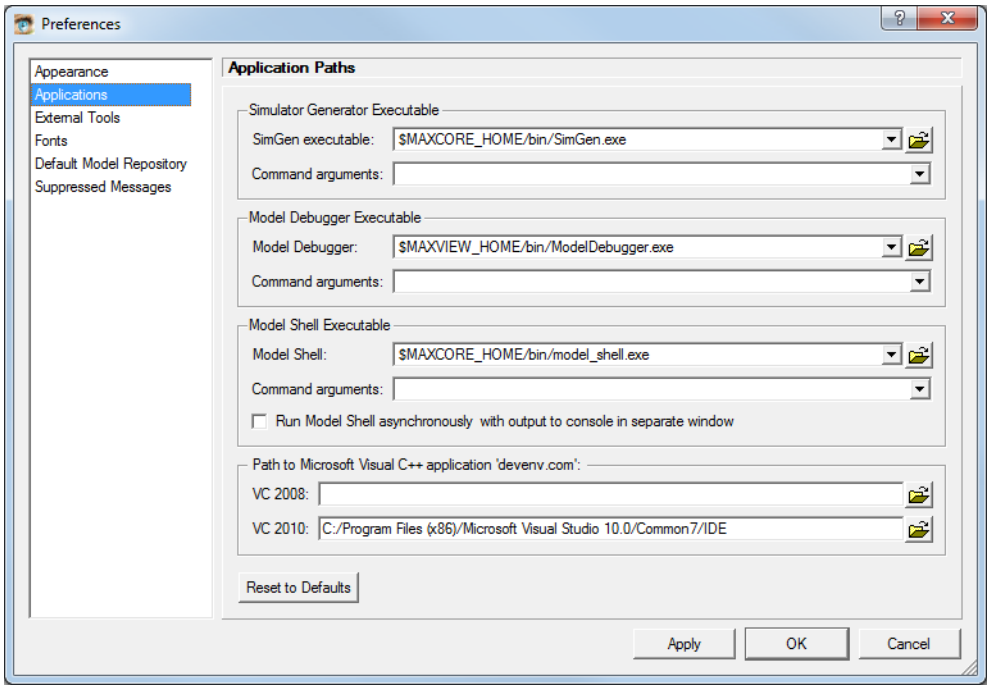

Figure 4-34 Preferences dialog, Applications group

Note

- On Microsoft Windows, the direction of the slash characters separating directories appears to be reversed. You can enter slash characters in either direction but are displayed as / in all cases. This does not affect operation.
- On Microsoft Windows, environment variables are displayed as `$MAXxxxx_HOME`. You can use this format instead of `%MAXxxx_HOME%`.

Use this entry to display the settings for application paths:

Simulator Generator Executable

Use the **SimGen** control to select the path to the `SimGen.exe` file.

Use the **Command arguments** control to specify additional command line options. See Appendix C *Building System Models in Batch Mode*.

Model Debugger Executable

Use the **Model Debugger** control to select the path to the Model Debugger executable.

Use the **Command arguments** control to specify additional command line options. See the *Model Debugger for Fast Models User Guide*, `http://infocenter.arm.com/help/topic/com.arm.doc.dui0314-/index.html`.

Note

Previous versions of System Generator used the `$MAXCORE_HOME` variable to specify the path for both the System Canvas and Model Debugger applications.

The separate path specifications enable using different versions of Model Debugger and provide more flexibility for installing Model Debugger separately from System Canvas.

Model Shell Executable

Use the **Model Shell** control to select the path to the Model Shell executable.

Use the **Command arguments** control to specify additional command line options. See the *Model Shell for Fast Models Reference Manual*, `http://infocenter.arm.com/help/topic/com.arm.doc.dui0457-/index.html`.

Note

- Check `Run Model Shell asynchronously` to enable starting a separate Model Shell instance with its own output window.
- Select the **Run in Model Shell** entry on the **Projects** menu to start simulation.

Path to Microsoft Visual Studio application 'devenv.com'

Use the browse controls to select the paths to the Microsoft Visual Studio `devenv.com` file. This path is used for building the model and for displaying the development environment.

For more information about how to specify the compiler to use for different models, see *Project Settings dialog* on page 4-60.

Click **Reset to Defaults** to reset the application paths. After changing a path, click **Apply** to save the changes.

Note

Under Linux, you can select the GCC C++ compiler that builds the model with the simgen command-line option `--gcc-path`.

Note

- The appearance of this dialog is different for the Linux version of System Canvas:

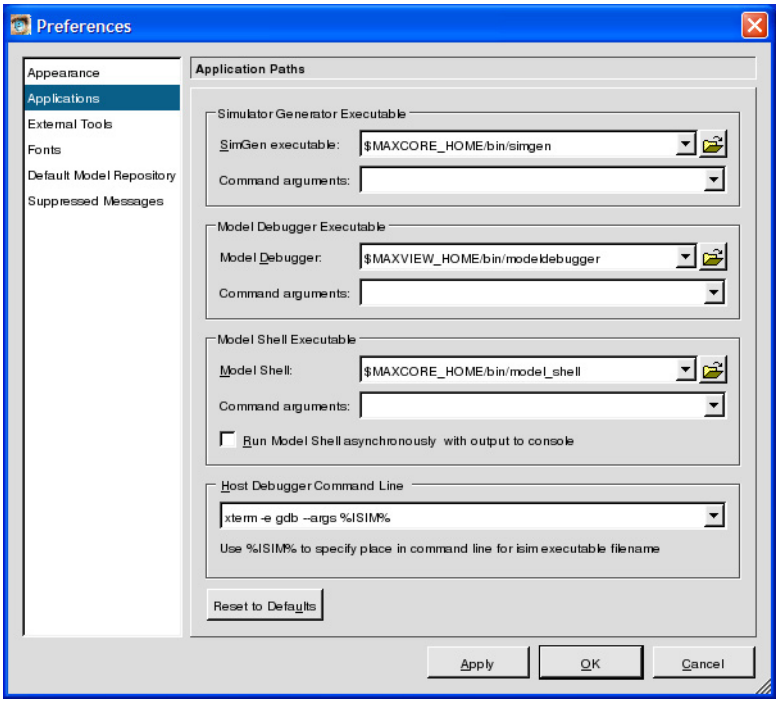

Figure 4-35 Preferences, Applications for Linux

- If **Run Model Shell asynchronously** is checked, the following command line is used:

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

- You can use **Host Debugger Command Line** to specify command-line options. The default text is:

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

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

See also

Reference

- *Simgen command-line options* on page C-3.

4.17.3 External Tools

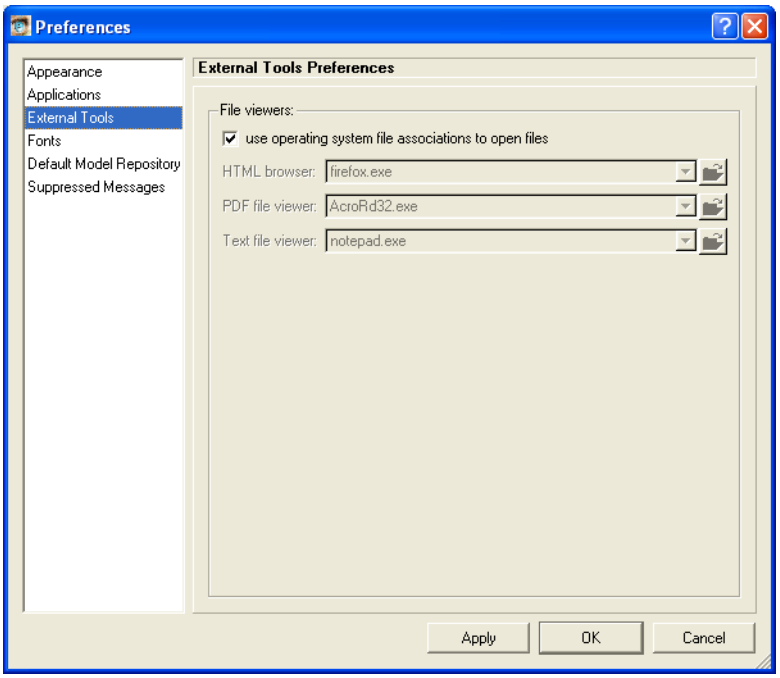

Figure 4-36 Preferences dialog, External Tools group

Note

The **use operating system file associations** checkbox is only available on Microsoft Windows, and is selected by default. This inactivates the external tool edit fields and buttons. To activate these fields, clear the checkbox. The default external tools are different on Linux.

Use these settings to configure the tools used to display Fast Models documentation. Documentation is accessible through:

- the System Canvas **Help** menu item. You can access the PDFs for the System Canvas and Fast Portfolio books this way.
- the documentation_file property in a component property listing. This property can point to a PDF, a text file, an HTML file, or `http://` link.

You can change the default external tools used to access the documentation. Click on the folder icon to open a browser, or use the drop down list to choose a previously-selected executable.

4.17.4 Fonts

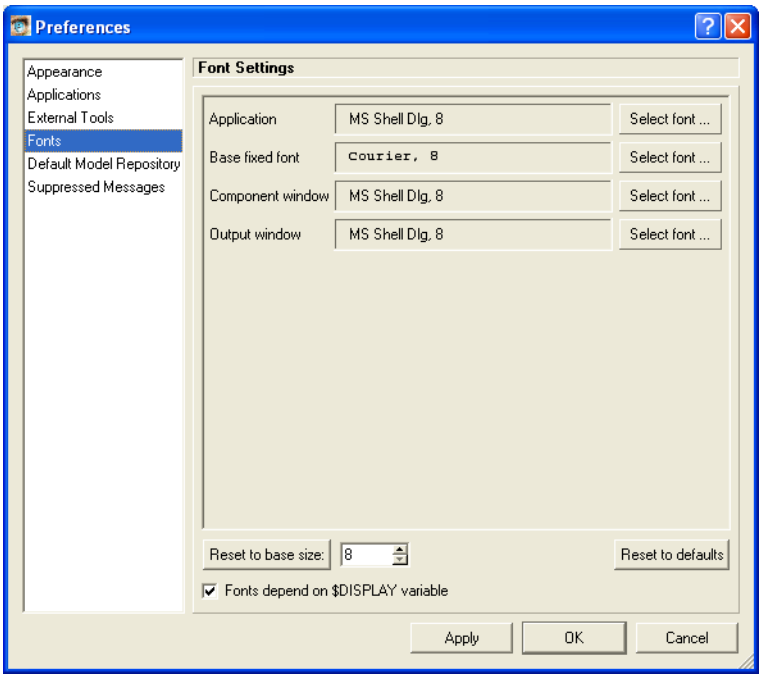

Figure 4-37 Preferences dialog, Fonts group

Use the controls in the **Font** panel to select font type and size for text displayed in the application windows:

Application This font is used in the application window.

Base fixed font

This font is used for text in the **Source** view of the Workspace window.

Block Diagram Component Name

This font is used in component title blocks in the Block Diagram window.

Fonts depend on $DISPLAY variable

Check this box to use the font specified in the `$DISPLAY` variable.

Reset to base size

Select a new font size from the spin control and click this button to reset all fonts sizes to the selected value.

Reset to defaults

Click this button to reset all the fonts back to the factory settings.

Note

If non-Latin characters are used in LISA code, the font selected for **Source windows** must support the character set used in the code. The default font might not support non-Latin characters.

4.17.5 Default Model Repository

Use this section to specify the Model Libraries that are added by default to new projects. See Figure 4-38.

Note

- The selected model repositories are added to a project when it is created. Changing the repository settings after a project has been created has no effect on the project.
- The paths to the repositories are hard coded into the `project_name.sgproj` file. To change the repositories for an existing project, open the file and edit the path.
- On Microsoft Windows, the direction of the slash characters separating directories appears to be reversed in the GUI. You can enter slash characters in either direction but they get displayed as / in all cases. This does not affect operation.
- Use environment variables, such as `$(PVLIB_HOME)`, to enable the same project file to be used on different workstations.

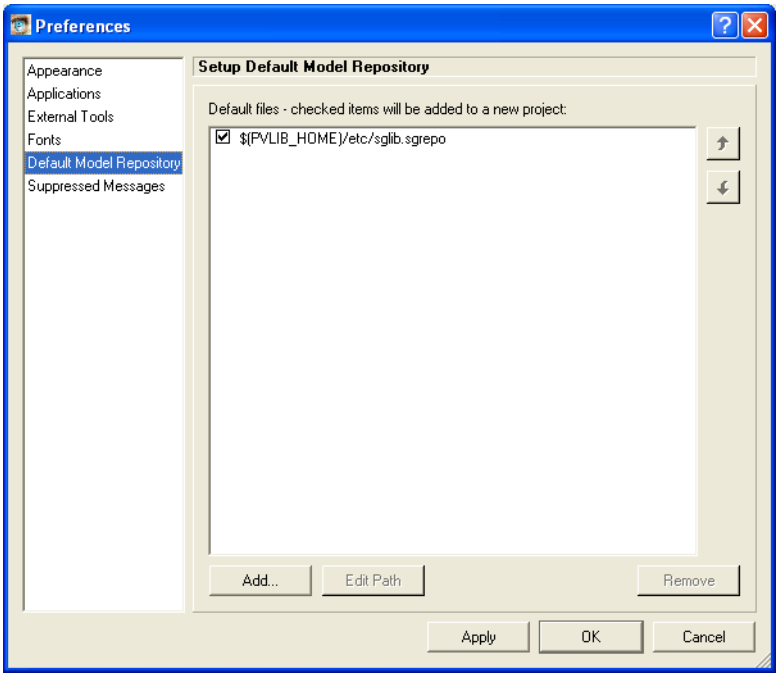

Figure 4-38 Preferences dialog, Model Repository

The dialog has the following controls:

Add Click the **Add** button to open a file selection dialog shown in Figure 4-39 on page 4-58 and add a new `sgrepo` repository file to the list. A repository file contains:

- a list of components
- the path to the LISA source for the component
- a list of library objects for the components.

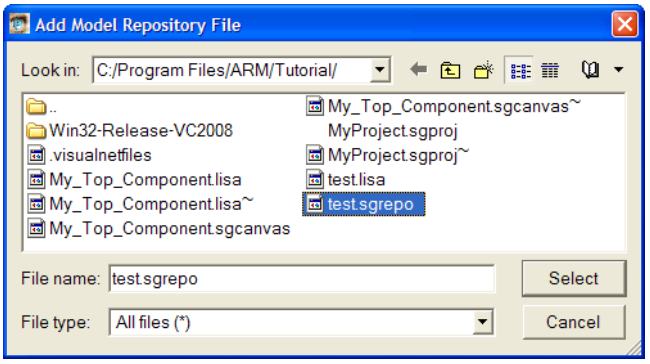

Figure 4-39 Add Model Repository File

A repository file can also consist of lists of paths to other repository files. This enables a hierarchical repository structure.

Select a directory to add all of the repositories in that directory to the list of repositories searched for components.

Remove Select a repository and click **Remove** to exclude the selected repository from new projects. The repository itself is not affected.

The default repository `$(PVLIB_HOME)/etc/sglib.sgrepo` cannot be deleted.

Edit Select a repository and click **Edit** to edit the path to the selected repository.

The path to the default repository `$(PVLIB_HOME)/etc/sglib.sgrepo` cannot be edited.

File checkboxes

Check the box to automatically include the repository in all new projects. Uncheck the box to prevent automatically including the repository in new projects, but keep the path to the repository available for later use.

Up/Down

Use the **Up** and **Down** buttons to change the order of repositories. File processing is based on the repository order.

Default Repositories can also contain a set of default configuration settings for one or more configurations. The values of these settings are appended to the corresponding default configuration values for all new projects. This mechanism enables presetting configuration parameters that are required by all projects that use the specified default model library. The following settings are affected:

- `Additional Include Directories`
- `Additional Compiler Settings`
- `Additional Linker Settings`.

Note

To enable using models from the Fast Model Portfolio in new projects immediately without being forced to add a default repository, the default entry `$(PVLIB_HOME)/etc/sglib.sgrepo` is present when System Canvas is started. This entry cannot be deleted. You can however uncheck the box next to the default path if you do not require the default repository in your new project.

4.17.6 Suppressed Messages

Use this section to suppress display of the selected messages. See Figure 4-40 on page 4-59.

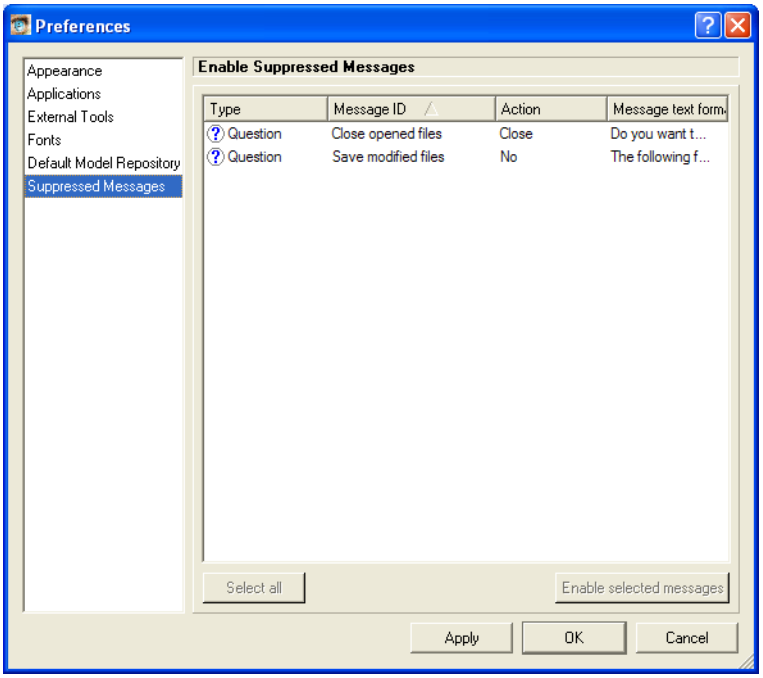

Figure 4-40 Preferences dialog, Suppressed Messages

To enable a suppressed message, select it in the list, then click **Enable selected messages**.

4.18 Project Settings dialog

Click the **Settings** button to display the project settings and customize the generation process or edit the LISA code of components. The default view is shown in Figure 4-41.

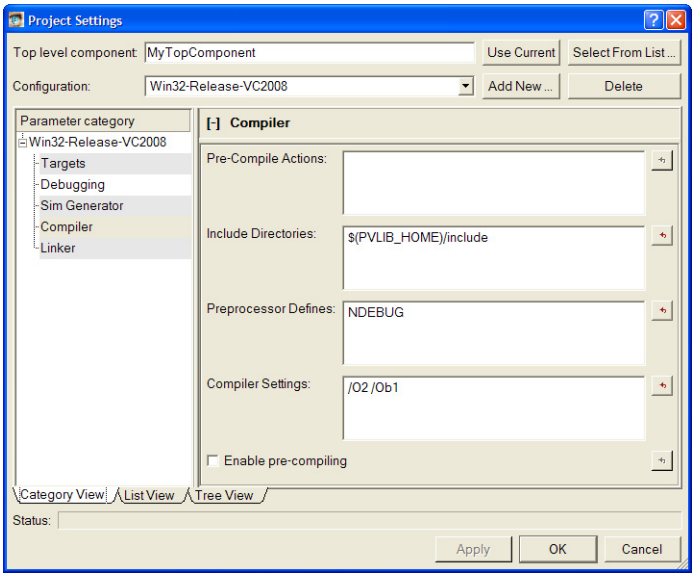

Figure 4-41 Project Settings dialog

4.18.1 Project top-level settings

The top part of the dialog enables you to configure project build options:

Top level component

The Top Level Component defines the root component of the system. Each component in a system can be set as the top component to build models from functional sub-blocks of a system.

The top level component can be set in the Project Settings dialog by one of the following methods:

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

Configuration

The `Configuration` control enables you to set parameters for the different configurations:

- Selecting an entry listed in **Configuration** drop-down list to use one of the existing configurations.
- Click **Add New** to create and new configuration.

  A dialog prompts for the name of the new configuration and an optional description. You can use the **Copy values from** button to select a configuration to copy the values from. This can be one of the existing configurations or a default set of configuration settings.

- Click **Delete** to delete the currently-selected configuration from the list of available configurations.

The value defaults to the currently-selected active configuration.

Selecting a configuration in this dialog does not set the configuration displayed in the **Select Active Project Configuration** drop-down box on the main window. The configuration set in this dialog is stored in the project file and is used if the system is rebuilt with the specified configuration. You can use this control to specify the options for all of the configurations that you use for a project and simplify switching from one active configuration to another.

Note

If you are building your system on Microsoft Windows, you must ensure that anyone else using your model system has the appropriate additional support libraries installed on their computer, otherwise the system fails to run:

- For a debug build, the appropriate version of Microsoft Visual Studio.
- For a release build, the appropriate version of the Microsoft Visual Studio redistributable package.

4.18.2 Parameters category

The **Parameter category** panel lists parameters for the selected build option. For information about the different display options for parameters in the **Category View** tab, see:

- *Category View* on page 4-62
- *List View* on page 4-67
- *Tree View* on page 4-68.

For more information on the parameters, see *Alphabetical list of project parameter IDs* on page 4-69.

To set the release options for example:

1. Click the **Category View** tab.

   Select the **Windows-Release** entry and select the operating system and link options from the **Platform/Linkage** dropdown menu:

   - **Linux** builds a 32-bit model for Linux
   - **Linux64** builds a 64-bit model for Linux
   - **Win32** builds a 32-bit model using the release run-time library for Microsoft Windows
   - **Win32D** builds a 32-bit model using the debug run-time library for Microsoft Windows
   - **Win64** builds a 64-bit model using the release run-time library for Microsoft Windows
   - **Win64D** builds a 64-bit model using the debug run-time library for Microsoft Windows.

2. Select the compiler to use for this configuration from the **Compiler** dropdown menu.

3. Enter a path into the **Build directory** field to select the directory where the build is performed. This directory contains the source code and the build library for the system model. If the path is not absolute, it is interpreted as being relative to the directory that contains the project file.

4. The **Configuration description** text box contains text that indicates the configuration options selected for release, compiler, and build directory.

Category View

Click the **Category View** tab to display the categories in the left pane and text boxes for the options in the right pane. Select an item in the **Parameter Category** view to display the controls for configuring the corresponding parameters:

Top-level configuration details

Select the top-most item in the panel to configure the project settings:

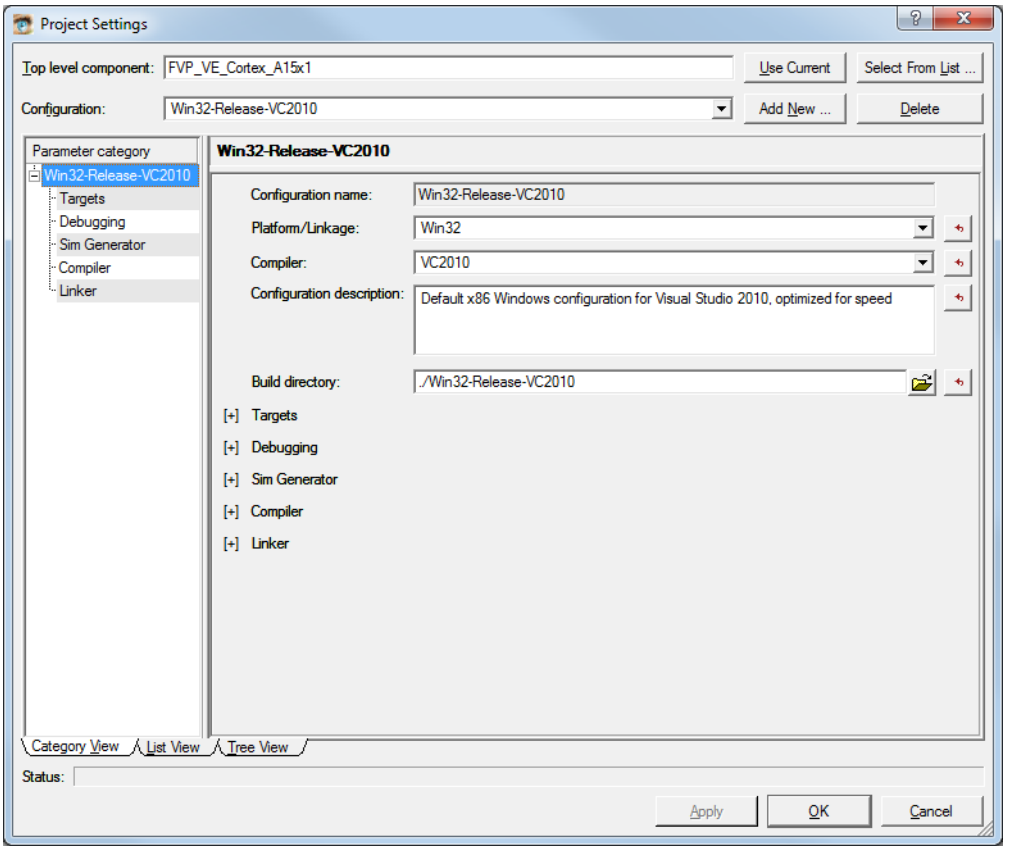

Figure 4-42 Project Settings dialog, Category View top level

The controls in the right panel correspond to the parameters listed in Table 4-4:

Table 4-4 Release parameters in the Category View tab

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

See *Alphabetical list of project parameter IDs* on page 4-69 for a description of the available parameters.

Targets

Select the **Targets** item in the panel to configure the build targets:

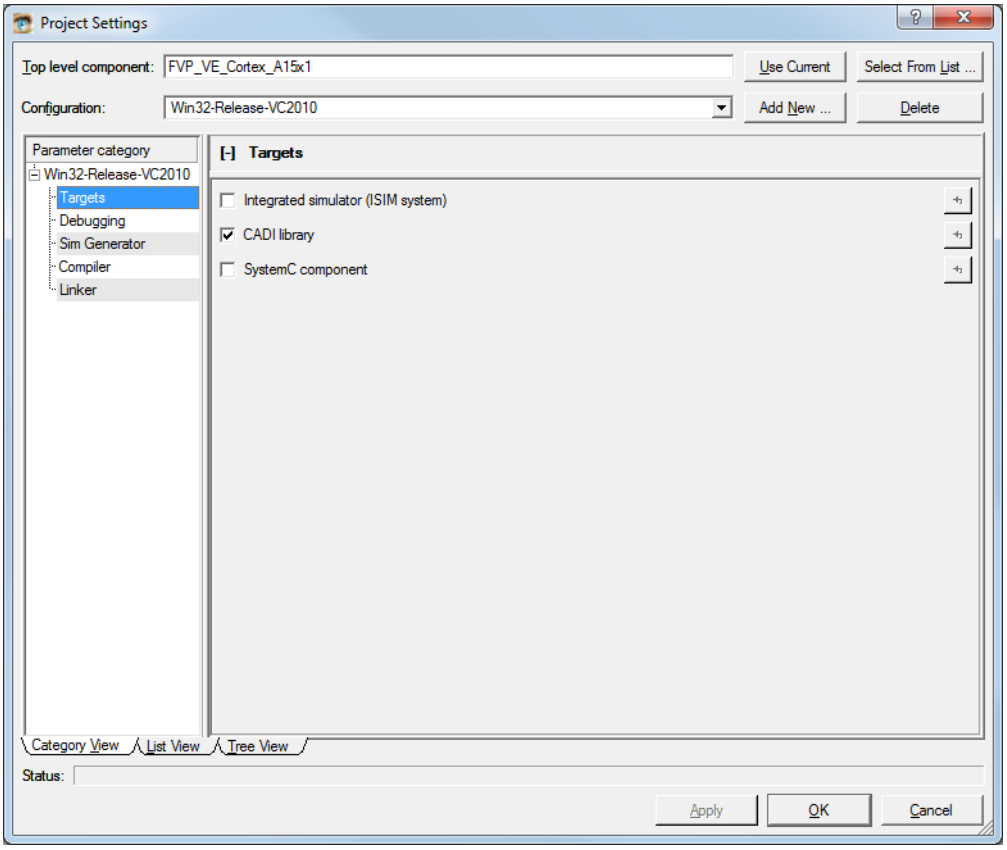

Figure 4-43 Project Settings dialog, Category View Targets

The controls in the right panel correspond to the parameters listed in Table 4-5:

Table 4-5 Target parameters in the Category View tab

Control name	Parameter
Integrated simulator (isim_system)	TARGET_ISIM
CADI library	TARGET_MAXVIEW
SystemC component	TARGET_SYSTEMC

See *Alphabetical list of project parameter IDs* on page 4-69 for a description of the available parameters.

Debugging

Select the **Debugging** item in the panel to configure the debug options:

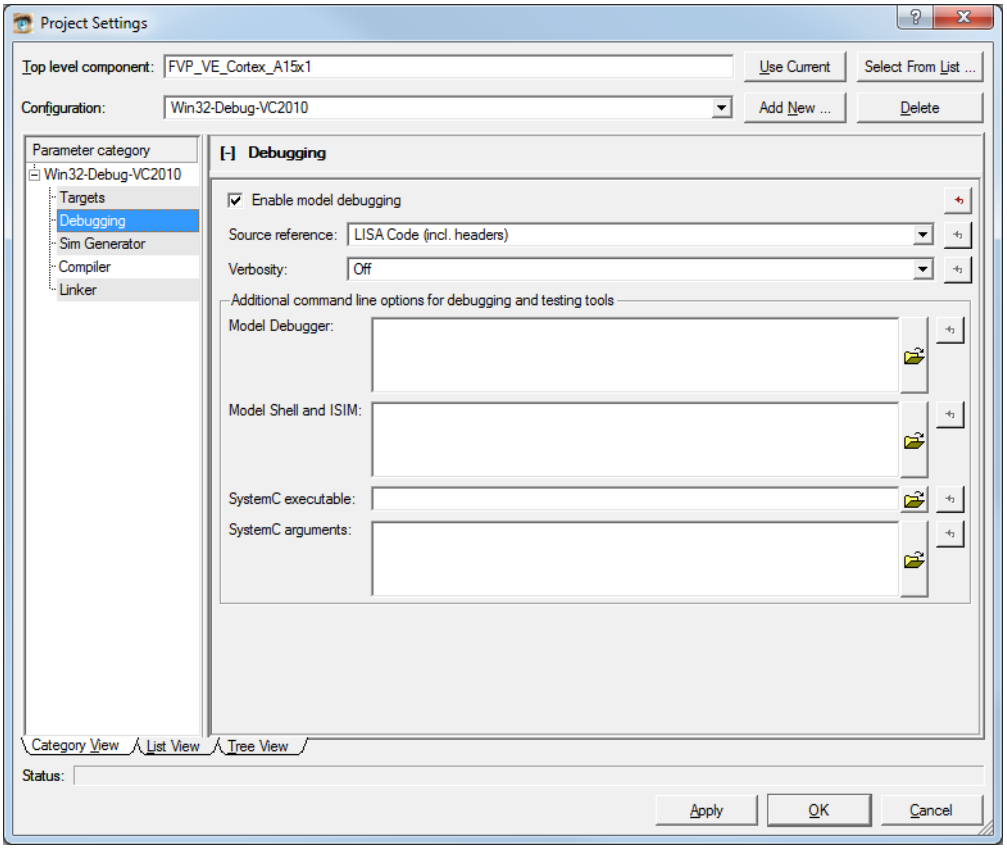

Figure 4-44 Project Settings dialog, Category View Debugging

The controls in the right panel correspond to the parameters listed in Table 4-6:

Table 4-6 Debugging parameters in the Category View tab

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

See *Alphabetical list of project parameter IDs* on page 4-69 for a description of the available parameters.

Sim Generator

Select the **Sim Generator** item in the panel to configure the debug options:

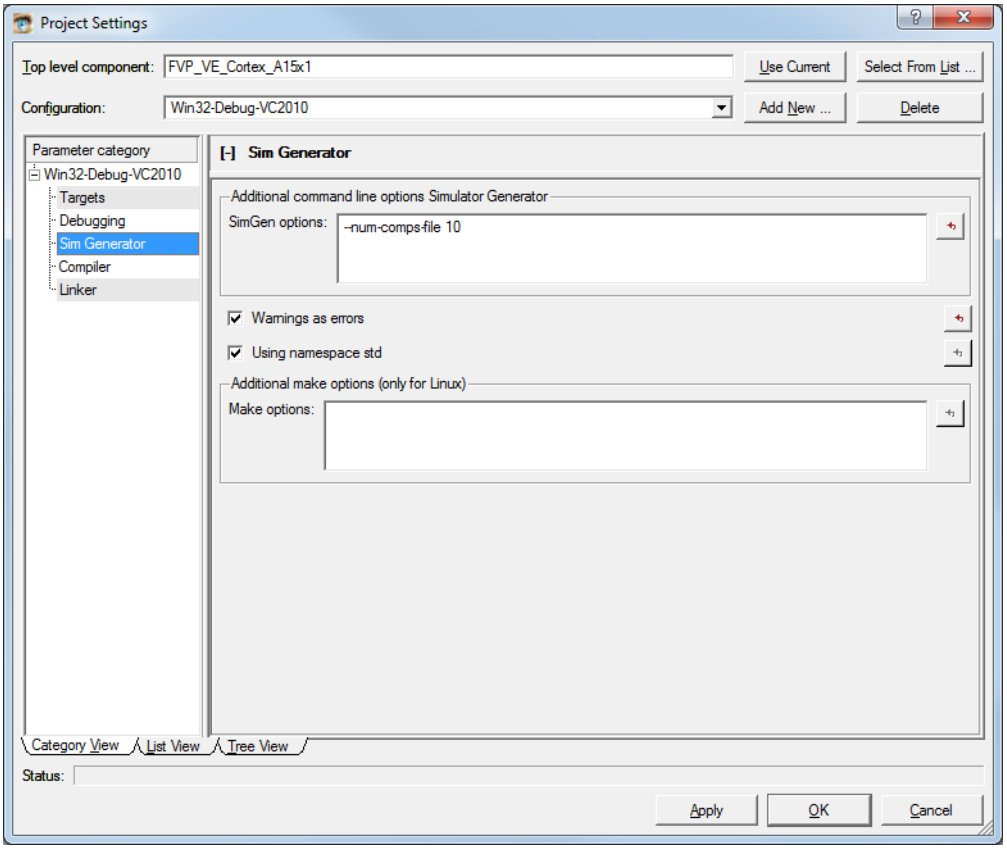

Figure 4-45 Project Settings dialog, Category View Sim Generator

The controls in the right panel correspond to the parameters listed in Table 4-7:

Table 4-7 Sim Generator parameters in the Category View tab

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 options:

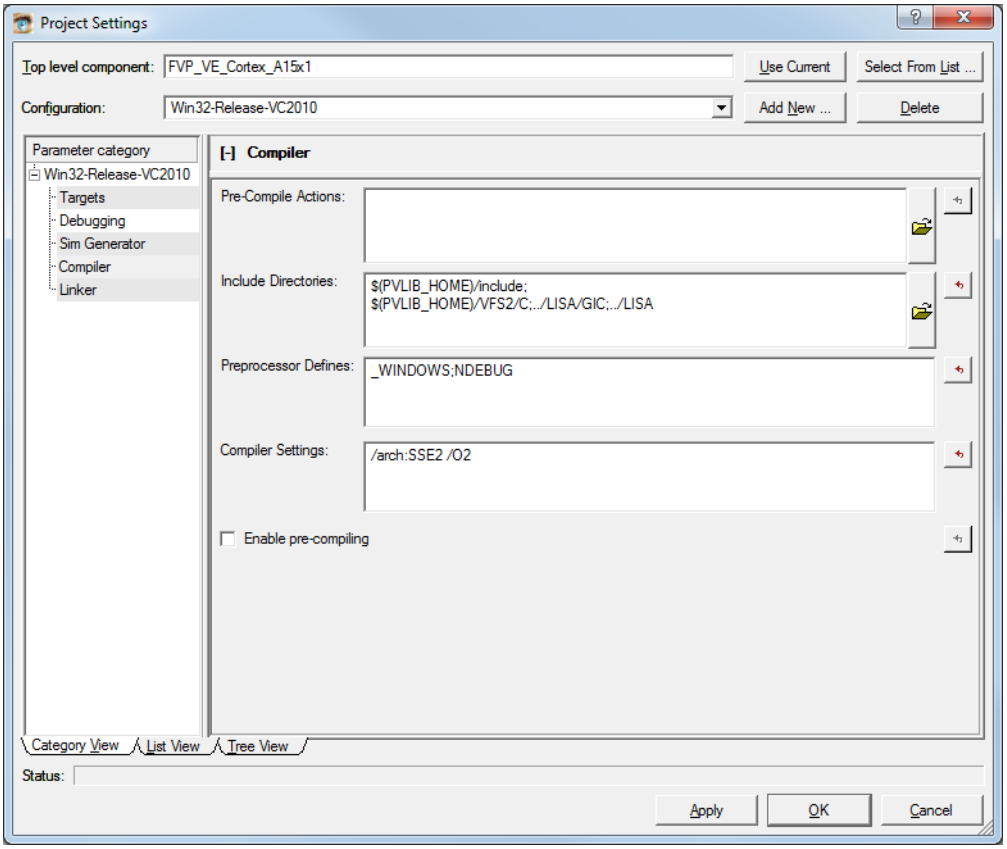

Figure 4-46 Project Settings dialog, Category View Compiler

The controls in the right panel correspond to the parameters listed in Table 4-8:

Table 4-8 Compiler parameters in the Category View tab

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

See *Alphabetical list of project parameter IDs* on page 4-69 for a description of the available parameters.

Linker

Select the **Linker** item in the panel to configure the linker options:

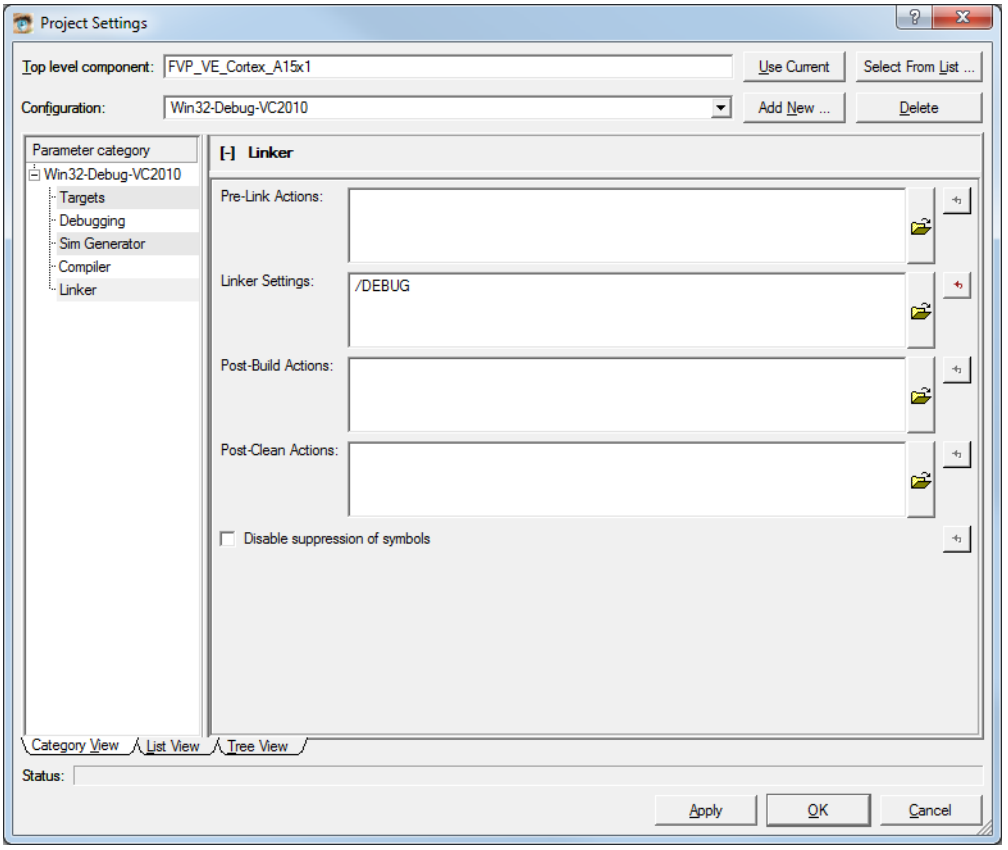

Figure 4-47 Project Settings dialog, Category View Linker

The controls in the right panel correspond to the parameters listed in Table 4-8 on page 4-66:

Table 4-9 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

See *Alphabetical list of project parameter IDs* on page 4-69 for a description of the available parameters.

List View

Click the **List View** tab to display all categories and their parameter values in a single flat list. See *Alphabetical list of project parameter IDs* on page 4-69 for a description of the available parameters.

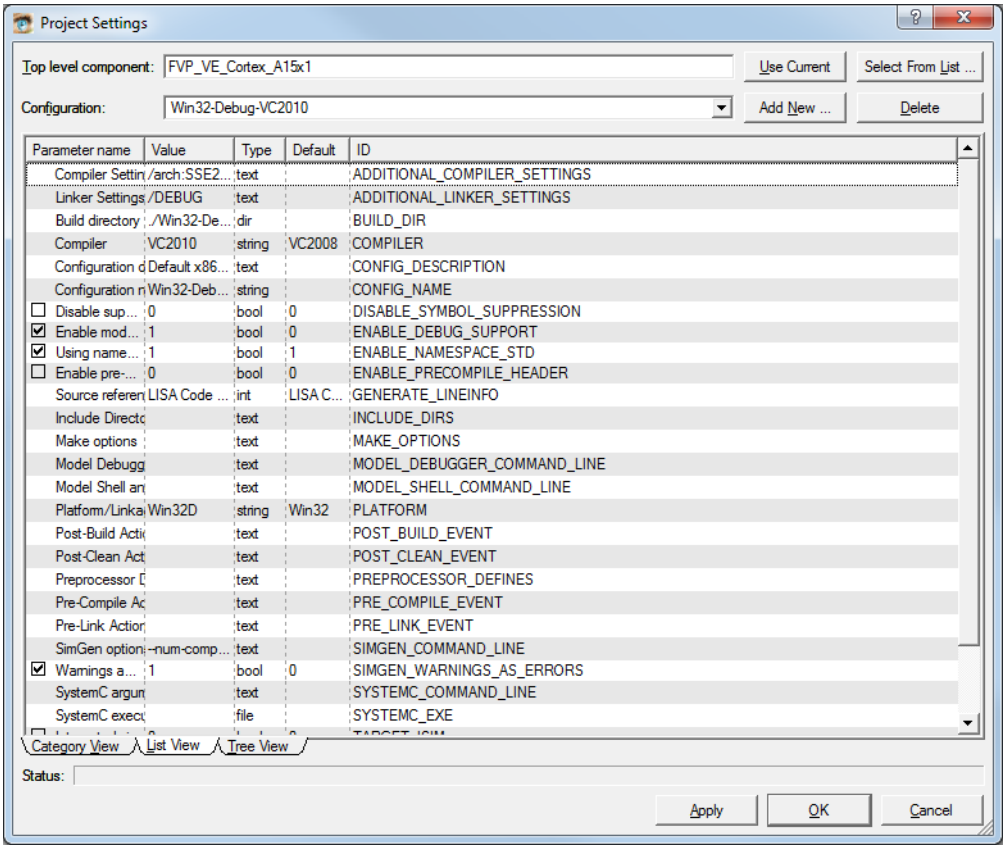

Figure 4-48 Project Settings dialog, List View

Tree View

Click the **Category View** tab to display all categories and their parameter values in a single hierarchical list. See *Alphabetical list of project parameter IDs* on page 4-69 for a description of the available parameters.

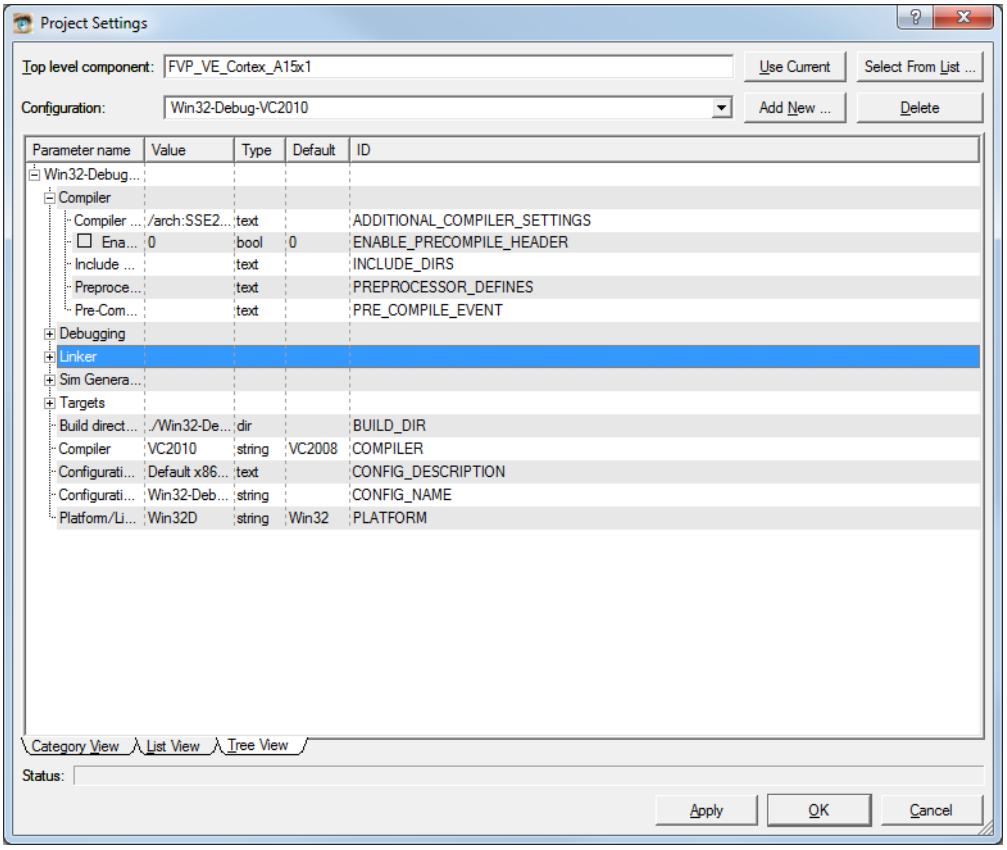

Figure 4-49 Project Settings dialog, Tree View

4.18.3 Alphabetical list of project parameter IDs

The configurable parameters for a project configuration are listed in Table 4-10.

Table 4-10 Full list of parameters shown in List View

Parameter name	Parameter ID	Default	Description
Compiler Settings	`ADDITIONAL_COMPILER_SETTINGS`	`""`	Operating system-specific additional compiler settings. For Microsoft Windows, consult the Visual Studio documentation on the compiler options.
Linker Settings	`ADDITIONAL_LINKER_SETTINGS`	`""`	Linker settings specified here are added to the default linker flags. For Microsoft Windows, consult the Visual Studio documentation on the linker options.
Build directory	`BUILD_DIR`	`""`	Build directory. If this is not an absolute path, it is relative to the position of the project file. For Microsoft Windows, this is typically either `./Windows-Debug` or `./Windows-Release`.

Table 4-10 Full list of parameters shown in List View (continued)

Parameter name	Parameter ID	Default	Description
Compiler	COMPILER	""	Compiler to use for this model: "VC2008" for Microsoft Visual Studio 2008 SP1 "VC2010" for Microsoft Visual Studio 2010 SP1 "gcc" for first gcc found in search path "gcc-4.1" for gcc version 4.1.2 "gcc-4.4" for gcc version 4.4.6
Configuration description	CONFIG_DESCRIPTION	""	More descriptive form of the configuration specified in CONFIG_NAME.
Configuration name	CONFIG_NAME	""	Name of the configuration.
Enable model debugging	ENABLE_DEBUG_SUPPORT	"0"	Setting this to 1 enables implementation defined debug support for debugging the model.
Enable namespace std	ENBALE_NAMESPACE_STD	"1"	Enable use of namespace std: • if 1 (true), using namespace std is generated and placed in the code • if 0 (false), you must specify the namespace. A user-specified namespace might reduce compilation time.
Enable pre-compiling	ENBALE_PRECOMPILE_HEADER	"0"	Enable pre-compiling of headers if true/1.
Source Reference	GENERATE_LINEINFO	"LISA Code (incl headers)"	Controls line redirection in the generated model source code: "LISA Code": source code only "LISA Code (incl. headers)": Source and header "Generated Code": No line redirection at all.
Additional Include Directories	INCLUDE_DIRS	""	Additional include directories. Multiple entries must be separated by a semicolon.
Model Debugger	MODEL_DEBUGGER_COMMAND_LINE	""	Additional options to be passed on the command line.
Model Shell	MODEL_SHELL_COMMAND_LINE	""	Additional options to be passed on the command line.
Platform/Linkage	PLATFORM	-	Platform for which this configuration is valid. One of "Linux", "Linux64", "Win32", "Win32D", "Win64", "Win64D".
Post-Built Actions	POST_BUILD_EVENT	""	Commands executed after building the model. Multiple commands must be separated by a semicolon.
Pre-Compile Actions	PRE_COMPILE_EVENT	""	Commands executed before compilation is started. Applies to Microsoft Windows only. Multiple commands must be separated by a semicolon.
Preprocessor Defines	PREPROCESSOR_DEFINES	""	Preprocessor defines. Multiple defines must be separated by semicolon.
Pre-Link Actions	PRE_LINK_EVENT	""	Commands executed before linking is started. Applies to Microsoft Windows only. Multiple commands must be separated by a semicolon.
SimGen options	SIMGEN_COMMAND_LINE	""	Additional options to be passed on the command line.

Table 4-10 Full list of parameters shown in List View (continued)

Parameter name	Parameter ID	Default	Description
Integrated simulator (isim_system)	TARGET_ISIM	"0"	If 1 (true), builds an executable with a statically-linked CADI system.
CADI library	TARGET_MAXVIEW	"1"	If 1 (true), build a CADI system dynamic library for running from Model Debugger.
SystemC component	TARGET_SYSTEMC	"0"	If 1 (true), build a SystemC component library.
Verbosity	VERBOSITY	"Off"	Verbosity level. Must be either "Sparse", "On", or "Off".
Warnings as errors	SIMGEN_WARNINGS_AS_ERRORS	"1"	If 1 (true), treat LISA parsing and compiler warnings as errors
SystemC arguments	SYSTEMC_COMMAND_LINE	""	Command line arguments for System C executable
SystemC executable	SYSTEMC_EXE	""	Name of final SystemC Executable. This executable is called by 'Run SystemC Executable'.

4.18.4 Project file contents

The project file is generated from the specified configuration options and describes the options used for each platform and the files that are required to build the model. File and directory names can be either absolute or relative to the project or repository file. You can use environment variables in filenames.

Caution

Files and directories are processed in the order they are encountered in the project file. For more information, see *File processing order* on page 1-7.

An individual file can have both file and directory actions associated with it. See *Directories in path statements* on page 4-73.

Japanese or Korean characters must not be used in project filename paths, as they can lead to failure to find the specified libraries.

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

Host platform

Specifies the operating system as:

- "Linux" for a 32-bit version of the build
- "Linux64" for a 64-bit version of the build
- "Win32" for a 32-bit release version of the build
- "Win32D" for a 32-bit debug version of the build
- "Win64" for a 64-bit release version of the build
- "Win64D" for a 64-bit debug version of the build.

Compiler Specifies the compiler to use as:

- "VC2008" for Microsoft Visual Studio 2008 SP1
- "VC2010" for Microsoft Visual Studio 2010 SP1
- "gcc" for gcc version found in search path (Linux)

- "gcc-4.1" for GCC 4.1.2, "gcc-4.4" for GCC 4.4.6 (Linux)

Additional compiler options might be added in future product versions.

Note

For Linux, the chosen compiler version only effects the files identified by the project file and repositories as described in *File/Path Properties dialog* on page 4-36. It does not select the gcc version used to build the model. The Simulator Generator always uses the gcc found in the search path instead. To enable the Simulator Generator to automatically select the libraries that match the current gcc compiler, use the compiler option `gcc`.

Note

For Microsoft Visual Studio 2008 and Visual Studio 2010, you must have Service Pack 1 (SP1) and the latest security upgrades installed.

Action Specifies the action to take for the specified file or directory as one or more of:

- "lisa" to process the file as a LISA file. This action cannot be applied to directories.
- "compile" to process the file as a C++ file

  If a directory is specified, all *.c, *.cpp, and *.cxx files in the directory are compiled.
- "ignore" to exclude the file or directory from the build and deploy process, such as a disabled file or project notes.
- "link" to link the file with existing files.

  If a directory is specified, on Microsoft Windows all *.lib and *.obj files in the directory are added to the linker input. On Linux, all *.a and *.o files are added.
- "deploy" to produce a deployable file.

  If a directory is specified, the entire directory and its subdirectories are recursively copied to the destination.

  Note

  "deploy" is the only action that acts recursively on subdirectories.
- "incpath" to include the directory in the list of include search paths specified by the -I option for the compiler. This is the default action for a directory.
- "libpath" to include the directory in the list of library search paths specified by the -L option for the compiler.

  Note

  "libpath" 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="WIN32"|"Win32D", compiler="VC2010", action="link"|"deploy";
```

The compiler options cannot be ORed together, but they can be omitted to indicate that more than one compiler is permitted:

```
path = ../src/my_windows_code.cpp, platform = "win32";
```

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

```
path = ../lib/release_2008/my_lib.lib, platform = "win32", compiler="VC2008";
path = ../lib/my_lib.lib, platform = "win32", compiler="VC2010";
path = ../src/my_windows_code.cpp, platform = "win32"; // both compilers can be used
```

Directories in path statements

Project files can contain directories in the `path` statement. Directories are differentiated from normal files by including the trailing / character. Platform and compiler filters might apply.

If an action is specified for a file that indicates an action associated with a directory, the directory path is formed by removing the filename from the full path.

The path specification:

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

causes `MyFile.lisa` to be used as the LISA source and the parent directory for `MyFile.lisa` to be added to both the include and library search paths.

If, for example, `MyFile.lisa` is in the directory `C:/ARM/MyProjects/Project_1/`, that directory path is added to the include and library search paths.

Example project file

Example 4-1 lists a typical project file and shows how the different configuration sections are used.

Example 4-1 Typical project file

```
sgproject "VP_DualCore.sgproj"
{
TOP_LEVEL_COMPONENT = "DualCoreSystem";
ACTIVE_CONFIG_LINUX = "Linux-Release-GCC-4.4";
ACTIVE_CONFIG_WINDOWS = "Win32-Release-VC2010";
config "Linux-Debug-GCC-4.1"
{
        ADDITIONAL_COMPILER_SETTINGS = "-march=pentium-m -mtune=nocona -mfpmath=sse -msse2 -ggdb3 -Wall";
        BUILD_DIR = "./Linux-Debug-GCC-4.1";
        COMPILER = "gcc-4.1";
        CONFIG_DESCRIPTION = "Default Linux configuration for GCC 4.1 with debug info support";
        CONFIG_NAME = "Linux-Debug-GCC-4.1";
        ENABLE_DEBUG_SUPPORT = "1";
        PLATFORM = "Linux";
        SIMGEN_COMMAND_LINE = "--num-comps-file 10";
}
config "Linux-Release-GCC-4.1"
{
        ADDITIONAL_COMPILER_SETTINGS = "-march=pentium-m -mtune=nocona -mfpmath=sse -msse2 -O3
-fomit-frame-pointer -Wall";
        BUILD_DIR = "./Linux-Release-GCC-4.1";
        COMPILER = "gcc-4.1";
        CONFIG_DESCRIPTION = "Default x86 Linux configuration for GCC 4.1, optimized for speed";
        CONFIG_NAME = "Linux-Release-GCC-4.1";
        PLATFORM = "Linux";
        PREPROCESSOR_DEFINES = "NDEBUG";
        SIMGEN_COMMAND_LINE = "--num-comps-file 50";
}
config "Linux-Debug-GCC-4.4"
{
        ADDITIONAL_COMPILER_SETTINGS = "-march=pentium-m -mtune=generic -mfpmath=sse -msse2 -ggdb3 -Wall";
        BUILD_DIR = "./Linux-Debug-GCC-4.4";
        COMPILER = "gcc-4.4";
        CONFIG_DESCRIPTION = "Default x86 Linux configuration for GCC 4.4 with debug information";
        CONFIG_NAME = "Linux-Debug-GCC-4.4";
        ENABLE_DEBUG_SUPPORT = "1";
        PLATFORM = "Linux";
        SIMGEN_COMMAND_LINE = "--num-comps-file 10";
}
config "Linux-Release-GCC-4.4"
{
        ADDITIONAL_COMPILER_SETTINGS = "-march=pentium-m -mtune=nocona -mfpmath=sse -msse2 -O3
-fomit-frame-pointer -Wall";
        BUILD_DIR = "./Linux-Release-GCC-4.4";
        COMPILER = "gcc-4.1";
        CONFIG_DESCRIPTION = "Default x86 Linux configuration for GCC 4.4, optimized for speed";
        CONFIG_NAME = "Linux-Release-GCC-4.4";
        PLATFORM = "Linux";
        PREPROCESSOR_DEFINES = "NDEBUG";
        SIMGEN_COMMAND_LINE = "--num-comps-file 50";
}
config "Linux64-Debug-GCC-4.4"
{
        ADDITIONAL_COMPILER_SETTINGS = "-ggdb3 -Wall";
```

```
        BUILD_DIR = "./Linux64-Debug-GCC-4.4";
        COMPILER = "gcc-4.4";
        CONFIG_DESCRIPTION = "Default x86_64 Linux configuration for GCC 4.4 with debug information";
        CONFIG_NAME = "Linux64-Debug-GCC-4.4";
        ENABLE_DEBUG_SUPPORT = "1";
        PLATFORM = "Linux64";
}
config "Linux64-Release-GCC-4.4"
{
        ADDITIONAL_COMPILER_SETTINGS = "-O3 -Wall";
        BUILD_DIR = "./Linux64-Release-GCC-4.4";
        COMPILER = "gcc-4.4";
        CONFIG_DESCRIPTION = "Default x86_64 Linux configuration for GCC 4.4, optimized for speed";
        CONFIG_NAME = "Linux64-Release-GCC-4.4";
        PLATFORM = "Linux64";
        PREPROCESSOR_DEFINES = "NDEBUG";
}
config "Win32-Debug-VC2008"
{
        ADDITIONAL_COMPILER_SETTINGS = "/arch:SSE2 /Od /RTCsu /Zi";
        ADDITIONAL_LINKER_SETTINGS = "/DEBUG";
        BUILD_DIR = "./Win32-Debug-VC2008";
        COMPILER = "VC2008";
        CONFIG_DESCRIPTION = "Default x86 Windows configuration for Visual Studio 2008 with debug information";
        CONFIG_NAME = "Win32-Debug-VC2008";
        ENABLE_DEBUG_SUPPORT = "1";
        PLATFORM = "Win32D";
        SIMGEN_COMMAND_LINE = "--num-comps-file 10";
}
config "Win32-Release-VC2008"
{
        vADDITIONAL_COMPILER_SETTINGS = "/arch:SSE2 /O2";
        BUILD_DIR = "./Win32-Release-VC2008";
        COMPILER = "VC2008";
        CONFIG_DESCRIPTION = "Default x86 Windows configuration for Visual Studio 2008, optimised for speed";
        CONFIG_NAME = "Win32-Release-VC2008";
        PLATFORM = "Win32";
        SIMGEN_COMMAND_LINE = "--num-comps-file 50";
}
config "Win32-Debug-VC2010"
{
        ADDITIONAL_COMPILER_SETTINGS = "/arch:SSE2 /Od /RTCsu /Zi";
        ADDITIONAL_LINKER_SETTINGS = "/DEBUG";
        BUILD_DIR = "./Win32-Debug-VC2010";
        COMPILER = "VC2010";
        CONFIG_DESCRIPTION = "Default x86 Windows configuration for Visual Studio 2010 with debug information";
        CONFIG_NAME = "Win32-Debug-VC2010";
        ENABLE_DEBUG_SUPPORT = "1";
        PLATFORM = "Win32D";
}
config "Win32-Release-VC2010"
{
        ADDITIONAL_COMPILER_SETTINGS = "/arch:SSE2 /O2";
        BUILD_DIR = "./Win32-Release-VC2010";
        COMPILER = "VC2010";
        CONFIG_DESCRIPTION = "Default x86 Windows configuration for Visual Studio 2010, optimised for speed";
        CONFIG_NAME = "Win32-Release-VC2010";
        PLATFORM = "Win32";
}
files
{
        path = "../LISA/DCVisualisation.lisa";
        path = "../LISA/DualCoreSystem.lisa";
```

```
        path = "$(PVLIB_HOME)/etc/sglib.sgrepo";
        path = "../../Common/LISA/common.sgrepo";
}
}
```

4.19 Protocol Properties dialog

Select a protocol from the **Protocols** list and right-click and select **Properties** to display the properties as shown in Figure 4-50 for the selected protocol:

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.

Info Descriptive information that was added when the file was added to the project.

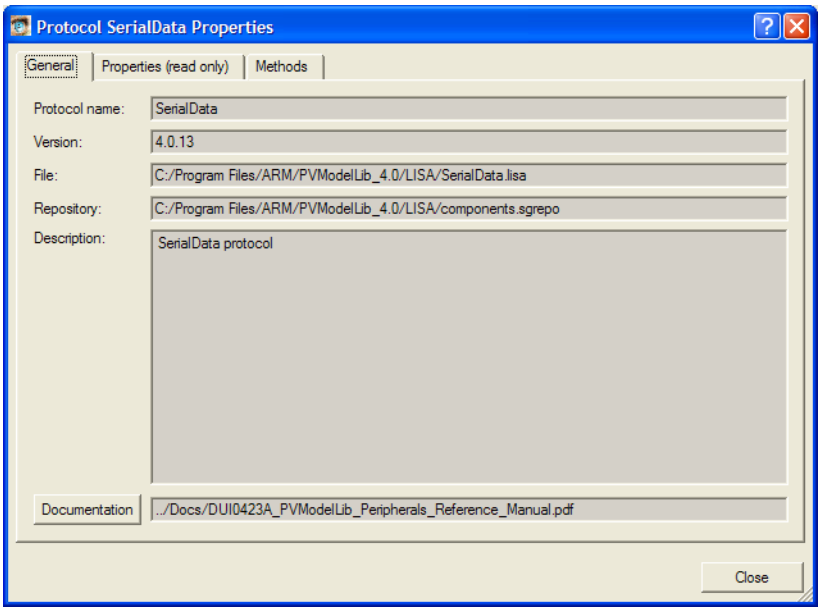

Figure 4-50 Protocol Properties dialog for SerialData

Click the **Methods** tab to display the LISA prototypes of methods, or behaviors, available for the protocol. The displayed values are for reference only and cannot be edited.

Click the **Properties** tab to display the properties for protocol. The displayed values are for reference only and cannot be edited.

4.20 Run Dialog

The Run Dialog can be used to specify the actions that are executed to run a selected target. This section describes the actions for the different targets and explains the additional options.

To display the dialog in Figure 4-51 click **Run** from the **Project** menu.

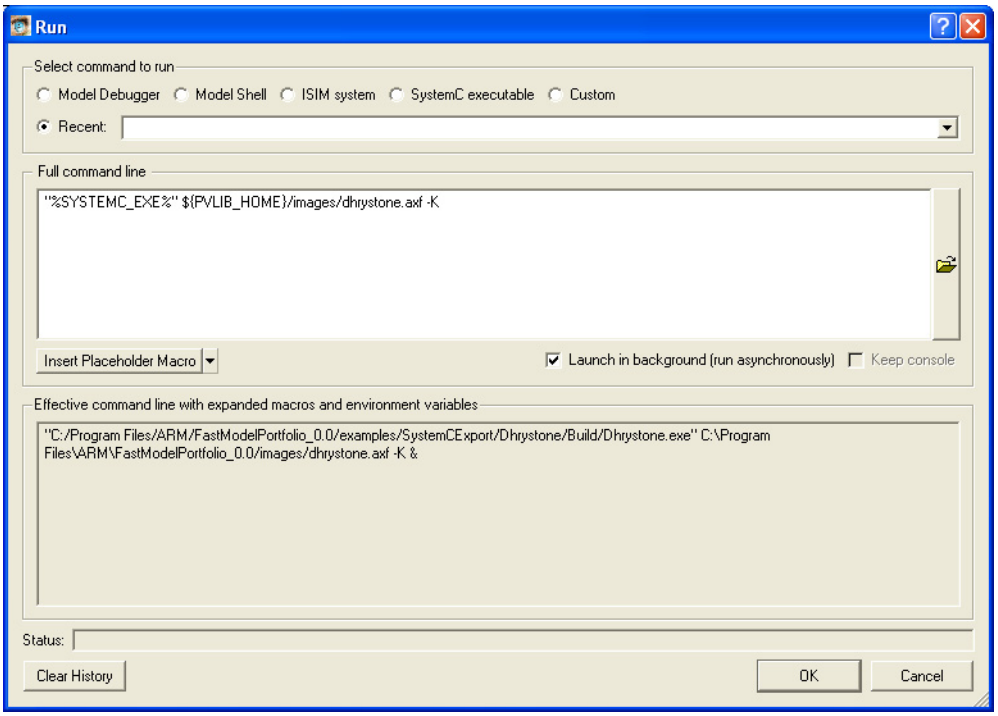

Figure 4-51 Run Dialog

The dialog contains of the following areas:

Select command to run

Select which executable you want to run.

Full Command Line

This field permits adjustment of the command line generated by System Canvas. You might specify additional parameters or change the location of the application which is loaded onto the executable.

Effective Command Line

Shows the complete command line to be executed with expanded macros and environment variables.

The dialog has the following controls:

Model Debugger

Click to run model in Model Debugger. The initial command line options are taken from project settings and user preferences. You can alter the options in Full Command Line.

Model Shell

Click to run model with Model Shell. The initial command line options are taken from project settings and user preferences. You can alter the options in Full Command Line.

ISIM system

Click to run model as ISIM system. The initial command line options are taken from project settings and user preferences. You can alter the options in Full Command Line.

SystemC executable

Click to run SystemC executable specified in configuration setting SYSTEMC_EXE and SYSTEMC_COMMAND_LINE. You can alter the options in Full Command Line.

Custom

Specify arbitrary command lines from scratch in Full Command Line.

Recent

Select recent command.

Insert Placeholder Macro

Insert macro or environment variable from drop-down list at the current cursor position in Full Command Line. Those macros are expanded to build the complete command line.

The following macros are supported:

`%CADI%`
: full absolute path of CADI dynamic library

`%ISIM%`
: full absolute path of ISIM executable

`%SYSTEMC_EXE%`
: full SYSTEMC_EXE executable pathname

`%BUILD_DIR%`
: relative path to build directory (relative to project path)

`%DEPLOY_DIR%`
: relative path to deploy directory (currently identical to %BUILD_DIR%)

`%PROJECT_DIR%`
: full absolute path to the directory of the project.

Launch in background

Run application asynchronously in a separate console window. Use this if the application requests user input or a big amount of output is expected.

Clear History

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

4.21 Self Port dialog

Use the Self Port dialog to add a port to the top-level component. To display the dialog in Figure 4-52, without having anything selected in the Block Diagram view click **Add Ports**, or click **Add Port** from the **Object** menu.

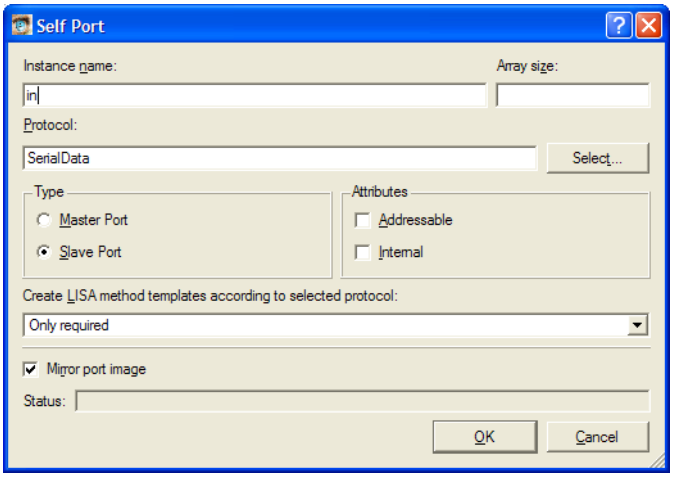

Figure 4-52 Self Port dialog

The dialog has the following controls:

Instance name Specify the name of the port.

Array Size Enter the number of ports is the new port is to be a port array. Leave the box empty, or enter 1, for normal ports.

Protocol Enter the name of the protocol for the port. To display a list of available protocols, click the **Select...** button.

Type Select **Master port** or **Slave Port**.

Attributes If required, specify the optional attributes for the port:

- **Addressable** for bus ports
- **Internal** for ports between subcomponents. The port is not visible if the current 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.

Only methods corresponding to the selected port type, that is, for either master or slave, are created.

Editing the existing port might result in the creation of new methods, but existing methods are not deleted.

Mirror port image

Reverse the direction of the port.

Chapter 5
SystemC Export

This chapter describes how to export a Fast Models system as a SystemC component. It contains the following sections:

Note

This chapter presumes the use of Single Instance export. ARM deprecates Single Instance export, but ARM recommends Multiple Instantiation export instead. See *SystemC Export with Multiple Instantiation*, installed with your software at `$PVLIB_HOME/Docs/GENC-010823_mi_ug.pdf`.

5.1 About SystemC export

The SystemC export feature of System Canvas generates a SystemC component that contains and wraps a Fast Models system. This involves two different simulation domains:

- Exported Virtual Subsystem (EVS), this is a Fast Model Virtual Platform exported into SystemC using the SystemC export feature from System Canvas.
- SystemC components.

The SystemC export feature generates one component and generates a SystemC wrapper for this component. The resulting component is the EVS. SystemC 2.2, TLM 2.0 and TLM 2.0.1 are supported by Fast Models.

Figure 5-1 shows how the Fast Models virtual platform interacts as an `SC_THREAD` with the SystemC master scheduler and communicates with other SystemC threads or modules.

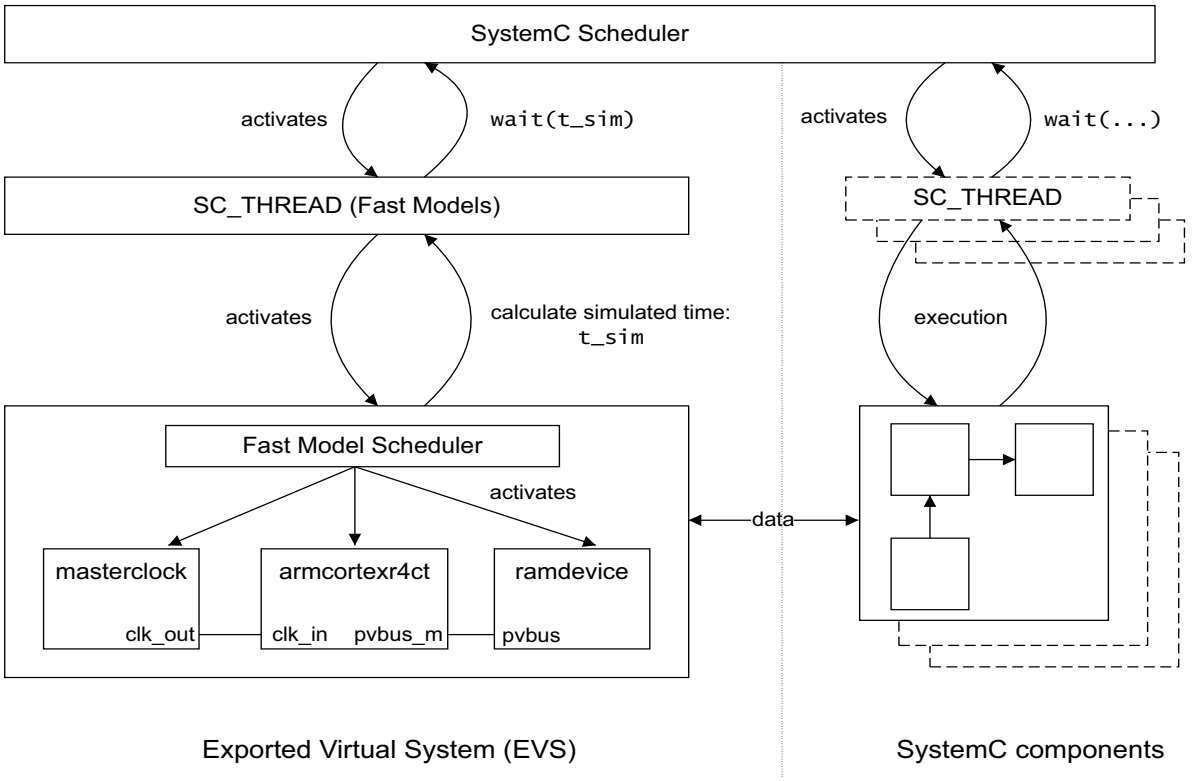

Figure 5-1 Scheduling between Fast Models and SystemC domains

The SystemC scheduler uses the `wait()` command to track the simulated time reported by the `SC_THREAD` and ensures that, over a long time period, all components are activated to simulate the same amount of time.

After the `SC_THREAD` (that encapsulates the Fast Models virtual platform) is activated, the Fast Models scheduler is invoked. It activates all components that are part of the subsystem until the (configurable) time quantum is finished. The actual time that has passed is calculated and the SystemC scheduler is notified.

Data can be exchanged between components in the Fast Models platform and other SystemC components by using either the *Direct Memory Interface* (DMI) or normal (blocking) transactions.

Note

The examples in this chapter assume the following:

- `main.cpp` is the file where the generated EVS, in addition to other SystemC components, are instantiated.
- `MySystem` is the name of the top level component of the Fast Models system
- `AConfig` is the build configuration name, for example, `Linux-Release-GCC-4.1` or `Win32-Release-VC2008`.

To export your own component, replace `main.cpp`, `MySystem`, and `AConfig` with the appropriate names for your model.

5.2 Building a SystemC component from System Canvas

This section describes prerequisites and build instructions to create an EVS from System Canvas.

5.2.1 Installing SystemC and TLM files

To build an EVS (SystemC Export) from the Fast Models platform, SystemC and TLM header files and libraries are required. This can be either done by installing the Third Party IP (TPIP) package which is available as a separate download package from the same location as the Fast Models package or using already installed packages on your site.

The Third Party IP package contains binary releases of the SystemC and TLM packages.The SystemC binary packages are link-compatible with the OSCI download version available from OSCI , `http://www.systemc.org`.

Note

The Fast Models SystemC examples rely on a certain directory structure for libraries and header files. This is different from the package that you can obtain from the OSCI web site because Fast Models supports a wider range of compilers than the OSCI package. If you want to use the original OSCI package in conjunction with the Fast Models SystemC examples, a set of patch files is required that is applied to the package that adjusts the directory names. To re-build the packages, follow the instructions from the README.txt file available in the `<TPIP>/OSCI/` source directory.

5.2.2 Building the EVS

1. System Canvas automatically uses the following include path:
   - `$(SYSTEMC_HOME)/include`
   - `$(TLM_HOME)/include/tlm`

   The `SYSTEMC_HOME` and `TLM_HOME` environment variables are set when the TPIP package is installed. This package is available as a separate download from the same location as the Fast Models package. If you require a different copy of SystemC or TLM, modify the variables before starting System Canvas.
2. In System Canvas, enable the target option **SystemC component** under the menu entry **Project → Project Settings → Targets**.
3. Add the ports you require on the generated SystemC component as normal Fast Models ports. See *Using the generated ports* on page 5-12 for more information on using ports.
4. Add the headers and library files specified for your operating system. See *Adding header files and libraries for Linux export* on page 5-6 or *Adding header files and libraries for Microsoft Windows export* on page 5-7.
5. Build the SystemC component. The output is:
   - static library `MySystem-sc_sg_wrapper-AConfig.a`.
     For Microsoft Windows, the library is `MySystem-sc_sg_wrapper-AConfig.lib`.
   - header file `sc_sg_wrapper_MySystem.h` in the Linux directory `AConfig/gen`
     For Microsoft Windows, the header file is located in the directory `AConfig/gen`.

   Note

   The SystemC examples that ship with the Fast Models Portfolio directly produce a SystemC executable by specifying Post Build Actions for the linker in the Project Settings dialog. This permits you to build and run these examples directly from System Canvas.

This is accomplished by customized Makefiles/vcprojs that compile the SystemC files of the user and link against the EVS. They are triggered in the Post-Build Actions of the `sgproj`.

5.2.3 Adding header files and libraries for Linux export

This section describes additional header files and libraries that are required for the SystemC export on Linux. This is split into headers and libraries that are required at build time and those that are required for packaging.

Header Files and libraries to build the EVS on Linux

The exported component requires specific header files and libraries, that is, AMBA-PV header files from `$MAXCORE_HOME/AMBA-PV/include` if AMBA-PV ports are used.

Note

Prior to Version 6.1 of Fast Models, certain runtime library and header files were moved out of the tools and into the portfolio.

The `include` path, usually `-I` in `CPPFLAGS` in `Makefiles`, is located in:

- `$PVLIB_HOME/include/fmruntime`

Runtime libraries that are linked to any final executable or shared object are located in:

- `$PVLIB_HOME/lib/Linux_GCC-X.Y/libfmruntime.a` or `-lfmruntime`

Note

`$(MAXCORE_HOME)/lib/Linux_GCC-4.1/libsc_sg_export.a` or `-lsc_sg_export` has to be replaced with `$(PVLIB_HOME)/lib/Linux_GCC-4.1/libfmruntime.a` or `-lfmruntime`.

References to `$MAXCORE_HOME/lib/Linux_GCC-4.1/*.a*` have to be removed from `Makefiles`.

Libraries to package the EVS on Linux

For more information on how to package the EVS, see *Packaging the model for use on Linux* on page A-3.

Note

The `SYSTEMC_HOME` and `TLM_HOME` environment variables are set when the Third Party IP package (TPIP) is installed. This package is available as a separate download from the same location as the Fast Models package. This package also contains the SystemC and TLM header files and libraries.

Depending on the example, additional libraries might be necessary (for instance, the Linux example requires the SDL library). Such libraries are searched in the same directory where the EVS is built.

5.2.4 Adding header files and libraries for Microsoft Windows export

This section describes additional header files and libraries that are required for the SystemC export on Microsoft Windows. This is split into headers and libraries that are required at build time and those that are required for packaging.

Headers and Libraries to build the EVS on Microsoft Windows

The exported component requires specific header files and libraries:

- the header file `$PVLIB_HOME/include/fmruntime/sc_sg_wrapper_base.h`
- all header files mentioned in all include sections of the protocols of the ports of the top level component, for example `amba_pv.h`
- SystemC header files from `$SYSTEMC_HOME/include`
- OSCI TLM header files from `$TLM_HOME/include/tlm`
- AMBA-PV header files, if AMBAPV ports are used, from `$MAXCORE_HOME/AMBA-PV/include`

Note

Prior to Version 6.2 of Fast Models, certain runtime library and header files were moved out of the tools and into the portfolio.

To make use of the new file locations for existing projects, you must:

- change `include` path (usually `AdditionalIncludeDirectories` in `*.vcproj`):
  - `$(MAXCORE_HOME)/include` -> `$(PVLIB_HOME)/include/fmruntime`
- change runtime libraries that are linked to any final executable or shared object (usually `AdditionalDependencies` in `*.vcproj`):
  - `libsc_sg_export.lib` -> `fmruntime.lib`
- remove the `${MAXCORE_HOME}/lib/Debug_2005` path from `AdditionalLibraryDirectories`
- remove the `${MAXCORE_HOME}/lib/Release_2005` path from `AdditionalLibraryDirectories`
- remove the `${MAXCORE_HOME}/lib/Debug_2008` path from `AdditionalLibraryDirectories`
- remove the `${MAXCORE_HOME}/lib/Release_2008` path from `AdditionalLibraryDirectories`.

Libraries to package the EVS on Microsoft Windows

See *Packaging the model for use on Microsoft Windows* on page A-3 for more details on how to package the EVS on Microsoft Windows.

Note

The `/vmg` option must be used in the project settings to correctly compile source code for use with SystemC.

The `SYSTEMC_HOME` and `TLM_HOME` environment variables are set when the *Third Party IP package* (TPIP) is installed. This package is available as a separate download from the same location as the Fast Models package. This package also contains the SystemC and TLM header files and libraries.

Depending on the example, additional libraries might be necessary. For example, the Linux example requires the SDL library. Such libraries are searched in the directory where the EVS is built.

5.3 Adding the generated SystemC component to a SystemC system

The following steps are required to add the generated System C component to your SystemC project:

- Link the generated library to your existing SystemC system using the appropriate additional libraries:
  - EVS Export Library `$PVLIB_HOME/lib/Win32_VC2008/Release/fmruntime.lib` for release builds using Microsoft VC2008
  - EVS Export Library `$PVLIB_HOME/lib/Win32_VC2008/Debug/fmruntime.lib` for debug builds using Microsoft VC2008
  - EVS Export Library `$PVLIB_HOME/lib/Win64_VC2008/Release/fmruntime.lib` for release builds using Microsoft VC2008
  - EVS Export Library `$PVLIB_HOME/lib/Win64_VC2008/Debug/fmruntime.lib` for debug builds using Microsoft VC2008
  - EVS Export Library `$PVLIB_HOME/lib/Win32_VC2010/Release/fmruntime.lib` for release builds using Microsoft VC2010
  - EVS Export Library `$PVLIB_HOME/lib/Win32_VC2010/Debug/fmruntime.lib` for debug builds using Microsoft VC2010
  - EVS Export Library `$PVLIB_HOME/lib/Win64_VC2010/Release/fmruntime.lib` for release builds using Microsoft VC2010
  - EVS Export Library `$PVLIB_HOME/lib/Win64_VC2010/Debug/fmruntime.lib` for debug builds using Microsoft VC2010
  - SystemC library from `$SYSTEMC_HOME/lib/Win32_VC2008/Release/SystemC.lib` for release builds using Microsoft VC2008
  - SystemC library from `$SYSTEMC_HOME/lib/Win32_VC2008/Debug/SystemC.lib` for debug builds using Microsoft VC2008
  - SystemC library from `$SYSTEMC_HOME/lib/Win64_VC2008/Release/SystemC.lib` for debug builds using Microsoft VC2008
  - SystemC library from `$SYSTEMC_HOME/lib/Win64_VC2008/Debug/SystemC.lib` for debug builds using Microsoft VC2008
  - SystemC library from `$SYSTEMC_HOME/lib/Win32_VC2010/Release/SystemC.lib` for release builds using Microsoft VC2010
  - SystemC library from `$SYSTEMC_HOME/lib/Win32_VC2010/Debug/SystemC.lib` for debug builds using Microsoft VC2010
  - SystemC library from `$SYSTEMC_HOME/lib/Win64_VC2010/Release/SystemC.lib` for debug builds using Microsoft VC2010
  - SystemC library from `$SYSTEMC_HOME/lib/Win64_VC2010/Debug/SystemC.lib` for debug builds using Microsoft VC2010.
- Include the generated header file of the generated SystemC component into `main.cpp`:

  ```
  #include "sc_sg_wrapper_MySystem.h"
  ```
- Instantiate the generated SystemC component:

  ```
  sc_sg_wrapper_MySystem mySystem("mySystem");
  ```
- If your generated system contains components that can load application files, such as processors, load the application files into them:

  ```
  mySystem.load_application_file("cpu0", "myApplication.elf");
  ```

  where *cpu0* is the instance name of a processor in the Fast Models system.

Note

The Cortex-A9MP and Cortex-A5MP require that the processor is specified by using a period (.) before the name. To load an application to cpu0, use a specification string like `Core.cpu0` where `Core` is the instance name. The Dhrystone example in the SystemC export directory uses this convention.

- If required, set parameters for the configurable components of the Fast Models system. For example, to set parameter `PAR` of component instance *cpu0* to value `true`:

  ```
  mySystem.set_parameter("cpu0.PAR", true);
  ```

  Note

  The Cortex-A9MP and Cortex-A5MP requires that the processor is specified if the parameter is set for a specific processor. Use a specification string like `Core.cpu0.PAR` where `Core` is the instance name.

- Set up the simulation frequency of the Fast Models system, in Hz:

  ```
  mySystem.set_frequency(1000000000);
  ```

  The parameter value must be the same frequency as the master clock connected to the system. The master clock is always 1Hz, and it is recommended that you set this to the same frequency as the fastest clock in the EVS simulation.

- Set up the execution quantum of the Fast Models system:

  ```
  mySystem.set_global_quantum(0.001);
  ```

  For this example, the Fast Models system runs for 100 ticks before SystemC `wait()` is called.

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

The following libraries are required:

- For use on Linux:
  - `libMAXCOREInitSimulationEngine.so.2` is required.
  - `libarmctmodel.so` is required.
  - `libSDL-1.2.so.0.11.2` is required if your model uses the PL041 AACI component or uses any visualization components.

  The search order is:

  1. Alongside the EVS `.so`
  2. Alongside the executable that loaded the EVS `.so`

- For use on Microsoft Windows:
  - `libMAXCOREInitSimulationEngine.2.dll` is required.
  - `SDL.dll` is required if your model uses the PL041 AACI component or uses any visualization components.
  - `armctmodel.dll` is required.

  The search order is:

  1. Alongside the current module, for example EVS `.dll` or isim system, depending on how the simulator was built.

2. In the PATH, which includes attempting the directory that contains the executable as per the Microsoft Windows default search behavior.

5.4 Using the generated ports

The generated SystemC component must have SystemC ports to enable communication with the SystemC world. These ports are generated automatically from the Fast Models ports of the top-level component.

The SystemC export feature automatically generates port wrappers that bind the SystemC domain to the Fast Models virtual platform as shown in Figure 5-2.

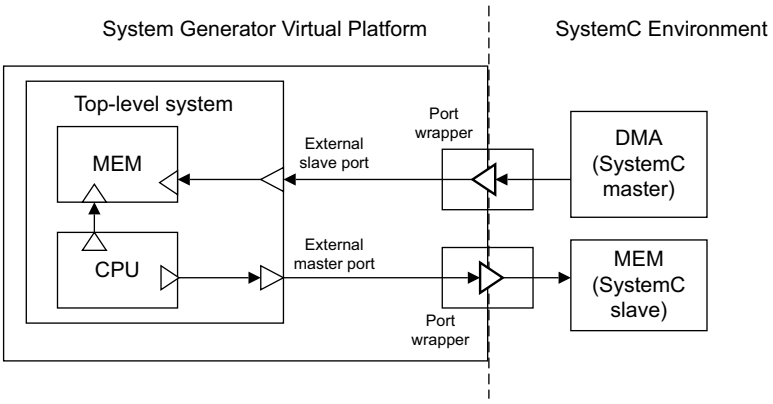

Figure 5-2 Port wrappers connect Fast Models and SystemC components

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).

For a list of supported protocols, their associated definitions, and bridges to the AMBA-PV protocol, see the *Fast Model Portfolio Peripheral Components Reference Manual*.

5.4.1 Protocol definition

This section describes requirements for the protocols that are used by the ports of the top level Fast Models component. It does not apply to other protocols that are not used to create SystemC ports. This section describes protocols that are based both on TLM 1.0 (for example signal protocols) and TLM 2.0 (for example bus 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 and any inconsistencies are found by the C++ compiler 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.

Example 5-1 Example TLM 1.0 protocol for exported SystemC component

```
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);
}
```

Example 5-2 Example TLM 2.0 bus protocol for exported SystemC component

```
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);
}
```

Example 5-2 defines a protocol that enables declaring ports that have a `read()` and a `write()` function. Master and slave ports can be declared using this protocol.

5.4.2 Properties for TLM1.0 based protocols

TLM1.0 based protocols are mapped to their SystemC counterparts using three properties in the LISA protocol definition. These properties must be set in the protocol description:

`sc_master_port_class_name`

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

`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 is specialized in the generated SystemC component for slave ports on the SystemC side. This class must declare the functions defined in the corresponding protocol, for example:

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

It must be derived in the SystemC component 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 &`

`sc_slave_export_class_name`

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

In Example 5-3:

- `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 is publicly derived.

Example 5-3 AMBAPV Signal protocol in Fast Models

```
protocol AMBAPVSignal {

  includes {
    #include <amba_pv.h>
  }

  properties {
    version = "1.0";
    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>";
  }
 . . .
```

The SystemC module ports must use the corresponding names in the SystemC code as shown in Example 5-4.

Example 5-4 AMBAPV Signal protocol in SystemC component class

```
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 AMBA PVBus component and the SystemC port shown in Example 5-4.

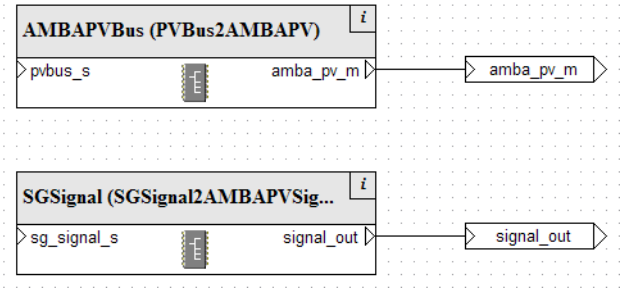

Figure 5-3 SGSignal component in System Canvas

5.4.3 Properties for TLM 2.0 based protocols

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

`sc_master_socket_class_name`

This is the class name of the SystemC class that is instantiated in the generated SystemC component for master sockets on the SystemC side. It is bound 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 is specialized in the generated SystemC component 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)
```

It must be derived in the SystemC component 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 is instantiated in the generated SystemC component for slave sockets on the SystemC side. It is bound 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 is specialized in the generated SystemC component for slave sockets on the SystemC side. It must also provide a constructor taking the argument:

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

Example 5-5 AMBAPV protocol in System Generator

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

  properties {
    version = "1.0";
    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>";
  }
```

The SystemC module sockets must use the corresponding names in the SystemC code as shown in Example 5-6.

Example 5-6 AMBAPV protocol in SystemC component class

```
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;
    ...
}
```

Note

Although the *Direct Memory Interface* (DMI) supports the ability to define `read_latency` and `write_latency` parameters for DMI accesses, Fast Models ignores any latencies in the DMI descriptor that are set by the target. `read_latency` and `write_latency` are estimates of the timing parameters for the corresponding memory accesses. They can either be used or ignored by the initiator, depending on the degree of timing accuracy that you want to model.

5.5 Example systems

This section describes the procedure to build and run the SystemC Export examples. These examples are installed with the Fast Model Portfolio and placed in `$PVLIB_HOME/examples/SystemCExport`.

Note

Before building the examples, you must set up Fast Model Portfolio in System Canvas.

You must also set up `systemc_home` and `tlm_home`.

The examples use the AMBA-PV protocol and components provided with the Fast Model Portfolio. The SystemC TLM AMBA-PV API is provided with the Fast Models package, in `$MAXCORE_HOME/AMBA-PV`.

5.5.1 Building Example systems

To build the SystemC executable select **Settings → Targets → SystemC Component** and build the project by pressing the **Build** button in System Canvas. For the SystemC examples that ship with Fast Models this generates an executable that consists of the project name followed by `.x` (Linux workstation) or `.exe` (Microsoft Windows workstation). Examples are `Dhrystone.x`, `Dhrystone2.x`, or `DMA.x`.

Note

The executable can also be built from the command line. Navigate to the `./Build` directory and type:

```
$MAXCORE_HOME/bin/simgen -p Dhrystone.sgproj -b --configuration <CONFIGURATION>
```

5.5.2 Running the Example system

You can start the executable from System Canvas by pressing the **Run** button and selecting the **SystemC executable** option button.

Note

The executable can also be started from the command line by executing the `run.sh` (Linux workstation) or `run.bat` (Microsoft Windows workstation) scripts. This automatically loads the correct application onto the EVS and start the SystemC simulation. Alternatively the generated executable can be started directly, followed by the application which is loaded onto the platform.

Dhrystone.x $PVLIB_HOME/images/dhrystone.axf

5.5.3 Running the examples with debug support

To start the examples with support for debugging, use the **Run** button drop-down list that appears when pressing the arrow on the right side to it. Select **Run...** and add the `-d` parameter to the **full command line** window.

Note

The executable can also be started from the command line of a shell. Use the normal command to start the executable, but add the `-d` option. For example, enter the following text to start the Dhrystone application from a shell with a debug server:

```
./run.sh -d
```

To debug the application, attach Model Debugger to the simulation. All Model Debugger features, for example source-level debugging, step, step over, and breakpoints, can be used to debug the application.

Figure 5-4 shows how the CADI debug server is linked into the generated SystemC component and enables debug access to the wrapped Fast Models virtual platform. The CADI interface enables attaching Model Debugger and performing interactive debug sessions. Debugging the wrapped native SystemC components is possible by using native debug methods of the platform in use, such as GDB on Linux or Microsoft Visual Studio on Microsoft Windows platforms.

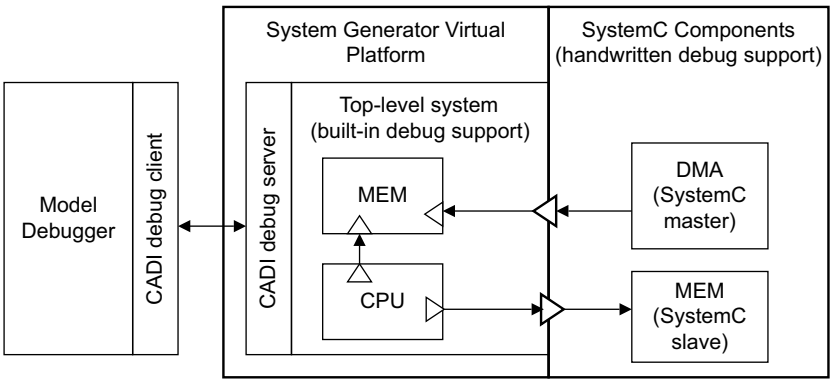

Figure 5-4 CADI client and server connects Model Debugger and system

Model Debugger is installed with the Fast Models installation. Start Model Debugger and select **Connect to Model** from the **File** menu to list all running simulations. Select the processor component and connect by double-clicking on the system to debug. See the *Model Debugger for Fast Models User Guide*, `http://infocenter.arm.com/help/topic/com.arm.doc.dui0314-/index.html` for more information on debugging targets.

5.5.4 Dhrystone example

This example, shown in Figure 5-7 on page 5-22, runs the Dhrystone benchmark application on a Fast Models system that consists of the following components:

- `Cortex-A9MP` processor
- `PVBusDecoder`
- `RAMDevice`
- `PVBus2AMBAPV` Bridge to memory on the SystemC side.

Note

Some of the memory in this example application is on the SystemC side.

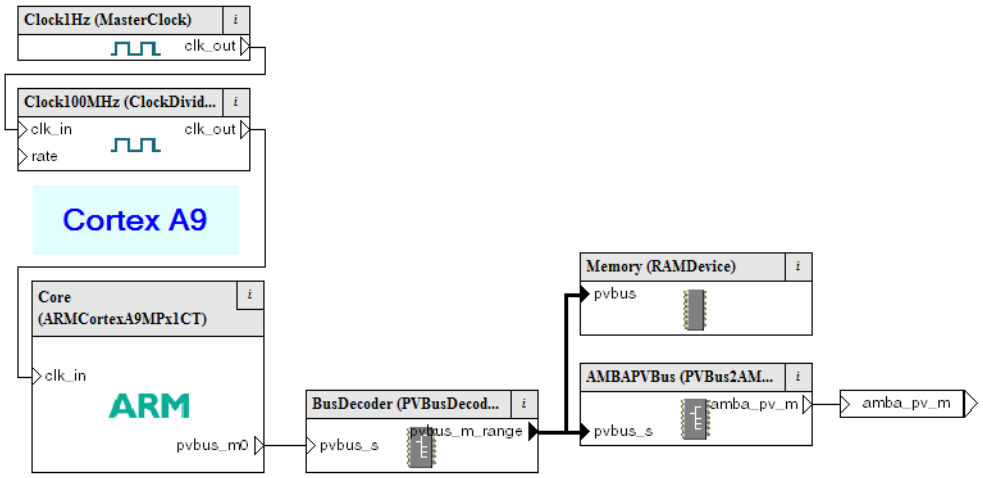

Figure 5-5 Dhrystone system example

The project file is `$PVLIB_HOME/examples/SystemCExport/Dhrystone/Build/Dhrystone.sgproj`.

Building the Dhrystone example

See *Building Example systems* on page 5-18 for how to build the example. Use `Dhrystone.sgproj` as the name for the project file.

Running the Dhrystone example

See *Running the Example system* on page 5-18 or *Running the examples with debug support* on page 5-18 for more instructions on how to run the example system.

Note

The dhrystone image which must be loaded onto the simulation executable is part of the TPIP package. The location is `<FastModelPortfolio>/images/dhrystone.axf` and is automatically used if the example is started from SGCanvas or using the provided scripts in the `Build` directory.

5.5.5 Dhrystone2 example

This example, shown in Figure 5-6, runs the Dhrystone benchmark application on a Fast Models system that consists of the following components:

- `ARM1176` processor
- `PVBusDecoder`
- `RAMDevice`.

Note

In this example, all of the application memory is on the Fast Models side.

The project file is `$PVLIB_HOME/examples/SystemCExport/Dhrystone/Build/Dhrystone2.sgproj`.

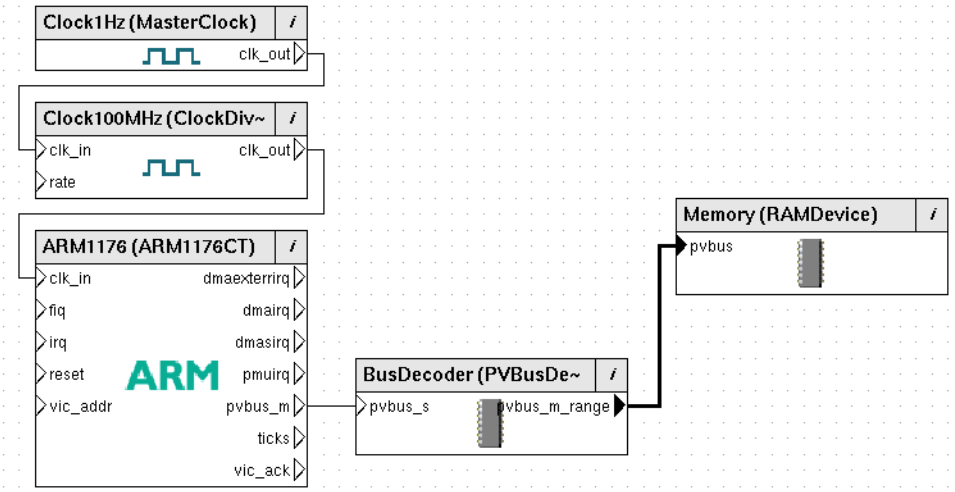

Figure 5-6 Dhrystone2 system example

Building the Dhrystone2 example

See *Building Example systems* on page 5-18 for how to build the example. Use `Dhrystone2.sgproj` as the name for the project file.

Running the Dhrystone2 example

See *Running the Example system* on page 5-18 or *Running the examples with debug support* on page 5-18 for more instructions on how to run the example system. The location of the

Note

The dhrystone image which must be loaded onto the simulation executable is part of the TPIP package. The location is `<FastModelPortfolio>/images/dhrystone.axf` and is automatically used if the example is started from SGCanvas or using the provided scripts in the `Build` directory.

5.5.6 DMA example

This example, shown in Figure 5-7 on page 5-22, is a modified version of the PV example from the OSCI TLM 2.0 package.

The Cortex-R4 programs a transfer and waits for an interrupt signaled by the PV DMA from the SystemC side to indicate the transfer is complete.

The Fast Models system contains the following components:

- `Cortex-R4` processor
- `PVBusDecoder`
- `RAMDevice` for the application memory
- `PVBus2AMBAPV` bridge to the SystemC side
- `AMBAPVSignal2SGSignal` bridge to forward end of transfer interrupts from SystemC into Fast Models
- `SGSignal2AMBAPVSignal` bridge to forward the end of transfer interrupts from Fast Models back into SystemC.
- `AMBAPV2PVBus` bridge from the SystemC side
- `DMA` transfer source memory.

The SystemC side contains:

- Simple PV DMA model
- Associated source and destination memories
- Pseudo master to receive end of transfer interrupt from Fast Models.

The project file is `$PVLIB_HOME/examples/SystemCExport/DMA/Build/DMA.sgproj`.

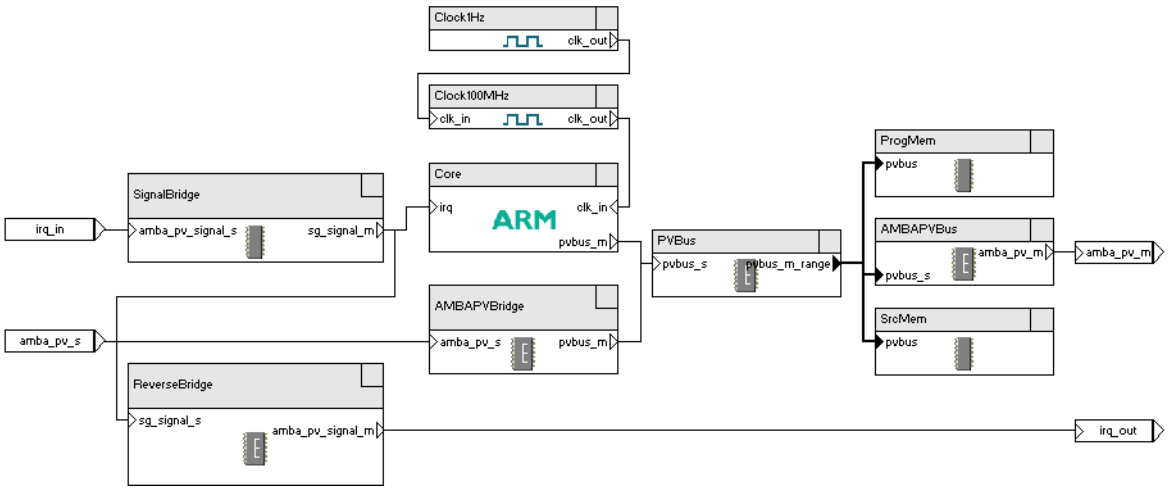

Figure 5-7 DMA system example

Building the DMA example

See *Building Example systems* on page 5-18 for how to build the example. Use `DMA.sgproj` as the name for the project file.

Running the DMA example

See *Running the Example system* on page 5-18 or *Running the examples with debug support* on page 5-18 for more instructions on how to run the example system.

5.5.7 EVS_Callbacks_Cortex-R4 example

The EVS_Callbacks_Cortex-R4 example, shown in Figure 5-9 on page 5-25, demonstrates the use of the callbacks that are described in *set_callbacks()* on page 5-32. It is based on the DMA example (See *DMA example* on page 5-21 for a description of the system). This example has two different execution modes, depending on whether the debug server is enabled or not.

Debug Server enabled

The size of the quantum is set to 10 simulation ticks and triggers the following callbacks:

- start_of_quantum
- proceed_quantum()
- simulation_stop()
- simulation_quit()

The simulation_quit() callback is triggered if the visualization unit is closed by pressing the close symbol (typically 'X' in the upper right corner, depending on your window manager). To properly reproduce the behavior it is recommended to attach a debugger to the debug server of the EVS and '`instruction step`' through the code. See `README.txt` in the top level directory of the example for more detailed instructions and a description of the expected output.

Note

The -d option must be passed to the executable to start the debug server.

No Debug Server enabled

The following callbacks are triggered if the example is started without the debug server being enabled (default case)

- start_of_quantum()
- wait()
- simulation_stop()

See `README.txt` in the top level directory of the example for more detailed instructions and a description of the expected output.

Building the EVS_Callbacks_Cortex-R4 example

See *Building Example systems* on page 5-18 for how to build the example. Use `EVS_Callbacks_Cortex-R4.sgproj` as the name for the project file.

Running the EVS_Callbacks_Cortex-R4 example

See *Running the Example system* on page 5-18 or *Running the examples with debug support* on page 5-18 for more instructions on how to run the example system.

5.5.8 Linux example

The Linux example shown in Figure 5-8 on page 5-24 is based on the Cortex-A8 EB platform. This variant of the system provides sufficient resource to run the operating system and a simple example application. The SystemC side contains a simple `CPTimers` model. The `Cortex-A8` component runs a Linux image.

The project file is `$PVLIB_HOME/examples/SystemCExport/LinuxSystem/Build/LinuxSystem.sgproj`

Figure 5-8 on page 5-24 shows the LinuxSystem with the bridges between Fast Models and SystemC. The `rate_hz_X` 64 integer value bridges permit you to configure the SystemC based timer module for dynamic clock configuration. The `int_X` integer bridge transmits the interrupt source signals for the processor. The `amba_pv_m` bridge connects to the peripheral memory-mapped registers of the timer component.

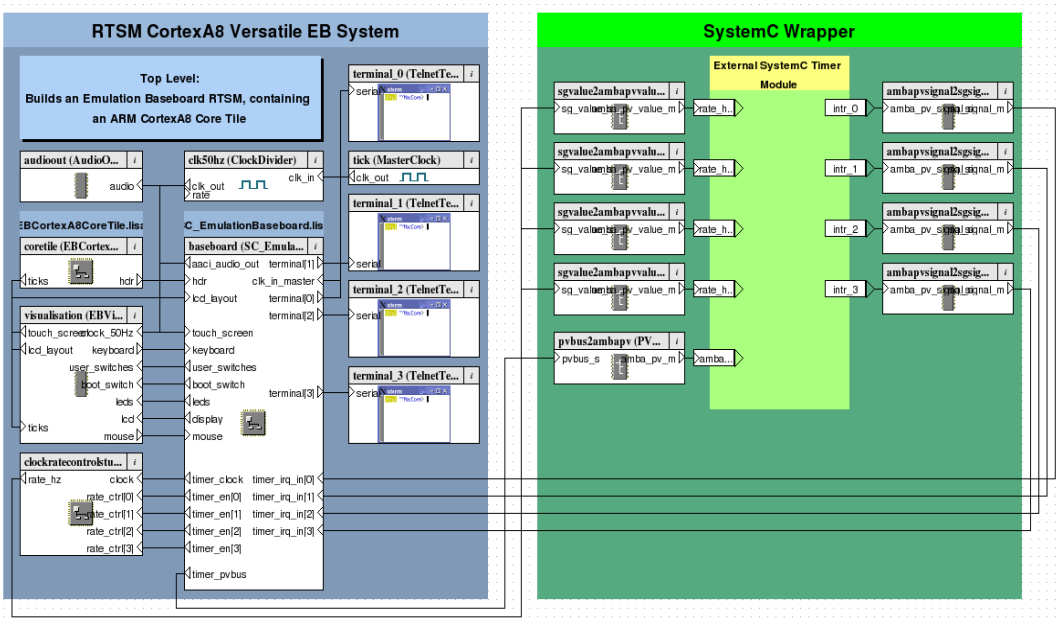

Figure 5-8 Linux example

Building the LinuxSystem example

See *Building Example systems* on page 5-18 for how to build the example. Use `LinuxSystem.sgproj` as the name for the project file.

Running the LinuxSystem example

See *Running the Example system* on page 5-18 or *Running the examples with debug support* on page 5-18 for more instructions on how to run the example system.

Note

The Linux image that must be loaded onto the simulation executable is part of the TPIP package. The locations for these images are:

- `<FastModelsPortfolio_X.Y>/images/FVP_EB_Linux\FVP_EB_CLI_EB_V6/*.axf`
- `<FastModelsPortfolio_X.Y>/images/FVP_EB_Linux\FVP_EB_CLI_EB_V7/*.axf`
- `<FastModelsPortfolio_X.Y>/images/FVP_EB_Linux\FVP_EB_CLI_EB_V7_SMP/*.axf`

You must load the image first, then load the filesystem for EB into the MMC using the `p_mmc_file` parameter. The folders of the file systems are located under:

- `<FastModelsPortfolio_X.Y>/images/FVP_EB_Linux\*`

The image is automatically used if the example is started from SGCanvas or if using the provided scripts in the `Build` directory.

5.5.9 Protocols example

The Protocols example, shown in Figure 5-9 on page 5-25, uses the AMBA-PV Signal and SignalState protocols and associated components provided with the Fast Model Portfolio. The example tests these two protocols. The AMBA-PV API on the SystemC side is provided by Fast Models.

The Fast Models system contains the following components:

- ARM1176 processor
- PVBusDecoder
- RAMDevice for the application memory
- bridges to the SystemC side
- bridges from the SystemC side.

The SystemC side contains a simple component model with associated master ports and slave exports.

Note

The Protocols example does not include support for debugging. The only purpose for this example is illustrating the use of additional protocols and corresponding bridges.

The project file is $PVLIB_HOME/examples/SystemCExport/Protocols/Build/Protocols.sgproj.

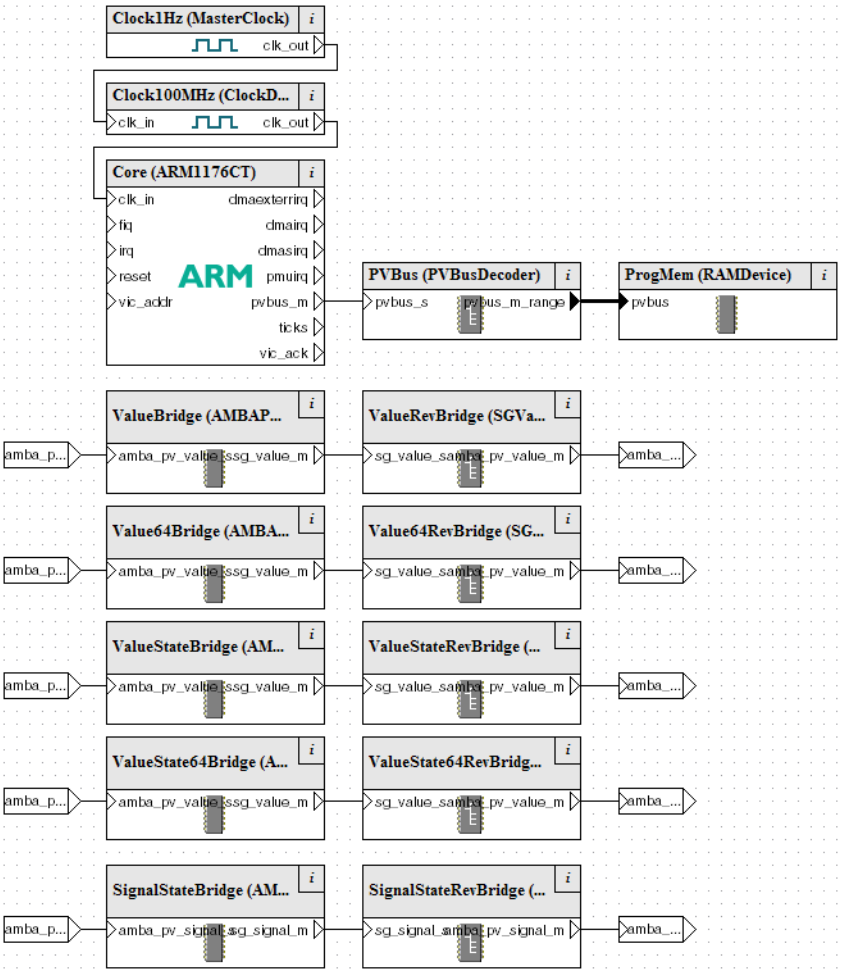

Figure 5-9 Protocols example

Building the Protocols example

To build the example, see *Building Example systems* on page 5-18. Use Protocols.sgproj as the name for the project file.

Running the Protocols example

For more instructions on how to run the example system, see *Running the Example system* on page 5-18.

5.5.10 EVS_Delay_Cortex-A8 Example

The Delay example describes how to delay the simulation startup of an EVS and setting an external reset signal from SystemC. There are two options to force the EVS to stop simulation until a certain condition is met. These are described in this section.

Overloading the wait() callback

This solution implements a derived `sc_sg_callback` object (see *sc_sg_callbacks class* on page 5-32 and *EVS_Callbacks_Cortex-R4 example* on page 5-22), which waits for the reset signal going low in its overloaded wait callback during the init phase. This is achieved by introducing an event which is triggered by the reset signal going low. The first quantum of the EVS starts immediately after the `wait()` callback returns from waiting for the event. For example, the reset signal was set to 0.

Example 5-7 Waiting for reset in 'wait()' callback

```
class callback : public sc::sc_sg_callbacks
{
public:
    // overloaded callbacks from the EVS (Exported Virtual Subsytem)
    void wait(double time, int mode);
}


void callback::wait(double time, int mode)
{
    if (mode == SG_MODE_INIT)
    {
        sc_core::wait(reset_event);
    }
    sc_core::wait(sc_core::sc_time(time, sc_core::SC_SEC)); /* do normal wait */
}
```

The SystemC signal `reset_event` is triggered in the example after 10 ns. At timestamp 11ns the signal is triggered and the `wait()` callback returns from its call permitting the EVS to start the simulation.

When the simulation starts, the `wait()` callback is called. This callback waits for an event to be set by some other component. The EVS simulation does not advance until the callback returns. Somewhere within the quantum it sets the `reset_event` which causes the `wait()` callback to return and unlock the EVS. The next time the SystemC scheduler activates the EVS it starts simulating its quantum.

Note

The advantage of this solution is that it effectively prevents the entire EVS from starting and there is no delay when the callback returns. The disadvantage is that for multiprocessor systems all processors are halted. If only a single processor is to be stopped then this solution is not applicable. In this case the reset port must be used.

Using the Reset port

The other solution is to connect a reset signal to the respective port within the EVS. Figure 5-10 shows the example system using an ARM Cortex-A8 where the reset port is connected by an `ambapvsignal2sgsignal` component. This example is based on the Dhrystone example with the difference that the reset port is routed by this bridge to another SystemC component.

Note

To drive the `reset` signal to '1' from the very beginning of the simulation, it is necessary to set this signal in the simulation phase `end_of_elaboration()`. If that is not done, it depends on the scheduler which component is activated first. This can result in a delay of one quantum before the EVS recognizes the reset signal being active.

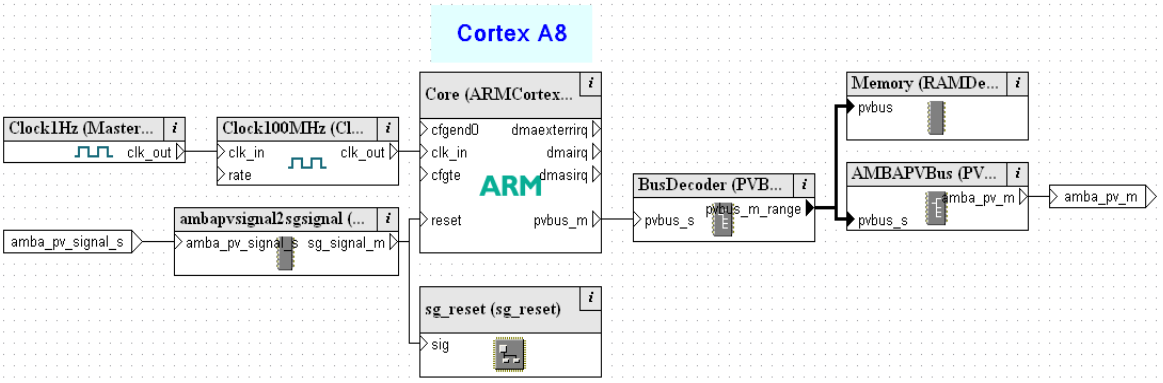

Figure 5-10 Reset Example system

To activate this behavior in the example `DELAY_WAIT` in the `main.cpp` file is required to be undefined. The drawback of this solution is that setting the reset signal is only recognized by the EVS at the end of its quantum. This is because of the nature of temporal decoupling. In contrast to the solution where the callback is overloaded this solution does not make use of a simulation API and therefore suffers from the effects of temporal decoupling.

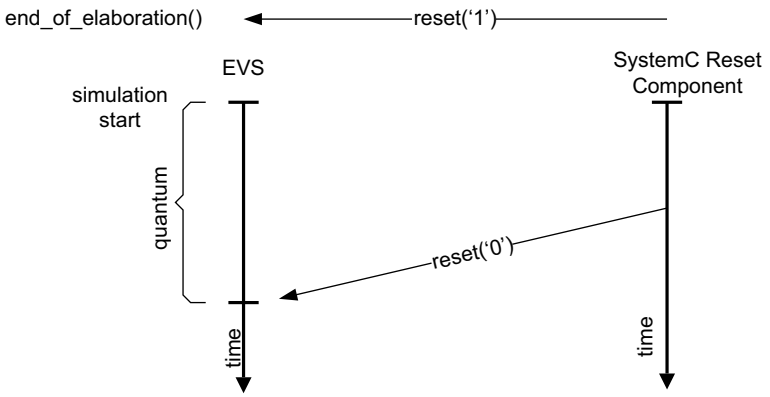

Figure 5-11 Timing diagram for driving the reset signal

Figure 5-11 on page 5-27 shows the timing diagram for the reset example using the reset signal. Before the simulation starts (`end_of_elaboration()`) the reset signal is set. Within the quantum of the SystemC Reset component the reset signal is set back to ‘0’. The EVS recognizes this change at the end of its quantum. This difference to the solution which uses the `wait()` callback has to be taken into account when choosing one or the other solution.

5.6 SystemC component API

This section describes the API functions for a SystemC component built by performing a SystemC export from a Fast Models system.

5.6.1 Component access function

There is a single component access function.

get_instance()

Use this function to access the singleton exported SystemC component. The format is:

```
static sc_sg_wrapper_base * get_instance()
```

5.6.2 Parameter access functions

Refer to component parameters by forming a dot (.) separated string that combines:

- the Fast Models hierarchical component instance name, but without the top level component of the Fast Models system
- the name of the parameter to be accessed.

For example:

```
MySubComponentA.MyParam
```

set_parameter()

Use this function to get parameters from components present in the Fast Models system. The formats are:

```
bool set_parameter(const std::string & name, const std::string & value)
bool set_parameter(const std::string & name, const char * value)
bool set_parameter(const std::string & name, bool value)
bool set_parameter(const std::string & name, int value)
bool set_parameter(const std::string & name, unsigned int value)
bool set_parameter(const std::string & name, long long value)
bool set_parameter(const std::string & name, unsigned long long value)
```

where:

`name` specifies the name of the parameter to be changed.

`value` specifies the value of the parameter to be changed.

The function returns `false` if the parameter is unknown, otherwise it returns `true`.

Note

MTI and trace parameters always return `true` even if they are invalid.

Note

Changes made to parameters within System Canvas take precedence over changes made with `set_parameter()`.

get_parameter_list()

Use this function to get a list of all parameters from components present in the Fast Models system. The format is:

```
std::map<std::string, std::string> get_parameter_list() const
```

5.6.3 Simulation control functions

The time to pass to SystemC `wait()` after the Fast Models system has been simulated for a quantum is determined by:

- the parameter value specified in the `set_global_quantum()` method
- the Fast Models simulation frequency specified by the parameter value used in the `set_frequency()` method. See Figure 5-12.

$$\text{yield (seconds)} \approx \frac{\text{round(f*t)}}{\text{f}}$$

Figure 5-12 Yield equation

where:

`f` is the frequency (Hertz)

`t` is the time set in `set_quantum_seconds` or `set_global_quantum` (seconds).

break_quantum()

```
void break_quantum(unsigned long delta = 0)
```

Note

This function is deprecated. Use `break_quantum_seconds()` instead.

break_quantum_seconds()

Use this function to instruct the Fast Models simulation to break its current simulation quantum. The format is:

```
void break_quantum_seconds(double intr_time = 0)
```

where `intr_time` specifies the number of seconds from now to when the simulation quantum is to be broken.

If `break_quantum_seconds()` is called, the Fast Models system simulation stops `intr_time` seconds from the method call. The SystemC `wait()` callback is called, with an `sc_time` value corresponding to the fraction of the quantum simulated so far, to enable simulating other SystemC components. If `intr_time` is zero, the simulation quantum is broken immediately, that is, as soon as the callee returns control to the simulation.

`break_quantum_seconds()` can be called more than once per simulation quantum. The simulation quantum is broken as many times as `break_quantum_seconds()` is called. After the quantum has been broken, the next quantum is a new/full quantum.

However, distinct calls to `break_quantum_seconds()` are merged if they refer to the same simulation timing point. This is done even though the calls might have come from within distinct quantums.

It is the responsibility of the SystemC scheduler, or any other component of the SystemC environment, to call `break_quantum_seconds()` with the appropriate parameter.

To synchronize Fast Models and SystemC environments, ARM recommends using `break_quantum_seconds()` instead of SystemC `wait()`. See also *sc_sg_break_quantum()* on page 5-32.

get_callbacks()

This function returns the callback object registered within the Fast Models simulation. The format is:

```
sc_sg_callbacks * get_callbacks() const
```

get_cycle_count()

This function returns the total number of cycles simulated by the Fast Models system since the start of the simulation. The format is:

```
unsigned long long get_cycle_count()
```

get_freq()

Note

This function is deprecated. Use `get_frequency()` instead.

get_frequency()

Use this function to retrieve the simulation frequency of the exported Fast Models system as set by `set_frequency`. The format is:

```
unsigned long long get_frequency() const
```

get_global_quantum()

This function retrieves the global simulation quantum time in seconds as stored in the TLM 2.0 final class `tlm_global_quantum`. The format is:

```
double get_global_quantum() const;
```

Note

If this function is called without including TLM 2.0 final headers, it falls back to using `get_quantum_seconds()`.

The frequency must have been set with the `set_frequency()` API call before using the quantum API functions.

get_quantum_seconds()

This function retrieves the time in seconds per local simulation quantum of the exported Fast Models subsystem. The format is:

```
double get_quantum_seconds() const;
```

Note

The frequency must have been set with the `set_frequency()` API call before using the quantum API functions.

get_quantum()

This function retrieves the local quantum of the exported Fast Models subsystem in simulation clock ticks. The format is:

```
unsigned long get_quantum()
```

Note

This function is deprecated. Use `get_quantum_seconds()` instead. (See *get_quantum_seconds()* on page 5-31). The frequency must have been set with the `set_frequency()` API call before using the quantum API functions.

get_rate()

Note

This function has been deprecated. Use `get_clock_rate()` instead for equivalent results.

get_slot()

Note

This function is deprecated. Use `get_global_quantum()` instead. (See *get_global_quantum()* on page 5-31.)

sc_sg_break_quantum()

Use this function to pass a time value in seconds to the break_quantum_seconds function of the wrapper component. If there is no wrapper instance, SystemC `wait()` is called. The format is:

```
void sc_sg_break_quantum(double intr_time = 0)
```

where `intr_time` specifies the number of seconds from now to when the simulation quantum is to be broken.

Note

This is a convenience function for break_quantum_seconds(). It can be replaced by the following call in the SystemC component:

```
sc::sc_sg_wrapper_base::get_instance()->break_quantum_seconds(intr_time);
```

set_callbacks()

Use this function to register a callback object within the Fast Models simulation. The format is:

```
void set_callbacks(sc::sc_sg_callbacks * cb)
```

where `cb` specifies a pointer to an instance of an `sc_sg_callbacks` derived class that defines these virtual methods. See Example 5-8.

Example 5-8 sc_sg_callbacks class

```
namespace sc
{
```

```
    class sc_sg_callbacks
    {
    public:
        virtual ~sc_sg_callbacks(){};

// the callbacks
    public:
        virtual void start_of_quantum(int /* mode */){};
           /*Called by the EVS
            *at the start of each quantum.
            *This default implementation does nothing.
            *mode - one of:
            *SG_MODE_INIT
            *SG_MODE_STOP
            *SG_MODE_RUN
            *SG_MODE_BPT */

        virtual void wait(double time, int mode);
           /*Called by the EVS to
            *synchronize with SystemC.
            *This default implementation calls sc_wait() with the
            *appropriate sc_time argument.
            *t - time to wait in seconds
            *mode - one of:
            *SG_MODE_INIT
            *SG_MODE_STOP
            *SG_MODE_RUN
            *SG_MODE_BPT*/
        virtual void proceed_quantum(int /* mode */){};
           /* Called by the EVS
            *before executing the System Generator scheduler
            *This default implementation does nothing.
            *mode - one of:
            *SG_MODE_INIT
            *SG_MODE_STOP
            *SG_MODE_RUN
            *SG_MODE_BPT */
        virtual void simulation_quit(int /* mode */ );
           /* Called by the EVS
            *before simulation quits.
            *This default implementation calls sc_stop().
            *mode - one of:
            *SG_MODE_INIT
            *SG_MODE_STOP
            *SG_MODE_RUN
            *SG_MODE_BPT */
        virtual void simulation_stop(int /* mode */ ){};
           /* Called by the EVS
            *after simulation stops, e.g. after application exit.
            *This default implementation does nothing.
            *mode - one of:
            *SG_MODE_INIT
            *SG_MODE_STOP
            *SG_MODE_RUN
            *SG_MODE_BPT */
    };

};
```

The callback object defines a set of methods that the Fast Models simulation calls at a predefined simulation time point.

The Fast Models simulator has its own predefined callbacks, with default implementations. The callbacks object given to the method can override the default implementations.

set_freq()

Note

This function has been deprecated. Use `set_frequency()` instead.

set_frequency()

Use this function to specify the frequency in Hz at which the EVS is to be simulated with regard to the SystemC simulation time, using this equation:

```
SystemC simulation time in seconds = EVS simulation ticks / frequency
```

The `set_frequency()` function controls an internal timer that determines the granularity of synchronization between the EVS and SystemC. The EVS internal quantum is set as a multiple of this frequency, based on the TLM 2.0 global quantum. When the EVS terminates its internal quantum, the EVS reports via `wait()callback` that it has simulated for a certain amount of SystemC time, as in the equation.

It is recommended that you call `set_frequency()` with a value that matches the fastest processor in the EVS simulation. The frequency of the faster processor refers to the clock rate received by that processor using the 1Hz MasterClock component with any additional rate changes because of `clockdiv` components. For example, if the EV contains two processors, core0 connects directly to the MasterClock and core1 connects using a `clockdiv` that multiples the rate by 2. It is recommended that `set_frequency` is set to 2.

Note

Setting the frequency too low might result in inaccuracy in the simulation, and might even result in the EVS not permitting any time to advance in SystemC, especially when the expression `TLM 2.0 global quantum in seconds * frequency` becomes too low.

set_global_quantum()

Use this function to define the global simulation quantum time in seconds. The EVS system is simulated for the specified time before calling SystemC `wait()` to permit simulating other SystemC components. The format is:

```
void set_global_quantum(double interval);
```

This method can be called before `sc_start()` or at any time during simulation. If called after `sc_start()`, the number of seconds per simulation slot is taken into account at the next simulation slot. It is highly recommended to call this function or `set_quantum_seconds()` before `sc_start()` to set a reasonable high quantum to achieve high simulation performance.

The frequency must have been set with `set_frequency()` API call before using the quantum API functions. If a quantum of 100000 is set the call looks like

```
/* Simulation quantum, i.e. seconds to run per quantum */
   <instance>.set_global_quantum(100000.0 / <instance>.get_frequency());
```

where `<instance>` refers to the name of the EVS instance.

set_quantum_seconds()

Use this function to define the local simulation quantum time of the exported Fast Models subsystem in seconds. The Fast Models system is simulated for the specified time before calling SystemC `wait()` to permit simulating other SystemC components. The format is:

```
void set_quantum_seconds(double interval);
```

This method can be called before `sc_start()` or at any time during simulation. If called after `sc_start()`, the number of seconds per simulation slot is taken into account at the next simulation slot.

The frequency must have been set with the `set_frequency()` API call before using the quantum API functions. When one of the local quantum setting API functions has been called, the global tlm quantum is no longer queried during simulation to update the local simulation quantum of the exported Fast Models subsystem.

If a quantum of 100000 is set the call looks like

```
/* Simulation quantum, i.e. seconds to run per quantum */
   <instance>.set_quantum_seconds(100000.0 / <instance>.get_frequency());
```

where `<instance>` refers to the name of the EVS instance.

set_quantum()

Note

This function is deprecated. Use `set_quantum_seconds()` instead.

set_rate()

Note

This function is deprecated. Use `set_frequency()` instead for equivalent results. (See *get_frequency()* on page 5-31.)

set_slot()

Note

This function is deprecated. Use `set_global_quantum()` instead. (See *get_global_quantum()* on page 5-31.)

stop()

Note

This function is deprecated. Use the SystemC function `sc_stop()` instead.

5.6.4 Application loading functions

This section describes the function used for loading an application file.

load_application_file()

Use this function to specify the application files to be loaded by Fast Models system components. This method must be called before starting the simulation. The specified application files are loaded into the respective components. If this method is called before the system components are initialized, the calls are deferred until simulation starts. The format is:

```
void load_application_file(const std::string & instance_name,
                           const std::string & file)
```

where:

`instance_name`
: is the instance name of the component

> **Note**
>
> If the component has subcomponents, such as the Cortex-A9MP and Cortex-A5MP, the subcomponent must be specified. Use a period (.) to separate the components. For example, if `Core` is the instance name, specify the first processor as `Core.cpu0`.

`file`
: specifies the application file to be loaded by the component.

5.6.5 Debug control functions

This section describes the different debug modes of the EVS.

set_debug_mode()

Use this function to specify whether a CADI debug server is launched when the simulation starts. This debug server permits connections with debuggers like Model Debugger to debug the source code of the applications loaded and executed by the Fast Models system processor. The format is:

```
void set_debug_mode(bool debug = true, bool start_simulation = false, bool cadi_log =
false)
```

where:

`debug`
: Set to `true` to specify that the simulation starts in debug mode. If set to `false`, debuggers cannot connect to the simulation.

`start_simulation`
: This parameter determines if the simulation starts running after the debug server is launched. If `start_simulation` is set to `false`, the default if debug mode is enabled, the debug server blocks the simulation from running and waits for debugger execution control commands to run the simulation.

> **Note**
>
> If the system is set up in non-debug mode, the simulation starts running by default.
>
> Calling `set_debug_mode()` after `sc_start()` is ignored.

`cadi_log`
: This parameter permits logging of all CADI calls and responses into an XML file for each component instance.

set_debug()

Note

`set_debug(bool debug = true)` is deprecated. Use `set_debug_mode(bool debug = true, bool start_simulation = false, cadi_log = false)` instead.

5.6.6 Accessing the CADI interface directly

There are two ways to access the CADI interface of the EVS.

- Connect with the CADI Client Integration Kit (CCIK). This method uses TCP/IP as transport layer.
- Connect directly to the CADI Interface.

The direct connection to the CADI Interface of the EVS is provided through a global symbol of the exported platform:

```
extern "C"{ CADI_WEXP eslapi::CADIBroker * CreateCADIBroker(); }
```

This enables getting the CADIBroker. The CADIBroker enables querying all CADIInterfaces that are maintained by the simulation.

Includes

The following header files must be included to access the CADIBroker:

```
"#include "eslapi/CADIFactory.h"
"#include "eslapi/CADI.h"
"#include "sc_sg_wrapper_<platform_name>.h"
```

For the include `sc_sg_wrapper_<platform_name>`, the term `<platform_name>` must be changed to the name of the EVS. This include file is generated into a subdirectory of the actual build path.

Accessing CADIBroker

An instance of the CADIBroker can be initialized by:

```
eslapi::CAInterface *ca_interface = CreateCADIBroker();
```

See the CADI documentation for more information on using the CADIBroker.

5.7 Scheduling of Fast Models and SystemC

This section describes:

- The function of the scheduler generated by Fast Models.

  The scheduler handles debugger requests and balances simulation time (quantum) between the EVS and SystemC models.

- The callbacks that are executed in the different simulation steps.

  Callbacks are described in more detail in *get_callbacks()* on page 5-31 and *set_callbacks()* on page 5-32.

Note

The following conventions are used in this section:

- EVS refers to the instance of the Fast Models platform exported to a SystemC model (Exported Virtual System)
- For graphics, callbacks are annotated with the text *Callback*

5.7.1 Standalone simulation without a debug server

If the simulation is started without a debug server, the scheduler operates as shown in Figure 5-13 on page 5-39. This is the default startup mode.

The simulation steps are:

Initializing the simulation

The simulation is started and initialized:

1. The SystemC simulation is started by calling `sc_start()` from the simulation environment.
2. A `wait()` callback is issued. The time parameter is zero and the mode parameter is `SG_MODE_INIT`.

   The `wait()` callback is issued only once immediately after the start of simulation.

   Note

   If the use-case of the EVS requires to prevent the EVS from starting its quantum you might register for the `wait()` callback and check for `SG_MODE_INIT`. When the EVS is permitted to start the callback can be returned.

Running the simulation

After the initial callback, the simulation enters a loop to execute the following simulation steps:

1. If stopped, the simulation exits.
2. The `start_of_quantum()` callback is issued with parameter `SG_MODE_RUN` if the quantum has not been already been interrupted.
3. EVS is simulated for a time quantum equivalent to the number of seconds specified by `set_quantum_seconds()`.
4. A `wait()` callback is issued with the time parameter equivalent to the actual simulated time in the `EVS`.

This is typically used to issue a SystemC `wait()` to hand over control to the SystemC model to simulate the same amount of time and synchronize the two models. This is the default implementation. A custom simulation environment can reimplement this callback.

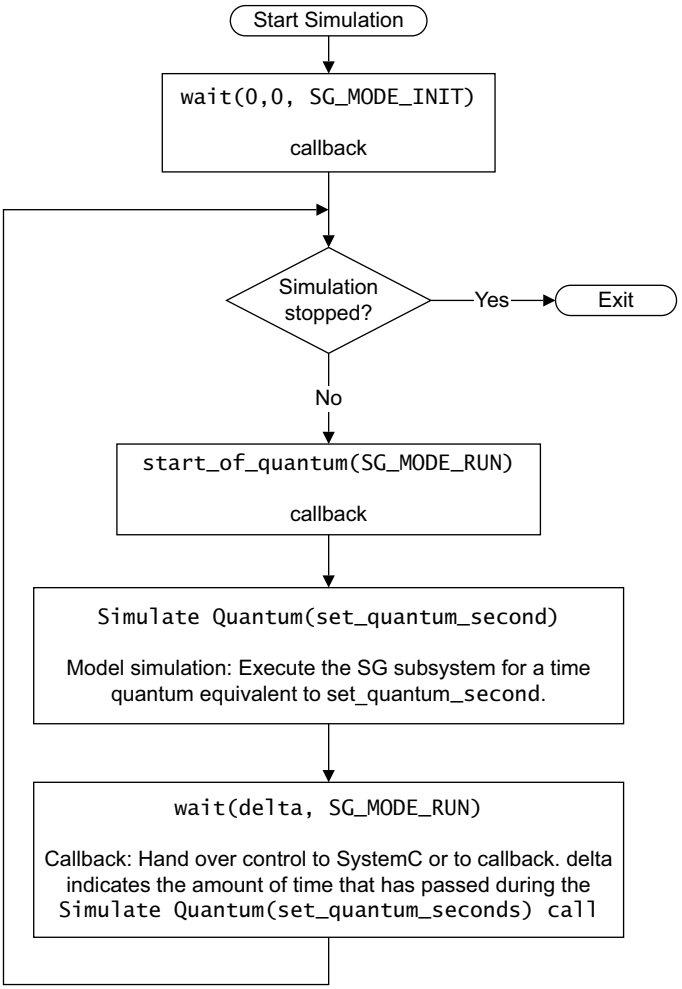

Figure 5-13 Simulation without a debug server

5.7.2 Simulation with a debug server

A debug server with a CADI 2.0 interface can be started with the exported model. The initialization is done in the SystemC simulation phase `before_end_of_elaboration()`. The debug server is therefore ready to accept incoming requests after sc_start() is called. The scheduler executed for this case is shown in Figure 5-14 on page 5-41.

After the System C simulation starts:

1. The `wait()` callback is initiated with mode parameter set to `SG_MODE_INIT`.
2. A check is made to determine whether a debugger command must be processed. Such commands are typically:
   - `Run` to change the simulation state from stop to running
   - `Stop` to change the state from running to stop.

3. The EVS simulation state is tested to determine if the EVS is still running:

EVS simulation running

The following steps are executed. These steps are equivalent to the case where no debug server is started:

1. If a previous quantum has executed, the `start_of_quantum()` callback is executed with parameter `SG_MODE_RUN`.

   If the previous quantum was interrupted by a breakpoint or stop command, the callback is not executed.

   A single step causes a breakpoint to be hit, so the callback is not typically executed after a single step.
2. The `proceed_quantum()` callback is executed before each execution of the Fast Models scheduler.

   This callback is especially useful for debugger integration that requires a notification before each start of simulation from the Fast Models subsystem.
3. EVS is simulated for a time quantum equivalent to the time specified by `set_quantum_seconds()`.
4. The `wait()` callback is issued with the time parameter equal to the actual simulated time in Fast Models. This is typically used to issue a SystemC `wait()` to cause the SystemC model to simulate the same amount of time and synchronize the two models. This is the default implementation. A custom simulation environment can reimplement this callback.

EVS simulation stopped

The following steps are executed:

1. A `wait()` callback is issued with the time parameter zero and the mode set to `SG_MODE_STOP`.
2. The simulation is then suspended for 20ms to permit other simulators to catch up.
3. The simulation resumes with the check for a pending debugger command that might have arrived.

Note

The following callback functions can be used to notify the end of EVS simulation:

`simulation_quit()`

Called before the simulation quits because of a user event such as pushing the close button of the visualization component in the `LinuxSystem` example.

The default implementation calls `sc_stop()`, but this can be overloaded by the SystemC user.

`simulation_stop()`

Called after the simulation stops because of, for example, application exit (`_sys_exit`).

The default implementation does nothing, but this can be overloaded by the SystemC user.

See also *Simulation control functions* on page 5-30.

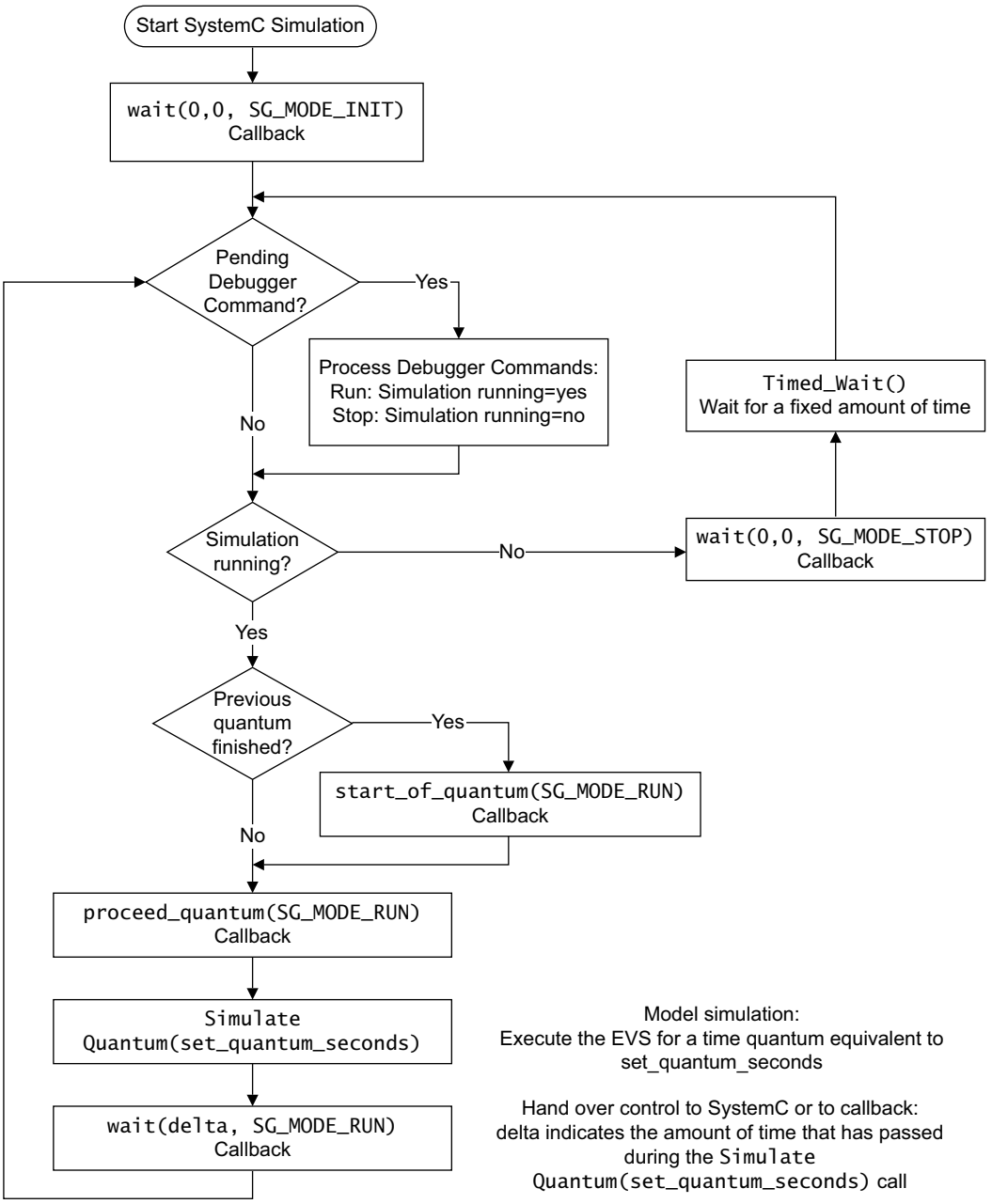

Figure 5-14 Standalone simulation with a debug server

5.8 Limitations

This section describes limitations of the current release.

Note

It is important to understand that the EVS do not provide any level of timing accuracy. These platforms have no implementation of timing or cycle accuracy, although the platforms are accurate on a functional level.

5.8.1 Single SC_THREAD and wait() calls

All exported models from within a FastModels EVS are exported within a single `SC_THREAD`. Therefore all initiators on the EVS side are seen from the SystemC scheduler as a single component.

Figure 5-15 shows a scenario where two processors in an EVS issue a transaction to peripherals modeled in SystemC (Peripheral 1 and 2). Both peripherals call `wait(10ms)` to finish the transaction. This results in an overall delay of 20ms. The correct delay would be 10ms if both processors in the EVS were modeled in a separate `SC_THREAD`.

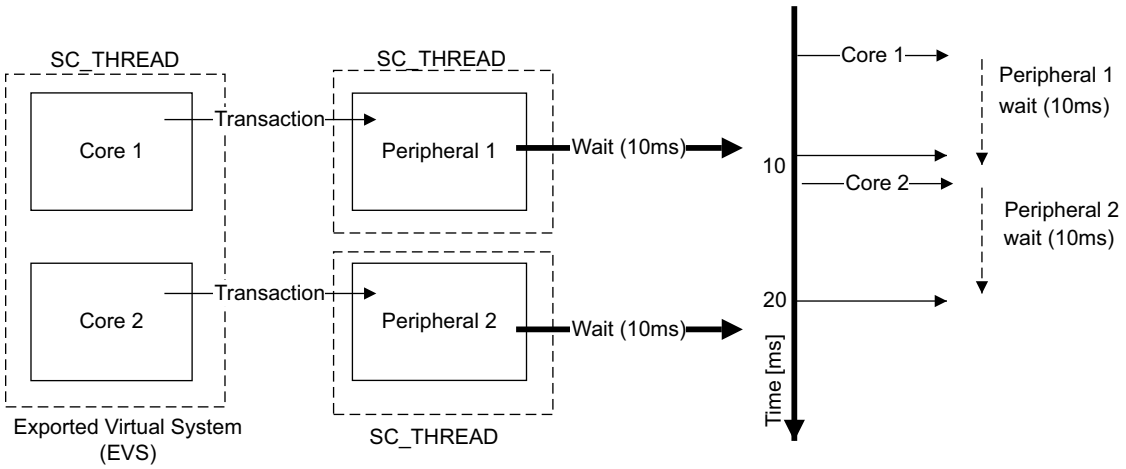

Figure 5-15 SC_THREAD wrapper for exported components

Note

Peripherals used with temporal decoupled models must not call `wait()`. It is recommended to model peripherals in a way to avoid `wait()` calls as EVS target functional correct behavior but is not timing accurate. This limitation is to be removed in a future release.

5.8.2 Single SC_THREAD and problems with re-entrancy

Figure 5-16 on page 5-43 shows two components in an EVS where Core 2 issues a blocking transaction to a peripheral (Peripheral 2). Peripheral 2 issues a blocking transaction to Peripheral 1 that in turn generates a blocking transaction to Core 1. The problem is that Transaction #1 blocks the `SC_THREAD` for the EVS and this results in a deadlock situation when Peripheral 1 tries to access Core 1. This limitation is to be removed in a future release.

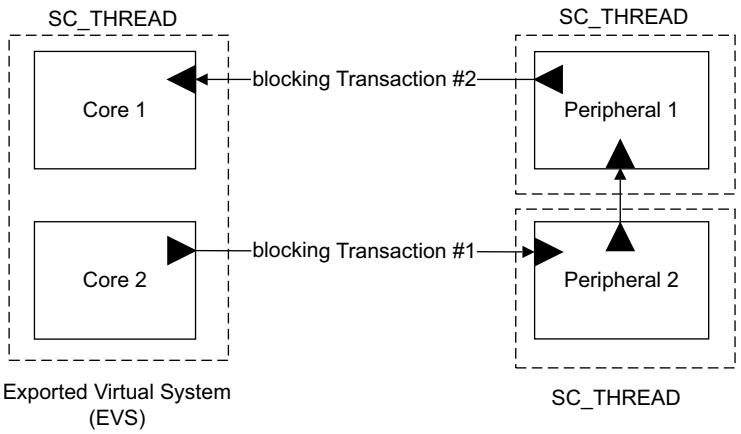

Figure 5-16 Blocking transactions in single SC_THREAD

5.8.3 Calling wait() in the middle of a transaction

If `wait()` is called in a middle of a transaction for externally triggered signals, it is not guaranteed that signals/transactions are always generated within the `SC_THREAD` context of the EVS. If a signal is triggered by an external event, such the WFI signal in the Cortex-A9, calling `wait()` in the behavior implementation of a peripheral that responds to this signal causes a runtime error. In Figure 5-17, Peripheral 1 issues a signal that causes the `wfi` signal on the processor to be set. The processor, directly responding to this signal change in its behavior (not buffered) generates a signal change on its `wfi` port with the context of `SC_THREAD` #1.

If it is required that the peripheral really issues a `wait()` in reaction to a signal that is changing, it must be buffered in the bridge between the EVS and SystemC. On the next activation of the bridge the signal can be set with the thread context of the EVS.

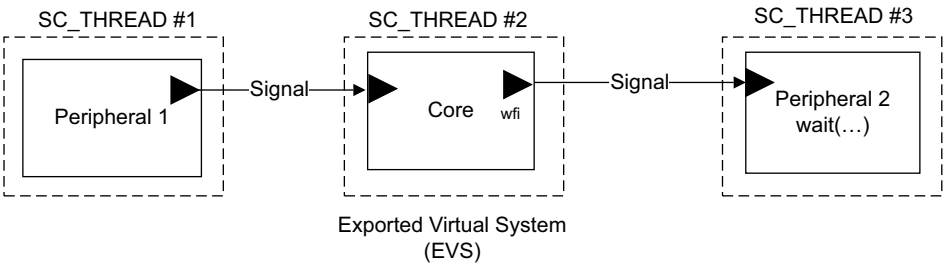

Figure 5-17 Calling wait() in the middle of a transaction

Note

The exported platform runs in temporal decoupled mode using a time quantum. Therefore targets (for example, Peripheral 2 in Figure 5-17) do not call `wait()`.

5.8.4 Timing annotation

TLM 2.0 supports the annotation of time to transactions. The EVS does not honor such timing as described by the scenarios shown in Figure 5-18 and Figure 5-19.

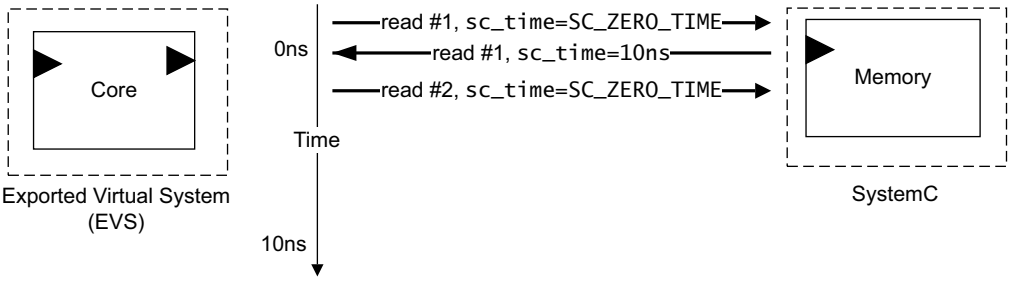

Figure 5-18 Transaction issued by EVS

For Figure 5-18, the component in the EVS processor issues a read transaction to the SystemC component Memory. The transaction immediately returns annotating a delay of 10ns. The EVS ignores this annotation and continues the execution (read transaction #2) for the time quantum as if the memory had responded with a `SC_ZERO_TIME` delay.

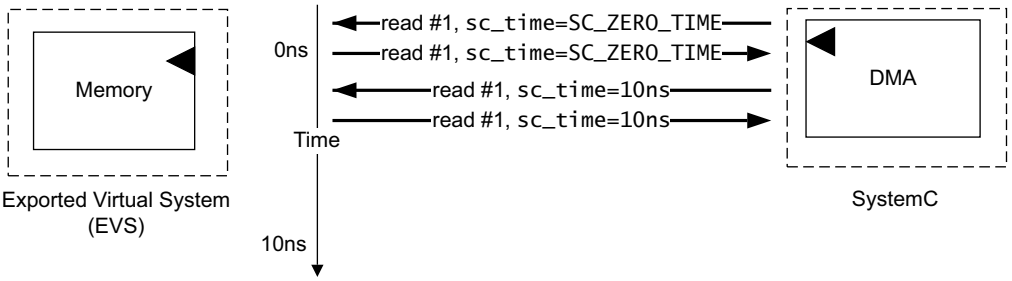

Figure 5-19 Transaction issued by SystemC

For Figure 5-19, a transaction is issued by a component in the SystemC domain (Core) that accesses a memory modeled in the EVS (Memory). The timing annotation of the transaction is never be changed by the EVS because it runs in a temporal-decoupled way that assumes all transactions are handled immediately. Figure 5-19 shows two read transactions that are issued by a DMA component. The first transaction, marked with `SC_ZERO_TIME`, is answered by the memory component in the EVS immediately. Therefore the timing annotation is not changed. The second transaction is similar and the annotation of 10ns is also not changed by the memory component. This behavior is inherent to the temporal decoupled fast models and does not change for future releases.

5.8.5 Code translation support for external memory

The processor components in the EVS make use of code translation for high simulation speed. Therefore they fetch data from external memory to translate it into host machine code. If the memory contents are changed outside of the scope of the processor, the data is inconsistent.

Figure 5-20 on page 5-46 shows this scenario. In step 1, the processor fetches the contents from the external memory by a *Direct Memory Interface* (DMI) read transaction. This code is translated into host machine code. If the memory content in the external memory is changed by the SystemC DMA controller, the processor is not notified. Therefore, the SystemC component is responsible to invalidate the DMI pointer so that the processor can reload the content.

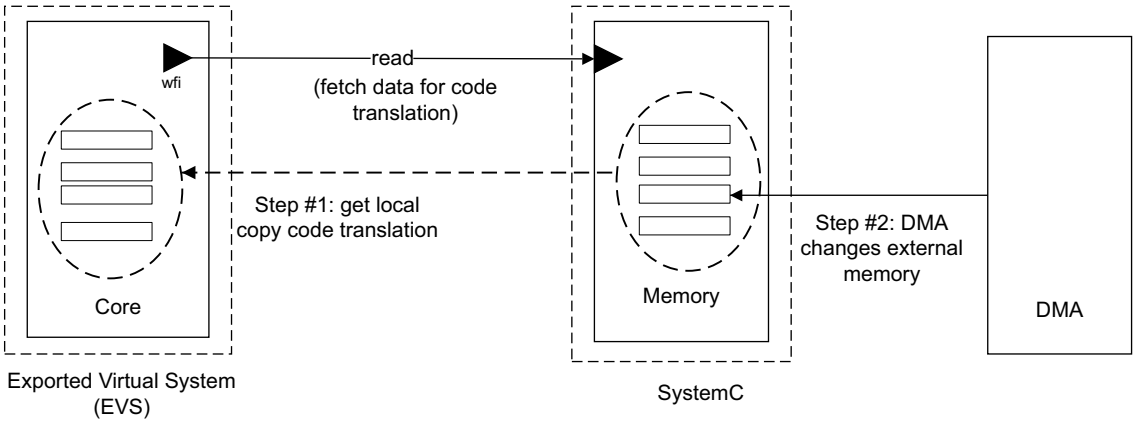

Figure 5-20 Local modification of external memory

Note

You must enable DMI accesses to instruction memory. If not, it is unsupported and:

- all accesses are modeled
- multiple spurious transactions are performed
- code translation is done per instruction instead of in blocks of instructions.

This results in performance being dramatically reduced.

5.8.6 Debug accesses

AMBA-PV offers two functions based on the TLM 2.0 base protocol to generate debug accesses:

- `debug_read(…)`
- `debug_write(…)`.

There is important limitation if such debug transactions are initiated from a SystemC master port to an EVS slave port such as shown in Figure 5-21.

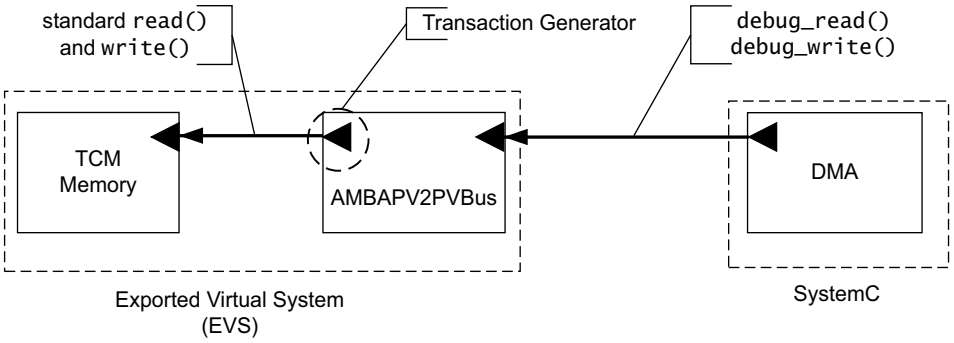

Figure 5-21 Debug accesses from a SystemC component

Debug transactions from a SystemC component (for Figure 5-21, DMA) can access `debug_read()` and `debug_write()` functions on the respective bus bridge (`ARMBAPV2PVBus`). However, the transactions that are generated on the bus connection are only standard non-debug transactions. The attached component cannot distinguish between a non-debug and a debug transaction, so it is assumed to be a normal transaction.

This limitation is to be removed in a future release.

Note

Debug accesses that are initiated by a master port in the EVS are correctly translated to debug accesses in the AMBAPV bridge (PVBus2AMBAPV).

Chapter 6
Creating a New Component

This chapter is a tutorial that covers the steps to create a new component. It contains the following sections:

6.1 Basic configuration

In addition to using the supplied components and integrating C libraries, you can also create your own custom LISA+ components. The LISA+ language is the base for new components and is described in the *LISA+ Language for Fast Models Reference Manual*, http://infocenter.arm.com/help/topic/com.arm.doc.dui0372-/index.html.

This section describes how to create a new LISA+ component and integrate it into the ARM dual processor system. The component doubles each character that is sent from the UART to the terminal. See Figure 6-1.

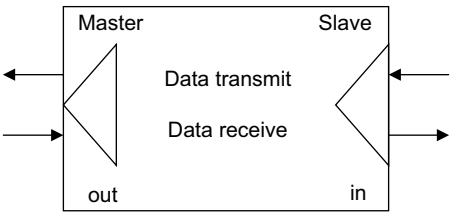

Figure 6-1 Character doubler component

The transmit direction of the component requires a dedicated behavior to double each character and treat escape sequences in a proper way.

6.1.1 Create a new LISA file

The new component is defined in a LISA source file:

1. Open the `VP_DualPlatform` project.

   Click on the **New** button to create an empty LISA file.

2. Navigate to the directory for the new file. For example:

   `$PVLIB_HOME\examples\VP_DualPlatform\LISA`

3. Enter a filename for the component. For example:

   `SerialCharDoubler.lisa`

   Click **Select** to set the filename.

4. The Workspace window shows an empty block diagram with the label `SerialCharDoubler?*`. The question mark indicated that the file is not part of the current system. The asterisk means that the file has not yet been saved.

5. Click the **Save** button or select **Save File** from the **File** menu.

6. Select **Add Current File** from the **Project** menu.

7. Click the **Files** tab of the Component window.

   The list now includes `SerialCharDoubler.lisa` as part of the project file `VP_DualPlatform.lisa`. The question mark and asterisk in the title are no longer present.

6.1.2 Resources section

The resources section describes the resources, including registers, memories, states, and other variables, of the component. In addition to annotated resources such as `REGISTER` and `MEMORY`, variable definitions can also be used.

The doubler component only requires a single variable that indicates if the current character sequence follows an escape character:

1. Select the **Source** tab in the Workspace for the file `SerialCharDoubler.lisa`.
2. Add the `resources` section to the code as shown in Example 6-1.

Example 6-1 Resource section with marker

```
// This file was generated by System Generator Canvas
// -------------------------------------------------
component SerialCharDoubler
{
    composition
    {
    }
    connection
    {
    }
    resources
    {
    // remember whether we are inside an escape sequence
    // which came through 'in'
       bool inEscape;
    }
}
```

6.2 Adding ports

Two ports are required for this component. See Figure 6-1 on page 6-2:

- the slave port receives the serial data
- the master port writes out the doubled characters.

Both ports use serial data and they must therefore use the SerialData protocol that is provided in Fast Model Portfolio.

Add the ports to the component:

1. In the Block Diagram view of the `SerialCharDoubler` component, right click into the blank diagram and select context menu item **Add port...**
2. The Self Port dialog is displayed. See Figure 6-2. Enter `in` for the ports **Instance Name**.

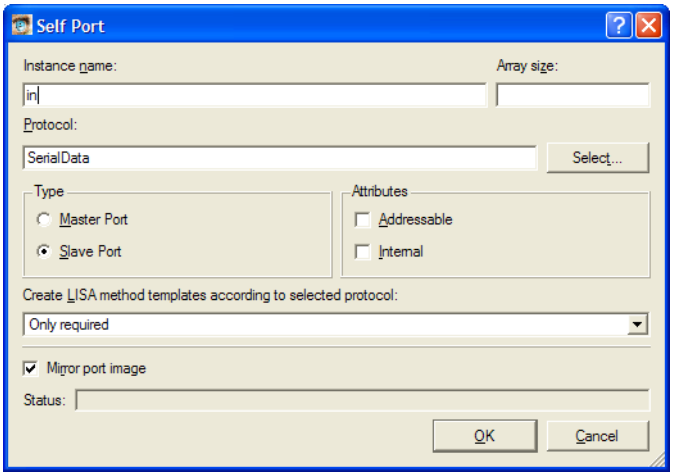

Figure 6-2 Self Port dialog

3. For **Type** select **Slave Port**.
4. To choose the protocol, click the **Select...** button **Protocol**. Select **SerialData** from the displayed list and click **OK**.

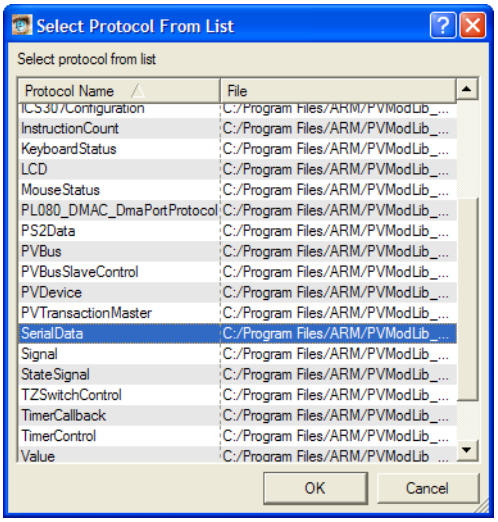

Figure 6-3 Select Protocol dialog

5. Leave the Self Port dialog by pressing the **OK** button. A port is added to the block diagram. Click to drop the port on the canvas.

6. Repeat the procedure for the `out` master port:
   - **Instance Name** is `out`
   - **Type** is `Master Port`
   - **Protocol** is `SerialData`.

7. Click the **Source** tab for the `SerialCharDoubler` component. The code is updated as shown in Example 6-2:

Example 6-2 Ports added to char doubler

```
// This file was generated by System Generator Canvas
// -------------------------------------------------
component SerialCharDoubler
{
    composition
    {
    }
    connection
    {
    }
    resources
    {
        // remember whether we are inside an escape sequence
        // which came through 'in'
        bool inEscape;
    }
    slave port<SerialData> in
    {
        behavior dataTransmit(uint16_t data):void
        {
            // TODO: place your code here
        }

        behavior dataReceive():uint16_t
        {
            // TODO: place your code here
        }

        behavior signalsSet(uint8_t signal):void
        {
            // TODO: place your code here
    }

        behavior signalsGet():uint8_t
        {
            // TODO: place your code here
        }
    }
    master port<SerialData> out;
    {
        behavior dataTransmit(uint16_t data):void
        {
            // TODO: place your code here
    }

        behavior dataReceive():uint16_t
        {
            // TODO: place your code here
```

```
        }

        behavior signalsSet(uint8_t signal):void
        {
            // TODO: place your code here
        }

        behavior signalsGet():uint8_t
        {
            // TODO: place your code here
        }
    }
}
```

6.3 Behavior section

The functionality of the component is expressed by behaviors. In the `SerialCharDoubler` component, you must manage two types of behavior:

Port behaviors

The implementations of behaviors supported by the used protocol. Slave ports implement slave port behaviors while master port behaviors are implemented for master ports.

Special-purpose behaviors

Certain reserved behaviors with a special meaning in components.

6.3.1 View the SerialData protocol

Examine the default SerialData protocol that is used by the master and slave ports:

1. Select the **Protocols** tab in the Component window.
2. Double-click on the `SerialData` protocol.
3. The protocol source is displayed in the Workspace window:

```
protocol SerialData{
 ...
    slave behavior dataTransmit(unit16_t data);
    slave behavior dataReceive():unit16_t;
    slave behavior signalsSet(uint8_t signal);
    slave behavior signalsGet():uint8_t;
}
```

4. The following characteristics apply to the `SerialData` protocol:
   - All behaviors are `slave` behaviors.
   - None of the behavior statements contains the keyword `optional`. The slave must therefore implement all of the behaviors.

6.3.2 Adding the new behavior to the SerialCharDoubler slave port

The slave port must alter the standard behavior for the transmit direction so that characters are doubled.

The receive direction is not changed and this requires only a call to the same behavior of master port `out` to transfer the data from the slave port to the master port.

The modification of the transmitted data is implemented in slave behavior `dataTransmit`. Select the **Source** tab to display the LISA source code for `SerialCharDoubler.lisa`, and modify the `SerialData` slave port `dataTransmit` behavior as shown in Example 6-3:

Example 6-3 Slave port behavior

```
slave port<SerialData> in
{
    behavior dataTransmit(uint16_t data)
    {
        // start of escape sequence?
        if(data == '\33')
        {
                inEscape = true;
```

```
                }
                if(inEscape)
                {
                    // leave escape sequences untouched (only send one char)
                    out.dataTransmit(data);
                }
                else
                {
                    // duplicate char on output
                    out.dataTransmit(data);
                    out.dataTransmit(data);
                }
                // end of escape sequence?
                if(isalpha(data))
                {
                    inEscape = false;
                }
            }
    ...
    }
```

The data is not doubled for escape sequences but is doubled for other characters. The end of escape sequences is detected by the occurrence of an alphabetic character and the resource `inEscape` is reset.

The `dataReceive()` behavior calls the respective behavior of the master port as shown in Example 6-4. Modify the behavior as shown:

Example 6-4 Slave port receive behavior

```
behavior dataReceive():uint16_t
{
    return out.dataReceive();
}
```

Signals are not effected by the doubling, so the behavior to set and get signals are wrappers on the master port behavior as shown in Example 6-5:

Example 6-5 Slave port get and set signals

```
behavior signalsSet(uint8_t signal)
{
    out.signalsSet(signal);
}

behavior signalsGet():uint8_t
{
    return out.signalsGet();
}
```

6.3.3 Master port

The character doubler component requires a master port to initiate incoming requests from the slave port and forward them to the connected slave port of another component.

The only action required in the code is declaring the master port:

```
master port<SerialData> out;
```

Remove or comment out any behavior definitions in the master port section in `SerialCharDoubler.lisa`.

The actual connection is done using connection mode in the Block Diagram view.

The master port uses the same protocol as the slave port. This is not a requirement, but is useful here because only one behavior is implemented, `dataTransmit`, and the rest is forwarded.

6.3.4 Special purpose behaviors

Certain behavior names have a special meaning in components. These special purpose behaviors are called implicitly and automatically in certain simulation phases. See Table 6-1 for a list.

Table 6-1 Special-purpose behaviors

Behavior	Description
`behavior init`	This behavior is called implicitly once when the simulation is started. The code in this behavior is executed only once. It is intended to allocate memory and other resources used during simulation.
`behavior reset(int level)`	This behavior is called whenever the simulation is reset by the user. Reset might be called multiple times. Memory must not be allocated in this behavior because it might be invoked several times.
`behavior terminate`	This behavior is the counterpart to `behavior init` and is called when the simulation terminates. It is intended to free any allocated memory and resources that were allocated in `behavior init`.

Init behavior

At system startup, each component is called for several phases of the simulation. The first call is the `init` phase. The `init` behavior implements what the component must do at this stage.

The `SerialCharDoubler` component only requires setting the boolean flag that indicates that the component is about to transmit an escape sequence. Place the code in Example 6-6 before the slave port definition in `SerialCharDoubler.lisa`:

Example 6-6 init behavior

```
behavior init
{
    inEscape = false;
}
```

Reset behavior

The doubler component `init` phase performs everything that is required at startup. There is, therefore, no requirement for a reset behavior.

Terminate behavior

The terminate behavior is typically responsible for freeing all resources allocated in the init phase. The doubler component does not allocate any resources in the init phase, the terminate behavior can therefore be omitted.

A different component type, for example a memory component, might allocate memory structures to represent data or register storage. Allocated structures must be freed in this phase.

6.4 Using the SerialCharDoubler component in the system

This section describes how to use the `SerialCharDoubler` component in the dual processor system.

Insert the new component in the dual processor system:

1. Remove the connection between the `uart1a` and the `term1` component.
2. Add the `SerialCharDoubler` component to the system.

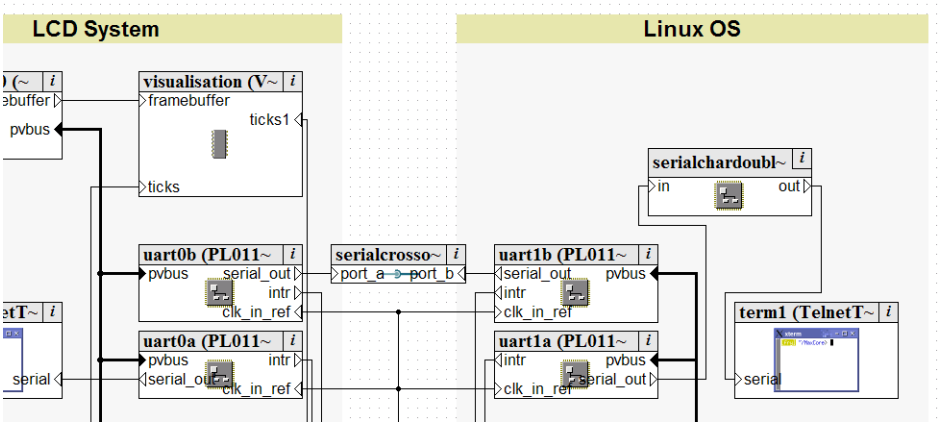

Figure 6-4 SerialCharDoubler inserted in dual processor system

3. Connect the `serial_out` port of the `uart1a` component to the `in` port of the `SerialCharDoubler` component.
4. Connect the `out` port of the `Serialchardoubler` component to the `serial` port of the `term1` component. See Figure 6-4.
5. Click the **Build** button on the toolbar to rebuild the system.
6. After the compilation is finished, click the **Debug** button.
7. Follow the steps already described in *Debugging the system* on page 2-20.
8. The output of the Linux terminal is doubled by `SerialCharDoubler` as shown in Figure 6-5.

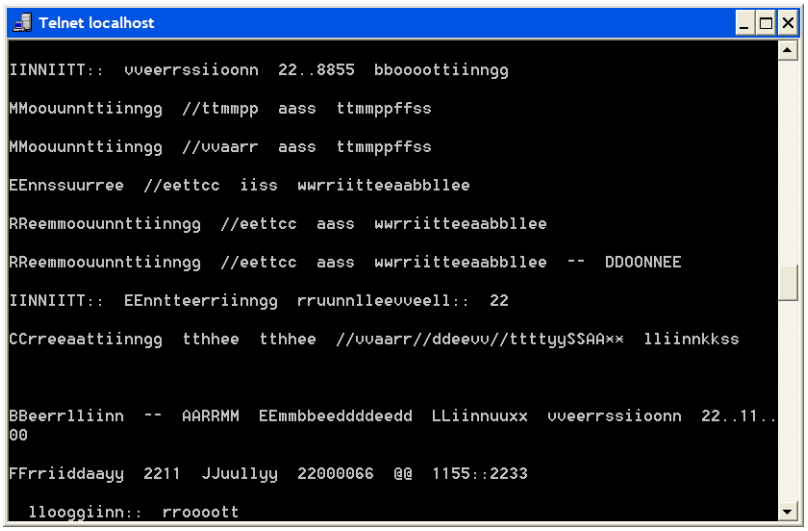

Figure 6-5 Linux booting with SerialCharDoubler component

Appendix A
Building and Running the EB FVP Example System

This appendix describes the steps required to build and run the supplied EB FVP platform example. It contains the following sections:

A.1 Using System Canvas to build the platform model

This section describes how to use System Canvas to build an EB FVP.

Note

Using other versions might not be successful. Filenames, interfaces and procedures might be different from those documented in this section.

Model projects supplied with one version of Fast Models are not necessarily compatible with a different version.

The model build subdirectories are in the `%PVLIB_HOME%\examples\FVP_EB` directory.

A.1.1 Building an EB Fixed Virtual Platform

This section uses the EB1176 Fixed Virtual Platform as an example. If you are building a different model, substitute your model name.

1. Start System Canvas for Fast Models.
2. Click the **Open** button on the System Canvas toolbar.
3. Navigate to the location of the EB1176 Fixed Virtual Platform project. This is `%PVLIB_HOME%\examples\FVP_EB\Build_EB1176\FVP_EB1176.sgproj`. Click **Open** to load the project.
4. By default, you generate a Release build of the model. To change to a Debug build, select the **Select Active Project Configuration** drop-down list on the System Canvas toolbar and change the configuration.
5. Click the **Build** button on the System Canvas toolbar to build the model.

   Note

   - Depending on your preference setting, a system check might be performed. Click **Proceed** to start the build.
   - Depending on the speed and processor loading of your computer, builds, particularly Release builds, can take several minutes to finish.

6. If you used the default project settings, the build generates a `Windows-Release-`*`compiler`*`\cadi_system_Windows-Release-`*`compiler`*`.dll` or `Linux-Release-`*`compiler`*`/cadi_system_Linux-Release-`*`compiler`*`.so` object.

   A `dll` object is generated on Microsoft Windows systems and a `so` object is generated on Linux systems.

   `compiler` is the name of the compiler used.

   If you created a Debug object, substitute `Debug` for `Release` in the directory and filenames.

   The object can be used with Model Shell or Model Debugger. See *Using Model Debugger to run a system model* on page A-5.

   You can also use the object with ARM Profiler or RealView Debugger. See *Using RealView Debugger to run a system model* on page A-8.

A.1.2 Packaging the system model for distribution

If you are running the model in the same environment where the model was created, all DLLs and support files are already available to the model.

If, however, you are packaging the system model for use on a system that does not have Fast Models installed, you must include the required DLLs and support files as part of the package.

Note

When the model is built, the output files are placed in a deployment subdirectory in the directory that contains the source project file. By default, the name of the deployment subdirectory matches the build options selected from the **Select Active Project Configuration** drop-down list on the System Canvas toolbar. To check the exact build directory that is being used, look at the **Build directory** field in the Project Settings dialog in System Canvas.

You can also create your own custom build configuration option. In this case, the name of the deployment directory matches the name of your custom build configuration.

For more information on deployment options for user-supplied files, see *File/Path Properties dialog* on page 4-36.

Packaging the model for use on Linux

The following steps must be followed if your model is to be used by others:

- Ship any user-defined files such as text files or application files
- Ship any shared libraries for custom components you created
- Ship the following files from the deployment directory for the model you built:
  - `libMAXCOREInitSimulationEngine.so.2`
  - `libarmctmodel.so`
  - `libSDL-1.2.so.0.11.2` (required if your model uses the PL041 AACI component or uses any visualization components).
- If Ethernet components are used, ensure that the end user has a TAP device configured.

Packaging the model for use on Microsoft Windows

The following steps must be followed if your model is to be used by others:

- Ship any user-defined files such as text files or application files
- Ship any DLLs for custom components you created
- Ship the following files from the deployment directory for the model you built:
  - `libMAXCOREInitSimulationEngine.2.dll`
  - `SDL.dll` (required if your model uses the PL041 AACI component or uses any visualization components)
  - `armctmodel.dll`.
- If Ethernet components are used, use ModelNetworking: `$PVLIB_HOME/ModelNetworking`.
- Ensure that the end user has one of the following applications installed:
  - a compatible version of Microsoft Visual Studio installed for running Debug builds (VC2008 SP1 or VC2010 SP1, and the latest security upgrades)
  - the Microsoft Visual Studio Redistributable Package (2008 SP1 or 2010 SP1) for Release builds.

A.2 Connecting to the model

This section describes the different ways to connect to a system model:

- *Using Model Shell to run the system model*
- *Using Model Debugger to run a system model* on page A-5
- *Using RealView Debugger to run a system model* on page A-8.

A.2.1 Using Model Shell to run the system model

To start the EB FVP using Model Shell, change to the directory that contains your model file and enter the following at the command prompt:

```
model_shell --cadi-server --model model_name [--config-file filename] [-C
instance.parameter=value] [--application app_filename]
```

where:

model_name is the name of the model file. By default the filename for Debug versions is typically `cadi_system_Windows-Debug-compiler.dll` on Microsoft Windows, or `cadi_system_Linux-Debug-compiler.so` on Linux, where *compiler* is the name of the compiler used to build the model.

filename is the name of your optional plain-text configuration file. Configuration files simplify managing multiple parameters. See *Using a configuration file* on page A-5.

instance.parameter=value

is the optional direct setting of a configuration parameter. See *Using the command line* on page A-5.

app_filename is the filename of an image to load to your model at startup.

Note

On Microsoft Windows, it might be necessary to add the directory that contains the Model Shell executable to your `PATH`. This location of the directory is typically:

```
$MAXCORE_HOME\FastModelTools_X.Y\bin
```

For example:

```
C:\Program Files\ARM\FastModelTools_X.Y\bin
```

You can also start Model Shell from the System Canvas GUI.

For more information on all Model Shell options, see the *Model Shell for Fast Models Reference Manual*, `http://infocenter.arm.com/help/topic/com.arm.doc.dui0457-/index.html`.

Starting the model opens the Fixed Virtual Platform CLCD display as described in *Using the CLCD window* on page A-13.

After the EB FVP starts, you can connect to it using a CADI-compliant debugger such as Model Debugger or RealView Debugger. See *Connecting to a running Model Shell using Model Debugger* on page A-6 and *Connecting to a running model using RealView Debugger* on page A-10.

Using a configuration file

To configure a model that you start from the command line with Model Shell, include a reference to an optional plain text configuration file as described in *Using Model Shell to run the system model* on page A-4.

Comment lines in the configuration file must begin with a # character.

Each non-comment line of the configuration file contains:

- the name of the component instance
- the parameter to be modified and its value.

  Boolean values can be set using either `true`/`false` or `1`/`0`. Strings must be enclosed in double quotes if they contain whitespace.

A typical configuration file is listed in Example A-1:

Example A-1 Configuration file

```
# Disable semihosting using true/false syntax
cluster.cpu0.semihosting-enable=false
#
# Enable the boot switch using 1/0 syntax
motherboard.sp810_sysctrl.use_s8=1
#
# Set the user switch position
motherboard.ve_sysregs.user_switches_value=1
```

Using the command line

You can define model parameters when you invoke the model by using the `-C` switch when starting Model Shell. You can also use `--parameter` as a synonym for the `-C` switch. See *Using Model Shell to run the system model* on page A-4. Use the same syntax as for a configuration file, but each parameter must be preceded by the `-C` switch.

A.2.2 Using Model Debugger to run a system model

This section describes the various ways you can use Model Debugger to connect to a model:

- *Starting Model Debugger from System Canvas*
- *Connecting to a running Model Shell using Model Debugger* on page A-6
- *Configuring the system from Model Debugger* on page A-7.

Starting Model Debugger from System Canvas

You can start the EB FVP using Model Debugger directly from System Canvas:

- Build and load your model project.
- Click the **Debug** button in the System Canvas toolbar.
- Model Debugger is displayed with the EB FVP preloaded.
- Before the running the model, you can user the Configure Model Parameters dialog to define instantiation parameters. See *Configuring the system from Model Debugger* on page A-7.

Alternatively you can start Model Debugger separately and load the model library yourself. If you are using Microsoft Windows, you must ensure that you have all required DLLs on your `PATH`. See the note at the end of *Building an EB Fixed Virtual Platform* on page A-2. For more information about using and configuring Model Debugger, see the *Model Debugger for Fast Models User Guide*, `http://infocenter.arm.com/help/topic/com.arm.doc.dui0314-/index.html`.

You can configure certain instantiation-time parameters using the Configure Model Parameters dialog before the model starts.

Connecting to a running Model Shell using Model Debugger

1. In Model Debugger, select **File → Connect to Model**.
2. The Connect Remote dialog shown in Figure A-1 is displayed.

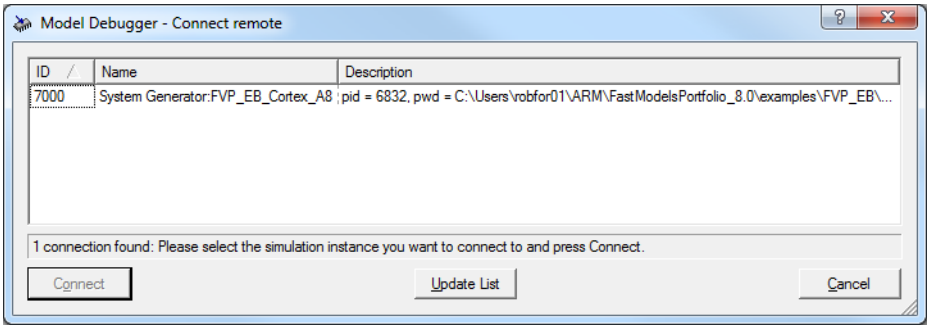

Figure A-1 Model Debugger Connect remote dialog

3. In the Connect Remote dialog, select the entry for the EB FVP.
4. Click **Connect**.
5. The Select Target dialog shown in Figure A-2 on page A-7 is displayed.

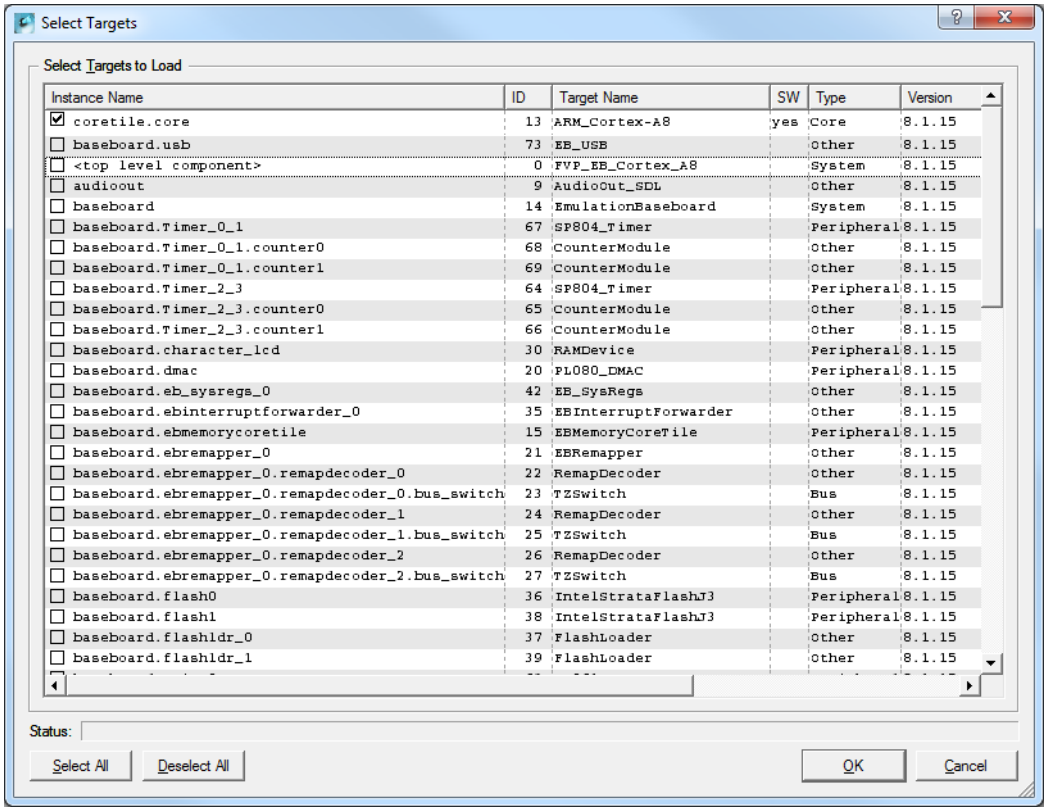

Figure A-2 Model Debugger Select Targets dialog

6. Choose the targets to load. By default, the processor corresponding to the ARM processor in your EB FVP is selected, and must be loaded. Click **OK**.

7. Model Debugger prompts you to load an application (image) to the model. Select the application image from the Load Application dialog and click **Open**.

You can now control and debug the model using Model Debugger. You can make multiple debugger connections to a single model instance.

To shut down the model, return to the command window that you used to start the model and press **Ctrl+C** to stop the CADI server. The Model Shell process must be in the foreground before you can shut it down.

Configuring the system from Model Debugger

If you load a model in Model Debugger, the Configure Model Parameters dialog shown in Figure A-3 on page A-8 is displayed to enable you to define instantiation-time parameters.

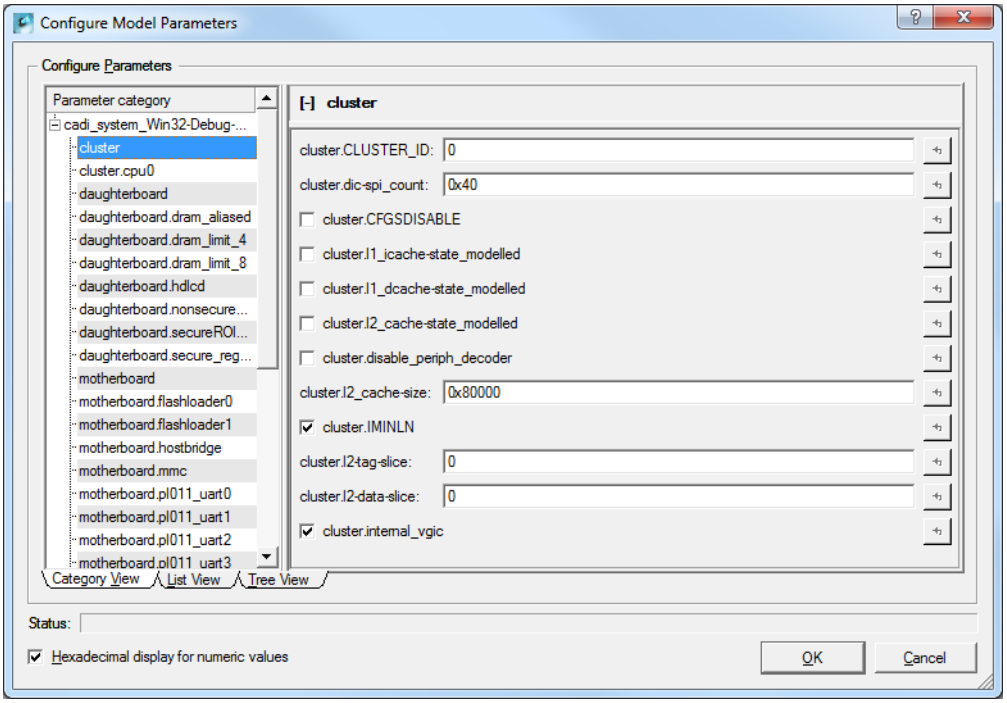

Figure A-3 Configure Model Parameters dialog

To view the configuration parameters:

1. Click **Category View** to expand the tree view in the left-hand Parameter category pane
2. Click a component in the list, for example **cluster**, to highlight the parameter category to modify
3. In the right-hand pane, select or deselect check boxes for boolean parameters or enter data such as strings or addresses in fields.

   Hovering the mouse pointer over a parameter shows a description of the parameter and its default value.

   You can change the numeric display from the default hexadecimal to decimal by clearing the **Hexadecimal display for numeric values** box in the lower left corner.

A.2.3 Using RealView Debugger to run a system model

For information about the ways you can connect to an EB FVP, see:

- *Starting the model as an FVP connection in RealView Debugger* on page A-9
- *Configuring the system from RealView Debugger* on page A-9
- *Connecting to a running model using RealView Debugger* on page A-10.

Note

You must not connect to more than one FVP at any one time in RealView Debugger. If you do, you might experience unexpected behavior or crashes.

Starting the model as an FVP connection in RealView Debugger

To add the EB FVP to the RealView Debugger Connect to Target window under the *Fixed Virtual Platform* (FVP) debug interface configuration:

1. In RealView Debugger, select **Target → Connect to Target...** to open the Connect to Target window.
2. Click the **Add** button beside the *Fixed Virtual Platform* (FVP) debug interface name. This opens the Model Configuration Utility window:
   - The FVP models distributed with RVDS are automatically displayed in the list based on the path set by the `ARM_FVP_PATH` variable.
   - If you have separately installed and built an EB FVP yourself using System Canvas, you can load the models from the `Build_EB`*`processor`*`\`*`platform-build-compiler`* subdirectory of the `%PVLIB_HOME%\examples\FVP_EB\`directory where:

     `processor` is one of the supplied processor models. such as ARM1176.

     `platform-build-compiler`
     is the platform, build type, and compiler, for example Linux-Release-GCC-4.1.

     See *Building an EB Fixed Virtual Platform* on page A-2.
3. Select the model to use in the Models pane on the left side of the Model Configuration Utility:
   - Configure the device parameters if required. See *Configuring the system from RealView Debugger*.
   - After you have finished, or if no parameters required configuration, click **OK**.
4. In the RealView Debugger Connect to Target window, double-click on your newly-created target to connect to it. If you are grouping targets by Configuration, expand the target connection tree view to see your target instance. Connecting to a target opens a CLCD window.

Configuring the system from RealView Debugger

In RealView Debugger, you can configure the EB FVP parameters before you connect to the model and start it:

- If you are adding the particular EB FVP to the RealView Debugger Connect to Target list, the Model Configuration Utility dialog box is displayed automatically.
- Alternatively you can right click on your existing target in the Connect to Target list and select **Configure...** to open the same dialog.

To view the configuration parameters, scroll down the list of devices shown in the upper right pane. Selecting a device populates the lower right pane with the device parameters.

To change a parameter value, select a boolean from the drop down list, or enter data such as strings or addresses by clicking in the relevant field. Hovering the mouse pointer over a device or parameter shows a description or additional information.

To change the numeric display from decimal to hexadecimal, right click on the parameter value in the lower right pane and select **Hexadecimal Display** from the context menu.

Connecting to a running model using RealView Debugger

You can use RealView Debugger to connect to an already running Model Shell instance of the EB FVP. You can make multiple debugger connections to a single model instance.

1. Start Model Shell, if it is not already running, as described in *Using Model Shell to run the system model* on page A-4.
2. In RealView Debugger, select **Target → Connect to Target...** to open the Connect to Target window.
3. Click the **Add** button beside the ModelProcess debug interface name.
4. The debugger detects any running CADI servers and displays them in a pop-up window. The coretile in your running EB FVP is automatically selected. Click **OK**.
5. In the RealView Debugger Connect to Target window, double-click on your newly-created ModelProcess target to connect to it.

A.3 Running an application on the system model

This section describes how to load and run an application on a system model. You can use one of the supplied example system models or a custom model you built by following the examples in *Using System Canvas to build the platform model* on page A-2.

The `brot.axf` example application runs on all versions of the EB FVP.

This demo application provides a simple demonstration of rendering an image to the CLCD display.

Source code is supplied:

- For Fast Model Portfolio, the examples are in the `%PVLIB_HOME%\images` directory.
- For RVDS, the source code is available in the directory `%ARMROOT%\RVDS\Examples\4.1\`*`nn`*`\<`*`platform_name`*`>\mandelbrot`, where *nn* is a version number.

Note

These applications are provided for demonstration purposes only and are not supported by ARM. The number of examples or implementation details might change with different versions of the system model.

For more information on debugging applications, see Chapter 3 *Debugging*, the *Model Debugger for Fast Models User Guide*, and the *RealView Debugger User Guide*.

A.3.1 Running the brot application in Model Debugger

To load and run the `brot.axf` image in Model Debugger:

1. Start Model Debugger and connect to the system model as described in *Using Model Debugger to run a system model* on page A-5.
2. Click the **Open** button on the main toolbar to open the Load Application dialog that enables you to select the application file to load.
3. Browse to the location of the `brot.axf` image. Click **Open** to load the image to the target.
4. Click the **Open** toolbar button again and browse to the location of the `brot.c` source file. Click **Open** to load the source into the Model Debugger source pane.
5. Click in the source pane and place a breakpoint on the `render` function on line 154 of the source code.
6. Press **F5**, or select **Run** from the **Debug** menu, until the CLCD window displays an image similar to that shown in Figure A-4 on page A-12.

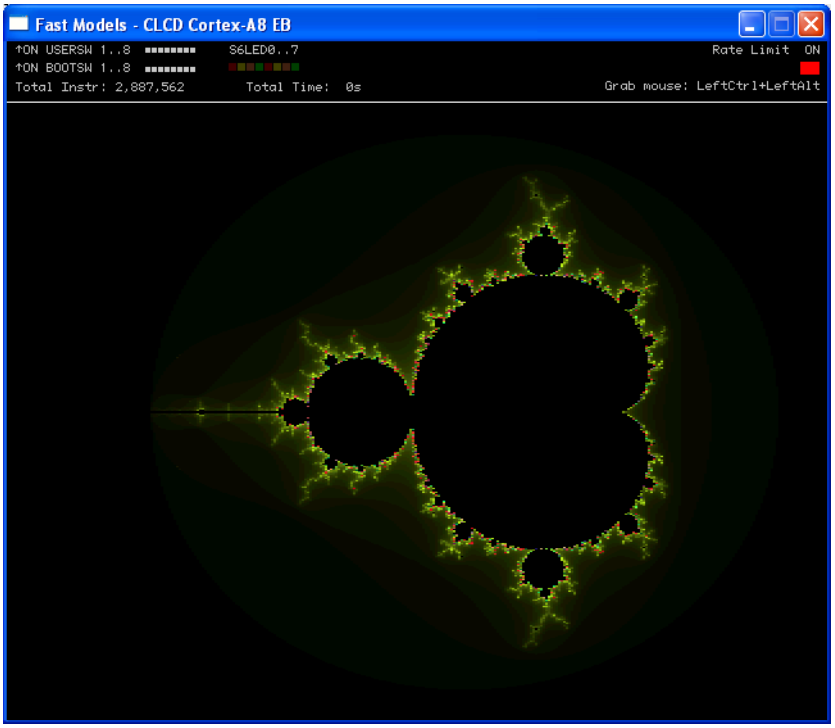

Figure A-4 CLCD window

7. Repeatedly press **F5** and notice how the CLCD window changes.

A.3.2 Running the brot application in RealView Debugger

The `brot.axf` example application runs on all versions of the EB FVP.

This demo application provides a simple demonstration of rendering an image to the CLCD display. Source code is supplied.

If you are using the Fast Model Portfolio, the examples are in the `%PVLIB_HOME%\images` directory.

In RVDS, the source code is available in the directory `%ARMROOT%\Examples\4.1\`*nn*`\platform\mandelbrot`, where *nn* is a version number.

Note

These applications are provided for demonstration purposes only and are not supported by ARM. The number of examples or implementation details might change with different versions of the system model.

It might be necessary to build the `brot.axf` image before loading the application. Instructions and build scripts are in `%ARMROOT%\Examples\4.0\`*nn*`\<`*platform_name*`>\mandelbrot`, where *nn* is a version number.

To load and run the `brot.axf` image in RealView Debugger:

1. Start RealView Debugger and connect to the system model as described in *Using RealView Debugger to run a system model* on page A-8.
2. Select **Load Image** from the **Target** menu. The Select Local Files to Load dialog is displayed.
3. Browse to the location of `brot.axf`, select it and click **Open**.

4. Select the **brot.c** tab in the RealView Debugger main window and place a breakpoint on the `render` function, as shown in Figure A-5.

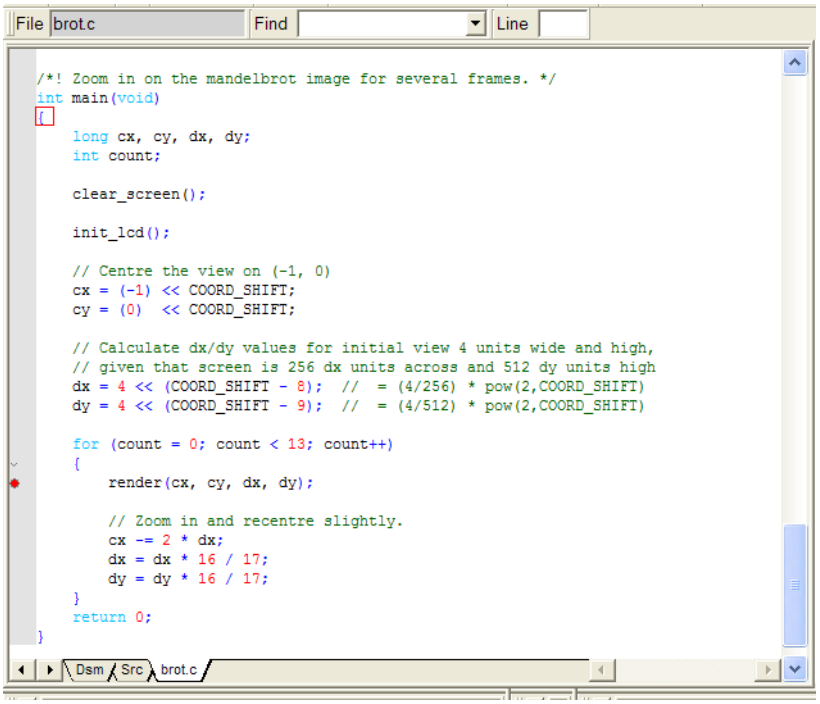

Figure A-5 Breakpoint in brot.c

5. Press **F5**, or select **Run** from the **Debug** menu, until the CLCD window is displayed as shown in Figure A-4 on page A-12.

6. Repeatedly press **F5** and observe the changes in the CLCD window.

A.3.3 Using the CLCD window

For both RealView Debugger and Model Debugger, the CLCD window is displayed when the FVP starts. The window title is `Fixed Virtual Platform CLCD`.

This window represents the contents of the simulated color LCD framebuffer. It automatically resizes to match the horizontal and vertical resolution set in the CLCD peripheral registers. For more information on the CLCD model components and other peripherals, see the *Fast Models Reference Manual*, `http://infocenter.arm.com/help/topic/com.arm.doc.dui0423-/index.html`.

Figure A-6 shows the EB FVP CLCD window immediately after being started. This is the default state.

Figure A-6 CLCD window at startup

The top section of the CLCD window displays the following status information:

USERSW Eight white boxes show the state of the EB User DIP switches:

These represent switch S6 on the EB hardware, USERSW[8:1], that is mapped to bits [7:0] of the SYS_SW register at address `0x10000004`.

The switches are in the off position by default. Click in the area above or below a white box to change its state.

BOOTSW Eight white boxes showing the state of the EB Boot DIP switches.

These represent switch S8 on the EB hardware, BOOTSEL[8:1], that is mapped to bits [15:8] of the SYS_SW register at address `0x100000004`.

The switches are in the off position by default.

Note

ARM recommends you configure the Boot DIP switches using the `boot_switch` model parameter rather than by using the CLCD interface.

Changing Boot DIP switch positions while the model is running can result in unpredictable behavior.

S6LED Eight colored boxes indicate the state of the EB User LEDs.

These represent LEDs D[21:14] on the EB hardware, that are mapped to bits [7:0] of the SYS_LED register at address `0x10000008`. The boxes correspond to the red/yellow/green LEDs on the EB hardware.

Total Instr A counter showing the total number of instructions executed.

The system model models provide a programmer's view of the system, so the CLCD displays total instructions rather than total processor cycles. Timing might differ substantially from the hardware because:

- the bus fabric is simplified
- memory latencies are minimized
- cycle approximate processor and peripheral models are used.

In general bus transaction timing is consistent with the hardware, but timing of operations within the model is not accurate.

Total Time A counter showing the total elapsed time, in seconds.

This is wall clock time, not simulated time.

Rate Limit A feature that disables or enables fast simulation.

The system model is highly optimized, so your code might run faster than it would on real hardware. This might cause timing issues.

If Rate Limit is enabled, the default, simulation time is restricted so that it more closely matches real time.

Click on the square button to disable or enable Rate Limit. The text changes from ON to OFF and the colored box becomes darker when Rate Limit is disabled. Figure A-6 on page A-13 shows the CLCD with Rate Limit disabled.

Note

You can control whether Rate Limit is enabled by using the `rate_limit-enable` parameter when instantiating the model.

If you click on the **Total Instr** or **Total Time** items in the CLCD, the display changes to show two different items as shown in Figure A-7 on page A-15. You can click on the items again to toggle between the original and alternative displays.

Figure A-7 CLCD window alternative display

Instr/sec Shows the number of instructions executed per second of wall clock time.

Perf Index The ratio of real time to simulation time. The larger the ratio, the faster the simulation runs. If you enable the Rate Limit feature, the Perf Index approaches unity.

You can reset the simulation counters by resetting the model.

If the CLCD window has focus:

- keyboard input is translated to PS/2 keyboard data.
- mouse activity over the window is translated into PS/2 relative mouse motion data. This is then streamed to the KMI peripheral model FIFOs.

Note

The simulator only sends relative mouse motion events to the model. As a result, the host mouse pointer does not necessarily align with the target OS mouse pointer.

You can hide the host mouse pointer by pressing the **Left Ctrl**+**Left Alt** keys. Press the keys again to redisplay the host mouse pointer. Only the **Left Ctrl** key is operational. The **Right Ctrl** key on the right of the keyboard does not have the same effect.

If you prefer to use a different key, use the `trap_key` configuration option. For more information, see the CADI parameter documentation in the *Fast Models Reference Manual*, http://infocenter.arm.com/help/topic/com.arm.doc.dui0423-/index.html.

Appendix B
Red Hat Linux Dependencies

This appendix describes the dependencies for Red Hat Linux. It contains the following sections:

B.1 About Red Hat Linux dependencies

Some library objects or applications have dependencies on other library files.

Fast Models requires some packages to be installed that are part of Red Hat Enterprise Linux, but which might not be installed by default.

If your Red Hat Enterprise Linux installation is subscribed to the Red Hat Network, or if you are using CentOS rather than Red Hat Enterprise Linux, dependencies might be installed from internet sources. Otherwise you must have your installation media present to install dependencies.

Some packages might have dependencies on other packages. If you install using the Add/Remove software GUI tool or the `yum` command line tool, these dependencies are automatically resolved. 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 at the terminal command line:

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

The following text is output to indicate that the library is contained by version `2.3.2-95.37` of `glibc` package:

```
glibc-2.3.2-95.37
```

B.2 Dependencies for Red Hat Enterprise Linux 4

The Red Hat preview release of GCC 4.1 packaged with Red Hat Enterprise Linux 4 is not compatible with Fast Models. To compile models on Red Hat Enterprise Linux 4, a compiler that is compatible with the official GCC 4.1.2 release from the Free Software Foundation is required. Please contact ARM if you are unable to source such a compiler.

Table B-1 lists package dependencies for Red Hat Enterprise Linux 4. Some of these are installed as part of a base installation.

Table B-1 Dependencies for Red Hat Enterprise Linux 4

Package	Required for
`glibc`	Fast Models Tools and virtual platforms
`libgcc`	Fast Models Tools and virtual platforms
`libstdc++`	Fast Models Tools and virtual platforms
`zlib`	Fast Models Tools and virtual platforms
`XFree86-libs`	Fast Models Tools and virtual platforms
`make`	Fast Models Tools only
`alsa-lib`	Fast Models virtual platforms only

B.3 Dependencies for Red Hat Enterprise Linux 5

ARM supports the use of the GCC 4.1 release packaged with Red Hat Enterprise Linux 5. ARM does not support the use of the Red Hat preview release of GCC 4.4 packaged with Red Hat Enterprise Linux 5. To use GCC 4.4 under Red Hat Enterprise Linux 5, a compiler that is compatible with the official GCC 4.4.4 release from the Free Software Foundation is required. Please contact ARM If you are unable to source such a compiler.

Table B-2 lists package dependencies for Red Hat Enterprise Linux 5. Some of these are installed as part of a base installation.

Table B-2 Dependencies for Red Hat Enterprise Linux 5

Package	Required for
glibc	Fast Models Tools and virtual platforms
libgcc	Fast Models Tools and virtual platforms
libstdc++	Fast Models Tools and virtual platforms
zlib	Fast Models Tools and virtual platforms
libICE	Fast Models Tools and virtual platforms
libSM	Fast Models Tools and virtual platforms
libX11	Fast Models Tools and virtual platforms
libXau	Fast Models Tools and virtual platforms
libXcursor	Fast Models Tools and virtual platforms
libXdmcp	Fast Models Tools and virtual platforms
libXext	Fast Models Tools and virtual platforms
libXfixes	Fast Models Tools and virtual platforms
libXrender	Fast Models Tools and virtual platforms
esound	Fast Models Tools and virtual platforms
make	Fast Models Tools only
gcc	Fast Models Tools only
gcc-c++	Fast Models Tools only
alsa-lib	Fast Models virtual platforms only
audiofile	Fast Models virtual platforms only

B.4 Dependencies for Red Hat Enterprise Linux 6

ARM supports the use of the GCC 4.4 release packaged with Red Hat Enterprise Linux 6.

Table B-3 lists package dependencies for Red Hat Enterprise Linux 6. Some of these are installed as part of a base installation.

Table B-3 Dependencies for Red Hat Enterprise Linux 6

Package	Required for
`glibc`	Fast Models Tools and virtual platforms
`glibc-devel`	Fast Models Tools only
`libgcc`	Fast Models Tools and virtual platforms
`make`	Fast Models Tools only
`libstdc++`	Fast Models Tools and virtual platforms
`libstdc++-devel`	Fast Models Tools only
`zlib`	Fast Models Tools and virtual platforms
`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 only
`xterm`	Fast Models virtual platforms only
`telnet`	Fast Models virtual platforms only

The 64-bit version of Red Hat Enterprise Linux does not install any 32-bit compatibility libraries. The Fast Models tools and models compiled for 32-bit targets require these libraries to be installed, so if required these additional libraries must be installed on 64-bit versions of Red Hat Enterprise Linux. See Table B-4.

Table B-4 32-bit libraries for use on 64-bit versions of RHEL

Package	Required for
glibc.i686	Fast Models Tools and virtual platforms
glibc-devel.i686	Fast Models Tools only
libgcc.i686	Fast Models Tools and virtual platforms
libstdc++.i686	Fast Models Tools and virtual platforms
libstdc++-devel.i686	Fast Models Tools only
zlib.i686	Fast Models Tools and virtual platforms
libXext.i686	Fast Models Tools and virtual platforms
libX11.i686	Fast Models Tools and virtual platforms
libXau.i686	Fast Models Tools and virtual platforms
libxcb.i686	Fast Models Tools and virtual platforms
libSM.i686	Fast Models Tools and virtual platforms
libICE.i686	Fast Models Tools and virtual platforms
libuuid.i686	Fast Models Tools and virtual platforms
libXcursor.i686	Fast Models Tools and virtual platforms
libXfixes.i686	Fast Models Tools and virtual platforms
libXrender.i686	Fast Models Tools and virtual platforms
libXft.i686	Fast Models Tools and virtual platforms
libXrandr.i686	Fast Models Tools and virtual platforms
libXt.i686	Fast Models Tools and virtual platforms
alsa-lib.i686	Fast Models virtual platforms only

Appendix C
Building System Models in Batch Mode

This appendix describes the *Simulation Generator* (simgen). It contains the following sections:

C.1 Introduction

The *Simulation Generator* (simgen) is a utility that generates a system model from a project.

Start simgen by either:

- pressing the **Build** icon in System Canvas to build the current project
- typing `simgen` on a command line and specifying the project options.

To display help for the command line options, start simgen with no parameters or specify help:

```
simgen --help
```

`simgen` accepts a system description as input and generates source code for a simulator as output.

The typical usage for building a platform simulator is:

```
simgen -p PROJECTFILE_NAME.sgproj --configuration CONFIGURATION_NAME -b
```

C.2 Simgen command-line options

The command-line options are listed in Table C-1:

Table C-1 Simgen command-line options

Short form	Option	Description
-	`--configuration` *NAME*	Specify the name for the configuration.
-	`--enable-warning` *NUM*	Enable warning number *NUM*. This overrides the `--warning-level` *LEVEL* option.
-	`--disable-warning` *NUM*	Disable warning number *NUM*. This overrides the `--warning-level` *LEVEL* option.
-	`--warnings-as-errors`	Treat LISA parsing and compiler warnings as errors.
-	`--num-comps-file` *NUM*	Specify the number of components that are generated into a single file. See *Using command-line options to decrease compilation time*.
-	`--num-build-cpus` *NUM*	Specify the number of processors used during build.
`-b`	`--build`	Build targets.
`-C`	`--clean`	Clean targets.
`-D`	`--define` *SYMBOL*	Define preprocessor symbol *SYMBOL*. You can also use *SYMBOL=DEF*
-	`--gcc-path` *PATH*	Under Linux, select the GCC C++ compiler that builds the model. Passes the full path of the chosen g++ executable to the Simulator Generator. Ensure this GCC version matches the GCC version in the model configuration. By default, simgen uses the g++ found in the search path.
`-h`	`--help`	Print help message that contains this list of command-line options and exit.
`-I`	`--include` *INC_PATH*	Add include path *INC_PATH*.
`-p`	`--project-file` *FILENAME*	Set simgen project file *FILENAME*.

`-v`	`--verbose` *ARG*	Set verbosity. *ARG* can be one of :
		`on`
		`sparse` (this is the default)
		`off`

`-w`	`--warning-level` *LEVEL*	Set warning level to *LEVEL*.

C.2.1 Using command-line options to decrease compilation time

Previously, all components were placed into a single file. Creating multiple files, however, can reduce the compilation time if only one component has changed and requires recompilation.

You can use the `num-comps-file` option to specify whether a single file or multiple files is used for components. The default is 1 and all components go into a single file.

If the value for `num-comps-file` is increased to, for example, 10 then the first 10 components go into the first file, the second 10 components go into the second file, and so on until all components are in files.

For development work where the source is modified frequently, setting `num-comps-file` to 1 can considerably reduce compilation time because only the file containing the changed module is recompiled. Alternatively, if the changes are such that all files must often be recompiled, then a high setting of, for example, 1000 reduces the compilation time. The trade-off is between little source code in many modules (1) or a lot of source in just one module (1000).

Note

If you are recompiling an existing project that is not to be modified, ARM recommends setting the value for `num-comps-file` to 1.

C.2.2 Interaction between num-comps-file and num-build-cpus

There is, however, an interaction between the `num-comps-file` and the `num-build-cpus` options. The value for `num-comps-file` indicates the maximum number of components per file.

If, for example, there are 40 components, `num-comps-file` is set to 25, and `num-build-cpus` is set to 4, the following action results:

- Four `.cpp` files are produced because there are four processors.
- Each `.cpp` file has 10 components instead of the maximum value of 25.

  Using only the `num-comps-file` option to calculate the components per file would result in only two `.cpp` files with one file containing 25 components and the second file containing 15. This split would only use two processors. A lower value is therefore used for the actual number of components per file to enable all four processors to be used.

C.2.3 Setting command-line parameters from System Canvas

To change the command-lines options from System Canvas:

1. Select **Preferences** from the **File** menu..
2. Add the command-line parameters in the **Command arguments** text box in the **Simulation Generator Executable** panel.
3. These settings are used if the **Build** toolbar button in System Canvas is clicked.