LISA+ Language for Fast Models

Version 5.0

Reference Manual

Copyright © 2007-2009 ARM Limited. All rights reserved.
ARM DUI 0372G (ID052609)

LISA+ Language for Fast Models
Reference Manual

Copyright © 2007-2009 ARM Limited. All rights reserved.

Release Information

Change history

Description	Issue	Confidentiality	Change
August 2007	A	Confidential	New document for System Generator LISA. Based on existing Core Generator LISA document.
February 2008	B	Confidential	Minor updates for release 3.2.
June 2008	C	Confidential	Updates for System Generator 4.0.
August 2008	D	Confidential	Updates for System Generator 4.0 SP1.
December 2008	E	Confidential	Updates for Fast Models 4.1.
March 2009	F	Non-Confidential	Updates for Fast Models 4.2.
April 2009	G	Non-Confidential	Updates for Fast Models 5.0.

Proprietary Notice

Words and logos marked with™ or ® are registered trademarks or trademarks owned by ARM Limited, except as otherwise stated below in this proprietary notice. Other brands and names mentioned herein may be the trademarks of their respective owners.

Neither the whole nor any part of the information contained in, or the product described in, this document may be adapted or reproduced in any material form except with the prior written permission of the copyright holder.

The product described in this document is subject to continuous developments and improvements. All particulars of the product and its use contained in this document are given by ARM in good faith. However, all warranties implied or expressed, including but not limited to implied warranties of merchantability, or fitness for purpose, are excluded.

This document is intended only to assist the reader in the use of the product. ARM Limited shall not be liable for any loss or damage arising from the use of any information in this document, or any error or omission in such information, or any incorrect use of the product.

Where the term ARM is used it means "ARM or any of its subsidiaries as appropriate".

Confidentiality Status

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

Unrestricted Access is an ARM internal classification.

Product Status

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

Web Address

`http://www.arm.com`

Copyright © 2007-2009 ARM Limited. All rights reserved.
Non-Confidential, Unrestricted Access

Copyright © 2007-2009 ARM Limited. All rights reserved.
Non-Confidential, Unrestricted Access

Contents
LISA+ Language for Fast Models Reference Manual

Preface

About this book x
Feedback xiii

Chapter 1 Introduction

1.1 General overview 1-2

Chapter 2 Components

2.1 About components 2-2
2.2 Resources section 2-6
2.3 Includes section 2-28
2.4 Composition section 2-29
2.5 Behavior sections 2-33
2.6 Port declarations 2-40
2.7 Connection section 2-44
2.8 Properties section 2-52
2.9 Debug section 2-54

Chapter 3 Communication with C++ Code

3.1 Accessing C++ constructs from LISA+ 3-2

Copyright © 2007-2009 ARM Limited. All rights reserved.

Unrestricted Access
Non-Confidential

3.2 Calling LISA+ behaviors from C++ code 3-6
3.3 Importing third party models 3-10

Chapter 4 Protocols
4.1 About protocols 4-2
4.2 Includes section 4-3
4.3 Properties section 4-4
4.4 Behavior prototypes 4-6

Appendix A Preprocessor
A.1 Scope A-2
A.2 Predefined symbols and macros A-5
A.3 Preprocessor statements A-7

 Copyright © 2007-2009 ARM Limited. All rights reserved.
Non-Confidential

List of Tables
LISA+ Language for Fast Models Reference Manual

	Change history	ii
	LISA+ terminology	xi
Table 2-1	Optional parameters for registers	2-7
Table 2-2	Debugger register access functions	2-13
Table 2-3	Parameters for memory, bus, and address space	2-18
Table 2-4	PARAMETER parameters	2-25
Table 2-5	Multiplier suffixes for integer properties and parameters	2-27
Table 2-6	Property values	2-52
Table 4-1	Property values	4-4

Copyright © 2007-2009 ARM Limited. All rights reserved.

Unrestricted Access

Non-Confidential

Copyright © 2007-2009 ARM Limited. All rights reserved.
Non-Confidential

Preface

This preface introduces the *LISA+ Language for Fast Models Reference Manual*. It contains the following sections:

- *About this book* on page x
- *Feedback* on page xiii.

Copyright © 2007-2009 ARM Limited. All rights reserved.
Non-Confidential, Unrestricted Access

About this book

This book describes the LISA+ language as used by the Fast Model Tools such as System Generator and System Canvas.

System Generator and System Canvas are part of the Fast Models Tools family of design and simulation products.

Intended audience

This book has been written for experienced hardware and software developers to aid using the LISA+ language to develop component models for the Fast Models environment. This language features described in this book are specifically targeted for component and system models and are used to generate fast simulators and other software development tools.

Organization

This book is organized into the following chapters:

Chapter 1 *Introduction*

Refer to this chapter for an introduction to the LISA+ language.

Chapter 2 *Components*

This chapter describes the LISA+ language functionality that is specific to creating components in System Canvas.

Chapter 3 *Communication with C++ Code*

This chapter describes how to call custom C++ code from LISA+ behavior code.

Chapter 4 *Protocols*

This chapter describes how to define protocols for communication between ports.

Appendix A *Preprocessor*

This appendix describes the preprocessor functionality.

Typographical conventions

The typographical conventions are:

italic Highlights important notes, introduces special terminology, denotes internal cross-references, and citations.

Copyright © 2007-2009 ARM Limited. All rights reserved.
Non-Confidential, Unrestricted Access

bold	Highlights interface elements, such as menu names. Denotes signal names. Also used for terms in descriptive lists, where appropriate.
`monospace`	Denotes text that you can enter at the keyboard, such as commands, file and program names, and source code.
`monospace` (mono underlined)	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>`

Terminology

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

LISA+ terminology

Term	Definition
Behavior	Each LISA+ component or protocol can have multiple behavior sections. These sections describe the behavior code in C.
Component	An individual sub-system element such as core, memory, bus, or peripheral, or a complete system or sub-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.
Code translation	Instruction set simulation technology. Functional accuracy and execution speed are key performance criteria.
CT core	A model of an ARM processor that makes use of code translation technology. CT core models translate ARM instructions on the fly and cache the translation to enable fast execution of ARM code.
External port	A port that is used to connect the subsystem to other components within a higher-level system.

Copyright © 2007-2009 ARM Limited. All rights reserved.
Non-Confidential, Unrestricted Access

LISA+ terminology (continued)

Term	Definition
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.
Protocol	A protocol defines ports in components that use the protocol to communicate with other components. Ports must use the same protocol if they are to be connected.
Resource	A section that allows private C/C++ variables, such as registers, to be declared within a component. These variables can be exposed if required.

Further reading

This section lists related publications by ARM®.

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

ARM publications

This book contains information that is specific to using the LISA+ language with Fast Model Tools. See the following documents for other relevant information:

- *Fast Model Tools User Guide* (ARM DUI 0370)
- *Model Debugger for Fast Models User Guide* (ARM DUI 0314)
- *Cycle Accurate Debug Interface Developer Guide* (ARM DUI 0444).

Copyright © 2007-2009 ARM Limited. All rights reserved.
Non-Confidential, Unrestricted Access

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 if appropriate.

Feedback on this book

If you have any comments on this book, send an e-mail to `errata@arm.com`. Give:

- the title
- the number
- the relevant page number(s) to which your comments apply
- a concise explanation of your comments.

ARM also welcomes general suggestions for additions and improvements.

Copyright © 2007-2009 ARM Limited. All rights reserved.
Non-Confidential, Unrestricted Access

Copyright © 2007-2009 ARM Limited. All rights reserved.
Non-Confidential, Unrestricted Access

Chapter 1
Introduction

This chapter introduces the LISA+ language and the enhancements present in LISA+. It contains the following section:

- *General overview* on page 1-2.

Copyright © 2007-2009 ARM Limited. All rights reserved.
Non-Confidential, Unrestricted Access

1.1 General overview

The *Language for Instruction Set Architectures* (LISA) language is specifically targeted for the description of instruction set architectures. LISA+ is an enhanced version of LISA that can be used to describe components and systems.

System Canvas and the LISA+ language provides a development environment for development of peripheral components or system designs. This system produces the following benefits:

- early software development
- hardware and software co-design
- dramatically shortened system exploration turnaround time
- highly accurate and non-ambiguous system specification
- maintainable system design.

1.1.1 Design methodology

System Canvas uses systems and components to create new library objects or executables.

Processor model development based on the Fast Models tools generation follows the design flow shown in Figure 1-1.

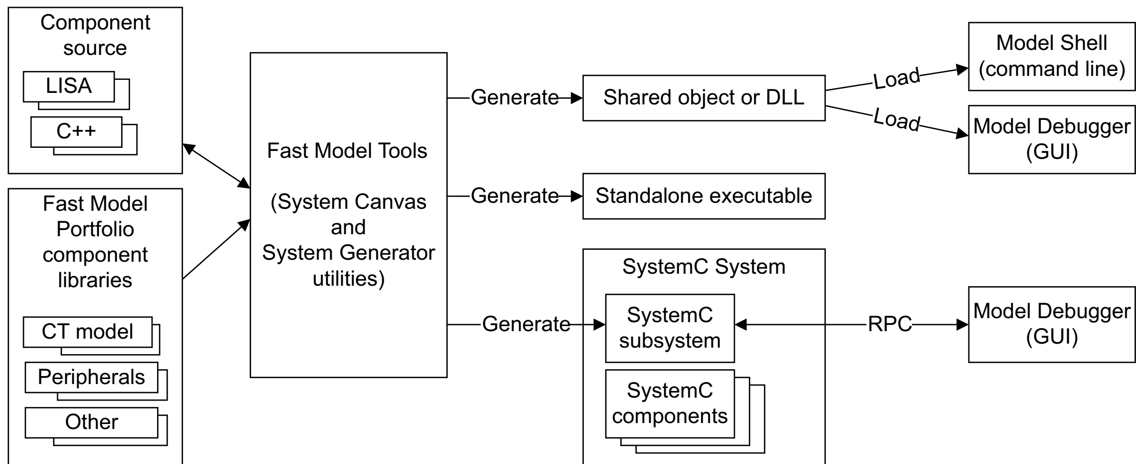

Figure 1-1 Fast Models Tools design flow

LISA+ source code for model components or systems is translated into C++ source code and compiled into library objects.

Copyright © 2007-2009 ARM Limited. All rights reserved.
Non-Confidential, Unrestricted Access

Chapter 2
Components

This chapter describes the sections within the component declaration. It has the following sections:

- *About components* on page 2-2
- *Resources section* on page 2-6
- *Includes section* on page 2-28
- *Composition section* on page 2-29
- *Behavior sections* on page 2-33
- *Port declarations* on page 2-40
- *Connection section* on page 2-44
- *Properties section* on page 2-52
- *Debug section* on page 2-54.

Copyright © 2007-2009 ARM Limited. All rights reserved.
Non-Confidential, Unrestricted Access

2.1 About components

A *component* is the fundamental LISA+ construct that is used to describe components and systems.

Components can have subcomponents and form a hierarchy. The top level component of a system, meaning the component that does not have any parent component, is sometimes also referred to as a *system*. There is, however, nothing special about a system component or top level component and they are declared in the same way as any other component.

Systems that have external ports can be used as a component in a higher-level system. The term *system* is sometimes used to distinguish a collection of connected components that does not include any external ports.

Components definitions can contain the following sections:

- resources (see *Resources section* on page 2-6)
- includes (see *Includes section* on page 2-28)
- composition (see *Composition section* on page 2-29)
- behavior (see *Behavior sections* on page 2-33)
- port (see *Port declarations* on page 2-40)
- connection (see *Connection section* on page 2-44)
- properties (see *Properties section* on page 2-52)
- debug (see *Debug section* on page 2-54).

Note

Except for the behavior and port sections, there cannot be more than one occurrence of each section type in a component.

Component names and keywords are case sensitive.

A component declaration uses the `component` keyword and can contain any of the sections listed in any order.

Example 2-1 shows an example of a component declaration:

Example 2-1 Component declaration

```
component MyComponent
{
    includes
    {
        // ...
```


Copyright © 2007-2009 ARM Limited. All rights reserved.
Non-Confidential, Unrestricted Access

```
    }

    resources
    {
        // ...
    }

    internal port<MyProtocol> port0
    {
        // ...
    }

    behavior init
    {
        // ...
    }
}
```

2.1.1 Integer types and state variables

The LISA+ language uses C/C++ code to describe the behavior and the state variables of a component.

Although native C integer types like `int` and `char` can be used in the description, it is often desirable to use integer types that have a defined bitwidth that is independent of the host architecture.

In LISA, the following integer types are always defined:

`uint8_t` 8 bit unsigned integer value

`int8_t` 8 bit signed integer value

`uint16_t` 16 bit unsigned integer value

`int16_t` 16 bit signed integer value

`uint32_t` 32 bit unsigned integer value

`int32_t` 32 bit signed integer value

`uint64_t` 64 bit unsigned integer value

`int64_t` 64 bit signed integer value.

Using these types is efficient because they do not impose any overhead over native C data types.

Copyright © 2007-2009 ARM Limited. All rights reserved.
Non-Confidential, Unrestricted Access

2.1.2 Using the message() function for debugging

Printing output to a window or to the console is often useful for debugging components. The LISA+ language features the built-in `message()` function that prints messages to the output window of the environment in which the simulation is executing.

Messages are forwarded through `CADISimulationCallback::simMessage()`.

Note

Message handling does not work in the `terminate()` simulation phase because the callback has already been disconnected, see *Displaying messages if no CADISimulationCallback is available* on page 2-5.

Messages are system wide and are forwarded without instance names. To indicate the originator of the message, prefix the message with the string returned by:

- `getInstanceName()` to include the name of the component in its parent component.
- `getInstancePath()` to include the component instance name from the top component. The top component name is not included. If this is called for a top component, an empty string is returned.

The `message()` function has C++ and C style prototypes.

C++ prototype

```
message(const std::string &msg, MessageType type);
```

is the C++ style prototype where:

`msg` is the message to display.

`type` characterizes the purpose or nature of the message.

The following symbols have been introduced to the LISA+ language and one of them can be passed as the type parameter:

MSG_FATAL_ERROR:

signals a fatal error. The error message is printed in the output window preceded with the text `FATAL ERROR`.

If the simulation is running, it is stopped.

If the simulation is in the `init()` or `reset()` phase, the simulation is prevented from running.

MSG_ERROR

indicates an error in the simulation. The message is displayed in the output window preceded by the text `ERROR`.

Copyright © 2007-2009 ARM Limited. All rights reserved.
Non-Confidential, Unrestricted Access

MSG_WARNING

indicates that the message is a warning. The message string is printed in the output window preceded by the text `WARNING`.

MSG_INFO

indicates that the message string is simply printed in the output window.

MSG_DEBUG

If a message is classified as being a debug message, it is only printed if the debug version of build is used. The message is preceded with the text `DEBUG`.

C prototype

```
message(MessageType type, const char *fmt, ...);
```

is the C-style prototype with a variable parameter list where:

`type` indicates the error type and has the same options as for the C++ prototype.

`fmt` is a format specification string. The options are the same as those used with the `printf()` family of functions.

An example of displaying an information message is:

```
message(MSG_INFO, "%s - caused this message \n", getInstanceName().cstr());
```

Displaying messages if no CADISimulationCallback is available

If a `CADISimulationCallback` is not available, use the `doString()` function.

If a `CADISimulationCallback` is available, but it does not accept messages, use `printf()`.

You can also use `printf()` to display messages that occur during `terminate()`.

Copyright © 2007-2009 ARM Limited. All rights reserved.
Non-Confidential, Unrestricted Access

2.2 Resources section

The resources section adds local definitions to the respective component:

- variable declarations that use the C/C++ syntax
- *annotated* resources that use the LISA+ syntax and parameters.

2.2.1 Plain C/C++ variable declarations

Plain C/C++ variables can be declared in the resource section in the same way as member variables of a C++ class. They usually contain the hidden state of the component. The state is hidden because this state is not visible in a debugger connected to this component.

If it is desirable to see certain state variables in the debugger, these state variables must be annotated with `REGISTER` keywords. See *Registers* on page 2-7.

2.2.2 Annotated resources

The following annotated resources can be defined in the resources section:

- `REGISTER` (see *Registers* on page 2-7)
- `MEMORY` (see *Memory* on page 2-17)
- `PARAMETER` (see *Parameters* on page 2-25).

The resource annotation has the following forms for defining single instances or arrays:

```
<resource_class>  [ <parameters> ]  identifier  ";"
<resource_class>  [ <parameters> ]  identifier  "[" <size> "]"  ";"
```


Copyright © 2007-2009 ARM Limited. All rights reserved.
Non-Confidential, Unrestricted Access

2.2.3 Registers

Use the REGISTER keyword to specify registers in the resources section. Registers are resources that store data and a bit-width and type can be specified for the register. The default data type is unsigned int and the default bit width is 32. Optional parameters that can be given to registers are listed in Table 2-1.

Table 2-1 Optional parameters for registers

Parameter	Type	Default	Description
address	integer	none	An address parameter is required to map memory-mapped register accesses. The address specified is translated into the unique register ID.
attribute	access type	read_write	Access type as one of: • read_write • read_only • write_only.
bitwidth	integer	32	Data bit-width. Admissible values are: • 8, 16, 32, and 64 for integer types. • 32 and 64 for floating point types • 1 for boolean types • the bitwidth is ignored for string registers.
description	string	""	Description of the resource
display_float_format	string	"%g"	Printf() format string for debugger display for floating point registers, only used for type(float) and display_format(float).
display_format	string	"hex"	Default display format for debuggers, supported formats are: hex, uint, int, bool, float, and string. Debuggers can always override this setting.
display_symbols	string-list	-	Comma-separated list of strings replacing the numerical display. Not all tools implement this feature.
dwarf_id	integer	-	Dwarf register ID. If not set, the register does not have a DWARF register ID. these are typically defined by the architecture ABI.
groups	string-list	-	List of register groups that the register is assigned to. Groups are separated by commas.
has_side_effects	boolean	false	Set to true if register access has side effects.
is_program_counter	boolean	false	Set to true if the resource is the program counter.

Copyright © 2007-2009 ARM Limited. All rights reserved.
Non-Confidential, Unrestricted Access

Table 2-1 Optional parameters for registers (continued)

Parameter	Type	Default	Description
`lsb_offset`(*Bit*)	-	-	Specifies the bit offset in the parent register. See *Component registers* on page 2-12.
`name`	string	resource name	Register name to display in, for example, a debugger. The default is: • the resource variable name for non-array resources, for example "`R`" for register R. • the resource variable name followed by the decimal index for register arrays, for example `R[0]`, `R[1]`, … `R[7]` for register array `R[8]`. The name string can contain a single `printf()` integer specifier for register arrays. For example: `REGISTER { name("R%u") } a[4];` results in registers R0, R1, R2 and R3 in the debugger.
`name_index_base`	integer	0	For register arrays with format specifiers in the name, such as `name("R%u")`, this parameter specifies the start index. For example, `name("R%u")` and `name_index_base(3)` sets the first register array element to R3.
`partof`(*ParentReg*)	-	-	Specifies the parent register. See *Component registers* on page 2-12.
`pv_port`	integer	`none`	Specifies the internal pv_port used to map read and write access to peripheral registers. The port must be a slave port of type `PVDevice`. If only one unique port of type `PVDevice` exists, this parameter can be omitted.
`read_behavior`	string	none	Specifies a user-defined read behavior to use if the automated mechanisms are not sufficient. Use of this behavior depends on the internal state.
`read_function`	string	none	Name of the debug read access behavior.
`read_mask`	integer	none	Specifies the value used for read accesses.
`read_sec_mask`	integer	none	Specifies the value used for read accesses. For secure accesses, this value overwrites other masks.
`reg_number`	integer	auto	CADI register ID. The value of `reg_number` can be any 32-bit unsigned integer constant except for the reserved value `0xFFFFFFFF`.
`reg_number_increment`	integer	1	Specifies the `reg_number` increment between array elements, starting with `reg_number`. Applies only to register arrays and if `reg_number` is specified.

Copyright © 2007-2009 ARM Limited. All rights reserved.
Non-Confidential, Unrestricted Access

Table 2-1 Optional parameters for registers (continued)

Parameter	Type	Default	Description
reset_value	<type>	0, 0.0, "", or false	Reset value. The value specified by the parameter is assigned to the register at initialization and reset execution. If the parameter is omitted, the register is reset with: • zero value for numeric type registers • false for boolean type registers • "" for string type registers. To avoid register initialization on reset, you must use UNINITIALIZED as the parameter argument. Admissible types are bool, int, uint, float, and string.
type	<type>	uint	Data type. Admissible types are bool, int, uint, float, and string.
virtual	boolean	false	Optimizes code generation by preventing the allocation of host memory for the resource. If virtual is true, no variable is generated for the register or memory, the resource must have read and write access functions, and must not be referenced in LISA+ code. By default, all registers and memory are not virtual. virtual does not apply to parameters. See Example 2-3 on page 2-10.
visible_in_debugger	boolean	true	Debug switch. Set to true to show the register in debugger or set to false to hide the register.
write_behavior	string	none	Specifies a user-defined write behavior to use if the automated mechanisms are not sufficient. Use of this behavior depends on the internal state.
write_function	string	none	Name of the debug write access behavior. **Note** The access functions must match the arguments and return type.
write_mask	integer	none	Specifies the value used for write accesses.
write_sec_mask	integer	none	Specifies the value used for write accesses. For secure accesses, this value overwrites other masks.

A register set R consisting of 32 registers that are 32 bits wide and a register file using the default types is specified as shown in Example 2-2 on page 2-10:

Copyright © 2007-2009 ARM Limited. All rights reserved.
Non-Confidential, Unrestricted Access

Example 2-2 Register set specification

```
resources
{
    REGISTER { bitwidth(32) } R[32];
    int a, b; // not visible in a debugger
    REGISTER { is_program_counter(true) } pc;
    REGISTER gpr[32];
    REGISTER { bitwidth(64), type(int) } accu;
    REGISTER { type(float) } fpr[16];
}
```

IDs in register arrays

If a register array is assigned an ID, the first register receives the value of the ID. Each subsequent register in the array is assigned the ID of the previous register incremented by one. For example:

```
    REGISTER { reg_number(5) } R[32];
```

Register `R[0]` is assigned ID 5, `R[1]` is assigned ID 6, `R[2]` is assigned ID 7, and so forth. You can use the `reg_number_increment` parameter to step between registers in an array, starting with `reg_number`. If, for example, `reg_number_increment` was set to 2 in the earlier example, `R[1]` would not be used.

Virtual registers

Example 2-3 shows how to use the `virtual` parameter. The parameter enables the abstract definition of a resource in much the same as C++ permits the definition of pure virtual functions. If virtual functions are used, the C++ programmer of the derived class is forced to implement them. Similarly, the LISA+ programmer is forced to implement a read and a write access behavior and not rely on the existence of a variable that has the resource name.

Example 2-3 Use of virtual parameter

```
component foo
{
    resources
    {
        REGISTER { type(uint32_t), virtual(true), read_function(GetStatus),
           write_function(SetStatus) } STATUS;
        REGISTER { type(uint32_t) } ENABLED;
        REGISTER { type(uint32_t) } ACTIVE;
```


Copyright © 2007-2009 ARM Limited. All rights reserved.
Non-Confidential, Unrestricted Access

```
    }

    behavior GetStatus(uint32_t id, uint64_t *data, bool doSideEffects) :
       AccessFuncResult
    {
        *data = ENABLED & ACTIVE;
        return ACCESS_FUNC_OK;
    }

    behavior SetStatus(uint32_t id, const uint64_t *data, bool doSideEffects) :
       AccessFuncResult
    {
        return ACCESS_FUNC_OK;
    }
}
```

In Example 2-3 on page 2-10, component `foo` has registers STATUS, `ENABLED` and `ACTIVE`:

- `STATUS` is visible to the debugger as a value that is the bit-wise AND operation of `ACTIVE` and `ENABLED`.
- There is no reason to access `STATUS` from LISA+ code, so the design is enforced.
- There is also no reason to write a value to `STATUS`, so using `SetStatus` simply returns the success flag.

Register access

Registers are accessed in the same way as C variables. See Example 2-4.

Example 2-4 Register assignment

```
behavior
{
    R[4] = a;
    R[b] = gpr[a];
}
```


Copyright © 2007-2009 ARM Limited. All rights reserved.
Non-Confidential, Unrestricted Access

Component registers

LISA+ supports registers that are embedded within other registers. That is, they are a component, or child, of a parent register. The implementation of component registers requires that:

- Component registers must be wholly embedded in their parent register. Each bit of the child register is found in the parent. Each component register therefore has exactly one parent.
- Component registers must have the same bit sequence as the parent. The bits of the component register are in the same order as the corresponding bits of the parent register. The bit sequence can be shifted in position, but cannot be split or manipulated in any other way.

The specification of the component register can be fully defined by the following parameters:

`partof(`*`Parent`*`)`

> *`Parent`* is the name of the parent register. This parameter is represented in LISA+ by the `partof` resource attribute. The value specified in `partof` is the name of the parent register.

`lsb_offset(`*`Bit`*`)`

> *`Bit`* is the *Least Significant Bit* (LSB) offset of the child in the parent register. This parameter is represented by the `lsb_offset` attribute that contains the LSB offset, in bits, of the child in the parent register. Specifying the LSB offset is optional and the default value is zero

The registers behave like an integer of the size specified in the bitwidth attribute. However, all modifications on a child or parent register affect the value of the corresponding parent or child registers. This is handled automatically. A child register can also be a parent. This enables component register relationships to extend to component register hierarchies. The consistency of the hierarchies is enforced automatically.

A simple example of a component register hierarchy is shown in Example 2-5 on page 2-13:

Copyright © 2007-2009 ARM Limited. All rights reserved.
Non-Confidential, Unrestricted Access

Example 2-5 Component register of a parent register

```
REGISTER { bitwidth(64) } RAX;
REGISTER { bitwidth(32), partof(RAX) } EAX;
REGISTER { bitwidth(16), partof(EAX) } AX;
REGISTER { bitwidth(8), partof(AX) } AL;
REGISTER { bitwidth(8), partof(AX), lsb_offset(8) } AH;
```

Debugger register access functions

The default functions that a debugger calls to access registers can be overridden by using behaviors that conforms to a specific prototype. The prototypes for these functions are listed in Table 2-2:

Table 2-2 Debugger register access functions

Function	Prototype
Register read function[a]	behavior <name>(uint32_t reg_id, uint64_t *data, bool side_effects) : AccessFuncResult
Register write function[b]	behavior <name>(uint32_t reg_id, const uint64_t *data, bool side_effects) : AccessFuncResult
String register read function	behavior <name>(uint32_t reg_id, string &data, bool side_effects) : AccessFuncResult
String register write function	behavior <name>(uint32_t reg_id, const string &data, bool side_effects) : AccessFuncResult

a. Refer to the CADIRegRead function description. See also the *Model Debugger for Fast Models User Guide*.

b. Refer to the CADIRegWrite function description. See also the *Model Debugger for Fast Models User Guide*.

Where:

reg_id holds the register ID of the register that is being accessed, that is, the argument used in the reg_number attribute. You can modify the array index step size by using reg_number_increment if you have a register array.

For each register with name *name*, a constant REGISTER_ID_*name* is generated. An array register has one id for the base and one for each index. The index ID is formed by appending an "_" and the index of the array entry.

Copyright © 2007-2009 ARM Limited. All rights reserved.
Non-Confidential, Unrestricted Access

`data` is a buffer in which read access functions must export to and write access functions must import from:

- If the bitwidth is less than or equal to 64, the data pointer points to a single 64 bit quantity.
- For larger registers, the data pointer points to an array of `uint64_t` that holds the entire register value. `data[0]` contains the least significant bits and so on in this case.

Note

The write access function prototypes declare the data parameter as `const`.

A separate pair of prototypes exist specifically for registers of type `string`. To simplify use, the data parameter is also of type string for the `string` access functions.

`side_effects` is a parameter indicating whether side effects of the access are enforced.

The `side_effects` parameter specifies whether a read or write involving a register or memory invokes specific side effects associated with that particular register or memory. The semantics of the `side_effects` parameter is slightly different for `read_function` and `write_function`. For `read_function`, the semantics for `side_effects` are:

`side_effects == false`

The read must only return the value of the register and not cause any other side effects. The debugger calls the function with `side_effects == false` to display the value of the register.

`side_effects == true`

The read returns the value of the register and causes side effects that are associated with reading the register. Invoking side effects while reading a register is not common. The debugger only calls `read_function` with `side_effects == true` if the user explicitly wants to trigger side effects.

For `write_function`, the semantics for `side_effects` are:

`side_effects == false`

The function might or might not cause side effects, depending on what the component can handle. Some side effects are required even if `side_effects == false` to keep the component in a consistent state. The only side effects invoked are those required to retain consistency.

Copyright © 2007-2009 ARM Limited. All rights reserved.
Non-Confidential, Unrestricted Access

The side effects are highly dependent on the modeled hardware. For example, if writing to a `SIZE` register adjusts the `ENDPTR` register, update the value of `SIZE` must reasonably also cause the side effect of updating `ENDPTR`. For this example, the `side_effects` parameter must be ignored for writes.

`side_effects == true`

The function causes all side effects that a normal bus write would cause. Invoking side effects for register writes is the most common use case.

Register and memory access functions must inform the calling code whether or not the access operation was successful. The following symbols have been added to the LISA+ language for this purpose:

`ACCESS_FUNC_OK`

The call was successful.

`ACCESS_FUNC_GeneralError`

This indicates an error that cannot be sufficiently explained by one of the other error return values.

`ACCESS_FUNC_UnknownCommand`

The command is not recognized.

`ACCESS_FUNC_IllegalArgument`

At least one of the argument values is illegal.

`ACCESS_FUNC_CmdNotSupported`

The command is recognized but not supported.

`ACCESS_FUNC_ArgNotSupported`

An argument to the command is recognized but not supported. For example, the target does not support a particular type of complex breakpoint.

`ACCESS_FUNC_InsufficientResources`

Not enough memory or other resources exist to fulfil the command.

`ACCESS_FUNC_TargetNotResponding`

A time out has occurred across the CADI interface and the target did not respond to the command.

Copyright © 2007-2009 ARM Limited. All rights reserved.
Non-Confidential, Unrestricted Access

`ACCESS_FUNC_TargetBusy`

The target received a request, but is unable to process the command. The call can be attempted again after some time.

`ACCESS_FUNC_BufferSize`

Buffer too small, for `char*` types.

`ACCESS_FUNC_SecurityViolation`

Request was not fulfilled because of a security violation.

`ACCESS_FUNC_PermissionDenied`

Request was not fulfilled because permission was denied.

In Example 2-6, registers R1 and R2 are assigned CADI ids and a shared read access function. The access function returns one's complement for R1 and two's complement for R2. If the access function is assigned to a different register, it reports an illegal argument.

Example 2-6 Read access function

```
resources
{
    REGISTER { read_function(my_read), reg_number(1) } R1;
    REGISTER { read_function(my_read), reg_number(2) } R2;
}

behavior my_read(uint32_t id, uint64_t *data, bool se) : AccessFuncResult
{
    if (id == 1)
        *data = ~R1;
    else if (id == 2)
        *data = ~R2 + 1;
    else
        return ACCESS_FUNC_IllegalArgument;

    return ACCESS_FUNC_OK;
}
```

Memory-mapped register access

PV peripherals implement memory-mapped register by connecting an external slave port of type `PVBus` to an internal slave port of type PVDevice.

Copyright © 2007-2009 ARM Limited. All rights reserved.
Non-Confidential, Unrestricted Access

Register IDs are specified explicitly and have the same value as the address offset. The register accesses are implemented by an internal read/write behavior. Memory-mapped port accesses and debug accesses are directed to this behavior.

If a unique internal port of type `PVDevice` exists, all memory-mapped register read and write operations over this port are directed to the automatically generated access functions.

If not already overwritten by `read_function` and `write_function` parameters the new access functions are also used for debug accesses.

Masks are used to influence the register access:

- There are masks for read and write operations.
- If the device distinguishes between secure and non-secure accesses, an additional set of read and write masks can be provided for secure accesses.
- If the mask parameter, is omitted full access is permitted.
- A zero mask ignores the access but returns a complete response.

Two additional tokens can be used to generate error responses:

- `ABORT` for an abort error response
- `DECODEABORT` for a decode error response.

If the automatic mechanisms are not sufficient, you can provide a local implementation that overrides the access behaviors. The arguments for the access behaviors are:

```
register_read_behavior(uint32_t reg_id, pv::ReadTransaction tx) :
    pv::Tx_Result
register_write_behavior(uint32_t reg_id, pv::WriteTransaction tx) :
    pv::Tx_Result
```

Each register resource has an ID that must be used in the read and write behaviors. See *Debugger register access functions* on page 2-13.

Note

Using memory-mapped register access features requires the Fast Models include files.

2.2.4 Memory

Memory resources are C-array-like constructs that make their contents visible to a debugger. They are always declared using the array syntax where the size of the array is the size of the memory.

Copyright © 2007-2009 ARM Limited. All rights reserved.
Non-Confidential, Unrestricted Access

The parameters listed in Table 2-3 can be given to memories:

Table 2-3 Parameters for memory, bus, and address space

Parameter	Type	Default	Description
`allow_unaligned_access`	boolean	`false`	Allow unaligned access. If this is true then accesses that are not naturally aligned, such as a 32-bit access on a non32-bit boundary, are allowed and have the expected result. If this is false such unaligned accesses are not allowed.
`attribute`	access type	`read_write`	Access type as one of: • `read_write` • `read_only` • `write_only`.
`description`	string	`""`	Description of the space
`endianness`	big or little	`little`	Select between little and big endianness.
`executable`	boolean	`false`	This flag identifies memory blocks that can hold executable code.
`mau_size`	integer	8	Size of the *Minimum Addressable Unit* (MAU) in bits. Admissible values are 8, 16, 32, and 64.
`paged`	boolean	`true`	Allow use of true paged memory. This means memory is not allocated completely at instantiation time, but instead in pages on demand. This typically results in lower overall memory usage but leads to slower memory access. If the array size is > `0x10000`, paging is enforced for the memory and the parameter is ignored.
`read_function`	string	none	Name of the debug read access behavior.
`space_id`	integer	-1	Specifies the memory space id. If not defined, `space_id` is automatically generated.

Copyright © 2007-2009 ARM Limited. All rights reserved.
Non-Confidential, Unrestricted Access

Table 2-3 Parameters for memory, bus, and address space (continued)

Parameter	Type	Default	Description
`supported_multiples_of_mau`	string	1	Permitted multiples of MAU for memory accesses. Multiple values must be separated by commas.
`virtual`	boolean	`false`	Optimizes code generation by avoiding the allocation of host memory for the resource. The resource must have read and write access functions and must not be referenced in LISA+ code. If this attribute is set to true, no variable is generated for this parameter. The read and write functions are responsible for providing and storing the value. The parameter is virtual because it is modeled through the read and write functions.
`write_function`	string	none	Name of the debug write access behavior.

The array size can be specified using:

- pure integer values
- Suffixes for Kilo, Mega, Giga, Tera, or Peta, for example, 1K, 1M, 1G, 1T, or 1P relative to the `mau_size`. These suffixes indicate multipliers of 2^{10}, 2^{20}, 2^{30}, 2^{40}, and 2^{50}.
- expressions, for example 2k-1.

Example 2-7 Resources definition

```
resources
{
    MEMORY { mau_size(8) } progmem[64k];
    MEMORY { mau_size(32) } datamem[128k-100];
}
```


Copyright © 2007-2009 ARM Limited. All rights reserved.
Non-Confidential, Unrestricted Access

Read and write accesses

Memory resources can be accessed using C-array like syntax as shown in Example 2-9:

Example 2-8 C array memory accesses

```
behavior load(uint32_t address, uint8_t &data)
{
    data = dmem[address];
}

behavior store(uint32_t address, uint8_t data)
{
    dmem[address] = data;
}
```

The size of the memory access is always one *Minimal Addressable Unit* (MAU). A 32-bit unsigned integer, for example, is the access size for a memory with a `mau_size` of 32.

The access functions listed in Example 2-9 can also be used to access memory:

Example 2-9 Read and write accesses

```
resource.read8( uint32_t address, uint8_t & destination );
resource.read16( uint32_t address, uint16_t & destination );
resource.read32( uint32_t address, uint32_t & destination );
resource.read64( uint32_t address, uint64_t & destination );
resource.write8( uint32_t address, uint8_t source );
resource.write16( uint32_t address, uint16_t source );
resource.write32( uint32_t address, uint32_t source );
resource.write64( uint32_t address, uint64_t source );
```

These access functions can be used to read or write 8, 16, 32 and 64-bit quantities from or to memory. The size of the access is implied by the function name. Only functions with a bitwidth greater than or equal to the `mau_size` of the memory can be used on a memory. If the bitwidth of the access is greater than the `mau_size` the result depends on the endianness of the memory. See Example 2-10 on page 2-21.

Copyright © 2007-2009 ARM Limited. All rights reserved.
Non-Confidential, Unrestricted Access

Example 2-10 Mixed bitwidth accesses

```
behavior load(uint32_t address, uint8_t &data)
{
        dmem.read8(address, data);
}

behavior store(uint32_t address, uint8_t data)
{
        dmem.write8(address, data);
}
```

Debugger memory read and write access functions

The access functions for memory are similar to the access functions for registers. You can control the way in which a memory resource is accessed by the debugger by defining your own debugger access functions. A maximum of one read-access function and one write-access function can be specified in the resources section.

Access functions are implemented as LISA+ behaviors in the same component that holds the memory resource. They are designated as access functions by the `read_function` and `write_function` resource parameters. Refer to the `CADIMemRead` and `CADIMemWrite` function descriptions for more information. See the *Model Debugger for Fast Models User Guide*.

```
resources
{
    MEMORY { mau_size(8), read_function(my_read) } progmem[64];
    MEMORY { mau_size(32), write_function(my_write) } datamem[64];
}
```

The access functions must conform to the following prototypes:

Read function

```
behavior read_function_name (uint32_t space_id,
                uint32_t block_id,
                uint64_t offset,
                uint32_t size_in_maus,
                uint64_t *data,
                bool side_effects,
                sg::MemoryAccessContext *mac) : AccessFuncResult
```

Write function

```
behavior write_function_name (uint32_t space_id,
                uint32_t block_id,
                uint64_t offset,
```


Copyright © 2007-2009 ARM Limited. All rights reserved.
Non-Confidential, Unrestricted Access

```
uint32_t size_in_maus,
const uint64_t *data,
bool side_effects,
sg::MemoryAccessContext *mac) : AccessFuncResult
```

where:

`space_id` is an integer value that is a unique identifier for a memory space.

`block_id` is an integer value that, for the specified memory space, is a unique identifier for a memory block.

`offset` is an absolute numerical offset into the space and block denoted by the `space_id` and `block_id` parameters. It designates the starting address for the memory access.

`size_in_maus` is the size of the access relative to the size of the MAU. A memory access might involve reading or writing multiple MAU quantities.

`data` is the buffer from which data is read or to which data is written. Its type is a pointer to a 64-bit unsigned integer. This size is equal to the largest MAU size currently supported and effectively eliminates endianness concerns. In the implementation, `data` is actually an array of `size_in_maus` size. Write functions protect this array by declaring the data pointer `const`.

`side_effects` is a parameter that indicates whether the side effects for the access must be enforced. The use of this parameter with memory reads and writes is completely analogous to its use with registers. See *Debugger register access functions* on page 2-13.

`MemoryAccessContext`

is a pointer to a `MemoryAccessContext` object. This provides extensibility to the prototype and the MemoryAccessContext class can be enriched if required. There are currently the following interfaces:

`GetAccessSizeInMaus()`

returns how many MAUs of memory area have to be read/written

`GetMauInBytes()`

returns the size of a MAU, measured in bytes

`GetMauInBits()`

returns the size of a MAU, measured in bits

Copyright © 2007-2009 ARM Limited. All rights reserved.
Non-Confidential, Unrestricted Access

Note

The reason for accessing these values though `MemoryAccessContext` rather than by coding them as constants is to lower the maintenance hazard that would arise if a memory attribute changes.

Example 2-11 shows an example of access functions:

Example 2-11 Read and write access functions

```
behavior my_read(uint32_t space_id,
                 uint32_t block_id,
                 uint64_t offset,
                 uint32_t size_in_maus,
                 uint64_t *data,
                 bool side_effects,
                 MemoryAccessContext *mac) : AccessFuncResult
{
    *data = progmem[offset];
    return ACCESS_FUNC_OK;
}

behavior my_write(uint32_t space_id,
                  uint32_t block_id,
                  uint64_t offset,
                  uint32_t size_in_maus,
                  const uint64_t *data,
                  bool side_effects,
                  MemoryAccessContext *mac) : AccessFuncResult
{
    datamem[offset] = (uint32_t) *data;
    return ACCESS_FUNC_OK;
}
```

In Example 2-12 on page 2-24, two memory resources, `m1` and `m2`, are declared and a read access function, `my_read`, is provided for them. It is possible for multiple memory spaces and blocks to share an access function because the distinction between them can be made at runtime by means of the `space_id` and `block_id` parameters. For Example 2-12 on page 2-24, this is accomplished with a simple `if` / `else` sequence.

After determining the resource to be accessed, data is copied into the data buffer. The for loop runs for `size_in_maus` times, copying one MAU quantity on each iteration and checking whether the accesses are within bounds or not.

Copyright © 2007-2009 ARM Limited. All rights reserved.
Non-Confidential, Unrestricted Access

Example 2-12 Implementation of a read access function

```
resources
{
    MEMORY { space_id(1), mau_size(8), read_function(my_read) } m1[64];
    MEMORY { space_id(2), mau_size(32), read_function(my_read) } m2[64];
}

behavior my_read(uint32_t space_id,
                 uint32_t block_id,
                 uint64_t offset,
                 uint32_t size_in_maus,
                 uint64_t *data,
                 bool side_effects,
                 MemoryAccessContext *mac) : AccessFuncResult
{
    if (space_id == 1)
    {
        for (int i = 0; (i < size_in_maus) && (offset + i < 64); ++i)
            data[i] = m1[offset + i];
    }
    else if (space_id == 2)
    {
        for (int i = 0; i < (size_in_maus) && (offset + i < 64); ++i)
            data[i] = m2[offset + i];
    }
    else
        return ACCESS_FUNC_IllegalArgument;

    return ACCESS_FUNC_OK;
}
```

Note

If you set the `virtual` resource parameter to `true`, you prevent memory from being allocated at run time. A virtual memory resource must provide debug read and write functions and cannot be directly accessed from LISA+ source code. If you do not define valid read and write access functions, or attempt to access the resource through LISA+ code, a build failure occurs.

Copyright © 2007-2009 ARM Limited. All rights reserved.
Non-Confidential, Unrestricted Access

2.2.5 Parameters

Parameters allow component configuration and parameterization at initialization time or run time of the system model.

Table 2-4 lists the parameters that can be given to the `PARAMETER` keyword. Array syntax is not supported for parameters.

Table 2-4 PARAMETER parameters

Parameter	Type	Default	Description
`default`	int or string	0 or ""	Default value is 0 for integers and the empty string for string parameters.
`description`	string	""	Plain text single line description of the parameter. The debugger might display this string next to the parameter to provide additional information about a parameter to the user.
`max`	int	`0x7FFFFFFFFFFFFFFF`	Maximum admissible value.
`min`	int	`0x8000000000000000`	Minimum admissible value. **Note** The minimum value (and therefore the maximum range) for a parameter depends on whether it is signed or unsigned. The maximum ranges are: • `[0x0, 0x7FFFFFFFFFFFFFFF]` for unsigned parameters • `[0x8000000000000000, 0x7FFFFFFFFFFFFFFF]` for signed parameters.
`name`	string	""	A text tag for the parameter that is displayed in the GUI. Any printable symbol except for # . and = can be used in `name`. Double quote characters within the tag must be escaped with \. If no `name` is specified, the parameter identifier is used.
`type`	int, bool, or string	`int`	Data type. **Note** The type can also be `intx_`*`size`* or `uintx_`*`size`* where *`size`* is one of 8, 16, 32, or 64.

Copyright © 2007-2009 ARM Limited. All rights reserved.
Non-Confidential, Unrestricted Access

Table 2-4 PARAMETER parameters (continued)

Parameter	Type	Default	Description
`read_function`	string	none	Name of the read access behavior.
`write_function`	string	none	Name of the write access behavior.
`runtime`	boolean	`false`	Switch between instantiation-time and run-time parameters. Instantiation-time parameters are set by the user before the system is instantiated and cannot be changed afterwards. This is the default. Run-time parameters can be changed during runtime.

The access functions have similar semantics as for registers. The access function prototypes are as follows:

- integer and bool parameters:

```
behavior my_read(uint32_t id, int64_t *data) : AccessFuncResult
behavior my_write(uint32_t id, const int64_t *data) : AccessFuncResult
```

- string parameters:

```
behavior my_read(uint32_t id, string &data) : AccessFuncResult
behavior my_write(uint32_t id, const string &data) : AccessFuncResult
```

For each parameter with name *`name`*, a constant `PARAMETER_ID_`*`name`* is generated. This is passed as `id` when the `PARAMETER` is read from or written to.

The default behavior for the `read_function` is to return the current value of the `PARAMETER`. The default behavior for the `write_function` is to set the value of the `PARAMETER`. If a `write_function` is specified, the parameter is no longer updated automatically. This update must be done in the write function.

Example 2-13 shows an example of using `PARAMETER` in a `resources` section.

Example 2-13 PARAMETER resource

```
resources
{
    PARAMETER { min(0), max(0xFFFF), default(0x80), name("Base Address")}
      baseAddress;

}
```


Copyright © 2007-2009 ARM Limited. All rights reserved.
Non-Confidential, Unrestricted Access

The parameter in the example would default to being called `baseAddress` if a `name` tag was not declared. When choosing parameter names or tags, you are strongly advised to adhere to the naming rules for C++ identifiers. This means you can use upper and lower case letters, numbers, and underscore characters. Avoid using hyphens, "-" in parameter names or tags. If you are supporting legacy code that uses hyphens in parameter names, you can use these old names within the `name` tag. However, the parameter name outside the braces must conform to C++ naming rules, and is what you must use in your LISA+ code.

Parameters that are integers and are entered in decimal format can have the multiplier suffixes listed in Table 2-5.

Table 2-5 Multiplier suffixes for integer properties and parameters

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

2.2.6 Accessing resources

All resources can be accessed in the component behavior by using the name of the resource.

2.2.7 Obsolete resources constructs

The `resource_mapping` section in the resource section of components is completely replaced by the `connection` section of components. See *Connection section* on page 2-44. The connection section enables using hierarchical systems in a clean way. A `resorce_mapping` section is ignored and results in a warning.

Copyright © 2007-2009 ARM Limited. All rights reserved.
Non-Confidential, Unrestricted Access

2.3 Includes section

The `includes` section is a dedicated place for `#include` preprocessor statements. See Chapter 3 *Communication with C++ Code* and Appendix A *Preprocessor*.

The declarations that result from the `#include` statements are visible in the bodies of the component behaviors. However, the `#include` statements are not expanded into the LISA+ code itself. This means that `#defines` coming from the `#include` statements are not visible in the LISA+ code and the included header files must not contain any LISA+ code.

Declarations in the `includes` sections can be made visible globally to other components. This means that declarations in the `includes` sections might conflict with other declarations in other `includes` sections of other components if they use the same names for different purposes. Therefore ARM recommends using only declarations in the `includes` section that do not conflict with other components declarations such as header files that are shared between components.

Copyright © 2007-2009 ARM Limited. All rights reserved.
Non-Confidential, Unrestricted Access

2.4 Composition section

The composition section enables hierarchical description of components.

The section lists all subcomponents of a component or system and defines the values of initialization-time and run-time parameters of the contained subcomponents.

Parameters of subcomponents are initialized in the composition section by specifying a comma-separated list of *name*=*value* statements in parentheses following the component type name. The name must be a published name. The name attribute is relevant for published names. See *Parameters* on page 2-25. The value can be either:

- a constant
- the parameter identifier of the enclosing component. In this case, the value of the parameter is forwarded from a component to its subcomponent. The parameter identifier is the identifier of the parameter in the resources section, not its name attribute.

Example 2-14 describes component MyComponent having two sub components mem1 and mem2 that are both of type MyMemory. The expression size=0x1000 sets compile time parameter size of component mem1 to 0x1000.

Example 2-14 Composition section

```
component MyComponent
{
    resources
    {
       PARAMETER mem2size;
    }
    composition
    {
        mem1: MyMemory(size=0x1000);
        mem2: MyMemory(size=mem2size, id=0x7000);
    }
    ...
}
```


Copyright © 2007-2009 ARM Limited. All rights reserved.
Non-Confidential, Unrestricted Access

2.4.1 Overriding component parameter attributes

Parameters of subcomponents that are set in the composition section of their parent component can not be set or configured in the debugger or run-time environment. They are hard coded in the system.

All other parameters can be configured in the debugger or when starting the run-time environment.

In Fast Models 4.1.20 or later however, you can override the `default` value for a parameter and, for integer parameters, the `min` and `max` values.

Overriding is done in the component instantiation statement in the form of assignments and must use the following syntax:

`parameter_name.attribute_name=attribute_value`

If done, the override assignments must be included in the normal parameter assignments in an instantiation statement as shown in Example 2-15

Example 2-15 Parameter overriding

```
component CPU
{
    resources
    {
        PARAMETER { default(0x8a0000) } itcm_size;
        PARAMETER { default(0xda0000), min(0x450000), max(0xff0000) } dtcm_size;
    }
}

component Board
{
    composition
    {
        cpu : CPU(itcm_size=0x440000, dtcm_size.default=0xaa0000, dtcm_size.min=0x600000);
    }
}
```

The code in Example 2-15 overrides the default value of `dtcm_size` and its `min` boundary. This means that the parameter is published and the user can set it, but:

- the user cannot specify a value less than `0x60000`
- the parameter is initialized with `0xaa0000` if the user does not specify anything.

Copyright © 2007-2009 ARM Limited. All rights reserved.
Non-Confidential, Unrestricted Access

For `min` and `max` attributes of integer parameters, the global rule also applies to the overridden variants. If the value specified during system instantiation is not within the [*min*, *max*] range, it is truncated to fit.

If a component overrides the `min` or `max` attribute of a parameter in a subcomponent, the new value can only restrict the [*min*, *max*] range:

- `min` can only be overridden with a greater value
- `max` can only be overridden with a smaller value.

If an integer parameter forwards another, whose [*min*, *max*] range is not identical, then the resulting range is the intersection of the restrictions:

- If this intersection is not identical to that of the forwarder, a warning is issued.
- If the intersection is void, an error is issued.

For Example 2-16, the LISA+ parser warns that the range of parameter `b_dtcm_size` is being restricted to `[0x450000, 0xef0000]`.

Example 2-16 Range restriction warning

```
component CPU
{
    resources
    {
        PARAMETER { default(0x8a0000) } itcm_size;
        PARAMETER { default(0xda0000), min(0x450000), max(0xff0000) } dtcm_size;
    }
}

component Board
{
  resources
  {
      PARAMETER { default(0xda0000), min(0x350000), max(0xef0000) } b_dtcm_size;
  }

  composition
  {
      // b_dtcm_size is the forwarder
      cpu : CPU(dtcm_size=b_dtcm_size);
  }
}
```

As is the case with parameter fixing and forwarding, an override assignment must use the *name* attribute of the parameter on the left-hand side, if specified. If the name is not a valid C identifier, it must be enclosed in double quotes. For example:

Copyright © 2007-2009 ARM Limited. All rights reserved.
Non-Confidential, Unrestricted Access

```
cpu : CPU("semihosting-enable".default=true);
```

 Copyright © 2007-2009 ARM Limited. All rights reserved.
Non-Confidential, Unrestricted Access

2.5 Behavior sections

Component descriptions can comprise multiple named behavior sections that specify the behavior of the component model. Behaviors can be considered to be similar to a function.

Behaviors of components always have a name and can also have an optional list of formal parameters and an optional return type as shown in Example 2-17:

Example 2-17 init behavior

```
component MyComponent
{
    // this behavior does not have parameters and has no return type
    behavior init
    {
        // initialize component
    }

    // this behavior has two parameters
    // the syntax is C-like
    behavior setPixel(uint32_t x, uint32_t y)
    {
    }
    // this behavior has a parameter and return a value of type 'bool'
    // the syntax for return types differs from C
    behavior isValidAddress(uint32_t address): bool
    {
        return address < memorySize;
    }
}
```

The C/C++ code in a behavior body can directly access all resources declared in the resource section.

2.5.1 Special-purpose behaviors

The following behaviors names have a specific meaning in components and are called implicitly and automatically in certain simulation phases:

`behavior init`

This behavior is called implicitly once when the simulation is started. The code in this behavior is only executed once. It is intended to allocate memory and other resources used during simulation.

Copyright © 2007-2009 ARM Limited. All rights reserved.
Non-Confidential, Unrestricted Access

`behavior reset(int level)`

This behavior is called whenever the simulation is reset by the user. Reset might be called multiple times. There are two reset levels:

- `MX_RESETLEVEL_SOFT` indicates that all state variables must be reset to their reset value but memory contents must not be cleared.
- `MX_RESETLEVEL_HARD` indicates that all state variables must be reset to their reset values and memories must also be cleared.

Memory must not be allocated in this behavior since it might be invoked several times.

Caution

All registers are normally reset to the values specified in their `reset_value` parameters immediately before `behavior reset` is run.

You can use `reset_value(UNINITIALIZED)` to prevent the register values being overwritten. See Table 2-1 on page 2-7.

`behavior loadApplicationFile(const string& filename)`

This behavior is invoked whenever a user loads an application file into a component using a debugger. The implementation of this behavior must load the application file.

`behavior terminate`

This behavior is the counterpart to `behavior init`. It is called when the simulation terminates. It is intended to free any allocated memory and resources that have been allocated in `behavior init`.

Note

See also *Controlling simulation from behaviors* on page 2-35.

2.5.2 Hierarchical behavior of special-purpose behaviors

The use of special purpose behaviors is optional. If they are missing for a component, the corresponding behaviors of all subcomponents of the component are called recursively. If a special purpose behavior is specified however, it is responsible for explicitly calling the subcomponents. This is easily done using the `composition` keyword as shown in Example 2-18 on page 2-35:

Copyright © 2007-2009 ARM Limited. All rights reserved.
Non-Confidential, Unrestricted Access

Example 2-18 Calling a subcomponent

```
behavior reset(int level)
{
   // reset subcomponents
   composition.reset(level);
   // reset state variables
   status = 0;
   counter = 0;
   control = 0;
}
```

A missing special purpose behavior B is equivalent to:

```
behavior B
{
   composition.B();
}
```

Note

If the `composition` statement is not present in a special purpose behavior, the corresponding behaviors of the subcomponents are never called and this might have undesirable effects.

A missing special purpose behavior is not equivalent to an empty special purpose behavior.

2.5.3 Controlling simulation from behaviors

Note

See also *Special-purpose behaviors* on page 2-33.

Any LISA+ behavior can use the following function calls to detect and control the simulation environment:

`simRun()` Start the simulation. This might be used from, for example, the `gui_callback()` function.

`simHalt()` Stop the simulation. This might be used if, for example, an error is detected.

Copyright © 2007-2009 ARM Limited. All rights reserved.
Non-Confidential, Unrestricted Access

Note

The simulation does not stop immediately. The actual shutdown might be as much as 200 cycles after the call to `simHalt()`.

`simShutdown()`

Stop the simulation and exit.

`simIsRunning()`

Returns true if the simulation is currently running. This might be used from, for example, the `gui_callback()` function.

`cadiRefresh()`

Notifies attached debuggers to refresh their state. The parameter is an int that combines one or more of the `CADI_REFRESH_REASON_X` flags (see the `CADITypes.h` file for details). For example:

```
behavior myBehavior()
{
  // do something that changes model state when in stop mode
  // ...
  cadiRefresh(eslapi::CADI_REFRESH_REASON_REGISTERS
             | eslapi::CADI_REFRESH_REASON_OTHER);
}
```

Caution

Do not call this function from behaviors that can be triggered as a response for refresh such as, for example, a register read. If called from the behavior, an endless loop results.

This function only has effect if the simulation is stopped.

2.5.4 LISA+ language elements in behaviors

Behaviors contain:

- C/C++ code
- additional syntactic constructs that have a special meaning in LISA+ that they do not have in C/C++.

There are the following types of behavior:

Port behaviors

behaviors that are declared inside a port.

Copyright © 2007-2009 ARM Limited. All rights reserved.
Non-Confidential, Unrestricted Access

Component behaviors and local behaviors

behaviors that are not declared inside a port.

Special purpose behaviors

component behaviors that have a special purpose such as init(), reset(), and terminate().

The following types of function call can occur in component and port behavior sections:

- to local behavior such as `foo()`
- to local port behavior such as `myport.foo()`
- to port behavior of a subcomponent such as `subcomp.aport.foo()`
- to special-purpose behaviors of a subcomponent such as `subcomp.init()`
- to special-purpose behaviors of all subcomponents such as `composition.init()`
- to `getInstanceName()` to return the instance name of the component.
- to `getInstancePath()` to return the instance name relative to the top-level component.

An example of using the `composition` keyword to call a special-purpose behavior for all subcomponents recursively is shown in Example 2-19:

Example 2-19 Calling special-purpose behavior

```
Behavior reset(int level)
{
   // reset subcomponents
   composition.reset(level);
   // reset state variables
   status = 0;
   counter = 0;
   control = 0;
}
```

The subcomponents are always called in the order they are declared in the `composition` section. Local behaviors, that is, non-port behaviors in the same component, are called by just using the name of the local behavior.

Subcomponent special-purpose behaviors

The special purpose behaviors of subcomponents can be called by prepending the subcomponent instance name and a dot before the behavior name to be called.

Copyright © 2007-2009 ARM Limited. All rights reserved.
Non-Confidential, Unrestricted Access

However, special purpose behaviors must always be called from the corresponding special purpose behavior in the parent component. Otherwise, the results are undefined.

Calling subcomponent behaviors explicitly enables defining a specific initialization order. See Example 2-20:

Example 2-20 Initialization order

```
component MyComponent
{
    composition
    {
        subcomp0: AnotherComponent
        subcomp1: AnotherComponent
        subcomp2: AnotherComponent
    }

     behavior init
        {
            // explicit initialization order
            subcomp2.init();
            subcomp0.init();
            subcomp1.init();
        }
}
```

Accessing ports

Access local ports (ports of the same component) by using the name of the port.

Access ports of subcomponents by using the syntax:

subcomponent_name.port_name.behavior

An example of calling a subcomponent port behavior is shown in Example 2-21:

Example 2-21 Accessing a subcomponent port

```
component MyComponent
{
    composition { subcomp: AnotherComponent }

     behavior set42_on_subcomp
        {
```


Copyright © 2007-2009 ARM Limited. All rights reserved.
Non-Confidential, Unrestricted Access

```
            subcomp.port0.set(42);
        }
}
```

Accessing the component instance name

The `getInstanceName()` function can be used to get the component instance name of the component. It returns a `std::string`. This is the instance name of the component in its parent component. The system component has no parent so the result is undefined, but most implementations return a generic string like "`system`".

This function is slow and is not recommended for use during simulation of a component that implements normal functionality. It might, however, be useful in component error messages and debugging output as shown in Example 2-22:

Example 2-22 Displaying component error messages

```
component MyComponent
{
     behavior init
        {
            cout << "initializing '" << getInstanceName() << "'" << endl;
        }
}
```

2.5.5 Scope

Component behaviors and port behaviors are both in the scope of the component and can directly access the resources of the component.

Copyright © 2007-2009 ARM Limited. All rights reserved.
Non-Confidential, Unrestricted Access

2.6 Port declarations

Communication between components is done with master and slave ports using *Transaction Level Modeling* (TLM). The ports use standard or user-defined protocols to communicate between components. Read and write accesses are always initiated from master ports.

The cores use this interface style to communicate with the peripherals. It can also be used for communication between peripherals. The communication connections are defined in the `connection` section of each component. See *Connection section* on page 2-44.

This kind of communication encapsulates each component behind an abstract interface. Components can easily be replaced by other components and it is generally easier to modify the structure of a system and to reuse components in other systems.

Components must interact during the simulation and this communication must be based on a defined protocol. LISA+ has the ability to define customized protocols that are tailored to the specific components and offer a clean interface. Ports of components can only be connected if they implement the same protocol.

More details on declaring, defining and using these protocols are given later. See Chapter 4 *Protocols*.

A port declaration has the format:

```
port_attributes port<protocol_name> instanceName[, instanceName2 …];
```

where:

`port_attributes`

can be a combination of the attributes `master`, `slave`, `internal` and `addressable`.

`protocol_name`

is the name of a protocol and can be considered to be a port type.

If a port has behavior that implements one or more protocol functions, the port declaration also has a body containing behavior declarations:

```
port_attributes port<protocol_name> instanceName
{
    behavior f {    }
    behavior g {    }
    // …
}
```


Copyright © 2007-2009 ARM Limited. All rights reserved.
Non-Confidential, Unrestricted Access

2.6.1 Master, slave, and internal ports

Each port has a master and a slave side. Some ports of a component are typically exposed to the system outside of the component as master ports, for example a memory port of a CPU. Such ports have the `master` port attribute.

Some ports are intended to be exposed to the outside system as slave ports, for example an interrupt request port of a CPU. Such ports have the `slave` attribute.

Some ports are only used internally in a component, for example to receive callbacks from a subcomponent. Such ports neither expose their master side nor their slave side. Such ports have the `internal` attribute.

All ports must be either `master` or `slave` or `internal`.

A port cannot be `master` and `slave` at the same time, meaning it can not expose the master and the slave side at the same time to the outside world.

You can add the `master` or `slave` attribute to the `internal` keyword to indicate how the internal port is to be used, but this is not required.

Master ports always only implement the master behaviors of a protocol and slave ports always implement the slave behaviors of a protocol. Because most protocols only have slave behaviors, typically only the slave port has behaviors.

Note

The implementation of protocol behaviors must be done inside the scope of the port declaration. All resources that have been declared within the component scope can be directly accessed.

2.6.2 Addressable ports

Addressable ports. such as bus ports, are declared by placing the keyword `addressable` before the keyword `port`:

```
master addressable port<myProtocol> myPort;
```

Use the `addressable` keyword to automatically create bus decoders for addressable slave ports. The `ADDRESS` keyword indicates the parameter of a protocol behavior that is the address.

Calls to a behavior that has a parameter with the `ADDRESS` keyword select the slave based on the value of the parameter. The code below shows a `read` behavior for a custom protocol:

Copyright © 2007-2009 ARM Limited. All rights reserved.
Non-Confidential, Unrestricted Access

```
protocol myProtocol
{
    slave behavior read(ADDRESS uint32_t addr, uint32_t *data);
}
```

See *ADDRESS arguments* on page 4-9 for more details on the `ADDRESS` keyword.

More information on connecting addressable port arrays is given later. See *Addressable port arrays* on page 2-48.

2.6.3 Port arrays

A LISA+ component can contain arrays of ports such as, for example, multiple interrupt ports in an IRQ controller.

It is useful to declare multiple instances of a port at the same time using the array construct:

```
slave port<MyType> access[2];
```

This declares two slave ports using the protocol `MyType`. In the behavior that implements the protocol `MyType`, an additional parameter of type `unsigned int` is available and denotes the port index. As an example protocol `MyType` has a behavior `read`:

```
protocol MyType
{
    slave behavior read(uint32_t addr, uint32_t &data)
}
```

Example 2-23 shows the parameter list and implementation of a `read` method.

Note

The `portIndex` parameter enables distinguishing between the different ports.

Example 2-23 Implementing read() for a port vector

```
slave port <MyType> access[2]
{
    behavior read( /* additional parameter */ unsigned int portIndex,
                uint32_t addr, uint32_t &data)
    {
        // implementation of read behavior
        if (portIndex == 0)
            // do something for port 0
            ;
        else
```


Copyright © 2007-2009 ARM Limited. All rights reserved.
Non-Confidential, Unrestricted Access

```
            // do something for port 1
            ;
    }
}
```

The length of the port must be a literal number. Expressions and parameter references are not allowed.

More details on connecting port arrays is available. See *Port array connections* on page 2-46.

2.6.4 Internal ports

Internal ports are normal ports that are not accessible from outside of the component. They are not visible in the parent component. They do not form a part of the component interface. They are rather an internal implementation detail of a component.

Internal ports are typically used to handle signals coming from master ports of subcomponents in the parent component.

It is not necessary to declare an internal port just to call port behaviors of a subcomponent. This can be done directly using the syntax: `asubcomponent.aport.abehavior (..)`

Example 2-24 Internal port

```
component MyComponent
{
    internal slave port<MyProtocol>
    {
        // …
    }
}
```


Copyright © 2007-2009 ARM Limited. All rights reserved.
Non-Confidential, Unrestricted Access

2.7 Connection section

The connection section is used to connect component ports with each other.

The scope of the connection section is defined by the composition section. Only the component's own ports and ports of components declared in the composition section can be connected in the connection section.

The connections section contains a list of connection statements. The syntax of a connection statement is shown in Example 2-25. `MAR` and `SAR` are the optional Master Address Range and Slave Address Range. The address range includes both the low address and high address.

`masterComponent` and `slaveComponent` can be the instance name of any subcomponent, as defined in the composition section, or the keyword `self` that stands for the component that contains the connection section. `masterComponent` is always the transaction initiator (master) and `slaveComponent` is the transaction receiver (slave).

Example 2-25 Connection section syntax

```
masterComponent.masterPort[MAR] => slaveComponent.slavePort[SAR];
```

For addressable ports an address range must be specified. See *Addressable ports* on page 2-41. No address range can be specified, however, for non-addressable ports.

The following rules apply for address ranges:

- If a range is only specified for the master, the range of the slave has the same size as the address range of the master and always starts from 0.
- If the address range of the slave is smaller then the address range of the master port, multiple addresses of the master port are linked to the same slave port address.

  If, for example, the master port has range 0 to `0x1FFF` and the slave port range is from 0 to `0xFFF`, the master port addresses `0x0001` and `0x1001` are both linked to address `0x0001` of the slave port.
- Overlapping address ranges are permitted but the order of the connection statements is significant. Later connection statements override earlier connection statements. The first connection statement, therefor, has the lowest priority. The priority of connections simplifies creating a default bus slave that covers the whole address space of the bus.

Example 2-26 on page 2-45 shows how ports are connected in the `connection` section.

Copyright © 2007-2009 ARM Limited. All rights reserved.
Non-Confidential, Unrestricted Access

Example 2-26 Connecting ports

```
component MyComponent
{
    composition
    {
        mem: MyMemory(size=0x1000);
        mem2: MyMemory(size=0x1000);
        otherComp: MyOtherComp;
        probe: MyProbe;
    }
    addressable master port<MyMemProtocol> memport;
    master port<MyOtherProtocol> otherPort;
    connection
    {
        // default bus slave comes first and gets all addresses which
        // are not overridden by the other connection statements
        self.memport[0..0xffffffff] => probe.access;
        // addressable master ports can have address ranges
        self.memport[0..0xfff] => mem.access[0..0xfff];
        // this is equivalent to => mem2.access[0..0xfff]
        self.memport[0x1000..0x1fff] => mem2.access;

        self.otherPort => otherComp.otherPort;
    }}
```

In Example 2-26, the port `memport` is an external port and `probe` is a component. The keyword `self` is used to identify that the external port `memport` is connected to the `access` port of the `probe` component.

2.7.1 Hierarchy in port connections

Components that have subcomponents might require exposing master ports of the subcomponent. This is possible for both slave and master ports.

Example 2-27 assumes a component `parent` with a subcomponent `sub`:

Example 2-27 Component hierarchy

```
component parent
{
    composition { subcomponent: sub }
    master port<masterType> forwardedMaster;
    slave port<slaveType> forwardedSlave;
    connection
```


Copyright © 2007-2009 ARM Limited. All rights reserved.
Non-Confidential, Unrestricted Access

```
    {
        subcomponent.subMaster => self.forwardedMaster;
        self.forwardedSlave => subcomponent.subSlave;
    }
}

component sub
{
    master port<masterType> subMaster;
    slave port<slaveType> subSlave;
}
```

The subcomponent ports `subMaster` and `subSlave` are forwarded and are visible to the outside with the names `forwardedMaster` and `forwardedSlave`. The keyword `self` is used to identify the external ports `forwardedSlave` and `forwardedMaster`.

This might seem to contradict the principle that master ports can only be connected to slave ports and vice versa. Both ports, however, have two sides, a master port also has a slave side where the method is being initiated. The same is true for slave ports that receive some kind of signal and then act as a master within the component.

2.7.2 Port array connections

LISA+ simplifies connecting port arrays by permitting port arrays in connection statements. Each connection statement consists of a left and right-hand side. There are therefore the following combinations:

- single port to single port
- port array to single port
- single port to port array
- port array to port array.

A single port can be either a port declared as single or a single element of a port array. Port arrays are used in connection statements as the array identifier without an index.

Example 2-28 shows the cases:

Example 2-28 Port array connections

```
protocol MyProtocol { /* protocol behaviors */ }

component Foo
{
    master port<MyProtocol> mPortArray[4];
}
```


Copyright © 2007-2009 ARM Limited. All rights reserved.
Non-Confidential, Unrestricted Access

```
component Bar
{
    slave port<MyProtocol> sPort;
    slave port<MyProtocol> sPortArray[4];
}

component MyComponent
{
    composition
    {
        foo : Foo;
        bar : Bar;
    }

    connection
    {
        // single port to single port
        foo.mPortArray[2] => bar.sPort;
        foo.mPortArray[2] => bar.sPortArray[3];

        // single port to port array
        foo.mPortArray[2] => bar.sPortArray;

        // port array to single port
        foo.mPortArray => bar.sPortArray[3];

        // port array to port array
        foo.mPortArray => bar.sPortArray;
    }
}
```

The following rules apply to array connections:

single-to-array connections

The master port is connected to every element of the slave port array.

array-to-single connection

Every element of the master port array is connected to the slave port.

array-to-array connections

Each element of the master port array is connected to the element of the slave port array that has the same index. This mode of connection is only valid for port arrays of equal size. If the arrays or not the same size, an error is issued.

Copyright © 2007-2009 ARM Limited. All rights reserved.
Non-Confidential, Unrestricted Access

2.7.3 Addressable port arrays

Port arrays can also be addressable. The declaration for addressable port arrays is the same as for port arrays except for the addition of the `addressable` keyword. The following declaration is for an array of five addressable master ports, named `pa`, that implement the `MyProtocol` protocol:

```
addressable master port<MyProtocol> pa[5];
```

Note

The `addressable` keyword can also be used with slave port arrays, but it has no effect.

Connection of addressable port arrays

Connecting individual elements of addressable port arrays requires an indexing scheme and specification of an address range. Because port index and port address ranges both appear in square brackets, it is necessary to observe a proper sequence between the two specifiers:

- The port index must be located after the array identifier.

  `m[2]` specifies the third port in port array `m`.

- Address ranges must be located after the port index.

  `m[0][0x0..0xF]` specifies address range `0x0`–`0xF` of the first port.

If you specify a connection in which a port array participates, you must supply an address range after the name of the port array. Example 2-29 shows how addressable port arrays are connected:

Example 2-29 Connection of addressable port arrays

```
protocol MyProtocol { /* protocol behaviors */ }

component Foo
{
addressable master port<MyProtocol> addr_mPortArray[4];
}

component Bar
{
slave port<MyProtocol> sPort;
slave port<MyProtocol> sPortArray[4];
}
```


Copyright © 2007-2009 ARM Limited. All rights reserved.
Non-Confidential, Unrestricted Access

```
component MyComponent
{
    composition
    {
        foo : Foo;
        bar : Bar;
    }

    connection
    {
        // single to single
        foo.addr_mPortArray[2][0x0..0xF] => bar.sPortArray[0x10..0x1F];
        foo.addr_mPortArray[2][0x0..0xF] => bar.sPortArray[3][0x10..0x1F];

        // single to array
        foo.addr_mPortArray[2][0x0..0xF] => bar.sPortArray[0x10..0x1F];

        // array to single
        foo.addr_mPortArray[0x0..0xF] => bar.sPortArray[3][0x10..0x1F];

        // array to array
        foo.addr_mPortArray[0x0..0xF] => bar.sPortArray[0x10..0x1F];
    }
}
```

Note

Addressable port arrays that appear in connection statements are expanded just like non-addressable ones, but with the addition of the address range specification

---

Note

Single-to-array connections are not applicable to addressable port arrays because the overlapping range conflicts are resolved by the order of declaration. This type of connection is always broken down to a sequence of single-to-single connections between the master port and each element of the slave port array, in ascending order. For the example given above, it would be:

```
foo.addr_mPortArray[2][0x0..0xF] => bar.sPortArray[0][0x10..0x1F];
foo.addr_mPortArray[2][0x0..0xF] => bar.sPortArray[1][0x10..0x1F];
foo.addr_mPortArray[2][0x0..0xF] => bar.sPortArray[2][0x10..0x1F];
// last declaration for the [0x0..0xF] range: has priority
foo.addr_mPortArray[2][0x0..0xF] => bar.sPortArray[3][0x10..0x1F];
```

Therefore, a single-to-array connection between addressable ports is equivalent to a single-to-single connection between the master port and the last element of the slave port array.

Copyright © 2007-2009 ARM Limited. All rights reserved.
Non-Confidential, Unrestricted Access

Addressable port arrays implementation

Addressable port arrays combine the features of addressable ports and port arrays. This means that slave methods must add the ADDRESS keyword before the address parameter and the extra unsigned int parameter for the array index:

Example 2-30 Implementation of addressable port arrays

```
protocol SimpleProtocol
{
    slave behavior read(ADDRESS uint32_t address);
}

slave port<SimpleProtocol> s[4]
{
    slave behavior read(unsigned int portIndex, ADDRESS uint32_t address)
    {
        // behavior implementation
    }
}
```

Using addressable port arrays in behaviors

Addressable port arrays are to be used in LISA+ behaviors just like normal port arrays. As is used in the connection section, the requirement to designate specific elements of a port array is managed by the use of indices in square brackets:

Example 2-31 Using addressable ports in behaviors

```
protocol SimpleProtocol
{
    slave behavior read(ADDRESS uint32_t address);
}

component Foo
{
    addressable master port<SimpleProtocol> addr_mPortArray[2];
}

component Bar
{
    slave port<SimpleProtocol> sPortArray[2]
    {
        slave behavior read(unsigned int portIndex, ADDRESS uint32_t address)
```


Copyright © 2007-2009 ARM Limited. All rights reserved.
Non-Confidential, Unrestricted Access

```
        {
            printf("index: %d, address: 0x%x\n", portIndex, address);
        }
    }
}

component Bar
{
    connection
    {
        foo.addr_mPortArray[0x0..0xF] => bar.sPortArray[0];
        foo.addr_mPortArray[0x10..0x1F] => bar.sPortArray[1];
    }

    behavior callRead()
    {
        foo.addr_mPortArray[0].read(0xA);
        foo.addr_mPortArray[1].read(0xE);
        foo.addr_mPortArray[0].read(0x12);
        foo.addr_mPortArray[1].read(0x1C);
    }
}
```

The standard output of behavior `callRead()` is:

```
index: 0, address: 0xaindex: 0, address: 0xeindex: 1, address: 0x12index: 1,
address: 0x1c
```


Copyright © 2007-2009 ARM Limited. All rights reserved.
Non-Confidential, Unrestricted Access

2.8 Properties section

The properties section of a component describes the properties of the component such as, for example, version or component type.

Some properties might only be relevant for a specific tool.

Example 2-32 shows the syntax of the properties section:

Example 2-32 Properties section

```
component MyComponent
{
    ...
    properties
    {
        version = “1.1.1”;
        component_type = “Memory”;
        description = “my component”;
        ...
    }
}
```

Table 2-6 lists the supported properties.

Table 2-6 Property values

Property	Default	Description
component_type	""	A string describing the type of component. This can be "Core", "Bus", "Memory" or any free-form text.
component_name	""	A string containing the name of component.
default_view	auto	Determines what is displayed if a component is opened in System Canvas. This can be either the block diagram (auto) or a source view (source).
description	""	This is a description of the component.
documentation_file	""	Filename or http link for the component documentation. For filenames, the path can be absolute or relative to the LISA+ file for the component. Supported file formats are pdf, txt, and html. Filenames can contain the * and ? wildcards.
executes_software	0 (1 for operation.main)	This Boolean property indicates that the component executes software and that application files can be loaded into this type of component.

Copyright © 2007-2009 ARM Limited. All rights reserved.
Non-Confidential, Unrestricted Access

Table 2-6 Property values (continued)

Property	Default	Description
has_cadi	1	If set to 1 (true), a CADI interface is generated for this component that enables connection of the target with a CADI compliant debugger. If set to 0 no CADI interface is generated for this component.
hidden	0	If set to 1 (true), the component is hidden. A hidden component is not shown in the System Canvas component list and cannot be added to a block diagram.
icon_file	""	File containing component's logo in xpm format. This icon is displayed in the System Canvas block diagram editor. The path is relative to the LISA+ file.
is_single_stepped	0	This property must be set to 0 (false) for systems that use the batch scheduler, that is, for systems that contain CT cores. The property must be set to 1 for simulation-cycle-based systems, that is, systems that must have behavior step called on every simulation cycle. Be aware that such systems are not necessarily cycle-accurate. **Note** Because this is a property of the whole system, this property is only significant for the top-level component of a system.
license_feature	""	License feature string required to run a virtual platform model that contains this component. This string must be present in the license file for the model.
loadfile_extension	""	Application filename extension for this target. Example: ".elf" or ".elf;*.hex"
small_icon_file	""	File containing icon shown in System Canvas list view. The component is displayed as a 12x12 pixel icon. The path is relative to the LISA+ file.
version	"1.0"	This is the version number of the component.

Copyright © 2007-2009 ARM Limited. All rights reserved.
Non-Confidential, Unrestricted Access

2.9 Debug section

The debug section is used to control generation of the debug interface for the component.

By default all components publish CADI interfaces in the CADI factory to enable connecting CADI-compliant debuggers. The CADI interfaces of these components collect the CADI information from their sub-components to enable access to the registers and memory to the hidden components.

The debug section enables control of the published CADI interfaces and the selection subcomponents that the CADI information is imported from.

Example 2-33 shows an example of including a debug section:

Example 2-33 Using a debug section

```
component MyComponent
{
    composition { a:comp_A; b:comp_B; c:comp_C;}
    debug
    {
        composition.publish;    // publish all components CADI ...
        a.unpublish;            // ... but do NOT publish CADI of 'a'
        composition.unimport;   // do NOT import any CADI info ...
        b.import;               // ... but import CADI info of 'b'
    }
}
```

The keywords have the following meaning:

publish The CADI of the subcomponent is published in the CADI factory and a CADI compliant debugger can connect to the component.

unpublish The CADI of the subcomponent is not published in the CADI factory and it is not possible for a CADI compliant debugger to connect to the component.

import The CADI information of the subcomponent is imported into the CADI interface of the component containing the debug section.

unimport The CADI information of the subcomponent is not imported into the CADI interface of the component containing the debug section.

composition All sub components of the component are effected by the statement.

Copyright © 2007-2009 ARM Limited. All rights reserved.
Non-Confidential, Unrestricted Access

The following rules apply:

- `unpublish` and `unimport` also affects all subcomponents of the components. In Example 2-34, the CADI of `aa` and `dd` are not published because the `unpublish` command issued to `aa` causes the CADI of `dd` is not published even though `aa` publishes `dd` in its debug section.
- The top-level component cannot be `unpublished` or `unimported`.

Example 2-34 Using unpublish and publish

```
component MyComponent
{
    composition { aa:comp_A; bb:comp_B; cc:comp_C;}
    debug
    {
        composition.publish; // publish all components CADI ...
        aa.unpublish; // ... but do NOT publish CADI of ‘aa’
    }
}
component Comp_A
{
    composition { dd:comp_D}
    debug
    {
        dd.publish; // publish CADI of component dd
    }
}
```


Copyright © 2007-2009 ARM Limited. All rights reserved.
Non-Confidential, Unrestricted Access

Copyright © 2007-2009 ARM Limited. All rights reserved.
Non-Confidential, Unrestricted Access

Chapter 3
Communication with C++ Code

This chapter describes how to call custom C++ code from LISA+ behavior code and how to call LISA+ behavior code from C++ code.

There are situations that might require calling custom C++ code from LISA+ behavior code, such as, for example to integrate your own existing C++ components into Fast Models systems. Similarly you might require access to LISA+ components from your C++ object. You can also incorporate third party C++ models into your system, even if you do not have access to the source code.

This chapter contains the following sections:

- *Accessing C++ constructs from LISA+* on page 3-2
- *Calling LISA+ behaviors from C++ code* on page 3-6
- *Importing third party models* on page 3-10.

Copyright © 2007-2009 ARM Limited. All rights reserved.
Non-Confidential, Unrestricted Access

3.1 Accessing C++ constructs from LISA+

All C++ constructs declared in a header file that is included in the `includes` section of a component can be used inside all behavior bodies, including ports, of the component without any special syntax.

LISA+ behaviors contain C++ code, so the syntax for accessing C++ functions and types is the same as for normal C++ code.

A commented LISA+ example demonstrating the concepts necessary to access C++ from LISA+ is given at the end of this section. See *LISA+ example* on page 3-5.

3.1.1 Changes required to your source code

To make C++ class declarations and definitions visible in LISA+ behaviors, add `#include` statements, referencing the C++ header files, to the `includes` section of the component. Everything defined in these header files is visible inside the bodies of all behaviors of the component. However, `#defines` defined in these headers are not visible outside of the behavior bodies and cannot affect conditional compilation of LISA+ code. Additional information on the `includes` section is available elsewhere in this document. See Appendix A *Preprocessor*.

Figure 3-1 on page 3-3 shows an example where a C++ component, `MyCPPComponent`, is imported into the LISA+ code for MyComponent. The LISA+ wrapper code references the C++ header for the `MyCPPComponent` model in the `includes` section. To access the C++ object, use pointers. In Figure 3-1 on page 3-3, a pointer, `*mycomp`, is used to access the `reset` function of the MYCPPComponent C++ component.

Copyright © 2007-2009 ARM Limited. All rights reserved.
Non-Confidential, Unrestricted Access

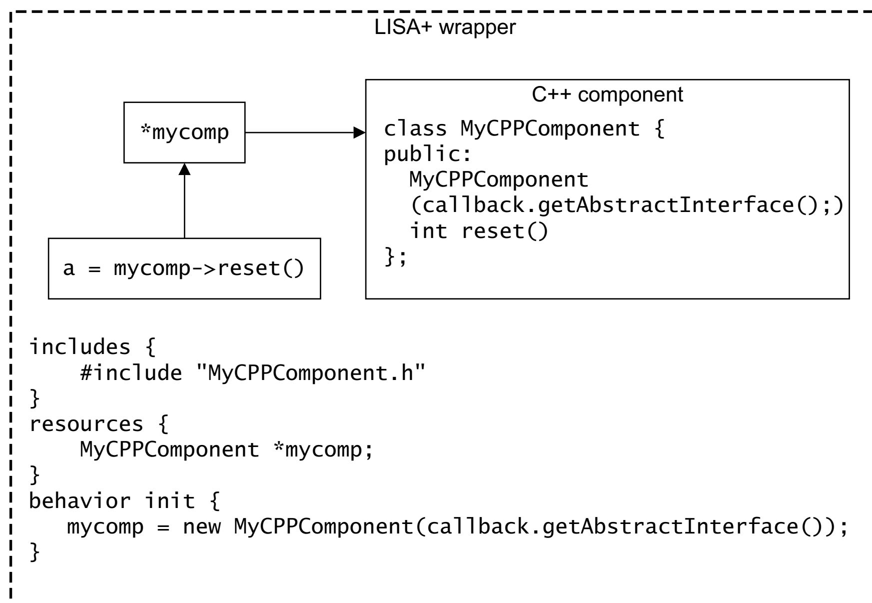

Figure 3-1 Relationship between LISA+ and C++ source

3.1.2 Changes required to your Fast Models project

In System Canvas, you must configure your project so that any imported C++ object header files can be located. Open the Projects Settings dialog shown in Figure 3-2 on page 3-4 by clicking **Project** → **Project Settings**. Add the path to your C++ header files in the **Include Directories** field in the **Compiler** parameter category. For more information on using System Canvas, see the *Fast Model Tools User Guide*.

Copyright © 2007-2009 ARM Limited. All rights reserved.
Non-Confidential, Unrestricted Access

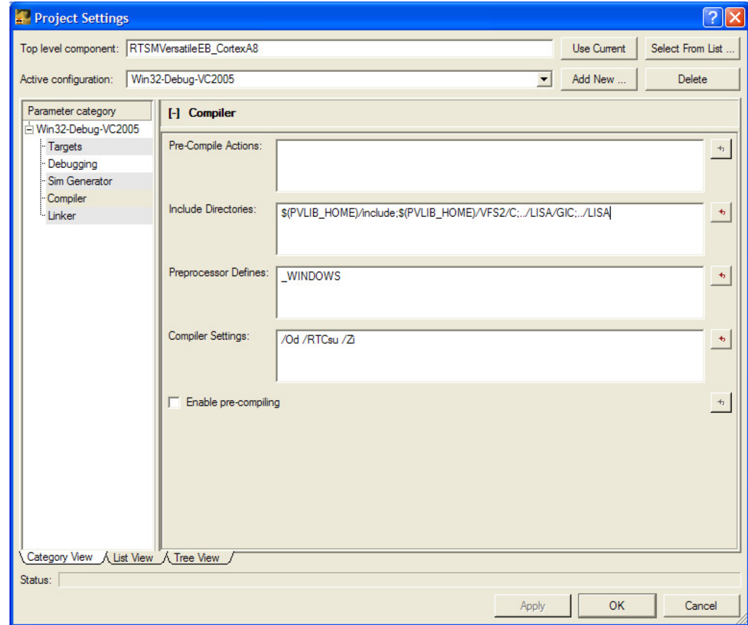

Figure 3-2 System Canvas Project Settings dialog

You must also include a path to the library file that contains your C++ object. Click **Project → Add Files...** to open the Add Files dialog. See Figure 3-3. Change the **File Type** to **Library and Object Files (*.lib; *.obj)** and locate your C++ object. Click **Open** to add the object to your Fast Models project.

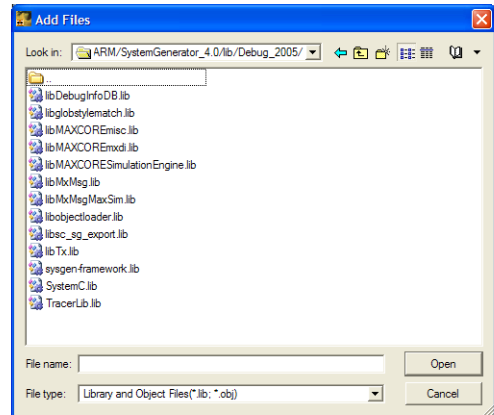

Figure 3-3 System Canvas Add Files dialog

Copyright © 2007-2009 ARM Limited. All rights reserved.
Non-Confidential, Unrestricted Access

3.1.3 LISA+ example

Example 3-1 demonstrates the LISA+ language features used to communicate with C++ code.

Example 3-1 Communicating with C++

```
component MyComponent{
includes
    {
       // make C++ declaration visible in behaviors and resources
        #include "MyCPPComponent.h"
    }
    resources
    {
        MyCPPComponent *mycomp;        // declare a pointer to a C++ class
    }
    // this internal port is used to allow the C++ code to talk to us
    internal port<MyInterface> callback
    {
    behavior signal()
        {
            // this behavior may be called by the C++ code
        }
    }
    behavior init
    {
         // create instance of C++ class. We pass a pointer to the 'callback'
         // port to allow the C++ code to call us back
         mycomp = new MyCPPComponent(callback.getAbstractInterface());
    }
    behavior reset(int level)
    {
        // C++ code can be directly called from LISA+ without any special
        // syntax
        mycomp->reset();
    }
    behavior terminate
    {
            delete mycomp;            // delete instance of C++ class
    }
}
// this protocol is used to allow the C++ code to call the LISA codeprotocol
MyInterface{
    behavior signal();
}
```


Copyright © 2007-2009 ARM Limited. All rights reserved.
Non-Confidential, Unrestricted Access

3.2 Calling LISA+ behaviors from C++ code

In some situations it might be necessary to call back into the LISA+ code from component C++ code. This is achieved by passing a pointer to a pure virtual interface class, called abstract interface, from the LISA+ code to the C++ code in `init`. During simulation, the C++ code can use this pointer to call back into the LISA+ code.

Every port, be it internal, master, or slave, can be called from the C++ code. However, non-port component behaviors cannot be called directly using this approach.

3.2.1 Importing models with callbacks

Callbacks provide a method that allows your C++ object to call LISA+ behaviors. Two conditions must be fulfilled in order for callbacks to work:

- Your LISA+ object must implement the necessary callback functions, through the `getAbstractInterface()` function. See *getAbstractInterface()* on page 3-7.
- The address of the LISA+ object must be passed to the C++ object, using a C++ header file. See *Abstract interface header file* on page 3-8.

A representation of the relationship between the callback function, C++ header file and LISA+ source is shown in Figure 3-4 on page 3-7.

Copyright © 2007-2009 ARM Limited. All rights reserved.
Non-Confidential, Unrestricted Access

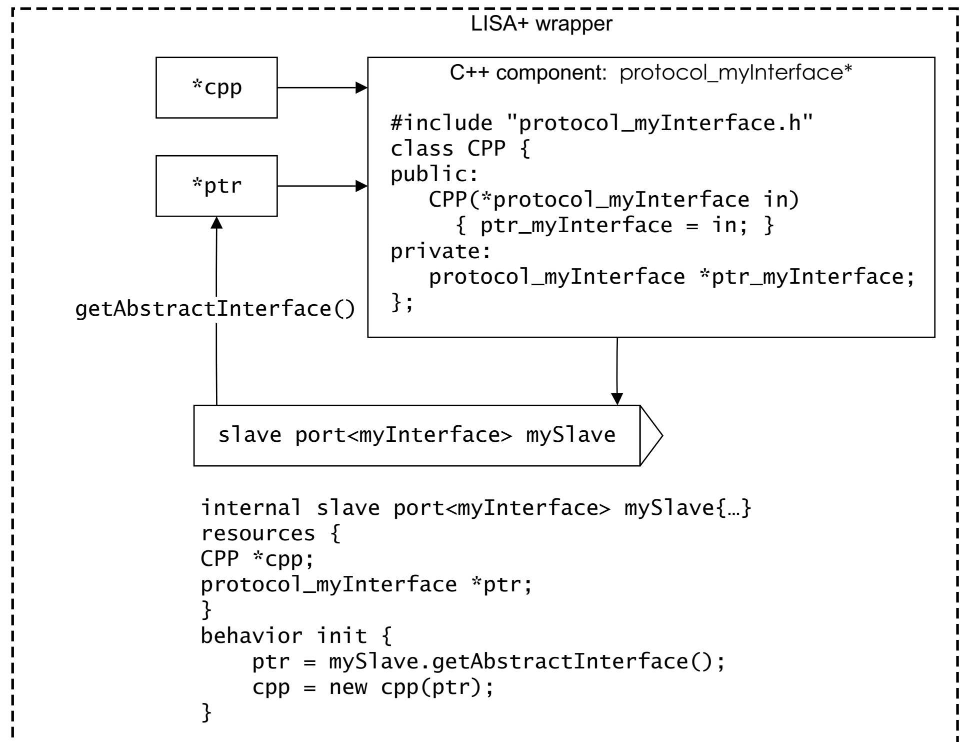

Figure 3-4 Relationship between C++ and LISA+ components in callbacks

3.2.2 getAbstractInterface()

Bridging between C++ and LISA+ components is done by calls through the `getAbstractInterface()` function. Figure 3-4 shows this function in relation to the C++ and LISA+ components. The function is implemented by LISA+ component ports and does the following:

- passes the port address of a LISA+ component to the C++ object
- returns a pointer of the type `protocol_`*`ProtocolName`* to the C++ component.

The LISA+ construct *`portinstance`*`.getAbstractInterface()` can be used in all component behaviors to get a pointer to the abstract interface class instance for a specific port *`portinstance`*. It returns a non-const pointer to a class named `protocol_`*`ProtocolName`*, where *`ProtocolName`* in the name of the protocol of the port. The LISA+ code can pass a pointer to this class to the C++ code upon component initialization and the C++ code can call the LISA+ code back through the abstract interface class and the port behaviors.

Copyright © 2007-2009 ARM Limited. All rights reserved.
Non-Confidential, Unrestricted Access

For the example shown in Figure 3-4 on page 3-7, use the LISA+ construct `mySlave.getAbstractInterface()` to get the pointer `*protocol_myInterface` to the mySlave port.

3.2.3 Abstract interface header file

To call methods of the abstract interface class, the C++ code must know the class declaration of the abstract interface class. This class definition is generated automatically by Simulator Generator into the directory that contains all source files generated by System Canvas. The name of the header file and the name of the abstract interface class is `protocol_ProtocolName.h` where *ProtocolName* is the name of the protocol for this abstract interface. There is one generated header file for each protocol in the system. The header file is always generated, even if it is not used by C++ code. In the example in Figure 3-4 on page 3-7, the C++ component header file is `protocol_myInterface.h`.

The abstract interface class and the header file is generated directly from the protocol definition in the LISA+ file. The interface class contains all behaviors of the protocol (master or slave) as virtual member functions that are in the same order as in the LISA+ protocol definition. Function names, parameters, return type and order are exactly the same in the LISA+ protocol definition and in the generated abstract interface class.

C++ code that is to be compiled independently of the LISA+ code can take a copy of this generated header file. By taking a copy of the header file, both the C++ code and the LISA+ code agree on the interface. This is exactly the same as the interface agreement between two C++ modules.

Whenever the LISA+ protocol definition is changed, the abstract interface class also changes and the interface header file used by the C++ code must also be updated.

If the C++ code includes the abstract interface header file and receives a pointer to such an interface, it can call any of the interface methods. The semantics of such an invocation depend on the type of port the abstract interface belongs to and its implementation.

Invoking a function in the abstract interface typically has the same semantics as if the LISA+ component containing the port invoked the corresponding behavior locally.

If the abstract interface belongs to a slave port, or an internal port that is not connected to any other ports, and the port implements the slave behaviors of the protocol, the C++ code can call all the slave behaviors on the port. This is the most typical use case.

For example, the LISA+ component can use a specific protocol to communicate with the C++ component. It can also give the C++ code access to certain resources by providing access functions for these resources in the protocol. The only purpose of such a protocol would be to communicate with a C++ code.

Copyright © 2007-2009 ARM Limited. All rights reserved.
Non-Confidential, Unrestricted Access

If the abstract interface belongs to a master port that is connected to one or more slave ports, invoking a function on the abstract interface results in calls to all slaves connected to the master port. If no slave is connected to the master port, or if none of the connected slaves implement the behavior being called, the results are undefined.

Copyright © 2007-2009 ARM Limited. All rights reserved.
Non-Confidential, Unrestricted Access

3.3 Importing third party models

You can include third party C++ models in your Fast Models system, without requiring access to the C++ source, provided that:

- you have access to compiled library files for the model
- the component interfaces are known
- the model has a callback interface.

To use a third party model in your system, you might be required to implement your own callback class to act as a bridge between the third party model and your LISA+ system. This is required if your third party model callback interface does not match the LISA+ protocols. See Figure 3-5.

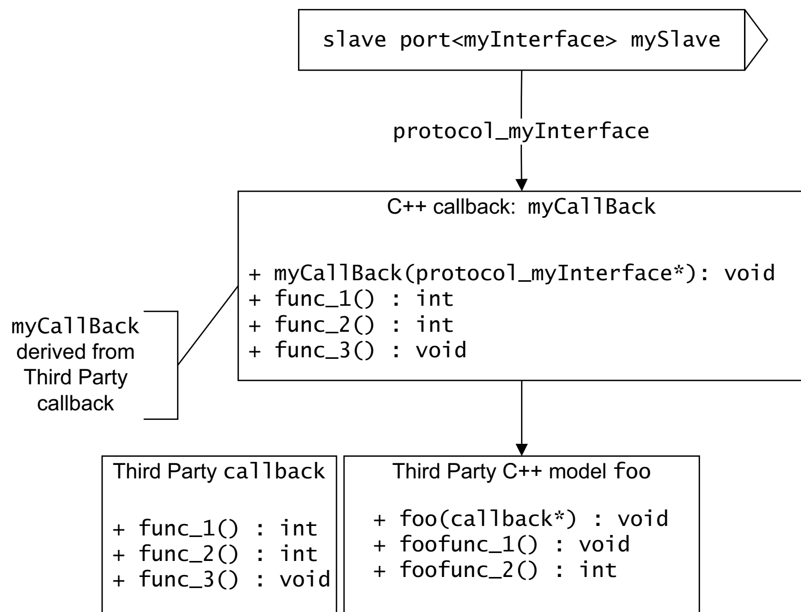

Figure 3-5 Third party model callback relationships

In Figure 3-5, the callback class you implement is called `myCallBack`. This class is derived from the callback class of the third party model and interfaces with your LISA+ protocol. A pointer to the `myCallBack` class is passed to the third party model as shown. The `myCallBack` class communicates with the C++ model using a LISA+ callback, `protocol_myInterface`.

Copyright © 2007-2009 ARM Limited. All rights reserved.
Non-Confidential, Unrestricted Access

Chapter 4
Protocols

This chapter describes the syntax of the protocol section. It has the following sections:

- *About protocols* on page 4-2
- *Includes section* on page 4-3
- *Properties section* on page 4-4
- *Behavior prototypes* on page 4-6.

Copyright © 2007-2009 ARM Limited. All rights reserved.
Non-Confidential, Unrestricted Access

4.1 About protocols

A protocol is defined by using the LISA+ keyword `protocol`:

```
protocol MyProtocol
{
    // protocol declaration
}
```

Protocols are always defined on the top-level in the LISA+ code.

By convention each protocol *`MyProtocol`* is defined in its own LISA+ file `MyProtocol.lisa`.

`MyProtocol` can be any valid C identifier. A protocol is used to define ports in components that use this protocol to communicate with other components, including parent-, sibling- and sub-components. Protocols can be also seen as *port types* as only ports that are using the same protocol can be connected.

A protocol definition can contain:

- an includes section
- a possibly empty list of behavior prototypes.
- a properties section.

Copyright © 2007-2009 ARM Limited. All rights reserved.
Non-Confidential, Unrestricted Access

4.2 Includes section

The `includes` section is an optional section inside the protocol declaration that is used to include C/C++ type definitions and constants that have been defined in external C/C++ header files.

The contents and semantics of the includes-section is the same as for the includes-section of components. It usually contains a list of preprocessor `#include` statements. These `#include` statements are not expanded into the LISA+ code. The `#include` statements are just associated with the protocol definition and the types declared in the included header files can be used in the protocol definition.

The scope of the declarations made directly or indirectly in the `includes` section can extend beyond the protocol definition that contains the `includes` section. The scope can span as far as all LISA+ files of a subsystem, including all other component and protocol definitions, but is only guaranteed to span the protocol definition itself.

Copyright © 2007-2009 ARM Limited. All rights reserved.
Non-Confidential, Unrestricted Access

4.3 Properties section

The `properties` section of a protocol describes properties such as, for example, version or base class.

Some properties might only be relevant for a specific tool. All properties are optional.

Example 4-1 shows the syntax of the properties section:

Example 4-1 Properties section

```
protocol MyProtocol
{
    ...
    properties
    {
        version = “1.1.1”;
        description = “my protocol”;
        ...
    }
}
```

Table 4-1 lists the supported properties:

Table 4-1 Property values

Property	Default	Description
`description`	`""`	This is a description of the protocol.
`version`	`"1.0"`	This is the version number of the protocol.
`documentation_file`	`""`	Filename or http link for the protocol documentation. For filenames, the path can be absolute or relative to the LISA+ file for the protocol. Supported file formats are `pdf`, `txt`, and `html`. Filenames can contain the * and ? wildcards.
`sc_slave_base_class_name`	`""`	Name of the SystemC base class for slave ports (slave exports and sockets for both TLM1 and TLM2).
`sc_slave_export_class_name`	`""`	Name of the SystemC class for slave ports (slave exports for TLM1 only).
`sc_master_port_class_name`	`""`	Name of the SystemC class for master ports (TLM1 only).

Copyright © 2007-2009 ARM Limited. All rights reserved.
Non-Confidential, Unrestricted Access

Table 4-1 Property values (continued)

Property	Default	Description
sc_master_base_class_name	""	Name of the SystemC base class for master ports (master sockets for TLM2 only).
sc_master_socket_class_name	""	Name of the SystemC class for master ports (master sockets for TLM2 only).
sc_slave_socket_class_name	""	Name of the SystemC class for slave ports (master sockets for TLM2 only).

4.3.1 Properties for SystemC export

The properties `sc_slave_base_class_name`, `sc_slave_export_class_name` and `sc_master_port_class_name` are only used if exporting a System Canvas generated system to SystemC.

These properties only appear in protocols that are specifically designed to declare SystemC ports in the top level component of the system that is to be exported to SystemC. They describe the mapping from System Canvas ports to the SystemC port classes. There is an Appendix on SystemC export in a separate book. See the *System Canvas for Fast Models User Guide*.

They are ignored if a port of this protocol is instantiated and is not in the top level component of a system or if no SystemC component is generated.

If a SystemC component is generated but the ports of protocols in the top level component do not have these properties set, the ports are ignored and a warning is issued.

Copyright © 2007-2009 ARM Limited. All rights reserved.
Non-Confidential, Unrestricted Access

4.4 Behavior prototypes

A protocol definition can contain zero or more behavior prototypes. Each behavior prototype declares a behavior of the enclosing protocol. These behavior prototypes are the main part of the protocol definition and define:

- the interface of the protocol
- the interface of the ports and components that use this protocol.

4.4.1 Syntax

The syntax of behavior prototypes is very similar to the syntax of behaviors in components. The main difference is that behavior prototypes:

- can have specific attributes
- are not required to have a body.

```
attributes behavior name[(formal_args)][:return_type];

attributes behavior name[(formal_args)][:return_type]
{
// default implementation. Can be empty
}
```

where:

`attributes` is a combination of `optional`, `master`, and `slave`. There are restrictions. See *Attributes* on page 4-7.

`name` can be any C identifier and is the name of the protocol behavior. Each name can only occur once in the protocol definition. Overloaded behavior prototypes are not allowed.

`formal_args` are the formal arguments of the behavior. The syntax is the same as for C++ function declarations. If user-defined types are used, they must be defined in a header file that is included in the includes-section.

The native C/C++ types and the native LISA+ types can be used without an include statement. Behaviors can optionally mark one argument as an address parameter by placing the `ADDRESS` keyword before the type of the formal argument.

If the argument list is empty, the opening and closing parentheses can be omitted. The names of the formal arguments can also be omitted. Variable number of arguments and default values are not allowed.

`return_type` is the type of the return value of that behavior. If the return type is `void`, it can be omitted. The colon `:` must also be omitted in this case.

Copyright © 2007-2009 ARM Limited. All rights reserved.
Non-Confidential, Unrestricted Access

4.4.2 Attributes

The `attribute` specifier for the behavior must be one of the following:

`master` the behavior must be implemented and is for a master port. A default implementation is not permitted.

`slave` the behavior must be implemented and is for a slave port. A default implementation is not permitted.

`optional master`
the behavior not required to be implemented. If it is implemented, it is for a master port. A default implementation can be provided as part of the prototype definition.

`optional slave`
the behavior not required to be implemented. If it is implemented, it is for a slave port. A default implementation can be provided as part of the prototype definition.

The `attribute` specifier for the behavior in the protocol definition must:

- exactly match the attributes in the port definition section
- be omitted in the port behavior definition.

In the large majority of cases, your protocol behavior is a slave, or optional slave, behavior. Master behaviors are not common.

Note

The presence of default behaviors is part of the protocol definition.

Mandatory behavior

For the following example of a behavior declaration in a protocol:

```
slave behavior f();
```

- Behavior `f()` must be implemented by all slave ports using this protocol.
- Calling `f()` invokes all behaviors `f()` in all slave ports connected to the same port.
- Behavior `f()` might not be implemented in master ports.
- The slave must provide read and write functions to access resources.
- A default implementation cannot be provided because the `optional` keyword is not used. If a default implementation is present, an error is generated.

Copyright © 2007-2009 ARM Limited. All rights reserved.
Non-Confidential, Unrestricted Access

Optional behavior without default implementation

For the following example of an optional behavior declaration in a protocol:

```
optional slave behavior f();
```

- Behavior `f()` might not be implemented by all slave ports that use this protocol. There is no default behavior, so a call to the behavior results in an error.
- Calling `f()` invokes all behaviors `f()` in all slave ports connected to the same port. A master might, for example, notify or query information from all connected slaves, but the handling of this is optional.
- A master must check whether at least one behavior is connected to a behavior in the specified port:

  ```
  if (myport.f.implemented()) myport.f();
  ```

  Note

  The use of the `implemented()` test is only recommended if the default implementation in protocol cannot be used. You are advised to ensure that optional behaviors in new code have default implementations. However, if you are using unmodified legacy code and the called behavior has the `optional` keyword but no default implementation, you must test for implementation.

- Calling a behavior on a port that is not implemented in any of the slaves causes a run-time error.

Optional behavior with default implementation

For the following example of an optional behavior declaration in a protocol:

```
optional slave behavior f()
{
// default implementation. Can be empty
}
```

- Behavior `f()` might not be implemented by all slave ports that use this protocol. If a slave does not implement the behavior, the default implementation is used instead. The default can be {} if no action is required.
- If behavior `f()` is implemented by the port, that implementation is used instead of the default implementation.
- Calling `f()` invokes all behaviors `f()` in all slave ports connected to the same port. A master might, for example, notify or query information from all connected slaves, but the handling of this is optional.

Copyright © 2007-2009 ARM Limited. All rights reserved.
Non-Confidential, Unrestricted Access

- It is not required that a master use the form:

  ```
  if (myport.f.implemented()) myport.f();
  ```

  to test for implementation of the behavior in a port. The default implementation means that `f.implemented()` returns `true` whether or not they have a local implementation.

- If a behavior returns a value, for example,

  ```
  optional slave behavior f() : uint8_t
  {
      // additional code can be present here, but is not required
      return 0;
  }
  ```

  the default implementation can return 0 or any another value that is valid in the context of the calling function.If the return value is undefined, the compiler generates a warning if such warnings are enabled. System Canvas does not issue a warning.

4.4.3 ADDRESS arguments

A protocol is a bus protocol if it enables a single master port to connect to multiple slave ports and the selection of the slave port is determined by the address of the access.

The simulator must inspect the address of the access to determine the destination slave. The address is specified as a parameter of the protocol behavior and must be annotated with the `ADDRESS` keyword as shown in Example 4-2:

Example 4-2 Using the ADDRESS keyword

```
protocol MyBusProtocol
{
    includes
    {
       // declare some user defined types we would like to use
        #include "TransactionMode.h"
   }

   slave behavior read(ADDRESS uint32_t addr): uint8_t;
   slave behavior write(ADDRESS uint32_t addr, uint8_t data);
   slave behavior setMode(const TransactionMode *mode);
}
```


Copyright © 2007-2009 ARM Limited. All rights reserved.
Non-Confidential, Unrestricted Access

The following rules apply to use of the `ADDRESS` keyword:

- The `ADDRESS` attribute can be omitted from the parameter list for the protocol behaviors.
- If present, the `ADDRESS` attribute can only be used with a single parameter of each protocol behavior.
- The `ADDRESS` parameter can be at any position in the argument list.
- `ADDRESS` is not a type specifier. The type of the address must be explicitly specified and an integer type of any size.
- The size of the integer type for each `ADDRESS` parameter must be the same for all of the behaviors in a protocol.
- It is valid to have behaviors with and without an `ADDRESS` parameter in the same protocol:
  - Calls to a behavior that has an `ADDRESS` parameter, `read()` for example, selects the slave based on the address.
  - Calls to a behavior that does not have an `ADDRESS` parameter, `reset()` for example, calls all connected slaves.

 Copyright © 2007-2009 ARM Limited. All rights reserved.
Non-Confidential, Unrestricted Access

Appendix A
Preprocessor

This appendix describes the C-like preprocessor statements that can be used in LISA+ source code. It has the following sections:

- *Scope* on page A-2
- *Predefined symbols and macros* on page A-5
- *Preprocessor statements* on page A-7.

Note

As in C, the preprocessor is applied to the source code before the underlying LISA+ parser is seeing the source code. The preprocessor statements are not part of the formal LISA+ language, but can be seen as a layer on top of the LISA+ language.

Unlike C, however, the LISA+ preprocessor interacts with the actual LISA+ language constructs. Some preprocessor features, such as macro replacement and includes, are deliberately disabled in certain contexts to ensure that tools that read, modify and write LISA+ code work correctly. There are, therefore, subtle restrictions to using preprocessor statement in LISA+ code. These restrictions are described in this appendix.

Copyright © 2007-2009 ARM Limited. All rights reserved.
Non-Confidential, Unrestricted Access

A.1 Scope

LISA+ source code can be split into the following main scopes that are significant for preprocessor processing:

Includes section and resources section code

Includes section and resources section code is all code between, but not including the outermost opening and the closing braces, of `includes` and `resources` sections.

Behavior code

Behavior code is all code between, but not including the outermost opening and the closing braces, of `behavior` sections. All bodies of `behavior` definitions are behavior code. This code consists of C/C++ code that has been extended by adding LISA+ keywords.

LISA+ top-level code

All LISA+ code that is not behavior code, includes section, or resources section code is LISA+ top-level code. This code consists only of LISA+ keywords and LISA+ constructs.

Each character of LISA+ source code belongs to exactly one of these scopes.

These scopes affect macro expansion and impose certain restrictions on some preprocessor statements.

A.1.1 LISA+ top-level code

Macro expansion and #include expansion is disabled in LISA+ top-level code.

The effect is that #include cannot be used to insert pieces of LISA+ top-level code into other LISA+ files. Preprocessor macros cannot be used to replace anything in the LISA+ top-level code.

However, all other preprocessor statements, such as conditional statements, can be used on the LISA+ top-level. Although macro replacement is disabled, macros are replaced in the expressions of `#if` and `#elif` to evaluate the conditional expressions.

Macros can use `#define` and `#undef` on the LISA+ top-level.

Copyright © 2007-2009 ARM Limited. All rights reserved.
Non-Confidential, Unrestricted Access

A.1.2 Includes section and resources section code

Macro expansion is enabled in Includes section and Resources section code. The preprocessor code in these sections has very special semantics since all preprocessor statements are processed twice:

1. During the first pass, the LISA+ code is processed by the LISA+ preprocessor and all preprocessor statements in these sections, and only in these sections, appear unmodified in the generated source code
2. The C++ compiler processes the preprocessor statements.

The two-stage processing has the following implications:

- `#include` statements work correctly because they are processed by the C++ compiler and all definition from the `#include` files are therefore visible in all behaviors of the same file. This is the primary purpose of the `includes` section and the reason for the special semantics for the preprocessor statements in this section.
- Conditional preprocessor statements only work if either:
  - the symbols used in the condition are defined in the same section itself
  - the symbols are defines outside of the LISA+ file, for example, using `-D` on the command line of the simulator generator or predefined defines.

  Conditional preprocessor statements will not work if the condition uses symbols that are defined on the LISA+ top-level.

The intended use of the includes section is to place `#include` and `#define` statements in this sections to make their declarations visible in the behavior code.

ARM recommends that the resources section does not contain any `#includes` statements to ensure compatibility with future implementations.

A.1.3 Behavior code

Macro expansion is enabled in behavior code (that is, code that is inside of a behavior body).

`#include` statements are ignored and do not have any effect. Conditional compilation works as expected.

Copyright © 2007-2009 ARM Limited. All rights reserved.
Non-Confidential, Unrestricted Access

A.1.4 Restrictions on top-level LISA

Disabling `#include` and macro expansion ensures that tools that can read, modify and write LISA+ code, like System Canvas, can write the LISA+ code without making the textual changes of the preprocessor to the LISA+ code permanent in the LISA+ files.

After `#includes` and macros are expanded, there is no obvious way to undo the expansion and this restricts how the top-level LISA+ is preprocessed.

The final goal of all tools that can edit LISA+ code is to keep as much as possible of the LISA+ source code untouched. Applying a preprocessor to source code always removes all preprocessor statements and macro occurrences from this source code. Performing this action on the original source code is not desirable.

In practice these restrictions are easy to live with. Most preprocessor symbols are used in the C++ code in behaviors where they work as expected. There are special includes-sections to make constants and types defined in header files visible in all behavior code. It is easy to avoid redundancy in the code by using the LISA+ language features rather than preprocessor macros.

Copyright © 2007-2009 ARM Limited. All rights reserved.
Non-Confidential, Unrestricted Access

A.2 Predefined symbols and macros

The following preprocessor symbols are always defined in all LISA+ files in all scopes:

`SYSTEM_GENERATOR_MAJOR_VERSION`

Major version of the Fast Model Tools. The value is an unsigned integer such as, for example, 2 for System Generator 2.3.044.

`SYSTEM_GENERATOR_MINOR_VERSION`

Minor version of the Fast Model Tools. The value is an unsigned integer such as, for example, 3 for System Generator 2.3.023.

`SYSTEM_GENERATOR_REVISION`

Revision of the Fast Model Tools. The value is an unsigned integer such as, for example, 44 for System Generator 2.3.044.

`SYSTEM_GENERATOR_VERSION`

Fast Model Tools version as string constant such as, for example, `"2.3.044"` for version 2.3.044.

`SYSTEM_GENERATOR_VERSION_AT_LEAST(`*major*`,`*minor*`,`*revision*`)`

This macro evaluates to `true` (1) if the Fast Model Tools version is at least *major.minor.revision*.

For example `SYSTEM_GENERATOR_AT_LEAST(2,1,57)` evaluates to 1 for System Generator 2.3.044 since 2.3.044 is greater than 2.1.057.

`SYSTEM_GENERATOR_VERSION_EQUALS(`*major*`,`*minor*`,`*revision*`)`

This macro evaluates to `true` (1) if the Fast Model Tools version is exactly *major.minor.revision*.

`linux` Defined as 1 if parsing LISA+ files or generating a simulator on Linux host systems. Undefined on all other host platforms.

`WIN32` Defined as 1 if parsing LISA+ files or generating a simulator on Windows host systems. Undefined on all other host platforms.

——— **Note** ———

The version symbols above are only defined for Fast Model Tools versions greater than or equal to 2.2.024. For earlier versions, the version symbols are undefined.

Copyright © 2007-2009 ARM Limited. All rights reserved.
Non-Confidential, Unrestricted Access

Preprocessor symbols that are typically predefined by C/C++ compilers on certain host platforms are not defined for LISA+ files, not even in behavior bodies. For example, the symbols `__cplusplus` or `__GNUC__` are not defined when LISA+ files are preprocessed. However, preprocessor symbols can always be set manually in the project settings for specific host platforms.

Copyright © 2007-2009 ARM Limited. All rights reserved.
Non-Confidential, Unrestricted Access

A.3 Preprocessor statements

This section describes the supported preprocessor statements and their LISA+ semantics. All statements have same syntax and semantics as the corresponding C preprocessor statements.

#define Define a macro. Macros can have arguments. Converting to strings (# operator) and concatenating (## operator) is supported. The macro is defined in all LISA+ source in the same file that follows the `#define` statement unless it is explicitly `#undefined`.

Macros can be redefined several times without warning if the redefinition is identical. Macro expansion is disabled in LISA+ top-level code. See *Scope* on page A-2. Macros defined on the LISA+ top-level do not affect conditional statements in the `includes` and `resources` section.

#undef Undefine a macro. The macro is undefined in all LISA+ code in the same file that follows this statement. It is not an error if the macro was not defined before this statement.

#if #elif #else #endif

The code enclosed by #if, #elif, #else, or #endif blocks is enabled or disabled depending on the value of the expression following the #if and #elif statements. If the expression evaluates to 0 the code is disabled, for all other values the code is enabled.

Undefined identifiers in the expression have a numerical value of 0. The expression `defined(SYM)` evaluates to 1 if the preprocessor symbol SYM is defined or 0 if it is not defined.

#ifdef #ifndef

These statements are a shortcut for the `#if defined(SYM)` and `#if !defined(SYM)` statements respectively.

#include Include statements are ignored by the preprocessor. See *Scope* on page A-2. However, `#include` statements in the `includes` sections of components have the desired effect of making the declarations in the header files visible in the behavior code.

#error The error message that follows the `#error` statement is printed. Processing of the LISA+ code by the tool is unsuccessful and the tool performs as if an error has occurred.

#warning The warning message that follows the `#warning` statement is printed. Processing of the LISA+ code by the tool is successful and the tool performs as normal.

Copyright © 2007-2009 ARM Limited. All rights reserved.
Non-Confidential, Unrestricted Access

Copyright © 2007-2009 ARM Limited. All rights reserved.
Non-Confidential, Unrestricted Access