LISA+ Language for Fast Models

Version 1.0

Reference Manual

arm

101092_0100_03_en

LISA+ Language for Fast Models

Reference Manual

Release Information

Document History

Issue	Date	Confidentiality	Change
A	31 May 2014	Non-Confidential	New document for Fast Models v9.0, from DUI0372O for v8.2.
B	30 June 2014	Non-Confidential	Replacement of Chapter 2.
C	30 November 2014	Non-Confidential	Update for v9.1.
D	28 February 2015	Non-Confidential	Update for v9.2.
E	31 May 2015	Non-Confidential	Update for v9.3.
F	31 August 2015	Non-Confidential	Update for v9.4.
G	30 November 2015	Non-Confidential	Update for v9.5.
H	29 February 2016	Non-Confidential	Update for v9.6.
I	31 May 2016	Non-Confidential	Update for v10.0.
J	31 August 2016	Non-Confidential	Update for v10.1.
K	11 November 2016	Non-Confidential	Update for v10.2.
L	17 February 2017	Non-Confidential	Update for v10.3.
1100-00	31 May 2017	Non-Confidential	Update for v11.0. Document numbering scheme has changed.
0100-00	31 August 2017	Non-Confidential	Document number has changed. Version number changed to 1.0.
0100-01	17 November 2017	Non-Confidential	Update for v11.2.
0100-02	23 February 2018	Non-Confidential	Update for v11.3.
0100-03	22 June 2018	Non-Confidential	Update for v11.4.

Non-Confidential Proprietary Notice

Arm Limited. Company 02557590 registered in England.

110 Fulbourn Road, Cambridge, England CB1 9NJ.

LES-PRE-20349

Confidentiality Status

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

Unrestricted Access is an Arm internal classification.

Product Status

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

Web Address

http://www.arm.com

Contents

LISA+ Language for Fast Models Reference Manual

Preface

About this book 7

Chapter 1 Introduction

1.1 General overview of System Canvas and LISA+ 1-10
1.2 Design methodology of System Canvas and LISA+ 1-11
1.3 Terminology of LISA+ 1-12

Chapter 2 LISA+ Components

2.1 About LISA+ components 2-14
2.2 LISA+ integer types and state variables 2-15
2.3 The LISA+ message function and debugging 2-16
2.4 Component resources section 2-18
2.5 Component includes section 2-34
2.6 Component composition section 2-35
2.7 Component behavior sections 2-38
2.8 Component port declarations 2-43
2.9 Component connection section 2-46
2.10 Component properties section 2-49
2.11 Component debug section 2-50
2.12 Component parameter export list 2-52

Chapter 3 Communication with C++ Code

3.1 Accessing C++ constructs from LISA+ 3-54

3.2 Calls to LISA+ behaviors from C++ code 3-57
3.3 Third party model import 3-59

Chapter 4 LISA+ Protocols

4.1 About LISA+ protocols 4-61
4.2 LISA+ protocol includes section 4-62
4.3 LISA+ protocol properties section 4-63
4.4 LISA+ protocol behavior prototypes 4-65

Appendix A LISA+ Preprocessor

A.1 About the LISA+ preprocessor Appx-A-70
A.2 LISA+ preprocessor scopes Appx-A-71
A.3 LISA+ preprocessing according to scope Appx-A-72
A.4 Predefined LISA+ symbols and macros Appx-A-73
A.5 LISA+ preprocessor statements Appx-A-74

Preface

This preface introduces the *LISA+ Language for Fast Models Reference Manual*.

It contains the following:

- *About this book* on page 7.

About this book

LISA+ Language Reference Manual. This document introduces the LISA+ language, which is used for writing models for components and systems. It describes communication with C++ code, and the LISA+ preprocessor.

Using this book

This book is organized into the following chapters:

Chapter 1 Introduction

This chapter introduces the document.

Chapter 2 LISA+ Components

This chapter describes the sections within the component declaration.

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.

Chapter 4 LISA+ Protocols

This chapter describes the syntax of the LISA+ protocol section.

Appendix A LISA+ Preprocessor

This appendix describes the C-like preprocessor statements that you can use in LISA+ source code, and how the LISA+ preprocessor works.

Glossary

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

See the *Arm® Glossary* for more information.

Typographic conventions

italic

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

bold

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

`monospace`

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

`monospace`

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

`monospace italic`

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

`monospace bold`

Denotes language keywords when used outside example code.

`<and>`

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

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

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

Feedback

Feedback on this product

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

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

Feedback on content

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

- The title *LISA+ Language for Fast Models Reference Manual*.
- The number 101092_0100_03_en.
- If applicable, the page number(s) to which your comments refer.
- A concise explanation of your comments.

Arm also welcomes general suggestions for additions and improvements.

Note

Arm tests the PDF only in Adobe Acrobat and Acrobat Reader, and cannot guarantee the quality of the represented document when used with any other PDF reader.

Other information

- *Arm® Developer*.
- *Arm® Information Center*.
- *Arm® Technical Support Knowledge Articles*.
- *Technical Support*.
- *Arm® Glossary*.

Chapter 1
Introduction

This chapter introduces the document.

It contains the following sections:

- *1.1 General overview of System Canvas and LISA+* on page 1-10.
- *1.2 Design methodology of System Canvas and LISA+* on page 1-11.
- *1.3 Terminology of LISA+* on page 1-12.

1.1 General overview of System Canvas and LISA+

The *Language for Instruction Set Architectures* (LISA) specifically targets the description of instruction set architectures. LISA+ is an enhanced version of LISA that describes components and systems.

System Canvas and LISA+ provide an environment for the development of peripheral components or system designs. Their benefits are:

- Early software development.
- Hardware and software co-design.
- Dramatically shortened system exploration turnaround time.
- Highly accurate and non-ambiguous system specification.
- Maintainable system design.

1.2 Design methodology of System Canvas and LISA+

System Canvas uses systems and components to create library objects or executables. It translates LISA+ source code for model components or systems into C++ source code, and compiles that into library objects.

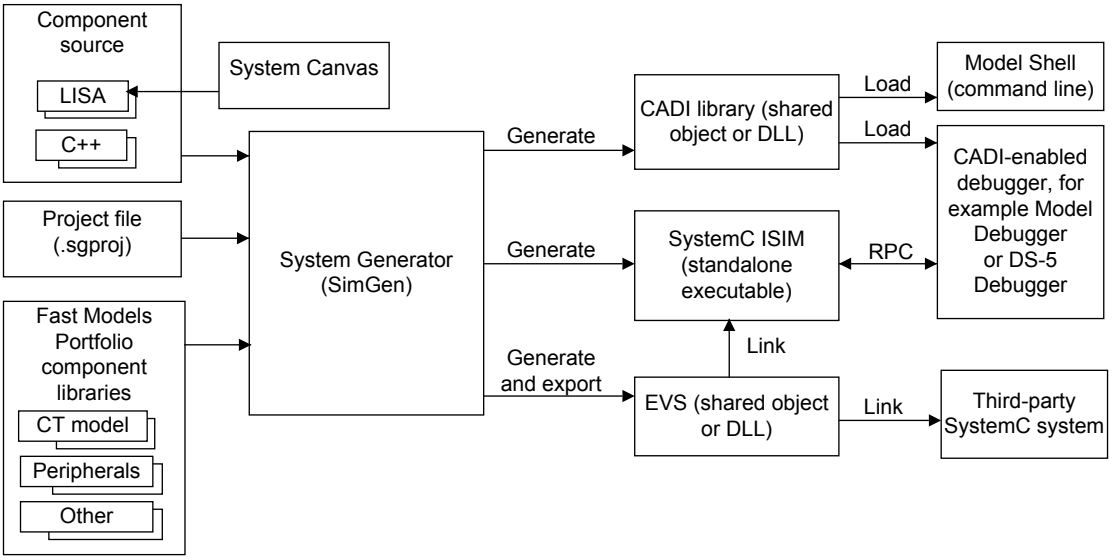

Figure 1-1 Fast Models Tools design flow for processor model development

1.3 Terminology of LISA+

The meanings of LISA+ terms.

Table 1-1 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, for example core, memory, bus, or peripheral, or a complete system or sub-system.
Connection	A link between two components. The connection is 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 core that makes use of code translation technology. CT core models translate instructions dynamically and cache the translation to enable fast execution of code.
External port	A port that connects the subsystem to other components within a higher-level system.
Internal port	Internal ports communicate with subcomponents and are not visible if the component is 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. To connect, ports must use the same protocol.
Resource	A section for declaring private C/C++ variables, for example registers, within a component. You can expose these variables if required.

Chapter 2
LISA+ Components

This chapter describes the sections within the component declaration.

It contains the following sections:

- *2.1 About LISA+ components* on page 2-14.
- *2.2 LISA+ integer types and state variables* on page 2-15.
- *2.3 The LISA+ message function and debugging* on page 2-16.
- *2.4 Component resources section* on page 2-18.
- *2.5 Component includes section* on page 2-34.
- *2.6 Component composition section* on page 2-35.
- *2.7 Component behavior sections* on page 2-38.
- *2.8 Component port declarations* on page 2-43.
- *2.9 Component connection section* on page 2-46.
- *2.10 Component properties section* on page 2-49.
- *2.11 Component debug section* on page 2-50.
- *2.12 Component parameter export list* on page 2-52.

2.1 About LISA+ components

A *component* is the fundamental LISA+ construct that describes components and systems.

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

Systems that have external ports can be used as components in a higher-level system. The term *system* can mean a collection of connected components with no external ports.

Component definitions can contain these sections:

- Resources.
- Includes.
- Composition.
- Behavior.
- Port.
- Connection.
- Properties.
- Debug.
- Parameter export list.

Except for the behavior and port sections, there can be only one of each section type in a component.

Note

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.

Component declaration

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

     resources
     {
          // ...
     }

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

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

2.2 LISA+ integer types and state variables

LISA+ uses C/C++ code to describe the behavior and the state variables of a component. Although you can use native C integer types like `int` and `char` in the description, it is often desirable to use integer types with a defined bitwidth that is independent of the host architecture.

LISA defines these integer types:

`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.

These types are efficient because they have no overhead over native C data types.

2.3 The LISA+ message function and debugging

This section describes debugging with the LISA+ message function.

This section contains the following subsections:

- *2.3.1 About the LISA+ message function and debugging* on page 2-16.
- *2.3.2 LISA+ message C++ prototype* on page 2-16.
- *2.3.3 LISA+ message C prototype* on page 2-17.

2.3.1 About the LISA+ message function and debugging

Printing output to a window or to the console is often useful for debugging components. LISA+ has a `message()` function that prints messages to the output window. It forwards messages through `CADISimulationCallback::simMessage()`.

Note

Message handling does not work in the `terminate()` simulation phase because the callback has already been disconnected.

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.

2.3.2 LISA+ message C++ prototype

The C++ style prototype has two parameters.

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

`msg`
is the message to display.

`type`

characterizes the purpose or nature of the message.

Pass one of these LISA+ symbols 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, this stops it in the same way as the `simHalt()` function. Note that the simulation does not stop immediately.

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

In SystemC systems, this maps to `SC_REPORT_FATAL()`.

MSG_ERROR

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

In SystemC systems, this maps to `SC_REPORT_ERROR()`.

MSG_WARNING

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

In SystemC systems, this maps to `SC_REPORT_WARNING()`.

MSG_INFO

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

In SystemC systems, this maps to `SC_REPORT_INFO()`.

MSG_DEBUG

indicates a debug message, which is only printed if the debug version of build is used. The message is preceded with the text `DEBUG`.

In SystemC systems, this maps to `SC_REPORT_INFO_VERB(SC_DEBUG, ...)`.

2.3.3 LISA+ message C prototype

The C-style prototype has a variable parameter list.

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

`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 is:

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

2.4 Component resources section

This section describes the resources section within the component declaration.

This section contains the following subsections:

- *2.4.1 About the component resources section* on page 2-18.
- *2.4.2 Plain C/C++ variable declarations* on page 2-18.
- *2.4.3 Annotated resources* on page 2-18.
- *2.4.4 Accessing resources* on page 2-33.
- *2.4.5 Obsolete resources constructs* on page 2-33.

2.4.1 About the component resources section

The resources section adds local declarations to a component.

- Variable declarations, which use the C/C++ syntax.
- Annotated resources, which use the LISA+ syntax and parameters.

2.4.2 Plain C/C++ variable declarations

Declare plain C/C++ variables in the resources 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.

To see certain state variables in the debugger, annotate these state variables with `REGISTER` or `MEMORY` keywords.

Related reference

Annotated resources - register parameters on page 2-19

Annotated resources - memory parameters on page 2-26

2.4.3 Annotated resources

This section describes register, memory, and parameter annotated resources.

Syntax of annotated resources

Use the `REGISTER`, `MEMORY`, and `PARAMETER` keywords to specify annotated resources in the resources section.

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

```
<resource_class>  [ <parameters> ]  identifier  “;”
```

```
<resource_class>  [ <parameters> ]  identifier  “[“ <size> “]”  “;”
```

Related reference

Annotated resources - registers on page 2-18

Annotated resources - memory on page 2-26

Annotated resources - parameters on page 2-32

Annotated resources - registers

This section describes LISA+ component registers.

Annotated resources - about registers

Registers are resources that store data.

You can specify a bit width and type for the register. The default data type is `unsigned int` and the default bit width is 32.

Register resource definition

Register set `R` consists of 32 registers that are 32 bits wide and a register file using the default types.

```
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];
}
```

Annotated resources - register parameters

Parameter names, types, default values, and descriptions of LISA+ component registers.

Table 2-1 Optional parameters for registers

Name	Type	Default	Description
`address`	Integer	`none`	Address maps to a unique register ID.
`attribute`	Access type	`read_write`	`read_write`, `read_only`, or `write_only.`
`bitwidth`	Integer	32	Data type bit-width: **Integer** 8, 16, 32, 64. **Floating point** 32, 64. **Boolean** 1. **String** -.
`description`	String	""	Description of the resource.
`display_float_format`	String	"%g"	`printf()` format string for debugger display for floating point registers, only 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. Typically, the architecture ABI defines these.
`groups`	String-list	-	Comma-separated list of register groups that the register is assigned to.
`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.
`lsb_offset(`*`Bit`*`)`	-	-	Bit offset in the parent register.

Table 2-1 Optional parameters for registers (continued)

Name	Type	Default	Description
`name`	String	resource name	Register name to display in, for example, a debugger. By default: **Non-array resources** `"R"` for register R, for example. **Array resources** Resource name then the decimal index. For example, `R[0]`, `R[1]`, … `R[7]` for register array `R[8]`. The name string can contain one `printf()` integer specifier. 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, for example `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`*`)`	-	-	Parent register.
`pv_port`	Integer	`none`	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, you can omit this parameter.
`read_behavior`	String	none	Read behavior that you define in case 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	Value for read accesses.
`read_sec_mask`	Integer	none	Value 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	`reg_number` increment between array elements, starting with `reg_number`. Applies only to register arrays and if `reg_number` is specified.
`reset_value`	<type>	0, 0.0, "", or `false`	Reset value for `bool`, `int`, `uint`, `float`, or `string`. The value specified by the parameter is assigned to the register at initialization and reset execution. To avoid register initialization on reset, use `UNINITIALIZED` as the parameter argument.
`type`	<type>	`uint`	Data type: `bool`, `int`, `uint`, `float`, or `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. `virtual` does not apply to parameters.
`visible_in_debugger`	Boolean	`true`	Debug switch. `true` shows the register in the debugger, `false` hides it.
`write_behavior`	String	none	Write behavior that you define in case the automated mechanisms are not sufficient. Use of this behavior depends on the internal state.

Table 2-1 Optional parameters for registers (continued)

Name	Type	Default	Description
`write_function`	String	none	Name of the debug write access behavior. The access functions must match the arguments and return type.
`write_mask`	Integer	none	Value for write accesses.
`write_sec_mask`	Integer	none	Value for write accesses. For secure accesses, this value overwrites other masks.

Related concepts

Annotated resources - component registers on page 2-23

Annotated resources - IDs in register arrays

If a register array has an ID, the first register receives the value of the ID. The ID of each subsequent register in the array is the ID of the previous register plus one.

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

Register `R[0]` has ID 5, `R[1]`has ID 6, `R[2]` has ID 7, and so forth. Use the `reg_number_increment` parameter to step between registers in an array, starting with `reg_number`. For example, if `reg_number_increment` is set to 2, `R[1]` is not used.

Annotated resources - virtual registers

The `virtual` parameter enables the abstract declaration of a resource. If you use the LISA+ `virtual` parameter, you must implement a read and a write access behavior and not rely on the existence of a variable that has the resource name.

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;
    }

    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;
    }
}
```

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` returns the success flag.

Annotated resources - register access

Access registers in the same way as C variables.

Register assignment

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

Annotated resources - 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:

- 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. You can shift the bit sequence, but cannot split or manipulate it in any other way.

The parameters that specify the component register are:

`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.

Component register resource definition

```
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;
```

Annotated resources - debugger register access functions

Override the default functions that a debugger calls to access registers, by using behaviors that conform to a specific prototype.

Table 2-2 Prototypes of debugger register access functions

Name	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] See the `CADIRegRead` function description.
[b] See the `CADIRegWrite` function description.

`reg_id`

holds the register ID of the register that is being accessed, that is, the argument used in the `reg_number` attribute. Modify the array index step size by using `reg_number_increment` if there is 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. Form the index ID by appending an "_" and the index of the array entry.

`data`

is a buffer that 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`s that holds the entire register value, starting with the *Least Significant Bit* (LSB) in `data[0]`.

Note

- The write access function prototypes declare the data parameter as `const`.
- A separate pair of prototypes exists specifically for registers of type `string`. The data parameter is a reference to a `std::string` for these 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 differ 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. Only make the debugger call `read_function` with `side_effects == true` if you want 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.

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 use these LISA+ symbols to inform the calling code whether or not the access operation succeeded:

`ACCESS_FUNC_OK`

The call was successful.

`ACCESS_FUNC_GeneralError`
An error that the other error return values do not explain.

`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 fulfill the command.

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

`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.

Read access function

Registers R1 and R2 receive 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.

```
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;
}
```

Related information

Model Debugger for Fast Models User Guide
Component Architecture Debug Interface Developer Guide

Annotated resources - memory-mapped register access

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

Register IDs are specified explicitly. They have the same value as the address offset. An internal read/write behavior implements the register accesses. 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
```

Use the ID of a register resource in the read and write behaviors.

Note

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

Related reference

Annotated resources - debugger register access functions on page 2-23

Annotated resources - memory

This section describes memory resources.

Annotated resources - about memory

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

Annotated resources - memory parameters

Parameter names, types, default values, and descriptions for memories, buses, and address spaces.

Table 2-3 Parameters for MEMORY resources

Name	Type	Default	Description
`allow_unaligned_access`	Boolean	`false`	Permit unaligned access. If this is `true` then accesses that are not naturally aligned, for example a 32-bit access on a non-32-bit boundary, are permitted and have the expected result. If this is `false` such unaligned accesses are not permitted.
`attribute`	Access type	`read_write`	`read_write`, `read_only`, or `write_only.`
`description`	String	`""`	Description of the space.
`endianness`	Big or little	`little`	Select between little and big endianness.
`executable`	Boolean	`false`	`true` for memory blocks that can hold executable code.

Table 2-3 Parameters for MEMORY resources (continued)

Name	Type	Default	Description
`mau_size`	Integer	8	Size of the *Minimum Addressable Unit* (MAU) in bits. Admissible values are 8, 16, 32, and 64.
`paged`	Boolean	`true`	Permit 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.
`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 `virtual` is `true`, no variable is generated for this parameter. The read and write functions provide and store the value. The parameter is virtual because the read and write functions model it.
`write_function`	String	none	Name of the debug write access behavior.

Specify the array size using:

- Pure integer values.
- Suffixes for Kilo, Mega, Giga, Tera, or Peta, that is, 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}, respectively.
- Expressions, for example 2k-1.

Memory resource definition

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

Annotated resources - read and write accesses

Access memory resources using C-array like syntax.

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

The memory access functions are:

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 );
```

Use these access functions to read or write 8, 16, 32 and 64-bit quantities from or to memory. The function name implies the size of the access. 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.

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);
}
```

Annotated resources - debugger memory read- and write-access functions

The access functions for memory are similar to the access functions for registers. You can attach debugger access functions to `REGISTER` and `MEMORY` resources.

The component that holds the memory resource implements the access functions as LISA+ behaviors. The `read_function` and `write_function` resource parameters designate the access functions. Define the debugger access functions as behaviors, but attach them in the resources statements (`REGISTER` or `MEMORY`), as here. See the `CADIMemRead` and `CADIMemWrite` function descriptions for more information.

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

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

`block_id`
: an integer value that, for the specified memory space, is a unique identifier for a memory block. This is unused and always 0.

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

`data`
: 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`
: 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.

`MemoryAccessContext`
: a pointer to a `MemoryAccessContext` object. This provides extensibility to the prototype, and the `MemoryAccessContext` class can be enriched if required. The current interfaces are:

  `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.

  Note

  Accessing these values though `MemoryAccessContext`, rather than by coding them as constants, lowers the maintenance hazard arising from a memory attribute change.

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;
}
```

Implementation of a read access function

Here are declarations of two memory resources, `m1` and `m2`, and a read access function, `my_read`, 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 run-time by means of the `space_id` and `block_id` parameters. Here, a simple `if` / `else` sequence distinguishes them.

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.

```
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 allocation 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.

Related reference

Annotated resources - debugger register access functions on page 2-23

Related information

Model Debugger for Fast Models User Guide

Annotated resources - parameters

Parameters permit component configuration and parameterization of the system model at initialization-time or in run-time. They do not support array syntax.

Table 2-4 Parameters for parameters

Parameter	Type	Default	Description
`default`	Integer 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.
`max`	Integer	`0x7FFFFFFFFFFFFFFF`	Maximum admissible value.
`min`	Integer	`0x8000000000000000`	Minimum admissible value. 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`	Integer, Boolean, or string	`int`	Data type. The type can also be `intx_`*`size`* or `uintx_`*`size`* where *`size`* is one of 8, 16, 32, or 64.
`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. You can set instantiation-time parameters before the system instantiates. You cannot change them afterwards, or query them with a debugger. This is the default. You can change run-time parameters during run-time.

The access functions have similar semantics to those of registers. The access function prototypes are:

- 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.

Parameter resource definition

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

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.

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

Table 2-5 Suffixes for parameter values

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

2.4.4 Accessing resources

Access the resources in the component behavior by using the name of the resource.

2.4.5 Obsolete resources constructs

The `connection` section of components completely replaces the `resource_mapping` section in the resource section of components.

The `connection` section enables using hierarchical systems in a clean way. *System Generator* (SimGen) does not use `resource_mapping` sections, but issues a warning.

Related reference

2.9.1 About the component connection section on page 2-46

2.5 Component includes section

The `includes` section is a dedicated place for `#include` preprocessor statements.

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. ARM recommends using unique names in the `includes` section that do not conflict with other component declarations, for example in shared component header files.

Related reference

Chapter 3 Communication with C++ Code on page 3-53
Appendix A LISA+ Preprocessor on page Appx-A-69

2.6 Component composition section

This section describes the component composition section.

This section contains the following subsections:

- *2.6.1 About the component composition section* on page 2-35.
- *2.6.2 Overriding component parameter attributes* on page 2-36.

2.6.1 About the component 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.

Initialize parameters of subcomponents 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. The `value` can be:

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

Composition section

`MyComponent` has two subcomponents `mem1` and `mem2` that are both of type `MyMemory`. The expression `size=0x1000` sets compile-time parameter `size` of component `mem1` to `0x1000`.

```
component MyComponent
{
    resources
    {
        PARAMETER mem2size;
    }

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

Related reference

Annotated resources - parameters on page 2-32

2.6.2 Overriding component parameter attributes

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

Parameters of subcomponents that are set in the composition section of their parent component cannot 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.

You can override parameters of subcomponents in Fast Models in the component instantiation statement in the form of assignments, in the syntax:

```
parameter_name.attribute_name=attribute_value
```

Include the override assignments in the normal parameter assignments in an instantiation statement.

Parameter overriding

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

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

The code overrides the default value of `dtcm_size` and its `min` boundary. This means that the parameter is published and you can set it, but:

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

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.

Range restriction warning

The LISA+ parser warns that the range of parameter `b_dtcm_size` is being restricted to `[0x450000, 0xef0000]`.

```
component Core
{
    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
      core : Core(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, enclose it in double quotes.

```
core : Core("semihosting-enable".default=true);
```

2.7 Component behavior sections

This section describes the behavior section within the component declaration.

This section contains the following subsections:

- *2.7.1 About the component behavior sections* on page 2-38.
- *2.7.2 Special-purpose behaviors* on page 2-38.
- *2.7.3 Hierarchical behavior of special-purpose behaviors* on page 2-39.
- *2.7.4 Controlling simulation from behaviors* on page 2-39.
- *2.7.5 LISA+ elements in behaviors* on page 2-40.
- *2.7.6 Scope of behaviors* on page 2-42.

2.7.1 About the component behavior sections

Component descriptions can comprise multiple named behavior sections that specify the behavior of the component model. Behaviors are similar to functions.

Behaviors of components always have a name and can also have an optional list of formal parameters and an optional return type.

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. You cannot have behaviors with different signatures but the same name.

2.7.2 Special-purpose behaviors

The behavior names have specific meanings in components. *System Generator* (SimGen) calls the behaviors implicitly and automatically in certain simulation phases.

`behavior init`

SimGen calls this behavior implicitly when the simulation starts. The code in this behavior only executes once, to allocate memory and other resources.

`behavior reset(int level)`

SimGen calls this behavior whenever you reset the simulation. SimGen might call reset multiple times. There is one defined reset level:

`MX_RESETLEVEL_HARD`

Reset all state variables to their reset values and clear memories.

Do not allocate memory in this behavior because SimGen might invoke it several times.

Caution

- All registers are normally reset to the values specified by their `reset_value` parameters immediately before `behavior reset` is run.
- You can use `reset_value(UNINITIALIZED)` to prevent the register values being overwritten.

`behavior loadApplicationFile(const string& filename)`

SimGen invokes this behavior whenever you load 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`. SimGen calls it when the simulation terminates, to free any memory and resources that `behavior init` allocated.

`behavior setupGlobalSystemAttributes()`

Use this behavior to define a parameter export list. This behavior only matters in the top level component.

Related reference

2.12 Component parameter export list on page 2-52
2.7.4 Controlling simulation from behaviors on page 2-39
Annotated resources - register parameters on page 2-19

2.7.3 Hierarchical behavior of special-purpose behaviors

Special purpose behaviors are optional. If they are missing from a component, the corresponding behaviors of all subcomponents of the component are called recursively. If you specify a special purpose behavior, it is responsible for explicitly calling the subcomponents, using the `composition` keyword.

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.

2.7.4 Controlling simulation from behaviors

Any LISA+ behavior can use one of a set of function calls to detect and control the simulation environment.

Peripheral components do not normally use these behaviors. Normally, the user controls the state of the simulation through a debugger.

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.

Note

The simulation does not stop immediately. The actual shutdown might be as much as 200 instructions 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).

```
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, for example a register read. An endless loop results.
- This function only has effect if the simulation has stopped.

Related concepts
2.7.2 Special-purpose behaviors on page 2-38

2.7.5 LISA+ elements in behaviors

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

These are the types of behavior:

Port behaviors
are declared inside a port.

Component behaviors and local behaviors
are not declared inside a port.

Special-purpose behaviors
are component behaviors, for example `init()`, `reset()`, and `terminate()`.

The following types of function call can occur in component and port behavior sections, to:
- Local behavior, for example `foo()`.
- Local port behavior, for example `myport.foo()`.
- Port behavior of a subcomponent, for example `subcomp.aport.foo()`.
- Special-purpose behaviors of a subcomponent, for example `subcomp.init()`.
- Special-purpose behaviors of all subcomponents, for example `composition.init()`.
- `getInstanceName()` to return the instance name of the component.
- `getInstancePath()` to return the instance name relative to the top-level component.

Special-purpose behavior calls

The use of the `composition` keyword to call a special-purpose behavior for all subcomponents recursively.

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

SimGen calls the subcomponents in the order of declaration in the `composition` section. SimGen calls local behaviors, that is, non-port behaviors in the same component, by using the name of the local behavior.

Subcomponent special-purpose behaviors

Call the special-purpose behaviors of subcomponents by putting the subcomponent instance name and a dot before the behavior name.

Always call subcomponent special-purpose behaviors 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.

Initialization order

```
component MyComponent
{
    composition
    {
        subcomp0: AnotherComponent
        subcomp1: AnotherComponent
        subcomp2: AnotherComponent
    }
     behavior init
        {
            // explicit initialization order
            subcomp2.init();
            subcomp0.init();
            subcomp1.init();
        }
}
```

Port behavior access

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
```

Subcomponent port access

How to call a subcomponent port behavior.

```
component MyComponent
{
    composition { subcomp: AnotherComponent }
     behavior set42_on_subcomp
        {
            subcomp.port0.set(42);
        }
}
```

Component instance name behavior access

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

This function is slow, so ARM does not recommend it for simulation of a component that implements normal functionality. It might, however, be useful in component error messages and debugging output.

Component error message display

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

2.7.6 Scope of behaviors

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

Behaviors that want to return pointers or instances of types that are defined in the resources section of the same component must use the `COMPONENT_CLASS_NAME()` construct to access the type in the resource section. `COMPONENT_CLASS_NAME(x)` represents the scope of component x, and when returning a pointer or an instance of a type that is declared in the resources section of component x, you must explicitly specify this scope when specifying the return type of the behavior.

Component scope with COMPONENT_CLASS_NAME(x)

```
component MyComponent
{
    resources
    {
        struct status_t
        {
            int i;
        };
        status_t status;
    }

    behavior getStatus():status_t  // Incorrect: status_t is unknown outside the
behavior.
    {
        return status;
    }

    behavior getStatus(): COMPONENT_CLASS_NAME(MyComponent)::status_t  // Correct:
prefix the return type with the scope.
    {
        return status;
    }
}
```

2.8 Component port declarations

This section describes component port declarations.

This section contains the following subsections:

- *2.8.1 About component port declarations* on page 2-43.
- *2.8.2 Master, slave, and internal ports* on page 2-43.
- *2.8.3 Port arrays* on page 2-44.
- *2.8.4 Internal ports* on page 2-44.

2.8.1 About component port declarations

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

The processors use these interfaces to communicate with the peripherals. Peripherals can use them to communicate with each other. The communication connections are defined in the `connection` section of each component.

This kind of communication encapsulates each component behind an abstract interface. Components can easily replace each other, 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.

A port declaration has the format:

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

`port_attributes`
: a combination of the attributes `master`, `slave`, `internal` and `addressable`.

`protocol_name`
: the name of a protocol. In a sense, 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 {    }
    // …
}
```

Related reference

2.9.1 About the component connection section on page 2-46

Chapter 4 LISA+ Protocols on page 4-60

2.8.2 Master, slave, and internal ports

Ports have a master side and a slave side. Ports must be `master`, `slave` or `internal`.

Some ports of a component are exposed to the system as master ports, for example a memory port of a processor. 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 processor. 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.

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.

2.8.3 Port arrays

A LISA+ component can contain arrays of ports, 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)
}
```

Implementing read() for a port vector

The parameter list and implementation of a `read` method, which uses the `portIndex` parameter to distinguish between the ports.

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

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

Related concepts

2.9.3 Port array connections on page 2-47

2.8.4 Internal ports

Internal ports are: normal ports that are not accessible from outside of the component, not visible in the component, not a part of the component interface, an internal implementation detail of a component.

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

It is not necessary to declare an internal port to call port behaviors of a subcomponent. Do this directly using the syntax:

```
asubcomponent.aport.abehavior (..)
```

Internal port

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

2.9 Component connection section

This section describes the component connection section.

This section contains the following subsections:

- *2.9.1 About the component connection section* on page 2-46.
- *2.9.2 Hierarchy in port connections* on page 2-47.
- *2.9.3 Port array connections* on page 2-47.

2.9.1 About the component connection section

The connection section is for connecting component ports with each other.

The composition section defines the scope of the connection section. You can only connect the ports of the component itself and ports of components declared in the composition section in the connection section.

The connection section contains a list of connection statements. The connection section syntax is:

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

MAR, SAR
: the optional Master Address Range and Slave Address Range. The address range includes both the low address and high address.

  Note

  All addresses in the master and slave address ranges, if present, must be plain numeric constants, that is, either hex or decimal. Do not use expressions, enums or preprocessor symbols.

`masterComponent and slaveComponent`
: 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).

Specify an address range for addressable ports. You cannot specify an address range, however, for non-addressable ports.

The following rules apply for address ranges:

- If a range is only 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 link 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` both link to address `0x0001` of the slave port.
- You can overlap address ranges but the order of the connection statements is significant. Later connection statements override earlier connection statements. The first connection statement, therefore, has the lowest priority. The priority of connections simplifies creating a default bus slave that covers the whole address space of the bus.

Connecting ports

How to connect ports in the `connection` section.

```
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 that
        // 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;
    }
}
```

The port `memport` is an external port and `probe` is a component. The keyword `self` identifies that the external port `memport` connects to the `access` port of the `probe` component.

2.9.2 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.

Component hierarchy

This example assumes a component `parent` with a subcomponent `sub`.

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

2.9.3 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 these 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.

Port array connections

```
protocol MyProtocol { /* protocol behaviors */ }
component Foo
{
    master port<MyProtocol> mPortArray[4];
}

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;
    }
}
```

These rules apply:

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.

Note

SimGen issues warning `W7538` when it detects all elements of a master port array are connected to a single slave port. Such *fan-ins* are valid, but are usually unintentional and can cause significant performance problems. To suppress the warning, denote the fan-in as explicit by adding `[*]` to the left side of the connection statement, for example:

```
a[*] => b;
```

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. The master must be equal to or smaller than the slave, otherwise an error is raised.

2.10 Component properties section

The `properties` section of a component describes the properties of the component, for example version or component type. Some properties might only be relevant to a specific tool.

Properties section

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

Table 2-6 Component properties

Property	Default	Description
`component_type`	`""`	A string of free-form text describing the type of component.
`component_name`	`""`	A string containing the name of the component.
`default_view`	auto	The System Canvas view is block diagram (auto) or source (source).
`deprecated`	`false`	Marks a component as deprecated. Platforms using this component must be built using the `--allow-deprecated` switch.
`description`	`""`	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.
`dso_safe`	`true`	If set, the component can be placed in the shared library part of the generated model. Otherwise, the component is placed in a static library.
`executes_software`	`0`	This Boolean property indicates that the component executes software and that application files can be loaded into this type of component.
`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 the logo in `xpm` format. This icon is displayed in the System Canvas block diagram editor. The path is relative to the LISA+ file.
`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"`	The version number of the component.

2.11 Component debug section

The debug section enables control of the published and imported *Component Architecture Debug Interface* (CADI) interfaces.

By default, all components publish CADI interfaces in the CADI factory to enable connecting CADI-compliant debuggers. By default, components do not import the CADI interfaces of their subcomponents.

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 are:

`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 subcomponents of the component are affected by the statement.

These rules apply:

- `unpublish` and `unimport` also affect all subcomponents of the components.
- The top-level component cannot be `unpublished` or `unimported`.

Using unpublish and publish
The CADIs of `aa` and `dd` are not published because the `unpublish` command issued to `aa` applies to the CADI of `dd`, even though `aa` publishes `dd` in its debug section.

```
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
    }
}
```

Empty or missing debug section

```
debug
{
    composition.publish;
    composition.unimport;
}
```

2.12 Component parameter export list

Use the parameter export list to limit the exposed set of CADI parameters of a virtual platform. It is a special purpose behavior. Place it in the top component as `setupGlobalSystemAttributes()`.

A parameter export list

```
behavior setupGlobalSystemAttributes()
{
    // hide parameters of 'processor' and its subcomponents
    hideParameter("*.processor.*");

    // but expose all parameters of component processor.core0
    exposeParameter("*.processor.core0.*");

    // but do not expose its semihosting parameters.
    hideParameter("*.processor.core0.semihosting*");
}
```

The methods for exposing and hiding parameters are:

`exposeParameter()`
Expose the given parameters.

`hideParameter()`
Hide the given parameters.

These rules apply:
- By default all parameters are exposed.
- The filtering patterns given as the parameters of `exposeParameter` and `hideParameter` can either contain fully qualified parameter names (including instance names) or wild card expressions.
- Filter patterns are evaluated in the order of their calls.

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.

It contains the following sections:

- *3.1 Accessing C++ constructs from LISA+* on page 3-54.
- *3.2 Calls to LISA+ behaviors from C++ code* on page 3-57.
- *3.3 Third party model import* on page 3-59.

3.1 Accessing C++ constructs from LISA+

This section describes the changes needed to access C++ constructs from LISA+, and an example.

This section contains the following subsections:

- *3.1.1 About accessing C++ constructs from LISA+* on page 3-54.
- *3.1.2 Changes required to your source code* on page 3-54.
- *3.1.3 Changes required to your Fast Models project* on page 3-54.
- *3.1.4 LISA+ example of accessing C++ constructs* on page 3-55.

3.1.1 About accessing C++ constructs from LISA+

You can use C++ constructs without special syntax inside behavior bodies, including ports, of a component, if the `includes` section of a component includes a header file that declares the constructs.

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

3.1.2 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.

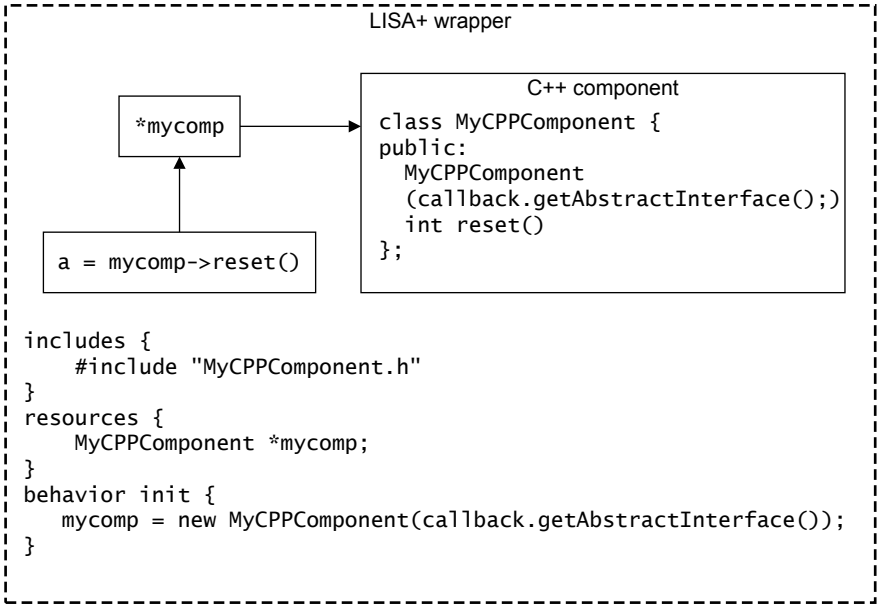

Figure 3-1 Relationship between LISA+ and C++ source

The figure shows an example where the LISA+ code for MyComponent imports a C++ component, `MyCPPComponent`. The LISA+ wrapper code references the C++ header for the `MyCPPComponent` model in the `includes` section. To access the C++ object, use pointers. A pointer, `*mycomp`, gives access to the `reset` function of MYCPPComponent.

3.1.3 Changes required to your Fast Models project

Configure your System Canvas project to locate C++ object header files.

Procedure

1. Open the **Projects Settings** dialog by clicking **Project** > **Project Settings** . Add the path to your C++ header files in the **Include Directories** field in the **Compiler** parameter category.
   Example:

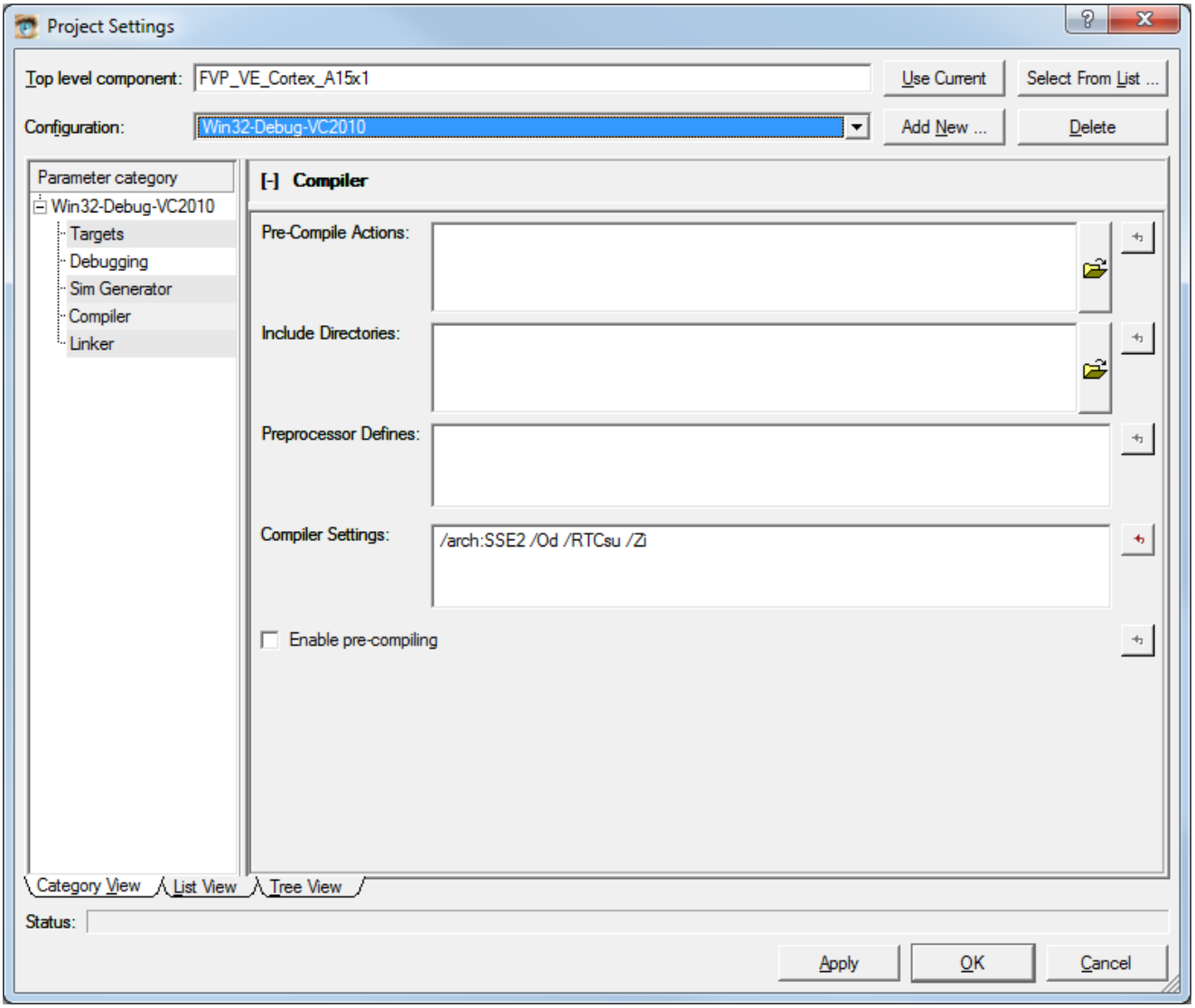

Figure 3-2 System Canvas Project Settings dialog

2. Include a path to the library file that contains your C++ object. Click **Project** > **Add Files...** to open the **Add Files** dialog. 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.

Related information

Fast Models User Guide

3.1.4 LISA+ example of accessing C++ constructs

A commented LISA+ example demonstrating the concepts necessary to communicate with C++.

```
component MyComponent{
includes
    {
       // make C++ declarations visible in behaviors and resources
        #include "MyCPPComponent.h"
    }

    resources
    {
```

```
        MyCPPComponent *mycomp;        // declare a pointer to a C++ class
    }

    // use this internal port to allow the C++ code to talk to us
    internal port<MyInterface> callback
    {
    behavior signal()
        {
            // the C++ code can call this behavior
        }
    }

    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)
    {
        // LISA+ code can call C++ code directly without any special
        // syntax
        mycomp->reset();
    }

    behavior terminate
    {
            delete mycomp;            // delete instance of C++ class
    }
}

// use this protocol to allow the C++ code to call the LISA code protocol
MyInterface{
    behavior signal();
}
```

Related reference

3.1.4 LISA+ example of accessing C++ constructs on page 3-55

3.2 Calls to LISA+ behaviors from C++ code

This section describes how to call LISA+ behaviors from C++ code.

This section contains the following subsections:

- *3.2.1 About calls to LISA+ behaviors from C++ code* on page 3-57.
- *3.2.2 Requirements for importing models with callbacks* on page 3-57.
- *3.2.3 getAbstractInterface()* on page 3-57.
- *3.2.4 Abstract interface header file* on page 3-58.

3.2.1 About calls to LISA+ behaviors from C++ code

Call back into LISA+ code from component C++ code by passing a pointer to a pure virtual interface class, the 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.

The C++ code can call every port, be it internal, master, or slave. However, it cannot call non-port component behaviors directly using this approach.

3.2.2 Requirements for importing models with callbacks

Callbacks provide a method that permits a C++ object to call LISA+ behaviors. It must fulfill two conditions for callbacks to work.

1. The LISA+ object must implement the necessary callback functions, through the `getAbstractInterface()` function.
2. You must pass the address of the LISA+ object to the C++ object, using a C++ header file.

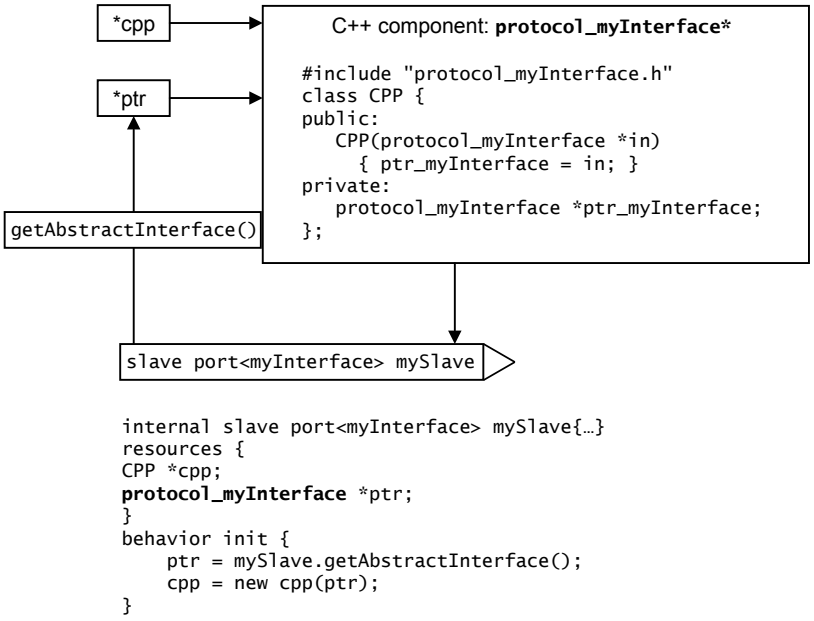

Figure 3-3 Relationship between C++ and LISA+ components in callbacks

Use the LISA+ construct `mySlave.getAbstractInterface()` to get the pointer `*protocol_myInterface` to the mySlave port.

The C++ component header file is `protocol_myInterface.h`.

3.2.3 getAbstractInterface()

Use this function to call between C++ and LISA+ components.

LISA+ component ports implement the function, which:

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

You can use the LISA+ construct *`portinstance`*`.getAbstractInterface()` in 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`* is the name of the protocol of the port. The LISA+ code can pass a pointer to this class to the C++ code on component initialization, and the C++ code can call the LISA+ code back through the abstract interface class and the port behaviors.

3.2.4 Abstract interface header file

System Generator (SimGen) automatically generates the definition of the abstract interface class into the directory that contains all of the source files that System Canvas generates.

To call methods of the abstract interface class, the C++ code must know the class declaration of the abstract interface class. The name of the header file and the name of the abstract interface class are `protocol_`*`ProtocolName`*`.h` and `protocol_`*`ProtocolName`*, 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.

SimGen generates the abstract interface class and the header file 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. The 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 you change the LISA+ protocol definition, the abstract interface class also changes and you must also update the interface header file that the C++ code uses.

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 call semantics depend on the type of port the abstract interface belongs to and the implementation of the port.

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 provide access functions for the C++ code to certain resources in the protocol. The only purpose of such a protocol would be to communicate with C++ code.

If the abstract interface belongs to a master port that connects 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.

3.3 Third party model import

You can include third party C++ models in your Fast Models system.

To include third party C++ models without access to the C++ source, you require:

- Compiled library files for the models.
- The model interfaces.
- A callback interface for the models.

To use a third party model in your system, you might require to implement your own callback class to bridge the third party model and your LISA+ system. For example, if your third party model callback interface does not match the LISA+ protocols.

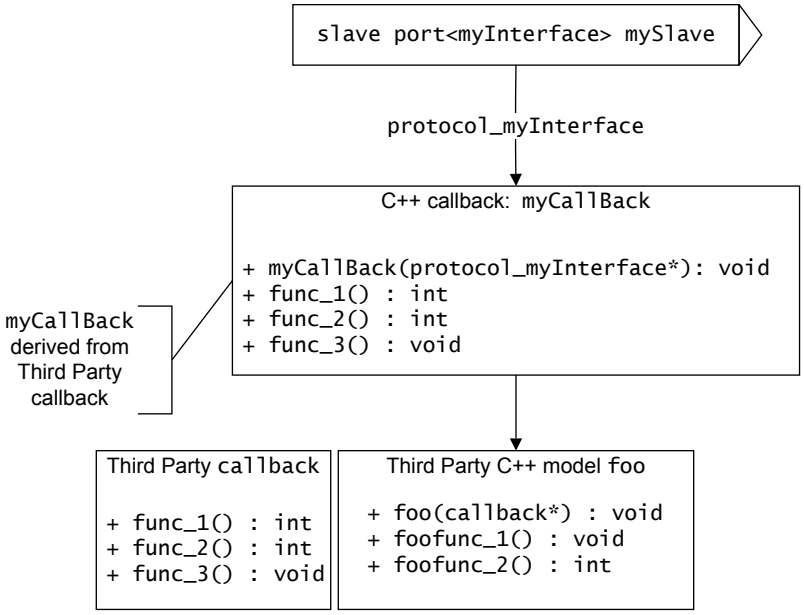

Figure 3-4 Third party model callback relationships

In the figure, the `myCallBack` callback class derives from the callback class of the third party model and interfaces with your LISA+ protocol. A pointer to the `myCallBack` class passes to the third party model. The `myCallBack` class communicates with the C++ model using a LISA+ callback, `protocol_myInterface`.

Chapter 4
LISA+ Protocols

This chapter describes the syntax of the LISA+ protocol section.

It contains the following sections:

- *4.1 About LISA+ protocols* on page 4-61.
- *4.2 LISA+ protocol includes section* on page 4-62.
- *4.3 LISA+ protocol properties section* on page 4-63.
- *4.4 LISA+ protocol behavior prototypes* on page 4-65.

4.1 About LISA+ protocols

The LISA+ keyword `protocol` declares a protocol.

```
protocol MyProtocol
{
    // protocol definition
}
```

Define protocols on the top level in the LISA+ code. By convention, each protocol *`MyProtocol`* has a separate LISA+ file `MyProtocol.lisa` to define it.

`MyProtocol` can be any valid C identifier. A protocol defines ports in components that use this protocol to communicate with other components, including parent-, sibling- and sub-components. You can think of protocols as *port types* because only ports that use the same protocol can connect.

A protocol definition can contain:

- An includes section.
- A possibly empty list of behavior prototypes.
- A properties section.

4.2 LISA+ protocol includes section

The `includes` section includes C/C++ type declarations and constants from external C/C++ header files. It is optional, and sits inside the protocol definition.

The contents and semantics of this includes section are the same as for the includes section of components. It usually contains a list of preprocessor `#include` statements. These `#include` statements do not expand into the LISA+ code. The `#include` statements associate with the protocol definition and you can use the types declared in the included header files 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 all LISA+ files of a subsystem, including all other component and protocol definitions, but is only guaranteed to span the protocol definition itself.

4.3 LISA+ protocol properties section

This section describes the properties section.

This section contains the following subsections:

- *4.3.1 LISA+ protocol properties section syntax and properties* on page 4-63.
- *4.3.2 LISA+ protocol properties for SystemC export* on page 4-63.

4.3.1 LISA+ protocol properties section syntax and properties

The `properties` section of a protocol describes properties, for example version and base class. Some properties might only be relevant for a specific tool. All are optional.

LISA+ properties section syntax

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

Table 4-1 LISA+ protocol properties

Property	Default	Description
`description`	`""`	Description of the protocol.
`version`	`"1.0"`	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.
`dso_safe`	`true`	If set, the component can be placed in the shared library part of the generated model. Otherwise, the component is placed in a static library.
`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).
`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).

4.3.2 LISA+ protocol properties for SystemC export

Use the properties `sc_slave_base_class_name`, `sc_slave_export_class_name` and `sc_master_port_class_name` if exporting a System Canvas generated system to SystemC.

These properties are for protocols that 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.

System Generator (SimGen) ignores them if you instantiate a port of this protocol outside of the top level component of a system or if no SystemC component is generated.

If a SystemC component is generated but you do not set these properties for the ports of protocols in the top level component, SimGen ignores the ports and issues a warning.

Related information

Fast Models User Guide

4.4 LISA+ protocol behavior prototypes

This section describes LISA+ protocol behavior prototypes, which declare a behavior of the enclosing protocol.

This section contains the following subsections:

- *4.4.1 About LISA+ protocol behavior prototypes* on page 4-65.
- *4.4.2 LISA+ protocol behavior prototype syntax* on page 4-65.
- *4.4.3 LISA+ protocol behavior prototype attributes* on page 4-65.
- *4.4.4 LISA+ protocol behavior ADDRESS arguments* on page 4-67.

4.4.1 About LISA+ protocol behavior prototypes

A protocol definition can contain zero or more behavior prototypes. Each behavior prototype declares a behavior of the enclosing protocol.

The 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.

4.4.2 LISA+ protocol behavior prototype syntax

The syntax of behavior prototypes is similar to the syntax of behaviors in components. The main differences are that behavior prototypes can have specific attributes, and do not require a body.

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

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

`attributes`
: a combination of `optional`, `master`, and `slave`. There are restrictions.

`name`
: the name of the protocol behavior, any C identifier. Each name can only occur once in the protocol definition. Overloaded behavior prototypes are not permitted.

`formal_args`
: the formal arguments of the behavior. The syntax is the same as for C++ function declarations. If you define types, define them in a header file that you include in the includes section.

  You can use the native C/C++ types and the native LISA+ types 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, you can omit the opening and closing parentheses. You can also omit the names of the formal arguments. Variable number of arguments and default values are not permitted.

`return_type`
: the type of the return value of that behavior. You can omit the return type if it is `void`. In this case, also omit the colon, `:`.

4.4.3 LISA+ protocol behavior prototype attributes

This section describes behavior prototype attributes.

About LISA+ protocol behavior prototype attributes

In most cases, a protocol behavior is a slave or optional slave behavior. Master behaviors are not common.

If set, the `attribute` specifier for the behavior must be one of:

`master`
: You must implement a behavior for a master port. A default implementation is not permitted.

`slave`
: You must implement a behavior for a slave port. A default implementation is not permitted.

`optional master`
: You do not have to implement this behavior for a master port. A default implementation can be provided as part of the prototype definition.

`optional slave`
: You do not have to implement this behavior 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 from the port behavior definition.

Note

- If there is no `attribute` specifier, none of these constraints apply. That is, the behavior is optional, and you can specify a default implementation if required.
- The presence of default behaviors is part of the protocol definition.

Mandatory LISA+ protocol behavior

A description of mandatory behavior.

For this behavior declaration in a protocol:

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

- All slave ports using this protocol must implement this behavior `f()`.
- Calling `f()` invokes all behaviors `f()` in all slave ports connected to the same port.
- Master ports might not implement behavior `f()`.
- The slave must provide read and write functions to access resources.
- Without the `optional` keyword, you must not specify a default implementation. The presence of a default implementation causes an error.

Optional LISA+ protocol behavior without default implementation

A description of optional behavior without default implementation.

For this optional behavior declaration in a protocol:

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

- Not all slave ports that use this protocol have to implement behavior `f()`. There is no default behavior, so a call to a missing 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

  ARM recommends the use of the `implemented()` test only if the default implementation in protocol is not usable. 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 none of the slaves implements causes a run-time error.

Optional LISA+ protocol behavior with default implementation

A description of optional behavior with default implementation.

For this optional behavior definition in a protocol:

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

- Not all slave ports that use this protocol have to implement behavior `f()`. If a slave does not implement the behavior, the default implementation is used instead. The default can be `{}` if no action is required.
- If the port implements behavior `f()`, 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.
- A master does not have to 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 there is a local implementation.
- If a behavior returns a value, the default implementation can return 0 or any another value that is valid in the context of the calling function.

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

  If the return value is undefined, the compiler generates a warning if you enable such warnings. System Canvas does not issue a warning.

4.4.4 LISA+ protocol behavior ADDRESS arguments

A description of the `ADDRESS` keyword.

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

The simulator must inspect the address of the access to determine the destination slave. Specify the address as a parameter of the protocol behavior annotated with the `ADDRESS` keyword:

LISA+ protocol behavior ADDRESS syntax

```
protocol MyBusProtocol
{
    includes
    {
       // declare your own types
        #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);
}
```

- You can omit the `ADDRESS` attribute from the parameter list for the protocol behaviors.
- If present, only use the `ADDRESS` attribute 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. Explicitly specify the type of the address as an integer type of any size.
- Match the size of the integer type for each `ADDRESS` parameter for all of the behaviors in a protocol.
- It is valid to have behaviors with and without an `ADDRESS` parameter in the same protocol:
  - A behavior with an `ADDRESS` parameter is `read()`, which selects the slave based on the address.
  - A behavior without an `ADDRESS` parameter is `reset()`, which calls all connected slaves.

Appendix A
LISA+ Preprocessor

This appendix describes the C-like preprocessor statements that you can use in LISA+ source code, and how the LISA+ preprocessor works.

It contains the following sections:

- *A.1 About the LISA+ preprocessor* on page Appx-A-70.
- *A.2 LISA+ preprocessor scopes* on page Appx-A-71.
- *A.3 LISA+ preprocessing according to scope* on page Appx-A-72.
- *A.4 Predefined LISA+ symbols and macros* on page Appx-A-73.
- *A.5 LISA+ preprocessor statements* on page Appx-A-74.

A.1 About the LISA+ preprocessor

The LISA+ preprocessor is like the C preprocessor, but not identical.

As in C, the LISA+ preprocessor processes the source code before the underlying LISA+ parser sees the source code. The preprocessor statements are not part of formal LISA+, but are like a layer on top of LISA+.

Unlike C, however, the LISA+ preprocessor interacts with the actual LISA+ constructs. Deliberately, the preprocessor disables some features, for example macro expansion and includes, in certain contexts so that tools that read, modify and write LISA+ code work correctly. Consequently there are subtle restrictions to using preprocessor statements in LISA+ code. This appendix covers them.

——— **Note** ———

The preprocessor generally ignores instructions on the very first line of a LISA file.

———————————

A.2 LISA+ preprocessor scopes

LISA+ source code has different preprocessor scopes.

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

`includes` and `resources`
: All of the code between but not including the outermost opening and the closing braces, of `includes` and `resources` sections.

`behavior`
: All of the code between but not including the outermost opening and the closing braces, of `behavior` sections. All bodies of `behavior` definitions are `behavior` code. The code is C/C++ with LISA+ keyword extensions.

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

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

A.3 LISA+ preprocessing according to scope

The LISA+ preprocessor treats the scopes differently.

- It never replaces or executes any `#includes`.
- It replaces macros only inside `behavior` code, nowhere else.
- It evaluates conditional compilation everywhere and macros evaluated in `#if` expressions.
- It ignores the preprocessor constructs (`#include`, `#define`, `#if`) in non-`behavior` sections, passing them on intact to the C++ compiler, which deals with them. It makes sure that the C++ compiler sees the same set of preprocessor symbols that it saw, so that the LISA+ preprocessor and the C++ compiler perform this conditional compilation in the same way.

Note

Use LISA+ features rather than preprocessor macros to avoid redundancy in the code.

A.4 Predefined LISA+ symbols and macros

ARM defines these preprocessor version symbols for all LISA+ files in all scopes, for Fast Model Tools versions greater than or equal to 2.2.024, but not for earlier versions.

`SYSTEM_GENERATOR_MAJOR_VERSION`
Major version of the Fast Model Tools. The value is an unsigned integer, for example 2 in *System Generator* (SimGen) 2.3.044.

`SYSTEM_GENERATOR_MINOR_VERSION`
Minor version of the Fast Model Tools. The value is an unsigned integer, for example 3 in SimGen 2.3.044.

`SYSTEM_GENERATOR_REVISION`
Revision of the Fast Model Tools. The value is an unsigned integer, for example 44 in SimGen 2.3.044.

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

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

`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 SimGen 2.3.044 because 2.3.044 is greater than 2.1.057.

`SYSTEM_GENERATOR_VERSION_EQUALS(`*`major,minor,revision`*`)`
`true` (1) if the Fast Model Tools version is exactly *`major.minor.revision`*.

`linux`
`true` (1) if parsing LISA+ files or generating a simulator on Linux host systems. ARM does not define it on other host platforms.

`WIN32`
`true` (1) if parsing LISA+ files or generating a simulator on Windows host systems. ARM does not define it on other host platforms.

ARM does not define preprocessor symbols that C/C++ compilers typically predefine on certain host platforms for LISA+ files, not even in behavior bodies. For example, ARM does not define the symbols `__cplusplus` or `__GNUC__` for the preprocessing of LISA+ files. However, you can set preprocessor symbols manually in the project settings for specific host platforms.

A.5 LISA+ preprocessor statements

LISA+ preprocessor statements have the 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) are 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 because of scope. Macros defined on the LISA+ top-level do not affect conditional statements in the `includes` and `resources` sections.

`#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`

Enable or disable the code enclosed by `#if`, `#elif`, `#else`, or `#endif` blocks, 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`

Shortcuts for the `#if defined(SYM)` and `#if !defined(SYM)` statements, respectively.

`#include`

Include statements are ignored by the LISA+ preprocessor because of scope. However, `#include` statements in the `includes` sections of components have the required effect of making the declarations in the header files visible in the behavior code.

`#error`

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

`#warning`

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