



Verification Continuum™ - Platform Architect and Virtualizer

## SystemC Modeling Library Reference Manual

---

## Copyright Notice and Proprietary Information

© 2024 Synopsys, Inc. ALL RIGHTS RESERVED.

This Synopsys software and all associated documentation are proprietary to Synopsys, Inc. and may only be used pursuant to the terms and conditions of a written license agreement with Synopsys, Inc. All other use, reproduction, modification, or distribution of the Synopsys software or the associated documentation is strictly prohibited.

Licensed Products communicate with Synopsys servers for the purpose of providing software updates, detecting software piracy and verifying that customers are using Licensed Products in conformity with the applicable License Key for such Licensed Products. Synopsys will use information gathered in connection with this process to deliver software updates and pursue software pirates and infringers.

### Destination Control Statement

All technical data contained in this publication is subject to the export control laws of the United States of America. Disclosure to nationals of other countries contrary to United States law is prohibited. It is the reader's responsibility to determine the applicable regulations and to comply with them.

### Disclaimer

SYNOPSYS, INC., AND ITS LICENSORS MAKE NO WARRANTY OF ANY KIND, EXPRESS OR IMPLIED, WITH REGARD TO THIS MATERIAL, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE.

### Trademarks

Synopsys and certain Synopsys product names are trademarks of Synopsys, as set forth at

<https://www.synopsys.com/company/legal/trademarks-brands.html>.

All other product or company names may be trademarks of their respective owners.

### Free and Open-Source Software Licensing Notices

If applicable, Free and Open-Source Software (FOSS) licensing notices are available in the product installation.

### Third-Party Links

Any links to third-party websites included in this document are for your convenience only. Synopsys does not endorse and is not responsible for such websites and their practices, including privacy practices, availability, and content.

[www.synopsys.com](http://www.synopsys.com)

# Contents

---

Preface .....	9
About This Manual .....	9
Documentation Conventions .....	10
Terminology .....	12
References .....	12
Customer Support .....	12
Synopsys Statement on Inclusivity and Diversity .....	13
 Chapter 1	
Introduction .....	15
1.1 SCML2 Introduction .....	15
1.2 Header Files .....	15
1.3 SCML2 FT Modeling Interfaces .....	15
1.3.1 Introduction to TLM2.0 .....	15
1.3.2 Extending TLM2.0 Base Protocol .....	18
1.4 SCML2 Modeling Objects .....	19
1.4.1 Storage Modeling Objects .....	19
1.4.2 Timing and Synchronization .....	21
1.4.3 Utility Objects .....	22
 Chapter 2	
Memory Objects .....	25
2.1 Introduction .....	25
2.2 Overview .....	25
2.2.1 Transaction Routing from Socket to Memories .....	25
2.2.2 Memory Map Modeling .....	26
2.2.3 Properties and Attributes .....	26
2.2.4 Behavior .....	26
2.2.5 Callbacks .....	27
2.2.6 Access Restrictions .....	29
2.2.7 Vectors of Memory Objects .....	34
2.3 memory .....	34
2.3.1 Types .....	35
2.3.2 Constructors .....	35
2.3.3 Properties .....	36
2.3.4 Behaviors .....	37
2.3.5 Callbacks .....	39
2.3.6 Access Restrictions .....	40
2.4 memory_alias .....	40
2.4.1 Types .....	40
2.4.2 Constructors .....	41
2.4.3 Properties .....	41

2.4.4 Behaviors .....	42
2.4.5 Callbacks.....	45
2.4.6 Access Restrictions .....	45
2.5 reg .....	46
2.5.1 Types.....	46
2.5.2 Constructors.....	46
2.5.3 Properties .....	46
2.5.4 Behaviors .....	47
2.5.5 Callbacks.....	51
2.5.6 Access Restrictions .....	53
2.6 bitfield .....	53
2.6.1 Types.....	54
2.6.2 Constructors.....	54
2.6.3 Properties .....	55
2.6.4 Behaviors .....	55
2.6.5 Callbacks.....	57
2.6.6 Access Restrictions .....	59
2.7 router .....	59
2.7.1 Types.....	59
2.7.2 Constructors.....	60
2.7.3 Properties .....	60
2.7.4 Behaviors .....	61
2.7.5 Callbacks.....	63
2.8 memory utilities .....	64
2.8.1 memory_index_reference.....	64
2.8.2 mappable_if .....	65
2.8.3 Callback Base Classes .....	66
2.8.4 Convenience Functions .....	68
2.9 Deprecated API's and Adapters .....	68
2.9.1 Callbacks.....	68
2.9.2 TLM2 Adapters .....	71
2.9.3 tlm2_gp_target_adapter.....	71
2.9.4 tlm2_gp_initiator_adapter .....	73

## Chapter 3

FT Model Interface APIs and Objects .....	75
3.1 Modeling Objects .....	75
3.1.1 Payloads .....	75
3.1.2 Sockets.....	77
3.1.3 Port Adaptors.....	77
3.1.4 Protocol States .....	97
3.1.5 Alignment in FT Protocols .....	98
3.2 Protocol Definitions .....	99
3.2.1 FT GFT Protocol Definition .....	99
3.2.2 FT AXI Protocol Definition .....	107
3.2.3 FT ACE Protocol Definition .....	116
3.2.4 FT AXI4 Stream Protocol Definition.....	118
3.2.5 FT CHI Protocol Definition .....	121
3.2.6 FT PCIe Protocol Definition .....	123
3.2.7 FT CXL Protocol Definition .....	126

<b>3.3 API Definitions .....</b>	<b>130</b>
3.3.1 FT GFT API Definition .....	130
3.3.2 FT AXI API Definition .....	134
3.3.3 FT ACE API Definition .....	141
3.3.4 FT AXI4 Stream API Definition.....	143
3.3.5 FT CHI API Definition .....	145
3.3.6 FT PCIe API Definition.....	149
3.3.7 FT CXL API Definition .....	149
<b>3.4 Protocol Checker.....</b>	<b>150</b>
3.4.1 Introduction .....	150
3.4.2 Features.....	150
3.4.3 Input Requirements .....	151
3.4.4 Getting Started.....	151
<b>Chapter 4</b>	
<b>Clock Objects.....</b>	<b>157</b>
4.1 Overview .....	157
4.2 Clocks and Reset.....	158
4.2.1 scml_clock .....	158
4.2.2 scml_divided_clock.....	162
4.2.3 Dynamic Clock Parameter Change and Reset .....	163
4.3 Modeling Objects for Clocks (Clock Objects).....	166
4.3.1 scml_clock_gate.....	166
4.3.2 scml_clock_counter.....	166
4.4 Base Classes .....	167
4.4.1 scml2::clocked_module .....	167
4.5 Modeling Objects for Base Classes (Modeling Objects) .....	169
4.5.1 scml2::clocked_timer.....	169
4.6 Convenience Classes .....	171
4.6.1 scml2::clocked_callback .....	171
4.6.2 scml2::clocked_event .....	173
4.7 Modeling Objects for Convenience Classes (Convenience Objects) .....	175
4.7.1 scml2::clocked_peq_container.....	175
4.7.2 scml2::clocked_peq .....	179
4.8 Code Example .....	183
4.8.1 Programmable Clock Peripherals.....	183
<b>Chapter 5</b>	
<b>Pulse TLM Modeling .....</b>	<b>185</b>
5.1 The Pulse Interface .....	185
5.1.1 The Pulse Definition Interface.....	186
5.1.2 The Pulse Observer Interface.....	189
<b>Chapter 6</b>	
<b>Modeling Utilities .....</b>	<b>191</b>
6.1 Port Utilities.....	191
6.1.1 dmi_handler.....	191
6.1.2 initiator_socket .....	192
6.1.3 Pin Callback Functions.....	195
6.1.4 Utility APIs.....	196

6.2 Commands . . . . .	197
6.2.1 scml_command_processor . . . . .	197
6.2.2 scml_loader . . . . .	199
6.3 Parameters . . . . .	200
6.3.1 scml_property . . . . .	200
6.3.2 scml_property_registry . . . . .	202
6.3.3 scml_property_server_if . . . . .	204
6.3.4 scml_simple_property_server . . . . .	206
6.4 Reporting . . . . .	207
6.4.1 status . . . . .	207
6.4.2 stream . . . . .	207
6.4.3 severity . . . . .	209
6.5 FastTrack . . . . .	209
6.5.1 FastTrack API . . . . .	210
6.5.2 FastTrack Categories . . . . .	212
6.5.3 Implicit FastTrack Messages . . . . .	216
6.5.4 Suppressing FastTrack Messages . . . . .	217

**Chapter 7**

Functional Coverage . . . . .	221
7.1 Coverage Semantics . . . . .	221
7.1.1 Functional Coverage Constructs . . . . .	222
7.1.2 Functional Coverage Exemptions . . . . .	222
7.1.3 Functional Coverage Calculation . . . . .	223
7.2 SCML Functional Coverage Reference . . . . .	224
7.2.1 Covergroup . . . . .	224
7.2.2 Coverage Point Base Class . . . . .	224
7.2.3 Storage Coverage Points . . . . .	225
7.2.4 Parameter and Status Coverage Points . . . . .	227
7.2.5 Clock Coverage Point . . . . .	227
7.2.6 Signal Port Coverage Point . . . . .	228
7.2.7 TLM Socket Coverage Point . . . . .	228
7.2.8 Generic Function Coverage Point . . . . .	228
7.2.9 Coverbins . . . . .	229
7.2.10 Default Bin . . . . .	230
7.3 Examples . . . . .	231

**Chapter 8**

Modeling Guidelines . . . . .	233
8.1 Requirements for a Virtual Prototype Model . . . . .	233
8.2 Virtual Prototype Model Content . . . . .	234
8.3 Introduction to SCML FT Modeling . . . . .	237
8.3.1 SystemC Transaction-Level Modeling . . . . .	237
8.3.2 Use Cases . . . . .	238
8.3.3 Fast Timed Modeling (FTM) Coding Style . . . . .	240
8.3.4 Modeling Concepts . . . . .	243
8.3.5 Creating SCML FT Models . . . . .	246
8.4 The SCML Modeling Guidelines for LT . . . . .	250
8.4.1 Modeling Methodology Guidelines . . . . .	250
8.4.2 Coding Style Guidelines . . . . .	256

8.5 Synchronization and Modeling for Speed .....	259
8.5.1 LT-Centric Simulation Techniques Overview .....	259
8.5.2 Debugging Temporally Decoupled Systems. ....	262
8.5.3 Modeling Fast Target and Router Peripherals ..	265
8.5.4 Optimizing Simulation Performance for FT Models ..	269
8.6 Getting Started .....	281
8.6.1 Modeling a Memory .....	282
8.6.2 Modeling an Interrupt Controller.....	289
8.6.3 Modeling a Watchdog Peripheral.....	301
8.6.4 Modeling a DMA. ....	316
8.6.5 Modeling a Cache .....	324
8.6.6 Example Timer Specification.....	333
Index .....	337



# Preface

---

The preface of the *SystemC Modeling Library Manual* describes:

- [About This Manual](#)
- [Documentation Conventions](#)
- [Terminology](#)
- [References](#)
- [Customer Support](#)
- [Synopsys Statement on Inclusivity and Diversity](#)

## About This Manual

This manual describes SystemC Modeling Library 1 (SCML1) and SystemC Modeling Library 2 (SCML2) modeling objects. It also provides the guidelines on how to use SCML to create virtual prototype models. SCML stands for *SystemC Modeling Library*; it is a C++ library of modeling components built on top of TLM2.0 and SystemC, which are modeling libraries as well.

It is assumed that you have some knowledge of SystemC.

This manual is organized as follows:

- [Introduction](#) gives an overview of the modeling objects and describes header files to be included.
- [Memory Objects](#) describes the SystemC Modeling Library (SCML) modeling objects.
- [FT Model Interface APIs and Objects](#) discusses the different FT modeling interfaces and objects.
- [Clock Objects](#) describes the SystemC Modeling Library (SCML) clock objects.
- [Pulse TLM Modeling](#) describes the pulse interface which is the basis for Pulse TLM modeling.
- [Modeling Utilities](#) describes the SCML modeling utilities.
- [Functional Coverage](#) describes the SCML functional coverage reference for SystemC TLM modeling.
- [Modeling Guidelines](#) explains the guidelines for FT and LT modeling.

## Documentation Conventions

This section lists and explains the documentation conventions used throughout this manual.

Convention	Description and Examples
<i>italic</i>	<p>Is used in running text for:</p> <ul style="list-style-type: none"><li>• GUI elements. For example: The <i>Enumeration</i> field contains a space-separated list of values.</li><li>• New terms. For example: A <i>protocol library</i> is a collection of protocol definitions.</li><li>• Web sites. For example: For more information, see <a href="http://www.eclipse.org">www.eclipse.org</a>.</li><li>• E-mail addresses. For example: Contact customer support via e-mail at <a href="mailto:vp_support@synopsys.com">vp_support@synopsys.com</a>.</li><li>• Manual names. For example: The preface of the <i>Analysis Manual</i> describes:</li></ul>
courier	<p>Is used for:</p> <ul style="list-style-type: none"><li>• Code text. For example: <pre>list_library_configurations myConfig</pre>In this example, myConfig is used.</li><li>• System messages. For example: JVM not found.</li><li>• Text you must type literally. For example: At the prompt, type go.</li><li>• Names (of environment variables, commands, utilities, prompts, paths, macros, and so on). For example: The build-options command sets build parameters.</li></ul>
<i>courier italic</i>	Indicates variables. For example: <i>scope</i> specifies a module, a channel, or a refined port.
<b>bold</b>	Serves to draw your attention to the text in question. For example: <pre>coreId = cwrSAGetCoreId("mycore");</pre>
[ ]	<p>Square brackets enclose optional items. For example: <pre>clean [-pch]</pre>If you must type a square bracket as part of the syntax, it is enclosed in single quotes. For example: <pre>'[ '--use-vector' ]'</pre></p>

Convention	Description and Examples
{ }	<p>Braces enclose a list from which you must choose one or more items. For example:</p> <pre data-bbox="470 297 1062 329">add {signalPattern   portPattern} ID</pre> <p>If you must type a brace as part of the syntax, it is enclosed in single quotes. For example:</p> <pre data-bbox="474 403 654 530">DECLARE '{' Item1 Item1 }'</pre>
	<p>A vertical bar separates items in a list of choices. For example:</p> <pre data-bbox="474 593 801 625">autoflush {on   off}</pre>
>	<p>A right angle bracket separates menu commands. For example:</p> <p>The <i>Project &gt; Update System Library</i> menu command is available.</p>
...	<p>A horizontal ellipsis in syntax indicates that the preceding expression may have zero, one, or more occurrences. For example:</p> <pre data-bbox="474 830 1057 861">build-options -option optionArgs ...</pre> <p>A horizontal ellipsis in examples and system messages indicates material that has been omitted. For example:</p> <pre data-bbox="474 973 1057 1094">::scsh&gt; dtrace add top1.signal_* \$t1 ::scsh&gt; dtrace add top1.clk_* \$t1 ... ::scsh&gt; dtrace flush *</pre>

## Terminology

<b>API</b>	Application Programmer's Interface
<b>ASI TLM WG</b>	Accellera Systems Initiative Transaction-Level Modeling Work Group
<b>AT</b>	In the context of PV, AT stands for Address Type. In the context of TLM2, AT stands for Approximately Timed.
<b>AV</b>	Architect's View
<b>CXL</b>	Compute Express Link
<b>DMA</b>	Direct Memory Access
<b>DT</b>	Data Type
<b>FIFO</b>	First In First Out
<b>IP</b>	Intellectual Property
<b>LT</b>	Loosely Timed
<b>PODT</b>	Plain Old Data Type
<b>PCI Express</b>	Peripheral Component Interconnect Express
<b>PV</b>	Programmer's View
<b>PWM</b>	Pulse Width Modulation
<b>SCML1</b>	SystemC Modeling Library 1
<b>SCML2</b>	SystemC Modeling Library 2
<b>STL</b>	Socket Transaction Language
<b>TLM</b>	Transaction-Level Modeling
<b>VPU</b>	Virtual Processing Unit

## References

This manual focuses on the use of SystemC, TLM2.0, and SCML for the creation of virtual prototype models. For more details on other use cases and detailed semantics of the libraries, see the following manuals:

- *SystemC Language Reference Manual*, IEEE standard 1666
- *IEEE Std. 1666 TLM-2.0 Language Reference Manual*

For details on interconnect components and the integration of processor models, see the *Integrating Third-Party Instruction-Accurate Models* manual.

## Customer Support

For technical support (regarding license keys, IP downloads, Host ID, Project ID, documentation or general support), contact the Support Center with a description of your question and supplying the debug information, using one of the following methods:

- Go to <https://solvnetplus.synopsys.com> and sign-in with your Synopsys SolvNetPlus credentials. Select *Cases* from the menu bar, and select *Create a New Case*. Provide the requested information, including:

- **Product L1:** Virtual Prototyping.
- **Product L2:** Select the product type that closest matches yours.
- **Case Type:** Select the case type from the drop-down menu.
- **Case Severity:** Select the case severity from the drop-down menu.
- **Subject:** Provide a brief summary of the issue or list the error message you have encountered.
- **Description:** For simulation issues, include the timestamp of any signals or locations in waveforms that are not understood.

After creating the case, attach the debug files you have created, if any.

- Or, send an e-mail message to [vp\\_support@synopsys.com](mailto:vp_support@synopsys.com). (your email will be queued and then, on a first-come, first-served basis, manually routed to the correct support engineer):
  - Include the Product name, Sub Product name, and Tool Version number in your email; so that it can be routed correctly.
  - For simulation issues, include the timestamp of any signals or locations in waveforms that are not understood.
  - Attach any debug files you created in the previous step.
- Or, telephone your local support center:
  - North America:  
Call 1-800-245-8005 from 7:00 AM to 5:30 PM Pacific time, Monday through Friday.
  - All other countries:  
<https://www.synopsys.com/support/global-support-centers.html>

## Synopsys Statement on Inclusivity and Diversity

Synopsys is committed to creating an inclusive environment where every employee, customer, and partner feels welcomed. We are reviewing and removing exclusionary language from our products and supporting customer-facing collateral. Our effort also includes internal initiatives to remove biased language from our engineering and working environment, including terms that are embedded in our software and IPs. At the same time, we are working to ensure that our web content and software applications are usable to people of varying abilities. You may still find examples of non-inclusive language in our software or documentation as our IPs implement industry-standard specifications that are currently under review to remove exclusionary language.



# Chapter 1

## Introduction

This chapter describes:

- [SCML2 Introduction](#)
- [Header Files](#)
- [SCML2 FT Modeling Interfaces](#)
- [SCML2 Modeling Objects](#)

### 1.1 SCML2 Introduction

SCML2 is an easy-to-use abstraction layer on top of SystemC and TLM2. It hides a lot of the complexity and common code that is required to correctly manage TLM2 transactions and it provides with modeling objects that handle common aspects of Virtual Prototype modeling.

### 1.2 Header Files

All SCML2 header files can be included by including `scml2.h`. The SCML2 headers enable the Memory objects as well as the FT Model interface APIs and objects.

All SCML2 logging header files can be included by including `scml2_diagnostics.h`.

The utility objects as well as the deprecated modeling objects are available after the `scml.h` file has been included.

```
#include "scml.h"
```

The clock modeling objects are available after the `scml_clock.h` file has been included.

```
#include "scml_clock.h"
```

### 1.3 SCML2 FT Modeling Interfaces

#### 1.3.1 Introduction to TLM2.0

The SCML2 FT modeling interfaces are based on the TLM2.0 standard, but extended to provide the support for additional hardware protocols while maintaining maximal interoperability and increased ease of use.

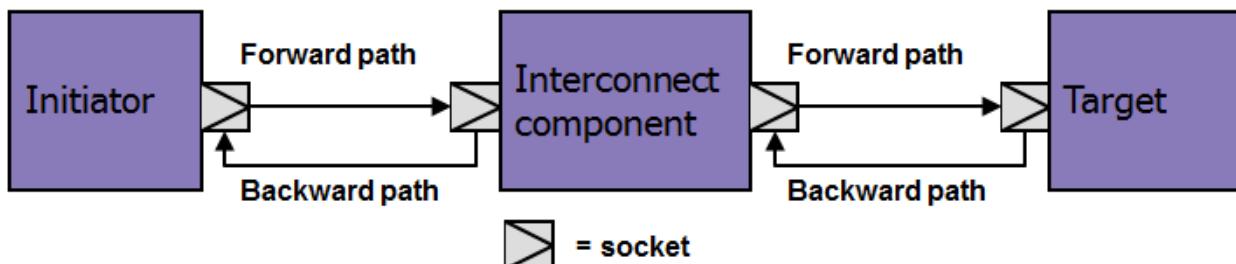
The basic communication elements defined in TLM2.0 are based on interfaces and sockets.

- *TLM2.0* defines communication between components through a combination of a *forward path* and a *backward path*. These paths are implemented as interface method calls, as supported in the IEEE SystemC standard. This approach allows both the initiator of a transaction as well as the target to control the progress and timing of a transaction.

- SystemC defines ports and exports as the endpoints of a communication connection, each capable of initiating or implementing a communication interface. TLM2.0 extends this with sockets which are a combination of a port and an export so that the combined forward and backward path can be tied together via a single communication connection.

Combining these together leads to components that have sockets as end points for communication, where each socket enables both the forward path and backward path interface definitions. By definition, a TLM2.0 protocol defines a point to point communication interface. Hence, it should always be possible to connect any TLM2.0 initiator to a TLM2.0 target as long as they use the same protocol definition. This is graphically represented below.

**Figure 1-1 TLM2.0 Background**



The interface definitions themselves each define a set of method calls:

- In the forward part, the following methods are defined:

```
void b_transport(TRANS &trans, sc_core::sc_time &t)
```

- **b\_transport**: Is used in the loosely timed coding style to implement a transaction exchange between initiator and target. When the call returns, the transaction is finished. There are two timing points: the start and end of a transaction. The call is blocking, which means the initiator should start the call from a `sc_thread`. The target is allowed to call `wait` in order to implement a delay. There is also a `sc_time` argument in the call which allows to annotate the timing delay for the end point of the transaction. The other argument of the call is the transaction payload.

```
enum tlm_sync_enum { TLM_ACCEPTED, TLM_UPDATED, TLM_COMPLETED} ;

tlm_sync_enum nb_transport_fw(TRANS &trans, PHASE &phase, sc_core::sc_time &t)
```

- **nb\_transport\_fw**: Is used in the approximately timed coding style and implements part of a transaction exchange between initiator and target. The call is non-blocking, which means that it is not allowed to call `wait` in the implementation, so that it can be called from an `sc_method`. Also, this call has an `sc_time` argument to allow timing annotation. The other arguments are payload and phase, where phase indicates the timing point or state of the protocol state machine. The return value is an indicator whether the payload and phase have been updated by the interface implementation before returning the call, or whether the transaction is completed.

```
bool get_direct_mem_ptr(TRANS& trans, tlm_dmi& dmi_data)
```

- **get\_direct\_mem\_ptr**: This call is used by the initiator to request direct access to the data storage that corresponds to the payload access request. The argument `dmi_data` is a data structure specific to make DMI requests, which is setup by the initiator and completed by the target in case DMI is allowed (which is indicated by the return value).

```
unsigned int transport_dbg(TRANS& trans)
```

- `transport_dbg` interface: This debugging call is used by the initiator to access storage in the system, all interconnect and target models should implement the debug interface so that there are no side-effects (no delay, event notifications, state change and so on).

In the backwards path the following methods are defined:

```
tlm_sync_enum nb_transport_bw(TRANS &trans, PHASE &phase, sc_core::sc_time &t)
```

- `nb_transport_bw`: This is the corresponding interface for the `nb_transport_fw`, now called from the target side of the connection.

```
void invalidate_direct_mem_ptr(sc_dt::uint64 start_range,  
                               sc_dt::uint64 end_range)
```

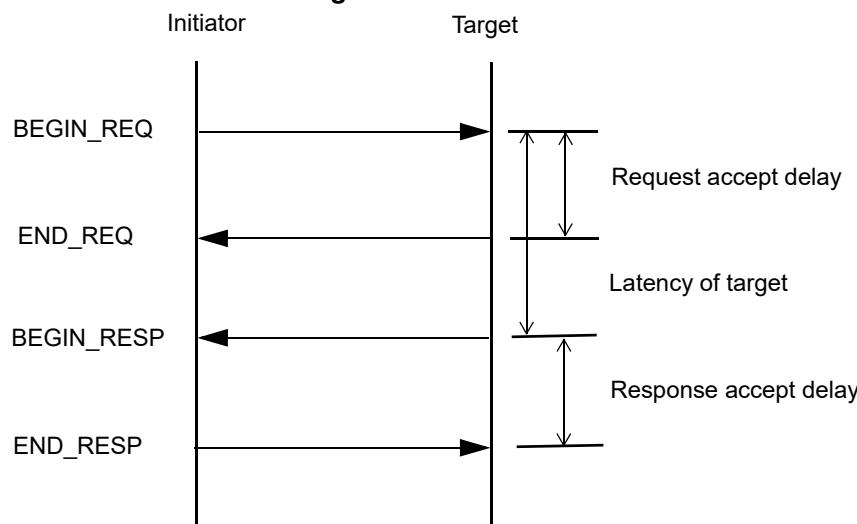
- `invalidate_direct_mem_ptr`: A method used by the target side of a connection to indicate that direct memory access to a certain address region is no longer allowed. This could be caused by a remap or reprogramming of the address regions in the interconnect components.

TLM2.0 also defines a base protocol, which contains a generic memory-mapped payload and defines a set of generic protocol timing points to be used for this protocol in the AT coding style. The payload definition contains address, data, data\_length, command, response and more attributes so that a generic memory mapped data transfer can be modeled. There are four timing points defined for the base protocol, they represent two transfers or basic exchanges between the initiator and the target.

- `BEGIN_REQ, END_REQ`: Represents the start of the transaction and indicates a request by the initiator to exchange data. The target responds when it is ready to accept the data.
- `BEGIN_RESP, END_RESP`: Represents the completion of the transaction and indicates the response by the target that it has consumed the transaction. The initiator responds when it has accepted the completion of the transaction.

The base protocol only allows one REQ or RESP transfer at a time, so an initiator cannot start another transaction until it has received an `END_REQ` or a target cannot complete an new transaction until it has received the `END_RESP` of the previous one. This is graphically represented below.

**Figure 1-2 Initiator and Target**



With these combined interfaces, a TLM2.0 socket supports both an LT and AT modeling style, even more: every TLM2.0 component is required to support both modeling styles. The FT coding style further builds on this aspect of the TLM2.0 standard, but provides ease of use and interoperability features through specialized sockets and payload definitions. The TLM2.0 interfaces are untouched and all modifications are done in a TLM2.0 compatible way so that interoperability with strict TLM2.0 components remains.

### 1.3.2 Extending TLM2.0 Base Protocol

The TLM2.0 base protocol is not intended to resemble any real life hardware protocol, but is intended as a generic interface. Key problems for the TLM2.0 base protocol are:

- It does not provide with timing points for the individual data beats of a burst transfer.
- It requires all address and data information to be available for writes at the start of the transaction.
- It is not possible to have concurrent read and write requests.

Therefore, it is required to have dedicated payload and phases to model a hardware protocol. The TLM2.0 standard intends that the base protocol can serve as bases to create payload and phase definitions that accurately model a real hardware protocol. The TLM2.0 standard also provides some basic rules and advice on how to do this by defining mandatory and ignorable extensions to the TLM2.0 base protocol. Extensions are additional payload attributes that can be added to the standard payload to add the additional features of real hardware protocols. It is also possible to extend or overwrite the phase definitions that are used in the non-blocking transport API calls. The standard requires mandatory extensions whenever a component is required to recognize and react to a payload or phase extension. The drawback of mandatory extensions is that they imply a change to the types used in the interfaces of the TLM2.0 sockets, which in turn makes it impossible to connect components with different mandatory extensions with each other. The benefit of this approach is that incompatible components are easily recognized.

The FT modeling interfaces take a different approach to this problem. Transactors or protocol conversion blocks are a major modeling pain when assembling a system and also a major cause for performance issues. Therefore, the FT modeling approach starts by requiring that all interfaces remain compatible with the TLM2.0 base protocol and rely on extended sockets and payload to provide the necessary protocol conversion logic so that conversions are only done when they are required and can be inserted automatically.

As a consequence the FT modeling interfaces use ignorable payload extensions:

- Each protocol defines a protocol state extension. This is a payload extension carrying the actual protocol timing label, indicating the current state in the protocol state machine.
- For each protocol, there can be protocol-specific attributes added via an extension. This should be limited to those attributes that do not map to the TLM2.0 base protocol. These extensions are ignorable in the sense that a model should assume they have a default value in case they are not present in the payload. At the same time, they are non-ignorable since a model can ignore the TLM2.0 base protocol semantics that are overruled by the protocol specific attributes.

There is one ignorable phase extension defined for the TLM2.0 base protocol: `FT_TIMING`. This is presented as an additional timing point indicating that the protocol state extension should be used to determine the actual current protocol state.

The FT modeling interfaces are compatible with the TLM2.0 standard in the sense that through the protocol conversion mechanisms, any TLM2.0 base protocol component will never notice that it is interfacing with a component that internally does not use the base protocol phases or parts of the payload.

This protocol conversion relies on an extended socket that has one additional user API namely `set_protocol`. This API indicates the protocol extensions that will be used inside the model. At the start of the simulation, the initiators and target sockets will check what protocol is used by the socket they are connected to and insert a protocol conversion if available. A protocol conversion to the TLM2.0 base protocol should always be available. When defining an FT protocol interface, the base protocol mapping is a part of the definition. Conversion from and to AXI and GFT protocol is also available as part of the extended socket semantics.

## 1.4 SCML2 Modeling Objects

SCML2 contains a wide range of modeling objects addressing different aspects of modeling. They are discussed in more detail in the coming sections.

### 1.4.1 Storage Modeling Objects

The storage modeling objects in SCML2 have been created in order to abstract and hide some of the tedious details of the TLM2.0 APIs while providing a simple mechanism to describe the memory map of a peripheral and to develop the behaviors that are associated with the different registers and bitfields of the component.

The following table provides an overview of the SCML2 modeling objects.

**Table 1-1 SCML2 Modeling Objects**

Modeling Object	Description
<code>memory</code>	Models memories and register files.
<code>memory_alias</code>	Models an alias for a memory region of another <code>memory</code> or <code>memory_alias</code> object.
<code>reg</code>	A register; it models a <code>memory_alias</code> object of size 1.
<code>bitfield</code>	Models an alias for a number of consecutive bits in a <code>reg</code> object.
<code>router</code>	Models a dynamic address decoder that can map a memory region to a region in another <code>memory</code> , <code>router</code> , or <code>tlm2_gp_initiator_adapter</code> object.

All the modeling objects and global functions are part of the `scml2` namespace.

The following table provides an overview of the SCML helper functions.

**Table 1-2 SCML2 Helper Functions**

Helper Functions	Description
<code>memory_index_reference</code>	Is an intermediate object used when dealing with indices on <code>memory</code> and <code>memory_alias</code> objects.
<code>mappable_if</code>	Is the definition of the interface to the memory objects. It is used by the target adapter and the router object.

These modeling objects are structured hierarchically as follows:

- Memories:
  - Memory: Is the top-level entry for a component memory map. It is the only modeling object that maintains actual storage. This modeling object can have associated behavior.

- `Memory_alias`: Refers to a section of the memory and can be used to define specialized behaviors per region or to provide logical names for different sections in the memory map.
- `Reg`: Is an alias of size one word.
- `Bitfield`: Is a subword region.
- `Router`: is used to model programmable forwarding of memory accesses to different internal memories or initiator ports.

The memory storage objects have a default behavior that corresponds to the TLM2.0 blocking transport and debug APIs, plus they implement the `get_direct_memory_ptr` interface. Memories can be connected to the FT sockets via an adapter which implements a couple of conversions to make sure the SCML2 memory semantics can remain simple and also takes care of the non-blocking to blocking transport conversion if needed. Through this combination of features, it becomes extremely simple to build a memory model. Simply instantiating an SCML2 memory and connecting it to a TLM2.0 target socket via an adapter is sufficient. All TLM2.0 APIs are taken care of automatically and are of no worry to the developer. Adding aliases and registers allows the creation of a meaningful memory map for the component. All address decoding will be automatically taken care of. Finally by registering callbacks to implement the register behavior it is very easy to construct the functional model of a component.

There are a set of default behaviors available that can be registered with the memory objects. These are behaviors like: `ignore`, `read_only`, `write_only`, `clear_on_read`, `word_only`, `error`, `set_on_read`, `clear_on_write_1`, `clear_on_write_0`, `set_on_write_1`, `set_on_write_0`. Some of these are only available for the register objects. For a complete list and API details, see "[Memory Objects](#)" on page 25.

As already mentioned, it is possible to override the default behavior (which is storage) by registering a callback with a storage object. The following callback types are available:

- `Transport callback`: This is a callback that reuses the TLM2.0 transport API arguments and gives full control over the interpretation and handling of the transaction payload. This can be used when additional extensions of the payload need to be accessed which are not handled by the storage objects.
- `Read and Write callbacks`: are callbacks that only override the read or write behavior. Various versions of these callbacks exist: they have a data pointer as argument, but variants with and without `byte_enables`, `sc_time` argument exist. There is also a tagged version that can be used to pass the index of the memory element that is accessed into the callback. The return value of the callbacks is the `tlm_response_status` attribute as used in the payload. The storage object will pass this back via the transaction payload.
- `Debug`: An API to override the default debug behavior of that storage object.

Callbacks can be registered and unregistered so that context-specific behavior can be enabled, or also as speed optimization when there is only need to have a certain behavior when the component is in a specific state. Callbacks disable DMI behavior and as a consequence have an impact on the simulation speed. This is typically not an issue since callbacks are typically associated with peripheral components that are accessed infrequently (compared to memory and caches).

It is possible to create additional "custom" callbacks that can be reused for different modules by creating a class that derives from `scml2::memory_callback_base` and that implements the `execute(payload, sc_time)` interface.

Callbacks are related to the LT coding style, this means it is allowed to call `wait()` in the callback implementation and there are different synchronization possibilities that can be indicated when a callback is registered.

- `NEVER_SYNCING`: This means that the call is non blocking by nature and that it will never call `wait`.

- SELF\_SYNCING: This means the callback is blocking and might call wait.
- AUTO\_SYNCING: In this case, the callback is blocking and may call wait. The memory object will synchronize with the SystemC time before calling the callback. The timing annotation passed to the callback will always be SC\_ZERO\_TIME.

For a more detailed overview of every callback and access function that is available for the storage objects, see “[Memory Objects](#)” on page 25.

#### 1.4.2 Timing and Synchronization

The timing and synchronization modeling objects are mostly related to the clocked modeling style and provide ease of use features to align the execution of models with clock boundaries since timing annotation on the TLM2.0 interfaces is not guaranteed to be aligned with the internal clock of a model.

The FT modeling style defines a set of additional clock objects and interfaces, but is created such that compatibility with the traditional SystemC clocked modeling style is maintained. Traditionally in SystemC, a clock connection is represented by a boolean signal. This means that clock ports will be represented by a `sc_in<bool>` port in SystemC. The FT modeling style maintains this style. So all clock connections are preferably done through signals and `sc_in/out<bool>` ports.

For full compatibility, clock generators should provide with an `scml_clock` object (see below) to provide the signal interface on the ports. This coding pattern allows to mix and match components that use a traditional SystemC clocked modeling style and the FT clock modeling style.

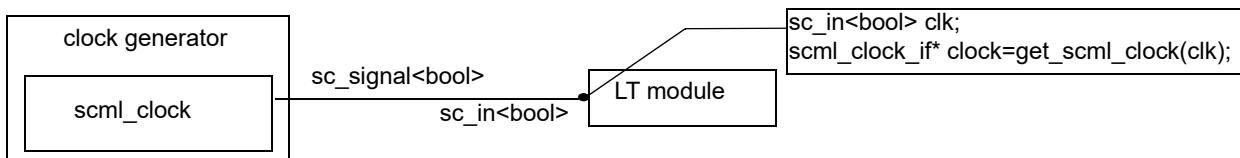
To retrieve the incoming clock from an `sc_in<bool>` input, the following API is provided:

```
// sc_in<bool> -> SCML clock
scml_clock_if* scml2::get_scml_clock(sc_core::sc_in<bool>& clk,
                                      bool allow_stubbed=false);
```

The API will retrieve the corresponding `scml_clock_if*` for the incoming clock. The second argument `allow_stubbed` controls the behavior if the signal interface is not bound to an SCML clock. If it is `false` or not provided and no SCML clock could be retrieved from the signal interface, then the API will fail by printing an error message and terminating the simulation. If `allow_stub` is set to `true`, it is possible to stub the input signal by binding it to any `bool` output port. In that case, the API will return a NULL pointer when no clock could be retrieved from the signal interface.

This coding pattern is supported by the GenericCIPLib clock generator that is provided with Platform Creator. It is very much advised that each component that has a clock output follows the same pattern to ensure that there is a signal interface as well as an `scml_clock` object available for each clock connection.

**Figure 1-3 The Coding Pattern**



The following clocked modeling objects are provided for the FT modeling style:

**Table 1-3 Clocked Modeling Objects**

Modeling Object	Description
scml_clock	Implements <code>sc_clock_if</code> . It is an optimized version of <code>sc_clock</code> .
scml_clock_gate	This is a module which takes a clock and an enable as inputs and produces a gated clock as output.
scml_divided_clock	This is a clock derived from another clock by multiplying the start time and/or the period with specified integer factors.
clocked_module	This is the base class for modules that want to receive SCML clock tick callbacks.
clocked_callback	This is a convenience class that forwards a clock tick callback to any member function of a module without the need to inherit from the <code>clocked_module</code> base class.
clocked_timer	This is a modeling object that provides a timer callback mechanism based on an SCML clock.
clocked_event	This is a convenience class that allows a SystemC method or thread to wait until a certain clock tick happens.
clocked_peq_container	This is a modeling object for TLM2 FT models using the non-blocking APIs. It buffers payload arriving in the model, like multiple outstanding transactions, possibly coming with different timing annotations from different initiators.
clocked_peq	This is a modeling object similar to the <code>clocked_peq_container</code> that can trigger a callback whenever an element from the payload buffer becomes available.

A detailed reference documentation of the clocked modeling objects can be found in “[Clock Objects](#)” on [page 157](#).

#### 1.4.3 Utility Objects

SCML2 also contains a set of ease of use objects that deal with various aspects of creating a component model. Since they do not really fit one of the main categories (timing and synchronization) an overview for all of them is given in this section.

The following table provides an overview of the SCML2 utility objects.

**Table 1-4 SCML2 Utility Objects**

Modeling Object	Description
<code>tlm2_gp_target_adapter</code>	Allows a <code>memory</code> object to bind to a <code>tlm2_target_socket</code> .
<code>tlm2_gp_initiator_adapter</code>	Allows a <code>router</code> object to map a memory region to a region on a <code>tlm2_initiator_socket</code> .
<code>dmi_handler</code>	Is a convenience object to do the bookkeeping of DMI pointers.
<code>initiator_socket</code>	Is a convenience socket that first tries to do a DMI access before doing a bus access.
<code>status</code>	Is a simple object that holds a status value in string format.
<code>stream</code>	Is the front-end object of SCML2 logging library. It formats the output and sends it to the back-end logger objects for processing.

Modeling Object	Description
<code>severity</code>	Holds a severity name and a value. Lower severity values mean a higher severity level.

For more details on the objects, see “[Memory Objects](#)” on page 25 and “[Modeling Utilities](#)” on page 191.



# Chapter 2

# Memory Objects

This chapter describes:

- [Introduction](#)
- [Overview](#)
- [memory](#)
- [memory\\_alias](#)
- [reg](#)
- [bitfield](#)
- [router](#)
- [memory utilities](#)
- [Deprecated API's and Adapters](#)

## 2.1 Introduction

The SCML memory objects serve several purposes:

- They provide with a default implementation of the TLM2 interfaces and generic payload.
- They provide with a mechanism to describe the memory map of a component.
- They should be used to model simple storage objects in a component.
- They have the necessary hooks to model the behaviors that are associated with an access to the elements in the memory map of a component.

The SCML memory objects only implement the Loosely Timed API's of the TLM2 interface, so they cannot interface directly with the TLM2 sockets, a `port_adapter` is required to handle timing and protocol conversion features of FT models (see “[Port Adaptors](#)” on page 77).

The interface used by the SCML memory objects is the `mappable_if` (see “[mappable\\_if](#)” on page 65).

## 2.2 Overview

This section provides a short overview of the different memory objects and their common features.

### 2.2.1 Transaction Routing from Socket to Memories

The `scml2::memory` is the top-level object to model the internal storage of a component, the other objects specify sub-regions within a memory. An `scml2::memory` can be bound to a target socket via a `port_adapter`. Multiple target sockets can be bound, each with its own `port_adapter`, to the same memory. All target sockets will see the memory at the same address.

To model multiple memory regions in a component, or when storage is seen at different locations from different sockets, an `scml2::router` can be used.

The `scml2::router` object is used to implement local address decoding and/or transaction forwarding in a model. A `router` can be bound to a target socket via a `port_adapter`. The `router` will maintain a map of address regions for the target port. Each address region is associated with a memory or initiator port to which the transactions in that address region need to be forwarded. This allows to connect multiple memories to a single target port at different addresses, but it also allows complex and dynamic transaction routing to memories and initiator ports.

## 2.2.2 Memory Map Modeling

The different memory objects can be used to model the memory map of a component.

The starting point for a memory map is the `scml2::memory` object. This object provides with the basic storage implementation, the handling of the TLM2 API's and the semantics of the generic payload. It is not possible to create a memory map consisting only of registers or bitfields, these objects need the `scml2::memory` object to convert generic payload transactions into simple register and bitfield accesses.

All other memory objects need an `scml2::memory` as parent object (directly or indirectly). A `memory_alias` represents a subrange in a memory and takes either a memory or another alias as parent. The address range represented by the alias should be smaller or have the same size as its parent object. The `scml2::reg` is intended to represent a single word in a memory map and can have a memory or an alias as parent. A bitfield represents a range of bits within a register, it can only have a register as parent.

Each memory object is templated by its datatype (`<DT>`). The datatype defines the size of the individual elements (or words) in the memory object. The size of that datatype (in bytes) is also used to determine the address range represented by the memory object. The `scml2::memory` object determines the base size for all objects in its hierarchy. It is possible, for example, registers in a memory to have a word size that is a multiple of the word size of the memory. This allows to model memory maps with varying sized registers, in such a case it is easiest to work with an `scml2::memory` with datatype `unsigned char` which allows registers of any other supported datatype to be used in that memory map.

## 2.2.3 Properties and Attributes

All memory objects have a name and a width. The width gives the size of the datatype in bytes. They also have a DMI attribute to control whether the TLM2 direct memory interface is enabled for that object. Each object can control this individually, a hierarchical object will forward its setting to all its children.

An `scml2::memory` has latency parameters. These are latency parameters that apply for all objects in that memory hierarchy.

All objects, except `scml2::reg` have a size attribute and all child object have a reference to their parent object.

## 2.2.4 Behavior

To access the content of the storage objects there are multiple APIs available to support the different situations where memory access could be required. The variants differ in the way callbacks are triggered and debugger watchpoints are intended to be triggered.

**Table 2-1 Behavior Type and Available APIs**

Type	API	Callback	Watchpoints
Simple	Put/get	No	Yes
Simple-Debug	Put/get_debug	No	No

Type	API	Callback	Watchpoints
Trigger-Callbacks	Put/get_with_triggering_callbacks	Regular	Yes
Trigger-Debug-Callbacks	Put/get_with_triggering_debug_callbacks	Debug	No

For each variant, there are also different signatures available.

**Table 2-2 Different Signatures of Each Variant**

Style	Arguments
TLM2	address, dataPtr, data_length, byte_enablePtr, enableLength
TLM2-Word	address, dataPtr, data_length
Word	index, DT
Sub-Word	index, DT, size, offset

In total, this gives 2x16 different access methods that are supported by the memory objects, summarized in the following table:

**Table 2-3 Access Methods Supported by Memory Objects**

Style	TLM2	TLM2-Word	Word	Sub-Word
Type				
Simple	x	x	x	x
Simple-Debug	x	x	x	x
Trigger-Callbacks	x	x	x	x
Trigger-Debug-Callbacks	x	x	x	x

## 2.2.5 Callbacks

The SCML memory objects provide with callbacks to model the behaviors that are associated with an access to the elements in the memory map of a component. Callbacks are methods in the component class that are registered with a memory object and that will be executed whenever there is an incoming transaction to the address range of that memory object. Callbacks can also be triggered via a Trigger-Callbacks or Trigger-Debug-Callbacks type of access to the memory object (see “Behavior” on page 26).

### 2.2.5.1 Callback Properties

In the memory map of a model, there can be multiple memory objects that cover the address range accessed by a transaction (for example, the top-level memory as well as a register). In such a case, SystemC Modeling Library will do the following:

1. To execute all behaviors that should be triggered by the transaction, the transaction will be "unrolled" to fit with the address range of the memory object with a callback: a burst access will be unrolled into accesses to the individual registers in the range, and an access to a register will be split into accesses to the bitfields it contains (provided at least one of them has a callback).

2. Only the callback of the "most refined" memory object will be called. That is the callback for the memory object that sits deepest in the object hierarchy (and that has a callback registered). This rule does not apply to bitfields, this means that when a register as well as one of its bitfields have a callback, then the register callback will be executed. It is up to the user to trigger the execution of the bitfield callbacks from the register callback (see "[Operators](#)" on page 50).

Following are the additional properties of callbacks:

- Callbacks can be registered either for debug or regular accesses and can be specialized for read or write accesses.
- The functions implementing a callback will be called as part of the TLM2 LT transport interface implementation. This means that calling `wait()` is allowed in a regular callback.
- In a callback, the address passed in the payload argument will be adjusted so that it is relative to the start of the memory object to which the callback is registered.
- Registering a callback (regular callback or a debug callback) to a memory object will disable DMI access to this memory object.

It is possible to extend SCML with additional callback mechanisms (see "[Callback Base Classes](#)" on page 66).

### 2.2.5.2 Registering Callbacks

Several convenience functions are defined to register predefined callbacks to a memory object. These functions are defined in the `scml2/memory_callback_functions.h` and `scml2/memory_debug_callback_functions.h` files, respectively.

The following functions are available to register a member method as a callback to a memory object:

#### Regular callback registration

```
set_callback(mem, SCML2_CALLBACK(method), syncType, tag)
set_read_callback(mem, SCML2_CALLBACK(method), syncType, tag)
set_read_no_store_callback(mem, SCML2_CALLBACK(method), syncType, tag)
set_write_callback(mem, SCML2_CALLBACK(method), syncType, tag)
set_word_read_callback(mem, SCML2_CALLBACK(method), syncType, tag)
set_word_write_callback(mem, SCML2_CALLBACK(method), syncType, tag)
set_post_write_callback(mem, SCML2_CALLBACK(method), syncType, tag);
```



`set_read_callback` will store the value returned to the initiator in the memory,  
`set_read_no_store_callback` will not store the return value. If a callback already calculates  
and stores the next value for the register, you need to use `set_read_no_store` variant, to  
prevent the new value from being overwritten when the call returns.

The following functions are available to register a member method as a debug callback to a memory object:

#### Debug callback registration

```
set_debug_callback(mem, SCML2_CALLBACK(method), tag)
set_debug_read_callback(mem, SCML2_CALLBACK(method), tag)
set_debug_write_callback(mem, SCML2_CALLBACK(method), tag)
```

Where the arguments represent the following:

<code>mem</code>	Is the memory object to which the callback will be registered.
------------------	--

SCML2_CALLBACK( <i>method</i> )	Is macro that helps pass the callback into the registration function. <i>method</i> is the name of the callback method, which should be a member of the model class of the memory object. The supported signatures of the callback method are explained in the sections of the relevant memory objects.
<i>syncType</i>	<p>Can be one of the following:</p> <ul style="list-style-type: none"> <li>• NEVER_SYNCING indicates that the callback is nonblocking and must never call <code>wait()</code>.</li> <li>• SELF_SYNCING indicates that the callback is blocking and may call <code>wait()</code>. The timing annotation is passed unmodified to the callback.</li> <li>• AUTO_SYNCING indicates that the callback is blocking and may call <code>wait()</code>. The memory object synchronizes before and after calling the callback. The timing annotation passed to the callback is always <code>SC_ZERO_TIME</code>.</li> </ul> <p>These types are defined in the <code>scml2/types.h</code> file.</p> <p>The Post predefined behavior callbacks do not support SELF_SYNCING callbacks (since the callback does not have a time argument)</p>
<i>tag</i>	Is an optional argument. When provided it is a user-provided integer that is passed to the callback. For example, so that the same callback can be attached to all registers in a register array.

### 2.2.5.3 Removing Callbacks

Callbacks can be removed at runtime. This is usually not needed, but to improve simulation performance it is a good idea to disable the callbacks for memory objects that are very often accessed (for example, in a polling loop) but where the behavior that is modeled in the callback is rarely needed. In such a case, a performance improvement is to remove the callback and register it again when the external event that the polling loop is checking for has happened.

The following API's are available to unregister callbacks from a memory object:

#### Unregistering regular callbacks

```
void remove_callback()
void remove_read_callback()
void remove_write_callback()
```

#### Unregistering debug callbacks

```
void remove_debug_callback()
void remove_debug_read_callback()
void remove_debug_write_callback()
```

### 2.2.5.4 Callback Methods

Every callback registration function accepts a few signatures for the callback methods. For a detailed list of the callback signatures, see sections: “[Callbacks](#)” on page 39, “[Callbacks](#)” on page 45, “[Callbacks](#)” on page 51, “[Callbacks](#)” on page 57, and “[Callbacks](#)” on page 63.

### 2.2.6 Access Restrictions

Access restrictions are a special type of callbacks. They are intended to control whether transactions can access memory objects or parts of them. Typically, access to certain registers or bitfields in a memory map

might be controlled by other registers or bitfields to prevent that the component is pushed into an illegal state through some external input.

Such behavior could be modeled using regular callbacks, but it can be tedious due to the "most refined" rule that applies to regular callbacks. Often the restriction applies for all bitfields in a register, forcing you to add the access check to all bitfield callbacks, or to replace the bitfield callbacks with a register callback.

### 2.2.6.1 Properties of Access Restrictions

Access restrictions have the following properties:

1. Access restriction callbacks are executed for an incoming transaction before any of the behavior (regular) callbacks are executed.
2. Access restrictions are executed for all memory objects in the memory map that have an address range that overlaps with the address range of the transaction.
3. To execute all registered access restrictions, the transaction will be "unrolled" to fit with the address range of the memory object with an access restriction: a burst access will be unrolled into accesses to the individual registers in the range, and an access to a register will be split into accesses to the bitfields it contains (provided at least one of them has an access restriction).
4. An access restriction determines which part of an access is restricted allowed by manipulating the `byte_enable` argument in the TLM2 Generic Payload. The byte enable uses `0xFF` per byte, for access restrictions the limitation that only values `0x0` and `0xFF` are allowed is lifted and the `byte_enable` value is used as a bit enable.
5. Access restrictions are cumulative: All access restrictions on the path to a certain memory object are executed. Each of them can further modify the `byte_enable` argument of the transaction.
6. When access is disabled for all bits of a memory object, the corresponding behavior callback will not be executed (this includes the default behavior of a memory object).
7. The restrict callback should take care of the data in the payload and the storage to ensure that any callback that might be executed works with the expected value. For example, in case a bitfield has an access restriction, but there is also a callback on the parent register, the access restriction may want to modify the transaction data so that the register callback gets the current value of the bitfield rather than what is specified in the original transaction payload.
8. The return value of an access restriction is used to update the transaction response field. The return type of a restrict callback is `scml2::access_restriction_result`. Possible values are:

<code>scml2::RESTRICT_NO_ERROR</code>	No change in transaction response, it is up to the callbacks to finalize the value (both restrict callbacks as well as behavior callbacks).
<code>scml2::RESTRICT_ERROR</code>	The transaction response is modified to <code>TLM_GENERIC_ERROR_RESPONSE</code> and the transaction is immediately abandoned. No further restrictions are checked and no behavior callbacks are executed.

### 2.2.6.2 Registering a Restriction

Several convenience functions are defined to register access restrictions to a memory object. These functions are defined in the `scml2/memory_restriction_functions_include.h` and `scml2/bitfield_restriction_functions_include.h` file, respectively.

The following functions are available to register a member method as an access restriction for a memory object:

#### Access restriction registration

```

set_restriction(mem, SCML2_CALLBACK(method), tag)
set_read_restriction(mem, SCML2_CALLBACK(method), tag)
set_write_restriction(mem, SCML2_CALLBACK(method), tag)

```

Where the arguments represent the following:

mem	Is the memory object to which the callback will be registered.
SCML2_CALLBACK(method)	Is the macro that helps pass the callback into the registration function. method is the name of the callback method, which should be a member of the model class of the memory object. The supported signatures of the callback method are explained in section “Access Restriction Methods” on page 31.
tag	Is an optional argument. When provided, it is a user-provided integer that is passed to the callback. For example, so that the same callback can be attached to all registers in a register array.

This is similar to the registration of a regular behavior callback with the exception that restrictions do not have a *syncType* as they are not allowed to influence the timing of the transaction.

### 2.2.6.3 Removing an Access Restriction

Access restrictions can be removed with the following API's:

#### API's to remove access restrictions

```

scml2::remove_restriction();
scml2::remove_read_restriction();
scml2::remove_write_restriction();

```

### 2.2.6.4 Access Restriction Methods

There are two forms of access restriction methods supported:

- A simple one available for registers, and bitfields, and
- a TLM2-based restriction method available for all memory objects.

The signature for these variants is as follows:

```

scml2::access_restriction_result MyRestrictFtion(DT& data, DT& bit_enables, int tag)
scml2::access_restriction_result MyRestrictFtion(tlm::tlm_generic_payload& trans, int tag)

```

With the following arguments:

**Table 2-4 Access Restriction Methods - Arguments**

Argument	Description
DT& data	Specifies the data argument, which is used to represent the data from the payload from the parts of the access that are not restricted, that is, the parts of the transaction that should be processed in the behavior callbacks. The data that should be assumed by the callbacks for the parts of the access that are restricted. This is the data that is already processed by the restriction.

Argument	Description
DT& bit_enables	Indicates which parts of the data vector are restricted (corresponding bits set to zero). This will influence whether callbacks are executed. Callbacks are not executed if all bit_enables are zero for the object the callback is associated with.
Int tag	Is an optional argument. A tag can be set when registering a callback, that same tag value will be passed to the access restriction each time it is called.
tlm::tlm_generic_payload& trans	Is the TLM transaction payload with updated byte_enables and address value (address is always relative to the base-address of the memory object), also streamin_width is already handled and converted into byte_enables. This variant gives access to all other payload attributes to decide on the restriction. It can be used to check for extensions or to deal with burst accesses.
scml2::access_restriction_result	The return value should be either scml2::RESTRICT_NO_ERROR or scml2::RESTRICT_ERROR. In the latter case, the transaction is immediately abandoned and a TLM_GENERIC_ERROR_RESPONSE will be returned.

## 2.2.6.5 Predefined Access Restrictions

For a number of frequent access restriction types, a predefined SCML2 API is provided to set a restriction on a memory object:

### Predefined Access Restrictions

```
scml2::set_ignore_restriction(mem, DT value = 0);
scml2::set_error_restriction(mem);
scml2::set_word_restriction(mem);
scml2::set_read_ignore_restriction(mem, DT value = 0);
scml2::set_read_error_restriction(mem);
scml2::set_read_word_restriction(mem);
scml2::set_write_ignore_restriction(mem, DT value = 0);
scml2::set_write_error_restriction(mem);
scml2::set_write_word_restriction(mem);
scml2::set_read_unmapped_error_restriction(mem)
scml2::set_write_unmapped_error_restriction(mem)
scml2::set_unmapped_error_restriction(mem)
scml2::set_read_unmapped_ignore_restriction(mem)
scml2::set_write_unmapped_ignore_restriction(mem)
scml2::set_unmapped_ignore_restriction(mem)
scml2::set_byte_error_restriction(mem)
scml2::set_short_error_restriction(mem)
scml2::set_int_error_restriction(mem)
scml2::set_byte_ignore_restriction(mem, DT value = 0)
scml2::set_short_ignore_restriction(mem, DT value = 0)
scml2::set_int_ignore_restriction(mem, DT value = 0)
scml2::set_read_byte_error_restriction(mem)
scml2::set_read_short_error_restriction(mem)
scml2::set_read_int_error_restriction(mem)
scml2::set_read_byte_ignore_restriction(mem, DT value = 0)
scml2::set_read_short_ignore_restriction(mem, DT value = 0)
scml2::set_read_int_ignore_restriction(mem, DT value = 0)
scml2::set_write_byte_error_restriction(mem)
```

```

scml2::set_write_short_error_restriction(mem)
scml2::set_write_int_error_restriction(mem)
scml2::set_write_byte_ignore_restriction(mem, DT value = 0)
scml2::set_write_short_ignore_restriction(mem, DT value = 0)
scml2::set_write_int_ignore_restriction(mem, DT value = 0)

```

These API's have the following properties:

- The predefined access restrictions can be set for all accesses, or only for read or write accesses.
- The predefined access restrictions can be set on any memory object (except the word-access limitation that cannot be set on a bitfield).
- The error restriction will set the transaction response to TLM\_GENERIC\_ERROR\_RESPONSE.
- The value argument for the scml2::set\_read\_ignore\_restriction will be stored in the data pointer of the transaction payload.
- The value argument for the scml2::set\_write\_ignore\_restriction should be set to the value of the memory object for which the restriction is set, in order to make sure that any callback on a memory object higher up in the memory map hierarchy does not overwrite that memory object.
  - So typically, scml2::set\_write\_ignore\_restriction(mem, (DT)mem);
  - Other values are possible, for example, passing 0 as value will implement a set\_0\_on\_write while still allowing for other behavior callbacks to be executed.
- A word access restriction will allow any transaction that is,
  - Word aligned
  - Data-length is a multiple of word-size
  - All byte\_enables are set

This means that an aligned 128-bit access in a 64-bit memory object will be allowed. Multiple register callbacks could be triggered, but all can assume the access is for the full word.

### 2.2.6.6 Access Restrictions Implementation Helper Functions

To facilitate the implementation of an access restriction, a number of helper functions are available that represent typical functionality in a restrict callback:

#### Access restriction helper functions:

```

scml2::restrict_all(DT& data, DT& bit_enables, DT& restrict_value = 0);
scml2::restrict_all<DT>(tlm::tlm_generic_payload& trans,
                        DT restrict_value = 0);
scml2::restrict_all_and_store(DT& data, DT& bit_enables, DT& restrict_value,
                             MEM_OBJECT<DT>& mem_obj);
scml2::restrict_some(DT& data, DT& bit_enables, DT& restrict_value,
                     DT in_data_mask);
scml2::restrict_some_and_store(DT& data, DT& bit_enables,
                             DT& restrict_value, DT in_data_mask,
                             MEM_OBJECT<DT>& mem_obj);

```

#### Properties of the helper functions:

- restrict\_all will set bit\_enables to 0 and pass the restrict\_value as data (by default set to 0).

- `restrict_all_and_store` will set `bit_enables` to 0, pass the `restrict_value` as data and store the `restrict` value in the memory object.
- `restrict_some` can be used to partially restrict the access. The `data_mask` should indicate the valid bits in the data (bits set to 1 in the `data_mask` will be restricted, that is, `bit_enables` will be set to 0). The `restrict_value` will be passed as data.
- `restrict_some_and_store` can be used to partially restrict the access. The `data_mask` should indicate the valid bits in the data (bits set to 1 in the `data_mask` will be restricted, that is, `bit_enables` will be set to 0). The `restrict_value` will be passed as data and also will be stored in the memory object.
- All convenience functions return `scml2::RESTRICT_NO_ERROR`.

#### 2.2.6.7 Access Restrictions Versus Behavior Callbacks

- Restrictions
  - All restrictions along the path of a transaction are checked: All elements in the memory hierarchy to the leaf nodes that get accessed can add restrictions.
  - Restrictions should manipulate `bit_enables/byte_enables` to implement the restrictions.
  - Are only allowed to further restrict (not to overwrite restrictions elsewhere in the path).
  - Restrictions should manipulate payload data to ensure consistency independent of other restrictions and behavior callbacks.
- Behavior callbacks
  - Only the 'most specified' behavior gets executed.
  - Behaviors have full access/control over transaction payload.
  - No need to care about any other callbacks or restrictions that might get executed.
  - Behavior callbacks will not be executed when `bit_enables/byte_enables` are all disabled for the memory object.
  - 'Word'-style behavior callbacks will not check whether all byte-enables are set if there is a restriction somewhere on the path to the callback.

#### 2.2.7 Vectors of Memory Objects

When managing vectors of memory objects, one can use the `scml2::vector` class which is API-compatible with `sc_core::sc_vector`, but has a different naming convention for the objects created in the array.

The names will be '`<name>_0, <name>_1, ..., <name>_n`'. For `sc_core::sc_vector`, these names would be '`<name>, <name>_1, ..., <name>_n`'.

Except for this difference, the two classes are intended to have identical behavior.

You can also use the `scml2::vector` class for other modeling objects (such as ports) to instantiate vectors of objects with the adapted naming convention.

### 2.3 memory

The `scml2::memory` represents the top-level object for the local memory or register file description of a model. The memory does the actual storage allocation and can be bound to a TLM2 target socket via an adapter. It implements the `mappable_if` interface which requires an object to implement the TLM2 LT

interfaces (`b_transport`, `transport_dbg` and `get_direct_mem_ptr`). The `mappable_if` is provided in SCML2 as a link between the storage objects and the TLM2 socket interfaces. For details, see “[mappable\\_if](#)” on page 65. A memory can have aliases and/or registers to specify the detailed memory hierarchy of the component.

The include file of the memory objects is `scml2/memory.h`.

It has the following properties:

- name and size, as provided with the constructor of the object. The name of the memory will be used by the debug and analysis tools to refer to this object.
- width: representing the datawidth that is stored in the memory. This is derived from the datatype template argument for the class.
- The `scml2::memory` also has a default read and write latency which will be used in the implementation of the default behavior. Any `b_transport` read or write call will get the latency implemented accordingly through timing annotation. This value will also be passed when the storage pointer is made available on a DMI request from an initiator. It is the only object with this parameter.
- The memory object also implements the index operation (`[]`) and has an initialize call to define an initial value for the storage.

### 2.3.1 Types

The memory class is templated with the underlying value type:

```
Template <typename DT> class memory
```

The following types are supported:

**Table 2-5 Supported Data types**

Supported Data types	Word size
<code>unsigned char</code>	8
<code>unsigned short</code>	16
<code>unsigned int</code>	32
<code>unsigned long long</code>	64
<code>sc_dt::sc_bignum&lt;128&gt;</code>	128
<code>sc_dt::sc_bignum&lt;256&gt;</code>	256
<code>sc_dt::sc_bignum&lt;512&gt;</code>	512

### 2.3.2 Constructors

The following constructor is available

```
memory(const std::string& name, unsigned long long size);
```

This creates a new memory. The size argument must be specified in words.

### 2.3.3 Properties

The following methods are available to access the properties of a memory:

- `const std::string& get_name() const`  
Returns the full hierarchical name of the memory object.
- `unsigned long long get_size() const`  
Returns the size of the memory object in words.
- `unsigned int get_width() const`  
Returns the width in bytes of the underlying data type of the memory object.
- `void set_default_read_latency(const sc_core::sc_time& t)`  
`const sc_core::sc_time& get_default_read_latency() const`  
`void set_default_write_latency(const sc_core::sc_time& t)`  
`const sc_core::sc_time& get_default_write_latency() const`  
Sets/gets the latency returned in the `tlm::tlm_dmi` structure of the `get_direct_mem_ptr()` call. If no callback is attached, this latency is also added to the timing annotation argument of the `b_transport()` call.
- `bool is_dmi_enabled()`  
Returns true if DMI accesses are allowed for the object, `false` otherwise. DMI is enabled by default.
- `void enable_dmi()`  
`void disable_dmi()`  
Enables/disables DMI accesses for the object.
- `const std::string& get_description() const`  
`void set_description(const std::string&)`  
Gets/sets the description for the memory object. This description can be displayed in debuggers.
- `bool has_default_read_behavior() const;`  
`bool has_default_write_behavior() const;`  
`bool has_default_debug_read_behavior() const;`  
`bool has_default_debug_write_behavior() const;`  
`bool has_never_syncing_read_behavior() const;`  
`bool has_never_syncing_write_behavior() const;`  
`bool is_dmi_allowed() const;`  
`bool is_dmi_read_allowed() const;`  
`bool is_dmi_write_allowed() const;`

These are a set of API's to query what type of behavior is associated with the memory and whether DMI is enabled.

- `bool has_default_restriction() const;`  
`bool has_default_read_restriction() const;`  
`bool has_default_write_restriction() const;`

These API's query whether there is a restriction set for the memory.

The memory object can be bound to a TLM2 target socket via a `port_adapter`. For details, see [Port Adaptors](#). A target port adaptor may be bound to an SCML2 memory, as shown below.

```
// bind adaptor to memory
scml2::memory my_memory("my_memory", 0x100);
(*my_port_adapter)(my_memory);
```

## 2.3.4 Behaviors

### 2.3.4.1 Initialization

The `initialize()` method can be used to put the specified initial value in the whole memory array:

```
void initialize (const DT& value = DT())
```

In case no argument is given, the value returned by the default constructor for the underlying data type is used.

### 2.3.4.2 Transport Calls

The following transport methods are available on the `memory` objects:

The `memory` object implements the following TLM2 methods:

```
void b_transport(tlm::tlm_generic_payload& trans, sc_core::sc_time& t)
unsigned int transport_dbg(tlm::tlm_generic_payload& trans)
bool get_direct_mem_ptr(tlm::tlm_generic_payload& trans,
                        tlm::tlm_dmi& dmiData)
```

These methods trigger the callbacks registered to the memory object hierarchy represented by the memory.

The `memory` object also implements the following TLM2-like methods:

```
void transport_without_triggering_callbacks(tlm::tlm_generic_payload& trans)
void transport_debug_without_triggering_callbacks(
    tlm::tlm_generic_payload& trans)
```

These transport methods do not trigger any callbacks. They access the current content of the `memory` object. The debug methods also do not trigger any watchpoints on the `memory` object.

### 2.3.4.3 Put/Get Access

The `memory` object supports the different put/get behaviors as described in “[Behavior](#) on page 26”. The following set of access functions is supported according to the different types and styles as defined in “[Behavior](#) on page 26”:

**Table 2-6 Pit/Get Access**

Type	API	Callback	Watchpoints
Simple	Put/get	No	Yes
Simple-Debug	Put/get_debug	No	No
Trigger-Callbacks	Put/get_with_triggering_callbacks	Regular	Yes
Trigger-Debug-Callbacks	Put/get_with_triggering_debug_callbacks	Debug	No

For each variant, there are also different signatures available:

**Table 2-7 Different Signatures of Each Variant**

Style	Arguments
TLM2	address, dataPtr, data_length, byte_enablePtr, enableLength
TLM2-Word	address, dataPtr, data_length
Word	index, DT
Sub-Word	index, DT, size, offset

The following table describes the arguments mentioned in the table 2-7.

**Table 2-8 Argument Descriptions**

Argument name	C++	description
address	unsigned long long address	Byte address as in TLM2 GP.
dataPtr	(const) unsigned char* data	Data array as in TLM2 GP.
data_length	unsigned int dataLength	The length of the transaction in bytes as in TLM2 GP.
byte_enablePtr	const unsigned char* byteEnablePtr	Specifies a byte enable array which can be 0 as in TLM2 GP.
enableLength	unsigned int byteEnableLength	Length of the byte enable array in bytes as in TLM2 GP.
index	unsigned long long index	The word index as specified by the DT template of the memory object.
DT	(const) DT& data	Is the data.
size	unsigned int size	Is the size of the access in bytes.
offset	unsigned int offset	Is the offset for the access in bytes.

Additional notes:

- The Trigger- Callback access functions take an additional `sc_time` argument. They return the TLM2 response status, returned by the triggered callback.
- The `get()` and `get_debug()` methods for word or subword accesses return the read data instead of passing it as an argument.
- The `put_debug_with_triggering_callbacks()` and `get_debug_with_triggering_callbacks()` calls must not use `byte_enables` (the `byteEnablePtr` must be 0) since TLM2 does not support byte enables for debug calls.
- The debug versions of these methods do not trigger watchpoints.

In total, this gives 2x16 different access methods that are supported by the memory objects, summarized in the following table:

**Table 2-9     Memory Objects**

Style	TLM2	TLM2-Word	Word	Sub-Word
Type				
Simple	x	x	x	x
Simple-Debug	x	x	x	x
Trigger-Callbacks	x	x	x	x
Trigger-Debug-Callbacks	x	x	x	x

For a complete list with all arguments, see the include file of the memory: scml2/memory.h.

#### 2.3.4.4    Operators

The following assignment operators are available:

```
reference operator[](unsigned long long index)  
DT operator[](unsigned long long index) const
```

The lvalue version of the *index* operator returns a `memory_index_reference` object that forwards all operations to the referenced memory object. The `const` version returns the current value.

```
iterator begin()  
const_iterator begin() const
```

Returns a random access iterator pointing to the first element in the memory object.

```
iterator end()  
const_iterator end() const
```

Returns a random access iterator pointing to the end of the memory object.

#### 2.3.5    Callbacks

The memory object supports the callbacks listed below. For more information on callbacks, see also “[Callbacks](#)” on page 27.

##### 2.3.5.1    Regular Callbacks

The memory object only supports the `set_callback` registration-style callback methods.

###### Regular callback registration

```
set_callback(mem, SCML2_CALLBACK(method), syncType, tag)  
set_read_callback(mem, SCML2_CALLBACK(method), syncType, tag)  
set_read_no_store_callback(mem, SCML2_CALLBACK(method), syncType, tag)  
set_write_callback(mem, SCML2_CALLBACK(method), syncType, tag)
```

It accepts callback methods with the following signatures:

```
void transportCallback(tlm::tlm_generic_payload&, sc_core::sc_time&, int tag)  
void transportCallback(tlm::tlm_generic_payload&, int tag)
```

Additional notes:

- In all the above callback methods, `tag` is an optional argument, only to be used when the callback is registered with a tag.

- The untimed callback (without the `sc_time` parameter) cannot be SELF\_SYNCING callbacks.

### 2.3.5.2 Debug Callbacks

The memory object support the following registration API's for debug callbacks:

#### Debug callback registration

```
set_debug_callback(mem, SCML2_CALLBACK(method), tag)
set_debug_read_callback(mem, SCML2_CALLBACK(method), tag)
set_debug_write_callback(mem, SCML2_CALLBACK(method), tag)
```

For debug callbacks, method must have one of the following signatures:

```
unsigned int transportCallback(tlm::tlm_generic_payload&)
unsigned int transportCallback(tlm::tlm_generic_payload&, int tag)
```

The return value is the number of consecutive bytes successfully read or written. If the access cannot be executed, 0 must be returned.

### 2.3.6 Access Restrictions

The memory object supports access restrictions as described in “[Access Restrictions](#)” on page 29. The memory supports all predefined access restrictions as specified in “[Predefined Access Restrictions](#)” on page 32, but it only supports TLM2-style access restriction callbacks:

```
scml2::access_restriction_result MyRestrictFtion(tlm::tlm_generic_payload& trans, int tag)
```

## 2.4 memory\_alias

The `scml2::memory_alias` object specifies a subregion of a `scml2::memory`. A `memory_alias` cannot exist on its own, it always needs a `scml2::memory` parent object. The alias does not come with its own storage and can have further aliases and/or registers. The alias cannot be bound to a TLM2 target socket (or the adapters) as it does not implement the `mappable_if`. The use of a `memory_alias` is to provide with a different name for a subregion to make sure the model reflects the design specification of the component, but also serves as a hook so that specialized behavior can be associated with part of the memory or register map of the component. The properties of an `memory_alias` are similar to the `memory` and provided via the `memory_base` base class.

The include file of the `memory_alias` objects is `scml2/memory_alias.h`.

It has the following properties:

- The `scml2::memory_alias` takes a name, parent memory, offset and size in the constructor. Offset indicates the start index for this alias within the range of the parent memory (or `memory_alias`).
- It has access functions for name, size, offset, parent and width. Exactly like these also exist for the `scml2::memory`.
- It also supports the index operator `[]` and the initialize call as exist for the `memory`.
- Callbacks that are associated with a `memory_alias` will override the behavior of the parent `memory`. Both the default behavior and the behavior that gets registered via the callback mechanism are overruled this way.

### 2.4.1 Types

The `memory_alias` class is templated with the underlying value type:

Template <typename DT> class memory\_alias

The following types are supported:

**Table 2-10 Supported Types**

Supported Datatype	Word size
unsigned char	8
unsigned short	16
unsigned int	32
unsigned long long	64
sc_dt::sc_bignum<128>	128
sc_dt::sc_bignum<256>	256
sc_dt::sc_bignum<512>	512

The memory alias should have a datatype where the word size is a multiple of that of its parent memory.

## 2.4.2 Constructors

The following constructors are available:

```
memory_alias(const std::string& name,
             memory<DT>& parent,
             unsigned long long offset,
             unsigned long long size)
memory_alias(const std::string& name,
             memory_alias<DT>& parent,
             unsigned long long offset,
             unsigned long long size)
```

Create a new `memory_alias` object. The `size` and `offset` argument must be specified in words.

## 2.4.3 Properties

The following methods are available to access the properties of a `memory_alias`:

- const std::string& `get_name()` const  
Returns the full hierarchical name of the of the `memory_alias` object.
- unsigned long long `get_offset()` const  
Returns the offset in words relative to the top-level memory object.
- unsigned long long `get_size()` const  
Returns the size of the `memory_alias` object in words.
- unsigned int `get_width()` const  
Returns the width in bytes of the underlying data type of the `memory_alias` object.
- memory\_base\* `get_parent()` const  
Returns a pointer to the parent memory or `memory_alias` object.

- `bool is_dmi_enabled()`  
Returns `true` if DMI accesses are allowed for the object, `false` otherwise. DMI is enabled by default.
- `void enable_dmi()`  
`void disable_dmi()`  
Enables/disables DMI accesses for the object.
- `const std::string& get_description() const`  
`void set_description(const std::string&)`  
Gets/sets the description for the `memory_alias` object. This description can be displayed in debuggers.
- `bool has_default_read_behavior() const;`  
`bool has_default_write_behavior() const;`  
`bool has_default_debug_read_behavior() const;`  
`bool has_default_debug_write_behavior() const;`  
`bool has_never_syncing_read_behavior() const;`  
`bool has_never_syncing_write_behavior() const;`  
`bool is_dmi_allowed() const;`  
`bool is_dmi_read_allowed() const;`  
`bool is_dmi_write_allowed() const;`

These are a set of API's to query what type of behavior is associated with the `memory_alias` and whether DMI is enabled.

- `bool has_default_restriction() const;`  
`bool has_default_read_restriction() const;`  
`bool has_default_write_restriction() const;`

These API's query whether there is a restriction set for the `memory_alias`.

## 2.4.4 Behaviors

### 2.4.4.1 Initialization

The `initialize()` method can be used to put the specified initial value in the whole memory array:

```
void initialize (const DT& value = DT())
```

In case no argument is given, the value returned by the default constructor for the underlying data type is used.

### 2.4.4.2 Transport Calls

The following transport methods are available on the memory objects:

The `memory_alias` object implements the following TLM2 methods:

```
void b_transport(tlm::tlm_generic_payload& trans, sc_core::sc_time& t) unsigned int
transport_dbg(tlm::tlm_generic_payload& trans)
```

These methods trigger the callbacks registered to the `memory_alias` object hierarchy represented by the memory.

The `memory_alias` object also implements the following TLM2-like methods:

```
void transport_without_triggering_callbacks(tlm::tlm_generic_payload& trans)
void transport_debug_without_triggering_callbacks()
```

```
    tlm::tlm_generic_payload& trans)
```

These transport methods do not trigger any callbacks. They access the current content of the memory\_alias object. The debug methods also do not trigger any watchpoints on the memory\_alias object.

#### 2.4.4.3 Put/Get Access

The memory\_alias object supports the different put/get behaviors as described in “[Behavior](#)” on page 26. The following set of access functions is supported according to the different types and styles as defined in “[Behavior](#)” on page 26:

**Table 2-11 Supported Access Functions**

Type	API	Callback	Watchpoints
Simple	Put/get	No	Yes
Simple-Debug	Put/get_debug	No	No
Trigger-Callbacks	Put/get_with_triggering_callbacks	Regular	Yes
Trigger-Debug-Callbacks	Put/get_with_triggering_debug_callbacks	Debug	No

For each variant, there are also different signatures available:

**Table 2-12 Different Signatures of Each Variant**

Style	Arguments
TLM2	address, dataPtr, data_length, byte_enablePtr, enableLength
TLM2-Word	address, dataPtr, data_length
Word	index, DT
Sub-Word	index, DT, size, offset

The following table describes the arguments mentioned in the table 2-12.

**Table 2-13 Argument Descriptions**

Argument name	C++	Description
address	unsigned long long address	Byte address as in TLM2 GP.
dataPtr	(const) unsigned char* data	Data array as in TLM2 GP.
data_length	unsigned int dataLength	The length of the transaction in bytes as in TLM2 GP.
byte_enablePtr	const unsigned char* byteEnablePtr	Specifies a byte enable array which can be 0 as in TLM2 GP.
enableLength	unsigned int byteEnableLength	Length of the byte enable array in bytes as in TLM2 GP.

Argument name	C++	Description
index	unsigned long long index	The word index as specified by the DT template of the memory object.
DT	(const) DT& data	Is the data.
size	unsigned int size	Is the size of the access in bytes.
offset	unsigned int offset	Is the offset for the access in bytes.

Additional notes:

- The Trigger-Callback access functions take an additional `sc_time` argument. They return the TLM2 response status, returned by the triggered callback.
- The `get()` and `get_debug()` methods for word or subword accesses return the read data instead of passing it as an argument.
- The `put_debug_with_triggering_callbacks()` and `get_debug_with_triggering_callbacks()` calls must not use `byte_enables` (the `byteEnablePtr` must be 0) since TLM2 does not support byte enables for debug calls.
- The debug versions of these methods do not trigger watchpoints.

In total, this gives 2x16 different access methods that are supported by the memory objects, summarized in the following table:

**Table 2-14 Supported Access Methods**

Style	TLM2	TLM2-Word	Word	Sub-Word
Type				
Simple	x	x	x	x
Simple-Debug	x	x	x	x
Trigger-Callbacks	x	x	x	x
Trigger-Debug-Callbacks	x	x	x	x

For a complete list with all arguments check the include file of the `memory_alias`:  
`scml2/memory_alias.h`.

#### 2.4.4.4 Operators

The following assignment operators are available:

```
reference operator[](unsigned long long index)
DT operator[](unsigned long long index) const
```

The lvalue version of the `index` operator returns a `memory_index_reference` object that forwards all operations to the referenced `memory` object. The `const` version returns the current value.

```
iterator begin()
const_iterator begin() const
```

Returns a random access iterator pointing to the first element in the `memory_alias` object.

```
iterator end()
const_iterator end() const
```

Returns a random access iterator pointing to the end of the `memory_alias` object.

## 2.4.5 Callbacks

The `memory_alias` object supports the callbacks listed below. For more information on callbacks, see also “[Callbacks](#)” on page 27.

### 2.4.5.1 Regular Callbacks

The `memory_alias` object only supports the `set_callback` registration-style callback methods.

#### Regular callback registration

```
set_callback(mem, SCML2_CALLBACK(method), syncType, tag)
set_read_callback(mem, SCML2_CALLBACK(method), syncType, tag)
set_read_no_store_callback(mem, SCML2_CALLBACK(method), syncType, tag)
set_write_callback(mem, SCML2_CALLBACK(method), syncType, tag)
```

It accepts callback methods with the following signatures:

```
void transportCallback(tlm::tlm_generic_payload&, sc_core::sc_time&, int tag)
void transportCallback(tlm::tlm_generic_payload&, int tag)
```

Additional notes:

- In all the above callback methods, `tag` is an optional argument, only to be used when the callback is registered with a tag.
- The untimed callback (without the `sc_time` parameter) cannot be SELF\_SYNCING callbacks.

### 2.4.5.2 Debug Callbacks

The `memory_alias` object support the following registration API's for debug callbacks.

#### Debug callback registration

```
set_debug_callback(mem, SCML2_CALLBACK(method), tag)
set_debug_read_callback(mem, SCML2_CALLBACK(method), tag)
set_debug_write_callback(mem, SCML2_CALLBACK(method), tag)
```

For debug callbacks, `method` must have one of the following signatures:

```
unsigned int transportCallback(tlm::tlm_generic_payload&)
unsigned int transportCallback(tlm::tlm_generic_payload&, int tag)
```

The return value is the number of consecutive bytes successfully read or written. If the access cannot be executed, 0 must be returned.

## 2.4.6 Access Restrictions

The `memory_alias` object supports access restrictions as described in “[Access Restrictions](#)” on page 29. The `memory_alias` supports all predefined access restrictions as specified in “[Predefined Access Restrictions](#)” on page 32, but it only supports TLM2-style access restriction callbacks:

```
scml2::access_restriction_result MyRestrictFtion(tlm::tlm_generic_payload& trans, int tag)
```

## 2.5 reg

The `scml2::reg` is a `memory_alias` of size 1. That is, it represents a subrange of the memory which is the same size as the width of the memory or in other words represents one storage location of the same size as the datatype template argument of the storage classes. As with the `memory_alias`, it does not implement its own storage and must have a parent memory or `memory_alias` and cannot be bound to the TLM2 target sockets (or adapters). On top of that it cannot have any other aliases and or registers, but it can have bitfields associated with itself. It is the leave node of the memory hierarchy. The register object has all the same properties and access methods as the `memory_alias` (name, parent, offset, index operator, callback overriding). The key additional feature for the register is that it can be used as a regular variable.

The include file of the `reg` objects is `scml2/reg.h`.

### 2.5.1 Types

The register class is templated with the underlying value type:

```
Template <typename DT> class reg
```

The following types are supported:

**Table 2-15 Supported Datatypes**

Supported Datatypes	Word Size
<code>unsigned char</code>	8
<code>unsigned short</code>	16
<code>unsigned int</code>	32
<code>unsigned long long</code>	64
<code>sc_dt::sc_bignum&lt;128&gt;</code>	128
<code>sc_dt::sc_bignum&lt;256&gt;</code>	256
<code>sc_dt::sc_bignum&lt;512&gt;</code>	512

The register should have a datatype where the word size is a multiple of that of its parent memory or `memory_alias`.

### 2.5.2 Constructors

The following constructors are available:

```
reg(const std::string& name,  
     memory<DT>& parent,  
     unsigned long long offset)  
reg(const std::string& name,  
     memory_alias<DT>& parent,  
     unsigned long long offset)
```

Create a new `reg` object. The `offset` argument must be specified in words.

### 2.5.3 Properties

The following methods are available to access the properties of a register:

- `const std::string& get_name() const`  
Returns the full hierarchical name of the of the `reg` object.
- `unsigned long long get_offset() const`  
Returns the offset in words relative to the top-level memory object.
- `unsigned int get_width() const`  
Returns the width in bytes of the underlying data type of the `reg` object.
- `memory_base* get_parent() const`  
Returns a pointer to the parent `memory` or `memory_alias` object.
- `bool is_dmi_enabled()`  
Returns `true` if DMI accesses are allowed for the object, `false` otherwise. DMI is enabled by default.
- `void enable_dmi()`  
`void disable_dmi()`  
Enables/disables DMI accesses for the object.
- `const std::string& get_description() const`  
`void set_description(const std::string&)`  
Gets/sets the description for the register object. This description can be displayed in debuggers.
- `bool has_default_read_behavior() const;`  
`bool has_default_write_behavior() const;`  
`bool has_default_debug_read_behavior() const;`  
`bool has_default_debug_write_behavior() const;`  
`bool has_never_syncing_read_behavior() const;`  
`bool has_never_syncing_write_behavior() const;`  
`bool is_dmi_allowed() const;`  
`bool is_dmi_read_allowed() const;`  
`bool is_dmi_write_allowed() const;`  
These are a set of API's to query what type of behavior is associated with the register and whether DMI is enabled.
- `bool has_default_restriction() const;`  
`bool has_default_read_restriction() const;`  
`bool has_default_write_restriction() const;`  
These API's query whether there is a restriction set for the register.

## 2.5.4 Behaviors

### 2.5.4.1 Initialization

The `initialize()` method can be used to put the specified initial value in the whole memory array:

```
void initialize (const DT& value = DT())
```

In case no argument is given, the value returned by the default constructor for the underlying data type is used.

## 2.5.4.2 Transport Calls

The following transport methods are available on the memory objects:

The `reg` object implements the following TLM2 methods:

```
void b_transport(tlm::tlm_generic_payload& trans, sc_core::sc_time& t) unsigned int  
transport_dbg(tlm::tlm_generic_payload& trans)
```

These methods trigger the callbacks registered to the `reg` object hierarchy represented by the memory.

The `reg` object also implements the following TLM2-like methods

These transport methods do not trigger any callbacks. They access the current content of the `reg` object. The debug methods also do not trigger any watchpoints on the `reg` object.

### **2.5.4.3 Put/Get Access**

The `reg` object supports the different put/get behaviors as described in section “[Behavior](#)” on page [26](#).

Unlike the memory object, the put and get methods of the reg object do not take an index argument (since this should always be 0). The following set of access functions is supported according to the different types and styles as defined in “[Behavior](#)” on page [26](#):

**Table 2-16 Supported Access Functions**

Type	API	Callback	Watchpoints
Simple	Put/get	No	Yes
Simple-Debug	Put/get_debug	No	No
Trigger-Callbacks	Put/get_with_triggering_callbacks	Regular	Yes
Trigger-Debug-Callbacks	Put/get_with_triggering_debug_callbacks	Debug	No

For each variant, there are also different signatures available:

**Table 2-17** Different Signatures of Each Variant

Style	Arguments
TLM2	address, dataPtr, data_length, byte_enablePtr, enableLength
TLM2-Word	address, dataPtr, data_length
Word	DT
Sub-Word	DT, size, offset

The following table describes the arguments mentioned in the table 2-16.

**Table 2-18 Argument Description**

Argument Name	C++	Description
address	unsigned long long address	Is the byte address as in TLM2 GP.
dataPtr	(const) unsigned char* data	Is the data array as in TLM2 GP.
data_length	unsigned int dataLength	Is the length of the transaction in bytes as in TLM2 GP.
byte_enablePtr	const unsigned char* byteEnablePtr	Specifies a byte enable array which can be 0 as in TLM2 GP.
enableLength	unsigned int byteEnableLength	Specifies the length of the byte enable array in bytes as in TLM2 GP.
DT	(const) DT& data	Specifies the data.
size	unsigned int size	Is the size of the access in bytes.
offset	unsigned int offset	Is the offset for the access in bytes.

Additional notes:

- The Trigger-Callback access functions take an additional `sc_time` argument. They return the TLM2 response status, returned by the triggered callback.
- The `get()` and `get_debug()` methods for word or subword accesses return the read data instead of passing it as an argument.
- The `put_debug_with_triggering_callbacks()` and `get_debug_with_triggering_callbacks()` calls must not use `byte_enables` (the `byteEnablePtr` must be 0) since TLM2 does not support byte enables for debug calls.
- The debug versions of these methods do not trigger watchpoints.

In total, this gives 2x16 different access methods that are supported by the memory objects, summarized in the following table:

**Table 2-19 Supported Access Methods**

Style	TLM2	TLM2-Word	Word	Sub-Word
Type				
Simple	x	x	x	x
Simple-Debug	x	x	x	x
Trigger-Callbacks	x	x	x	x
Trigger-Debug-Callbacks	x	x	x	x

For a complete list with all arguments, see the include file of the register: `scml2/reg.h`.

#### 2.5.4.4 Triggering Callbacks on Bitfields

The register also has a set of API's to trigger the callbacks on its bitfields. These API's can be used for example, from within a callback.

##### API's to trigger bitfield behavior from register callbacks

```
bool put_with_triggering_bitfield_callbacks(const DT& data, sc_core::sc_time& t)
bool put_with_triggering_bitfield_callbacks(const DT& data, const DT& bitMask,
                                             sc_core::sc_time& t)
bool get_with_triggering_bitfield_callbacks(DT& data, sc_core::sc_time& t)
bool get_with_triggering_bitfield_callbacks(DT& data, const DT& bitMask,
                                             sc_core::sc_time& t)
bool put_debug_with_triggering_bitfield_callbacks(const DT& data, const DT& bitMask)
bool put_debug_with_triggering_bitfield_callbacks(const DT& data)
bool get_debug_with_triggering_bitfield_callbacks(DT& data)
bool get_debug_with_triggering_bitfield_callbacks(DT& data, const DT& bitMask)
```

## 2.5.4.5 Operators

The following assignment operators are available:

```
iterator begin()
const_iterator begin() const
```

Returns a random access iterator pointing to the reg object.

```
iterator end()
const_iterator end() const
```

Returns a random access iterator pointing to the end of the reg object.

A reg object can be converted to the underlying data type:

```
operator DT() const
```

The following assignment operators are available:

```
reg& operator=(DT value)
reg& operator =(const reg& r)
```

The following arithmetic assignment operators are available and behave as defined for the underlying data type:

```
reg& operator+=(DT value)
reg& operator-=(DT value)
reg& operator/=(DT value)
reg& operator*=(DT value)
reg& operator%=(DT value)
reg& operator^=(DT value)
reg& operator&=(DT value)
reg& operator|=(DT value)
reg& operator>>=(DT value)
reg& operator<<=(DT value)
```

The following prefix and postfix decrement and increment operators are available:

```
reg& operator--()
DT operator--(int)
reg& operator++()
DT operator++(int)
```

The register object has all the same properties and access methods as the memory\_alias (name, parent, offset, index operator, callback overruling). The key additional feature for the register is that it can be used as a regular variable.

```
operator DT() const;
reg& operator=(DT value);
reg& operator =(const reg& r);
reg& operator+=(DT value);
reg& operator-=(DT value);
reg& operator/=(DT value);
reg& operator*=(DT value);
reg& operator%=(DT value);
reg& operator^=(DT value);
reg& operator&=(DT value);
reg& operator|==(DT value);
reg& operator>>=(DT value);
reg& operator<<=(DT value);
reg& operator--();
DT operator--(int);
reg& operator++();
DT operator++(int);
```

## 2.5.5 Callbacks

The reg object supports the callbacks listed below. For more information on callbacks, see also “[Callbacks](#)” on page 27.

### 2.5.5.1 Regular Callbacks

Every callback registration function accepts a few signatures for the callback methods:

#### Regular callback registration

```
set_read_no_store_callback(reg, SCML2_CALLBACK(method), syncType, tag)
set_callback(reg, SCML2_CALLBACK(method), syncType, tag)
set_read_callback(reg, SCML2_CALLBACK(method), syncType, tag)
set_write_callback(reg, SCML2_CALLBACK(method), syncType, tag)
set_word_read_callback(reg, SCML2_CALLBACK(method), syncType, tag)
set_word_write_callback(reg, SCML2_CALLBACK(method), syncType, tag)
set_post_write_callback(reg, SCML2_CALLBACK(method), syncType, tag);
```

The set\_callback registration function accepts callback methods with the following signatures:

```
void transportCallback(tlm::tlm_generic_payload&, sc_core::sc_time&, int tag)
void transportCallback(tlm::tlm_generic_payload&, int tag)
```

The set\_read\_callback and set\_read\_no\_store\_callback registration functions accept callback methods with the following signatures:

```
bool readCallback(DT& data, const DT& byteEnables, sc_core::sc_time&, int tag)
bool readCallback(DT& data, const DT& byteEnables, int tag)
bool readCallback(DT& data, const DT& byteEnables, sc_core::sc_time&,
                  const scml2::tlm2_gp_extensions& extensions int tag)
bool readCallback(DT& data, const DT& byteEnables,
```

```
const scml2::tlm2_gp_extensions& extensions int tag)
```

The `set_write_callback` registration function accepts callback methods with the following signatures:

```
bool writeCallback(const DT& data, const DT& byteEnables, sc_core::sc_time&, int tag)
bool writeCallback(const DT& data, const DT& byteEnables, int tag)
bool writeCallback(const DT& data, const DT& byteEnables, sc_core::sc_time&,
                  const scml2::tlm2_gp_extensions& extensions int tag)
bool writeCallback(const DT& data, const DT& byteEnables,
                  const scml2::tlm2_gp_extensions& extensions int tag)
```

The `set_word_read_callback` registration function accepts callback methods with the following signatures:

```
bool wordReadCallback(DT& data, sc_core::sc_time&, int tag)
bool wordReadCallback(DT& data, int tag)
bool wordReadCallback(DT& data, sc_core::sc_time&,
                      const scml2::tlm2_gp_extensions& extensions int tag)
bool wordReadCallback(DT& data, const scml2::tlm2_gp_extensions& extensions int tag)
```

The `set_word_write_callback` registration function accepts callback methods with the following signatures:

```
bool wordWriteCallback(const DT& data, sc_core::sc_time&, int tag)
bool wordWriteCallback(const DT& data, int tag)
bool wordWriteCallback(const DT& data, sc_core::sc_time&,
                      const scml2::tlm2_gp_extensions& extensions int tag)
bool wordWriteCallback(const DT& data,
                      const scml2::tlm2_gp_extensions& extensions int tag)
```

The `set_post_write_callback` registration function accepts callback methods with the following signature:

```
void postWriteCallback(int tag)
```

Additional notes:

- The `bool` return value indicates whether the access was successful. The transport style callbacks should use the TLM2 response status.
- In all the above callback methods, `tag` is an optional argument, only to be used when the callback is registered with a tag.
- Streaming burst accesses are unrolled into word accesses and subword accesses are converted into word accesses with byte enables.
- The `byteEnables` mask will contain `0xff` for enabled bytes and `0x0` for disabled bytes.
- The `wordReadCallback` and `wordWriteCallback` types are for word accesses only. For unaligned accesses or subword accesses, an error response is returned.
- The untimed callbacks (without the `sc_time` parameter) cannot be `SELF_SYNCING` callbacks.

### 2.5.5.2 Debug Callbacks

The `reg` object support the following registration API's for debug callbacks.

## Debug callback registration

```
set_debug_callback(reg, SCML2_CALLBACK(method), tag)
set_debug_read_callback(reg, SCML2_CALLBACK(method), tag)
set_debug_write_callback(reg, SCML2_CALLBACK(method), tag)
```

For debug callbacks, callback must have one of the following signatures:

```
unsigned int transportCallback(tlm::tlm_generic_payload&)
unsigned int transportCallback(tlm::tlm_generic_payload&, int tag)
```

The return value is the number of consecutive bytes successfully read or written. If the access cannot be executed, 0 must be returned.

### 2.5.5.3 Predefined Callbacks

The following functions are available to register specific read and write behavior to a register.

set_clear_on_read(reg)	Clears all bits of the memory when the memory is read.
set_set_on_read(reg)	Sets all bits of the memory when the memory is read.
set_clear_on_write_0(reg)	Clears all bits to which the bit 0 is written.
set_clear_on_write_1(reg)	Clears all bits to which the bit 1 is written.
set_write_once(reg)	Sets all bits to which the bit 0 is written.
set_set_on_write_0(reg)	Sets all bits to which the bit 0 is written.
set_set_on_write_1(reg)	Sets all bits to which the bit 1 is written.
set_write_once_error(reg)	Allows a value to be written to the memory once. All subsequent writes will return an error. See the note text given below.
set_write_once_ignore(reg)	Allows a value to be written to the memory once. All subsequent writes will be ignored. See the note text given below.

The write\_once callback function variants return a reference counted object of type scml2::write\_once\_state. This object can be used to reset the state, such that the memory becomes writable again, by calling reset() on it.

### 2.5.6 Access Restrictions

The register object supports access restrictions as described in “Access Restrictions” on page 29. The register supports all predefined access restrictions as specified in “Predefined Access Restrictions” on page 32, it supports both styles of access restriction callbacks:

```
scml2::access_restriction_result MyRestrictFtion(DT& data, DT& bit_enables, int tag)
scml2::access_restriction_result MyRestrictFtion(tlm::tlm_generic_payload& trans, int tag)
```

## 2.6 bitfield

The bitfield object is intended to access subword ranges in a register. It does not have its own storage, and must have a register as parent object. Bitfields are slightly different from the other storage objects, since they can be smaller than the 8-bit minimal access size of a TLM2 transport access. Like a register, they

cannot have any other aliases or registers; or by bound to the TLM2 target sockets. They also have a limited set of callbacks, only the `read` and `write` callbacks are allowed (no transport callbacks or callbacks with byte-enables and so on). Moreover, they have a limited list of access methods - only the ones with `datatype` as argument.

Other properties of `bitfield` object are:

- The `bitfield` can only be constructed with a register as parent. It has `name`, `offset`, and `size` parameters.
- It is possible to use it as a regular variable like a register, and all operators are overloaded.
- When a callback is registered with a `bitfield`, this will override the behavior of the parent register.
- When both the register as well as the `bitfield` have a callback registered, the register callback will be triggered. The rule of *most refined behavior* stops at the register. The register has additional APIs to call the `bitfield` callbacks from within the register callback implementation.
- `bitfield` objects can be attached to `reg` objects or other `bitfield` objects, to alias some of the bits in the parent object.
- The include file of the `bitfield` objects is `scml2/bitfield.h`.

## 2.6.1 Types

The `bitfield` class is templated with the underlying value type:

```
Template <typename DT> class bitfield
```

The following types are supported:

**Table 2-20 Supported Datatypes**

Supported Datatypes	Word Size
<code>unsigned char</code>	8
<code>unsigned short</code>	16
<code>unsigned int</code>	32
<code>unsigned long long</code>	64
<code>sc_dt::sc_bignum&lt;128&gt;</code>	128
<code>sc_dt::sc_bignum&lt;256&gt;</code>	256
<code>sc_dt::sc_bignum&lt;512&gt;</code>	512

When instantiating a `bitfield` object, it should have the same template value as its parent register.

## 2.6.2 Constructors

The following constructors are available:

```
bitfield(const std::string& name,
         reg<DT>& reg,
         unsigned int offset,
         unsigned int size)
bitfield(const std::string& name,
         bitfield<DT>& b,
```

```

        unsigned int offset,
        unsigned int size)

```

Creates a new bitfield. The `offset` and `size` arguments must be specified in bits.

### 2.6.3 Properties

The following methods are available to access the properties of a bitfield:

- `const std::string& get_name() const`  
Returns the full hierarchical name of the bitfield object.
- `unsigned long long get_offset() const`  
Returns the offset (in bits) of the bitfield start bit in the register.
- `unsigned long long get_size() const`  
Returns the size of the bitfield object in bits.
- `reg<DT>* get_register() const`  
Returns a pointer to the parent register object.
- `const std::string& get_description() const`  
`void set_description(const std::string&)`  
Gets/sets the description for the bitfield object. This description can be displayed in debuggers.

- `bool has_default_read_behavior() const;`  
`bool has_default_write_behavior() const;`  
`bool has_default_debug_read_behavior() const;`  
`bool has_default_debug_write_behavior() const;`  
`bool has_never_syncing_read_behavior() const;`  
`bool has_never_syncing_write_behavior() const;`  
`bool is_dmi_read_allowed() const;`  
`bool is_dmi_write_allowed() const;`

These are a set of APIs to query what type of behavior is associated with the `memory_alias` and whether DMI is enabled.

- `bool has_default_read_restriction() const;`  
`bool has_default_write_restriction() const;`

These APIs query whether there is a restriction set for the bitfield.

### 2.6.4 Behaviors

#### 2.6.4.1 Put/Get Access

The bitfield object supports a limited set of the different put/get behaviors as described in “[Behavior on page 26](#)”. The set of access functions supported according to the different types and styles defined in “[Behavior” on page 26](#) are detailed in the following table:

**Table 2-21 Access Functions**

Type	API	Callback	Watchpoints
Simple	Put/get	No	Yes

Type	API	Callback	Watchpoints
Simple-Debug	Put/get_debug	No	No
Trigger-Callbacks	Put/get_with_triggering_callbacks	Regular	Yes
Trigger-Debug-Callbacks	Put/get_with_triggering_debug_callbacks	Debug	No

For each variant, there are also different signatures available:

**Table 2-22 Signatures Available for Access Functions**

Style	Arguments
Bitfield access	DT

Additional notes:

- The Trigger-Callback access functions take an additional `sc_time` argument. They return the TLM2 response status, returned by the triggered callback.
- The `get()` and `get_debug()` methods for word or subword accesses return the read data, instead of passing it as an argument.
- The debug versions of these methods do not trigger watchpoints.

In total, this gives  $2 \times 4$  different access methods that are supported by the `memory` objects, summarized in the following table:

**Table 2-23 Access Methods Supported by Memory Objects**

Style	Bitfield Access
Type	
Simple	x
Simple-Debug	x
Trigger-Callbacks	x
Trigger-Debug-Callbacks	x

#### 2.6.4.2 Operators

A bitfield object can be converted to the underlying data type:

```
operator DT() const
```

The following assignment operators are available:

```
bitfield& operator=(DT value)
bitfield& operator =(const bitfield& b)
```

The following arithmetic assignment operators are available and behave as defined for the underlying data type:

```
bitfield& operator+=(DT value)
bitfield& operator-=(DT value)
```

```
bitfield& operator/=(DT value)
bitfield& operator*=(DT value)
bitfield& operator%=(DT value)
bitfield& operator^=(DT value)
bitfield& operator&=(DT value)
bitfield& operator|=(DT value)
bitfield& operator<<=(DT value)
bitfield& operator>>=(DT value)
```

The following prefix and postfix decrement and increment operators are available:

```
bitfield& operator--()
DT operator--(int)
bitfield& operator++()
DT operator++(int)
```

## 2.6.5 Callbacks

The `bitfield` object supports the callbacks listed in this section. For more information on callbacks, also see “[Callbacks](#)” on page 27.

### 2.6.5.1 Regular Callbacks



**Note** Standard `bitfield` callback implementations should not read or change any other bitfield value except the one they are registered on. The order of callback execution on the bitfields is undefined, hence reading or writing to another bitfield can lead to unintended situations. Use the post write callbacks for reading and adapting consistent values of multiple bitfields after the complete read/write operation has finished.

Every callback registration function accepts a few signatures for the callback methods:

#### Regular callback registration

- `set_read_no_store_callback(bitfield, SCML2_CALLBACK(method), syncType, tag)`
- `set_read_callback(bitfield, SCML2_CALLBACK(method), syncType, tag)`
- `set_write_callback(bitfield, SCML2_CALLBACK(method), syncType, tag)`
- `set_post_write_callback(bitfield, SCML2_CALLBACK(method), syncType, tag);`

The `set_read_callback` and `set_read_no_store_callback` registration functions accept callback methods with the following signatures:

- `bool readCallback(DT& value, sc_core::sc_time&, int tag)`
- `bool readCallback (DT& value, int tag)`
- `bool readCallback (DT& value, sc_core::sc_time&, const scml2::tlm2_gp_extensions& extensions int tag)`
- `bool readCallback (DT& value, const scml2::tlm2_gp_extensions& extensions int tag)`

The `set_write_callback` registration function accepts callback methods with the following signatures:

- `bool writeCallback(const DT& value, sc_core::sc_time&, int tag)`
- `bool writeCallback(const DT& value, int tag)`

- `bool writeCallback(const DT& value, sc_core::sc_time&, const scml2::tlm2_gp_extensions& extensions int tag)`
- `bool writeCallback(const DT& value, const scml2::tlm2_gp_extensions& extensions int tag)`

The `set_post_write_callback` registration function accepts callback methods with the following signature:

- `void postWriteCallback(int tag)`

Additional notes:

- The `bool` return value indicates that the access was successful.
- In all the above callback methods, `tag` is an optional argument, only to be used when the callback is registered with a tag.
- The untimed callbacks (without the `sc_time` parameter) cannot be SELF\_SYNCING callbacks.

### 2.6.5.2 Debug Callbacks

The `bitfield` object supports the following registration APIs for debug callbacks:

**Debug callback registration:**

- `set_debug_callback(bitfield, SCML2_CALLBACK(method), tag)`
- `set_debug_read_callback(bitfield, SCML2_CALLBACK(method), tag)`
- `set_debug_write_callback(bitfield, SCML2_CALLBACK(method), tag)`

For debug callbacks, callback must have one of the following signatures:

- `bool readCallback(DT& value, int tag)`
- `bool writeCallback(const DT& value, int tag)`

The `bool` return indicates whether the debug access was successful.

### 2.6.5.3 Predefined Callbacks

The following functions are available to register-specific read, and write behavior to a register.

<code>set_clear_on_read(bitfield)</code>	Clears all bits of the memory when the memory is read.
<code>set_set_on_read(bitfield)</code>	Sets all bits of the memory when the memory is read.
<code>set_clear_on_write_0(bitfield)</code>	Clears all bits to which the bit 0 is written.
<code>set_clear_on_write_1(bitfield)</code>	Clears all bits to which the bit 1 is written.
<code>set_set_on_write_0(bitfield)</code>	Sets all bits to which the bit 0 is written.
<code>set_set_on_write_1(bitfield)</code>	Sets all bits to which the bit 1 is written.

The `write_once` callback function variants return a reference counted object of type `scml2::write_once_state`. This object can be used to reset the state, such that the memory becomes writable again, by calling `reset()` on it.

## 2.6.6 Access Restrictions

The `bitfield` object supports access restrictions as described in “[Access Restrictions](#)” on page 29. The `bitfield` supports all predefined access restrictions as specified in “[Predefined Access Restrictions](#)” on page 32, it supports both styles of access restriction callbacks:

- `scml2::access_restriction_result MyRestrictFtion(DT& data, DT& bit_enables, int tag)`
- `scml2::access_restriction_result MyRestrictFtion(tlm::tlm_generic_payload& trans, int tag)`

## 2.7 router

The `router` is a SCML2 storage object that is intended to be used to create a configurable model, where memory access calls can be redirected to different storage components. Examples of models that can be built using a `router` are caches, memory controller, even interconnect components, or a dynamic address decoder. A `router` is similar to an `scml2::memory`, so it can be instantiated as a top-level memory object, but it does not implement any storage. It also does not have a default behavior; it requires a transport callback to implement its behavior. The key feature is, however, the additional mapping interface which allows routing certain memory regions to a `scml2::memory`, or an initiator socket (or anything implementing the `mappable_if`). Once a region is mapped, all accesses to that region will automatically be forwarded to the memory or socket without being evaluated in the `router` again. Only when the region is unmapped by the `router`, the transport implementation will be called again.

The include file of the `router` objects is `scml2/router.h`.

Additional properties of the `router` object are:

- It is possible to have a different mapping for reads and writes.
- The `router` has a limited set of put/get access methods, basically the ones that are similar to the TLM2 APIs.
- The map API maps a region of a certain size starting at the base address in the `router`, to a region in the destination, implementing the `mappable_if` interface starting from address offset. The `bool` return value will indicate failure if the region is already mapped, out of range, or if the request is not word-aligned.
- A `router` object is similar to a `memory` object, but it has no associated storage and no default behavior. A callback must be registered to the `router` object that implements the desired behavior of the accesses to this memory range.
- A `router` object can map a memory region to a region into a `memory` object, another `router` object, a `tlm2_gp_initiator_adapter` object, or an object that implements the `mappable_if` interface. Accesses to mapped regions do not trigger the attached callback, but are automatically forwarded to the destination object.
- The `router` object implements the `mappable_if` object, which means that it can be the destination for a mapped range of a `router` object.
- A `router` object cannot have aliases and/or registers.

### 2.7.1 Types

The `router` class is templated with the underlying value type:

```
Template <typename DT> class router
```

The following types are supported:

**Table 2-24 Supported Datatypes**

Supported Datatypes	Word Size
unsigned char	8
unsigned short	16
unsigned int	32
unsigned long long	64
sc_dt::sc_bignum<128>	128
sc_dt::sc_bignum<256>	256
sc_dt::sc_bignum<512>	512

## 2.7.2 Constructors

The following constructor is available:

```
router(const std::string& name, unsigned long long size)
```

Creates a new router. The *size* argument must be specified in words.

## 2.7.3 Properties

The following methods are available to access the properties of a memory\_alias:

- `const std::string& get_name() const`  
Returns the full hierarchical name of the of the router object.
- `unsigned long long get_offset() const`  
Always returns 0 for a router object.
- `unsigned long long get_size() const`  
Returns the size of the router object in words.
- `unsigned int get_width() const`  
Returns the data width of the router object in bytes.
- `bool is_dmi_enabled()`  
Returns true if DMI accesses are allowed for the object, false otherwise. DMI is enabled by default.
- `void enable_dmi()`  
`void disable_dmi()`  
Enables/disables DMI accesses for the object.
- `const std::string& get_description() const`  
`void set_description(const std::string&)`  
Gets/sets the description for the memory\_alias object. This description can be displayed in debuggers.

The router object can be bound to a TLM2 target socket via a port\_adapter (see “[Port Adaptors](#)” on page [77](#)). A target port adaptor may be bound to an SCML2 router as shown below.

```
// bind adaptor to memory
scml2::router my_router("my_router", 0x100);
(*my_port_adapter)(my_router);
```

## 2.7.4 Behaviors

### 2.7.4.1 Mapping APIs

The following methods are available to map or unmap memory regions:

- `bool map(unsigned long long base,  
 unsigned long long size,  
 mappable_if& destination,  
 unsigned long long offset);`

Maps the memory range [base, base + size] of the socket to which the `router` object is bound to the memory range [offset, offset + size] of the destination.

Possible destinations are: `memory`, `router`, `tlm2_gp_initiator_adapter`, and objects implementing the `mappable_if` interface.

The `base`, `size`, and `offset` arguments must all be specified in bytes.

If the mapping succeeds, `true` is returned; otherwise `false` is returned. The mapping will fail if:

- the mapped range overlaps with a previously mapped range, or
- if the mapped range is outside the memory range of the `router` object, or
- if the base address or size of the mapped range is not aligned with the width of the `router` object.

After mapping a region to a destination, all accesses coming to this region are automatically forwarded to this destination. If a callback is registered to `scml_router`, it is not invoked. If a burst goes across the boundary of a mapped region, then the burst is unrolled.

The `map()` method maps the memory range both for read access and write accesses.

- `bool map_read(unsigned long long base,  
 unsigned long long size,  
 mappable_if& destination,  
 unsigned long long offset);`

Same as the `map()` method, but the memory range is mapped only for read accesses.

- `bool map_write(unsigned long long base,  
 unsigned long long size,  
 mappable_if& destination,  
 unsigned long long offset);`

Same as the `map()` method, but the memory range is mapped only for write accesses.



**Note** Mapped ranges for read and for write accesses are completely independent. A mapped range for read accesses is allowed to overlap with a mapped range for write accesses.

- `bool unmap(unsigned long long base);`

Unmaps a previously mapped range for both read and write accesses.

Returns `true` if a mapped range is found and removed, otherwise `false`.

- `bool unmap_read(unsigned long long base);`

Same as the unmap( ) method, but the memory range is unmapped only for read accesses.

- `bool unmap_write(unsigned long long base);`

Same as the unmap( ) method, but the memory range is unmapped only for write accesses.

- `void unmap_all();`

Unmaps all previously mapped memory regions.



**Note** Mapping memory regions can be done:

- statically from the constructor of the module,
- or dynamically from the attached callback or from another SystemC thread.

#### 2.7.4.2 Transport Calls

The router object implements the following TLM2 methods:

- `void b_transport(tlm::tlm_generic_payload& trans, sc_core::sc_time& t)`  
`unsigned int transport_dbg(tlm::tlm_generic_payload& trans)`
- `bool get_direct_mem_ptr(tlm::tlm_generic_payload& trans,`  
`tlm::tlm_dmi& dmiData)`

These methods trigger the callbacks registered to the router object.

#### 2.7.4.3 Put/Get access

The router object supports some of the put/get behaviors as described in “[Behavior](#)” on page 26. So, only access methods that trigger callbacks are supported. The following set of access functions is supported according to the different types and styles as defined in “[Behavior](#)” on page 26:

**Table 2-25 Access Functions**

Type	API	Callback	Watchpoints
Trigger-Callbacks	Put/get_with_triggering_callbacks	Regular	Yes
Trigger-Debug-Callbacks	Put/get_with_triggering_debug_callbacks	Debug	No

For each variant, there are also different signatures available:

**Table 2-26 Signatures Available for Access Functions**

Style	Arguments
TLM2	address, dataPtr, data_length, byte_enablePtr, enableLength
TLM2-Word	address, dataPtr, data_length
Word	index, DT
Sub-Word	index, DT, size, offset

In the table 2-26, the arguments are as follows:

**Table 2-27 Arguments for Signatures of Access Functions**

Argument Name	C++	Description
address	unsigned long long address	Specifies the byte address as in TLM2 GP.
dataPtr	(const) unsigned char* data	Specifies the data array as in TLM2 GP.
data_length	unsigned int dataLength	Specifies the length of the transaction in bytes as in TLM2 GP.
byte_enablePtr	const unsigned char* byteEnablePtr	Specifies a byte enable array which can be 0 as in TLM2 GP.
enableLength	unsigned int byteEnableLength	Specifies the length of the byte enable array in bytes as in TLM2 GP.
index	unsigned long long index	Specifies the word index as specified by the DT template of the <code>memory</code> object.
DT	(const) DT& data	Specifies the data.
size	unsigned int size	Specifies the size of the access in bytes.
offset	unsigned int offset	Specifies the offset for the access in bytes.

Additional notes:

- The Trigger- Callback access functions take an additional `sc_time` argument. They return the TLM2 response status, returned by the triggered callback.
- The `put_debug_with_triggering_callbacks()` and `get_debug_with_triggering_callbacks()` calls must not use `byte_enables` (the `byteEnablePtr` must be 0), since TLM2 does not support byte enables for debug calls.
- The debug versions of these methods do not trigger watchpoints.

In total, this gives 2x8 different access methods that are supported by the `memory` objects, summarized in the following table:

**Table 2-28 Access Methods Supported by Memory Objects**

Style	TLM2	TLM2-Word	Word	Sub-Word
Type				
Trigger-Callbacks	x	x	x	x
Trigger-Debug-Callbacks	x	x	x	x

For a complete list with all arguments, see the `include` file of the `memory`: `scml2/router.h`.

## 2.7.5 Callbacks

The `router` object supports the callbacks listed in this section. For more information on callbacks, also see “[Callbacks](#)” on page [27](#).

### 2.7.5.1 Regular Callbacks

The `router` object only supports the `set_callback` registration-style callback methods.

## Regular callback registration:

- `set_callback(mem, SCML2_CALLBACK(method), syncType, tag)`
- It only accepts callback methods with the following signature:
- `void transportCallback(tlm::tlm_generic_payload&, sc_core::sc_time&, int tag)`



**Note** In all the above callback methods, tag is an optional argument, only to be used when the callback is registered with a tag.

### 2.7.5.2 Debug Callbacks

The memory object supports the following registration APIs for debug callbacks:

#### Debug callback registration

- `set_debug_callback(mem, SCML2_CALLBACK(method), tag)`

For debug callbacks, method must have one of the following signatures:

- `unsigned int transportCallback(tlm::tlm_generic_payload&)`
- `unsigned int transportCallback(tlm::tlm_generic_payload&, int tag)`

The return value is the number of consecutive bytes successfully read or written. If the access cannot be executed, 0 must be returned.

## 2.8 memory utilities

This section describes:

- [memory\\_index\\_reference](#)
- [mappable\\_if](#)
- [Callback Base Classes](#)
- [Convenience Functions](#)

### 2.8.1 memory\_index\_reference

The `memory_index_reference` object is returned by the `lvalue` version (non-const version) of the index operator (`operator[]`) of `memory` and `memory_alias` objects.

The `memory_index_reference` object forwards all operations to the referenced `memory` object.

The include file of the `memory_index_reference` objects is `scml2/memory_index_reference.h`.

The following sections describe:

- [Types](#)
- [Access Methods](#)
- [Operators](#)

#### 2.8.1.1 Types

The following type definitions are available:

```
typedef DT data_type
typedef memory_index_reference<DT> reference
```

### 2.8.1.2 Access Methods

The following access methods are available:

```
void put(const DT& value)
DT get() const

void put_debug(const DT& value)
DT get_debug() const
```

### 2.8.1.3 Operators

A `memory_index_reference` object can be converted to the underlying data type of the referenced [memory](#) object:

```
operator DT() const
```

The following assignment operators are available:

```
reference& operator=(DT value)
```

The following arithmetic assignment operators are available and behave as defined for the underlying data type of the referenced [memory](#) object:

```
reference& operator+=(DT value)
reference& operator-=(DT value)
reference& operator/=(DT value)
reference& operator*=(DT value)
reference& operator%=(DT value)
reference& operator^=(DT value)
reference& operator&=(DT value)
reference& operator|=(DT value)
reference& operator>>=(DT value)
reference& operator<<=(DT value)
```

The following prefix and postfix decrement and increment operators are available:

```
reference& operator--()
DT operator--(int)
reference& operator++()
DT operator++(int)
```

## 2.8.2 mappable\_if

The `mappable_if` object is the abstract interface that must be implemented by an object to be able to act as a destination for a mapped range of a [router](#) object.

The include file of the `mappable_if` objects is `scml2/mappable_if.h`.

The following section describes:

- [TLM API Methods](#)

### 2.8.2.1 TLM API Methods

The following methods must be implemented:

```
std::string get_mapped_name() const = 0
```

Should return the name of the mapped destination. For a [memory](#) or [router](#) object, this is the name of the object. For a [t1m2\\_gp\\_target\\_adapter](#) object, this is the name of the TLM2 initiator socket.

```
void register_bw_direct_mem_if(tlm::tlm_bw_direct_mem_if* bwInterface) = 0  
void unregister_bw_direct_mem_if(tlm::tlm_bw_direct_mem_if* bwInterface) = 0
```

Is called to register/unregister a pointer to a [tlm\\_bw\\_direct\\_mem\\_if](#) object. When the object that inherits from the [mappable\\_if](#) has to invalidate the DMI pointers, it has to call [invalidate\\_direct\\_mem\\_ptr\(\)](#) on each registered interface.



**Note** If a [tlm\\_bw\\_direct\\_mem\\_if](#) object is registered multiple times, it must only be stored once and the invalidate call must only be called once.

The following TLM2 API methods (see the *IEEE Std 1666 TLM-2.0 Language Reference Manual*) must be implemented:

```
void b_transport(tlm::tlm_generic_payload& trans, sc_core::sc_time& t) = 0  
bool get_direct_mem_ptr(tlm::tlm_generic_payload& trans, tlm::tlm_dmi& dmiData)  
= 0  
unsigned int transport_dbg(tlm::tlm_generic_payload& trans) = 0
```

## 2.8.3 Callback Base Classes

This section describes:

- [memory\\_callback\\_base](#)
- [memory\\_debug\\_callback\\_base](#)
- [router\\_callback\\_base](#)
- [router\\_debug\\_callback\\_base](#)
- [bitfield\\_read\\_callback\\_base](#)
- [bitfield\\_write\\_callback\\_base](#)
- [bitfield\\_debug\\_read\\_callback\\_base](#)
- [bitfield\\_debug\\_write\\_callback\\_base](#)

### 2.8.3.1 [memory\\_callback\\_base](#)

Base class for regular callbacks of [memory](#), [memory\\_alias](#), or [reg](#) objects.

The following virtual methods must be implemented:

```
void execute(tlm::tlm_generic_payload& trans, sc_core::sc_time& t) = 0
```

Implementation of the callback behavior.

```
bool has_never_syncing_behavior() const = 0
```

Returns `true` if the callback never synchronizes, otherwise `false`.

The include file of the [memory\\_callback\\_base](#) objects is `scml2/memory_callback_base.h`.

### 2.8.3.2 [memory\\_debug\\_callback\\_base](#)

Base class for debug callbacks of [memory](#), [memory\\_alias](#), or [reg](#) objects.

The following virtual method must be implemented:

```
unsigned int execute(tlm::tlm_generic_payload& trans) = 0
```

Implementation of the callback behavior.

The include file of the `memory_debug_callback_base` objects is `scml2/memory_debug_callback_base.h`.

### 2.8.3.3 `router_callback_base`

Base class for regular callbacks of `router` objects.

The following virtual methods must be implemented:

```
void execute(tlm::tlm_generic_payload& trans, sc_core::sc_time& t) = 0
```

Implementation of the callback behavior.

```
bool has_never_syncing_behavior() const = 0
```

Returns true if the callback never synchronizes, otherwise false.

The include file of the `router_callback_base` objects is `scml2/router_callback_base.h`.

### 2.8.3.4 `router_debug_callback_base`

Base class for debug callbacks of `router` objects.

The following virtual method must be implemented:

```
unsigned int execute(tlm::tlm_generic_payload& trans) = 0
```

Implementation of the callback behavior.

The include file of the `router_debug_callback_base` objects is `scml2/router_debug_callback_base.h`.

### 2.8.3.5 `bitfield_read_callback_base`

Templated base class for regular read callbacks of `bitfield` objects. The class has one template parameter, which is the data type of the `bitfield` object.

The following virtual methods must be implemented:

```
bool read(DT& value, sc_core::sc_time& t) = 0
```

Implementation of the callback behavior.

```
bool has_never_syncing_behavior() const = 0
```

Returns true if the callback never synchronizes, otherwise false.

The include file of the `bitfield_read_callback_base` objects is `scml2/bitfield_read_callback_base.h`.

### 2.8.3.6 `bitfield_write_callback_base`

Templated base class for regular write callbacks of `bitfield` objects. The class has one template parameter, which is the data type of the `bitfield` object.

The following virtual methods must be implemented:

```
bool write(const DT& value, sc_core::sc_time& t) = 0
```

Implementation of the callback behavior.

```
bool has_never_syncing_behavior() const = 0
```

Returns true if the callback never synchronizes, otherwise false.

The include file of the `bitfield_write_callback_base` objects is `scml2/bitfield_write_callback_base.h`.

### 2.8.3.7 `bitfield_debug_read_callback_base`

Templated base class for debug read callbacks of `bitfield` objects. The class has one template parameter, which is the data type of the `bitfield` object.

The following virtual method must be implemented:

```
bool read(DT& value) = 0
```

Implementation of the callback behavior.

The include file of the `bitfield_debug_read_callback_base` objects is `scml2/bitfield_debug_read_callback_base.h`.

### 2.8.3.8 `bitfield_debug_write_callback_base`

Templated base class for debug write callbacks of `bitfield` objects. The class has one template parameter, which is the data type of the `bitfield` object.

The following virtual method must be implemented:

```
bool write(const DT& value) = 0
```

Implementation of the callback behavior.

The include file of the `bitfield_debug_write_callback_base` objects is `scml2/bitfield_debug_write_callback_base.h`.

## 2.8.4 Convenience Functions

The following convenience functions are available in `scml2/utils.h`:

```
template <typename DT> DT extract_bits(const DT& v, unsigned int sizeBits,  
                                         unsigned int offsetBits)
```

Returns `sizeBits` bits from offset `offsetBits` of the data word `v`. `offsetBits` and `sizeBits` are specified in bits.

Little endian bit ordering is used (the offset of the 1sb is 0).

```
template <typename DT> DT insert_bits(const DT& v, const DT& rhs,  
                                         unsigned int sizeBits, unsigned int offsetBits)
```

Inserts `sizeBits` bits of the data passed in `rhs` at offset `offsetBits` in the data word `v` and returns the result. `offsetBits` and `sizeBits` are specified in bits.

Little endian bit ordering is used (the offset of the 1sb is 0).

## 2.9 Deprecated API's and Adapters

### 2.9.1 Callbacks

The following convenience callback functions are deprecated:

### 2.9.1.1 Disallow Access to Memory Object

The following functions are available to disallow the access to a memory object. These callbacks register a callback of type `memory_disallow_access_callback`.

#### Predefined API's to disallow access to a memory object

```
set_ignore_access(mem)
set_ignore_read_access(mem)
set_ignore_write_access(mem)
set_disallow_access(mem)
set_disallow_read_access(mem)
set_disallow_write_access(mem)
set_read_only(mem)
set_write_only(mem)
```

Where `mem` is the memory object to which the callback will be registered.

When an access is ignored, an *ok* response is returned; when an access is disallowed, an error response is returned. Reading a write-only memory or writing a read-only memory will also return an error response.

When an access is ignored or disallowed, the contents of the memory is not updated after the access.

The following functions are available to disallow the debug access to a memory. These callbacks register a debug callback of type `memory_disallow_debug_access_callback`. The callback ignores the access and returns 0. When an access is ignored or disallowed, the contents of the memory is not updated after the access.

#### Predefined API's to disallow debug accesses to a memory object

```
set_disallow_debug_access(mem)
set_disallow_debug_read_access(mem)
set_disallow_debug_write_access(mem)
```

where `mem` is the memory object to which the callback will be registered. When an access is ignored or disallowed, the contents of the memory is not updated after the access.

### 2.9.1.2 Callbacks on Predefined Behaviors

The following functions are available to register a user-defined callback, in combination with the predefined behaviors, where the user callback is called before the behavior callback:

#### Pre predefined callback registration

```
set_clear_on_read_callback(mem, object, callback, name, syncType)
set_clear_on_read_callback(mem, object, callback, name, syncType, tag)
set_word_clear_on_read_callback(mem, object, callback, name, syncType)
set_word_clear_on_read_callback(mem, object, callback, name, syncType, tag)
set_set_on_read_callback(mem, object, callback, name, syncType)
set_set_on_read_callback(mem, object, callback, name, syncType, tag)
set_word_set_on_read_callback(mem, object, callback, name, syncType)
set_word_set_on_read_callback(mem, object, callback, name, syncType, tag)
set_clear_on_write_0_callback(mem, object, callback, name, syncType)
set_clear_on_write_0_callback(mem, object, callback, name, syncType, tag)
set_word_clear_on_write_0_callback(mem, object, callback, name, syncType)
set_word_clear_on_write_0_callback(mem, object, callback, name, syncType, tag)
set_clear_on_write_1_callback(mem, object, callback, name, syncType)
set_clear_on_write_1_callback(mem, object, callback, name, syncType, tag)
```

```

set_word_clear_on_write_1_callback(mem, object, callback, name, syncType)
set_word_clear_on_write_1_callback(mem, object, callback, name, syncType, tag)
set_write_once_ignore_callback(mem, object, callback, name, syncType)
set_write_once_ignore_callback(mem, object, callback, name, syncType, tag)
set_write_once_error_callback(mem, object, callback, name, syncType)
set_write_once_error_callback(mem, object, callback, name, syncType, tag)
set_word_write_once_ignore_callback(mem, object, callback, name, syncType)
set_word_write_once_ignore_callback(mem, object, callback, name, syncType, tag)
set_word_write_once_error_callback(mem, object, callback, name, syncType)
set_word_write_once_error_callback(mem, object, callback, name, syncType, tag)
set_set_on_write_0_callback(mem, object, callback, name, syncType)
set_set_on_write_1_callback(mem, object, callback, name, syncType)
set_set_on_write_1_callback(mem, object, callback, name, syncType, tag)
set_word_set_on_write_0_callback(mem, object, callback, name, syncType)
set_word_set_on_write_1_callback(mem, object, callback, name, syncType)
set_word_set_on_write_1_callback(mem, object, callback, name, syncType, tag)

```

The following functions are available to register a user-defined callback, in combination with the above defined behaviors, where the user callback is called after the behavior callback:

### Post predefined callback registration

```

set_post_clear_on_write_0_callback(mem, object, callback, name, syncType)
set_post_clear_on_write_0_callback(mem, object, callback, name, syncType, tag)
set_post_clear_on_write_1_callback(mem, object, callback, name, syncType)
set_post_clear_on_write_1_callback(mem, object, callback, name, syncType, tag)
set_post_write_once_ignore_callback(mem, object, callback, name, syncType)
set_post_write_once_ignore_callback(mem, object, callback, name, syncType, tag)
set_post_write_once_error_callback(mem, object, callback, name, syncType)
set_post_write_once_error_callback(mem, object, callback, name, syncType, tag)
set_post_set_on_write_0_callback(mem, object, callback, name, syncType)
set_post_set_on_write_1_callback(mem, object, callback, name, syncType)
set_post_set_on_write_1_callback(mem, object, callback, name, syncType, tag)

```

Where:

<i>mem</i>	Is the memory object to which the callback will be registered.
<i>object</i>	Is a pointer to the class containing the callback method.
<i>callback</i>	<p>Is a pointer to a member function of the object class. It must have one of the following signatures:</p> <p>For the regular callbacks, it should have one of the <code>transportCallback</code>, <code>readCallback</code>, or <code>wordReadCallback</code> signatures, as listed above.</p> <p>For the post callbacks, it should be either:</p> <pre> void postWriteCallback() void postWriteCallback(int tag) </pre>
<i>name</i>	Is a string specifying the name of the callback function.

<code>syncType</code>	<p>Can be one of the following:</p> <ul style="list-style-type: none"> <li>• NEVER_SYNCING indicates that the callback is nonblocking and must never call <code>wait()</code>.</li> <li>• SELF_SYNCING indicates that the callback is blocking and may call <code>wait()</code>. The timing annotation is passed unmodified to the callback.</li> <li>• AUTO_SYNCING indicates that the callback is blocking and may call <code>wait()</code>. The memory object synchronizes before calling the callback. The timing annotation passed to the callback is always SC_ZERO_TIME.</li> </ul> <p>These types are defined in the <code>scml2/types.h</code> file.</p> <p>The Post predefined behavior callbacks do not support SELF_SYNCING callbacks (since the callback does not have a time argument).</p>
<code>tag</code>	Is a user-provided integer that is passed to the callback.

## 2.9.2 TLM2 Adapters

The `tlm2_gp_target_adapter` and `tlm2_gp_initiator_adapter` are deprecated in favor of the `port_adapters` described in [Port Adaptors](#).

### 2.9.3 `tlm2_gp_target_adapter`

`scml2::tlm2_gp_target_adapter` is an adapter that is used to bind a memory object to a tlm2 target socket. It takes a TLM2 GP transaction and forwards it to the memory objects. The adapter takes care of burst accesses (burst unrolling so that the different regions are accessed correctly) it also takes care of the AT to LT conversion for the TLM2 base protocol. It ensures that all accesses to memory objects can be executed as LT accesses (b\_transport semantics, that is, calling `wait()` and so on is allowed). The adapter does not touch any extensions so ignorable extensions are forwarded.

The adapter also implements the backward DMI interface, this means that the memories will use the target adapter to issue the invalidate DMI pointer calls to the initiator.

The `tlm2_gp_target_adapter` can be bound to any object implementing the `mappable_if`, for example, an `scml2::memory`.

The `tlm2_gp_target_adapter` is specific to the TLM2 base protocol, other protocol definitions may need their own adapter implementation to take care of specific burst unrolling features or for their specific AT to LT conversion. To create an adapter, it is required to create an object that implements the following:

- The `tlm::tlm_fw_transport_if` so that it can be bound to a target socket.
- A binding operation () with a `mappable_if` so that the storage objects can be bound to it.

The `tlm2_gp_target_adapter` object is used to bind an object that implements the `mappable_if` (for example, a `memory` or `router` object) to a `tlm_target_socket`.

All TLM2 API methods are forwarded to the object bound to the adapter.

The include file of the `tlm2_gp_target_adapter` objects is `scml2/tlm2_gp_target_adapter.h`.

```
template <unsigned int BUSWIDTH>
class tlm2_gp_target_adapter : public sc_core::sc_object,
    public tlm::tlm_fw_transport_if<>,
    public tlm::tlm_bw_direct_mem_if
{
    typedef tlm::tlm_base_target_socket<BUSWIDTH,
        tlm::tlm_fw_transport_if<>,
        tlm::tlm_bw_transport_if<>,
```

```

N,
POL> socket_type;

tlm2_gp_target_adapter(const std::string& name, socket_type& s);
void operator()(mappable_if& destination);
...
};

```

The following sections describe:

- [Types](#)
- [Constructors](#)
- [Binding](#)
- [Custom Forwarding](#)

#### 2.9.3.1 Types

The `tlm2_gp_target_adapter` class is templated with the `BUSWIDTH`:

```
template <unsigned int BUSWIDTH> class tlm2_gp_target_adapter
```

The `BUSWIDTH` must be the same as the `BUSWIDTH` of the TLM2 target socket to which the adapter is bound.

#### 2.9.3.2 Constructors

The following constructor is available:

```
tlm2_gp_target_adapter(const std::string& name,
                      tlm::tlm_base_target_socket<BUSWIDTH>& s)
```

Creates a target adapter and binds it to the TLM2 target socket.

#### 2.9.3.3 Binding

The following method is available to bind objects that inherit from the `mappable_if` object to the `tlm2_gp_target_adapter` object:

```
void operator()(mappable_if& destination)
```

Binds an object to the adapter class.

#### 2.9.3.4 Custom Forwarding

By default, `tlm2_gp_target_adapter` always forwards all TLM2 API methods to the first bound `mappable_if` object. It is possible to register a user-defined function to forward transactions to other bound `mappable_if` objects. This is done by calling `set_select_callback()`, and passing an SCML2 callback method which returns a `mappable_if` pointer for a given TLM payload.

For example:

```
MyModule(sc_module_name name) ... {
    ...
    adapter(memory1);
    adapter(memory2);
    set_select_callback(adapter, SCML2_CALLBACK(selectMemory));
    ...
}
```

```

scml2::mappable_if* selectMemory(tlm::tlm_generic_payload& trans) {
    if (...) {
        // Change the transaction address, and forward to memory 1
        trans.set_address(4);
        return &memory1;
    }
    else {
        // Change the transaction address, and forward to memory 2
        trans.set_address(8);
        return &memory2;
    }
}

```

## 2.9.4 tlm2\_gp\_initiator\_adapter

`scml2::tlm2_gp_initiator_adapter` is an object that is used to map a memory region of a `router` object to a `tlm_initiator_socket`. It implements the `mappable_if`, so you can use it as a destination for the map API's of the `router` and bind it to an initiator socket.

The `tlm2_gp_initiator_adapter` object is used to map a memory region of a `router` object to a `tlm_initiator_socket`.

The `tlm2_gp_initiator_adapter` object binds to the `tlm_initiator_socket` and implements the `mappable_if`.

The include file of the `tlm2_gp_initiator_adapter` objects is  
`scml2/tlm2_gp_initiator_adapter.h`.

```

template <unsigned int BUSWIDTH, int N = 1,
          sc_core::sc_port_policy POL = sc_core::SC_ONE_OR_MORE_BOUND>
class tlm2_gp_initiator_adapter : public sc_core::sc_object,
                                 public mappable_if,
                                 public tlm::tlm_bw_transport_if
{
...
tlm2_gp_initiator_adapter(const std::string& name, socket_type& s);
...
};

```

The following sections describe:

- [Types](#)
- [Constructors](#)

### 2.9.4.1 Types

The `tlm2_gp_initiator_adapter` is templated with the `BUSWIDTH`:

```
template <unsigned int BUSWIDTH> class tlm2_gp_initiator_adapter
```

The `BUSWIDTH` must be the same as the `BUSWIDTH` of the TLM2 initiator socket to which the adapter is bound.

#### 2.9.4.2 Constructors

The following constructor is available:

```
tlm2_gp_initiator_adapter(const std::string& name,  
                           tlm::tlm_base_initiator_adapter<BUSWIDTH>& s)
```

Creates an initiator adapter and binds it to the TLM2 initiator socket.

# Chapter 3

## FT Model Interface APIs and Objects

This chapter describes:

- [Modeling Objects](#)
- [Protocol Definitions](#)
- [API Definitions](#)
- [Protocol Checker](#)

### 3.1 Modeling Objects

This section covers the generic FT modeling extensions to the TLM2.0 base protocol standard. The different interface modeling APIs are introduced with a short description of their usage.

- [Payloads](#)
- [Sockets](#)
- [Port Adaptors](#)
- [Protocol States](#)
- [Alignment in FT Protocols](#)

#### 3.1.1 Payloads

```
scml2::ft_generic_payload
```

The FT modeling style comes with a new payload for transactions. This payload contains the same attributes and methods as the TLM2.0 base protocol standard payload, but is extended with additional APIs to access extensions. These APIs have been added to provide a link to the automated protocol conversion logic.

```
#define ADDnSET_EXT(payload, ext_type, attr_val)
#define SET_EXT_ATTR(payload, ext_type, attr_val)
#define GET_EXT_ATTR(payload, ext_type, attr_inst)
```

The APIs are encapsulated in a set of MACROS for ease of use.

- ADDnSET\_EXT: Adds the extension to the payload and sets value.
- SET\_EXT\_ATTR: Gets the extension from the payload and sets value.
- GET\_EXT\_ATTR: Gets the extension value for this payload.

While these MACROS provide ease of use, they do not always give the best simulation performance, for that purpose some of the underlying APIs are interesting to be used:

```
template <typename extension_type, typename attr_type >
extension_type *ft_generic_payload::get_extension_attr(extension_type*& tlm2Ext,
attr_type &attr_val)
```

- Retrieves the extension with type `extension_type` from the payload and extracts the value `attr_val`.
- This API does not do any protocol conversion. This can be used to access the protocol state since the conversion of this extension always happens in the socket, so no additional conversion checks are required.
- The extension pointer (`tlm2Ext`) can be reused, which avoids another lookup for the extension as would happen when using the macro's.

```
template <typename extension_type, typename attr_type >
void ft_generic_payload::set_extension_attr(extension_type*& tlm2Ext,
attr_type attr_val)
```

- Updates the value of the extension with `attr_val`. Since the TLM2.0 base protocol non-blocking transport APIs use the standard payload, it is required to do a conversion on the API argument.

```
ft_generic_payload* get_payload(tlm::tlm_generic_payload *tmp);
```

- This does a safe conversion from `tlm_generic_payload` to `ft_generic_payload`.

The TLM2.0 standard advises that transactions are pooled so that the memory management via reference counting (acquire and release APIs) can be supported. For this purpose, the SCML2 library provides with a generic memory manager that implements the TLM2 standard memory management interface (`claim()` and `free()`).

```
scml2::mem_manager<TYPE>
```

The FT payload relies on a modified extension for the additional APIs listed above. As with the FT payload, the extension modifications is limited to additional functionality so that full compatibility with the standard TLM2.0 extensions is maintained.

```
template <typename extension_type, typename attr_type >
class base_extension : public tlm::tlm_extension<extension_type>
```

The additional features for this extension base class are:

- Support for memory management. The memory management is fully controlled via the payload so that there is no need for additional memory management code anywhere in the models.
- An additional access API with support for protocol conversion. This API allows to reuse an extension pointer that was accessed via the `get_extension_attr` call of the payload. The API sets the value for the attribute and also checks if there is any protocol attribute conversion that needs to be maintained.

```
template <typename extension_type, typename attr_type >
void base_extension::set_attribute(tlm::tlm_generic_payload *payload, attr_type val)
```

The SCML2 payload extensions are used to create additional protocols and protocol attributes. They can also be used to create custom extensions to carry additional information with a transaction, this can be debugging and analysis information but is not limited to that.

An extension can be defined with the following macro:

```
DECLARE_EXTENSION(my_extension_type, unsigned int, 0);
```

The parameters for this extension definition are:

- The extension type that is created.
- The type for the attribute that is held in the extension.
- The default value for the attribute (only meaningful for types that support value assignment).

Custom extensions are used in the same way as any predefined extension:

```
unsigned int my_extension_value;
payload.get_extension_attr(my_extension_type, my_extension_value);
```

### 3.1.2 Sockets

Similar to the payload and the extension, also the sockets used in the FT modeling style are an extended version of the standard TLM2.0 sockets. The following sockets are defined for FT:

```
scml2::ft_initiator_socket
scml2::ft_target_socket

scml2::simple_initiator_socket
scml2::simple_target_socket
```

These sockets have additional API(s):

- `set_protocol`: It is required to use this API when using any of the protocol-specific extensions defined for FT modeling. The `set_protocol` API indicates the protocol extensions that will be used by this port and is used to drive the automated protocol conversion logic.
- `set_clock`: This API should be called by initiators and targets at the end of elaboration to specify the clock period at which they are operating. This will enable accurate tracing by TLM2 Port Trace and correct checking by Protocol Checker utility, see “[Protocol Checker](#)” on page 150.
  - The `scml2::ft_xxxx` sockets are similar to the sockets defined in the TLM2.0 standard.
  - The `scml2::simple_xxxx` sockets are similar to the sockets defined in the TLM2.0 utilities. They should be used when you have multiple ports each requiring a different implementation of the interfaces. They allow to register any member function of a component as an implementation of the transport calls (as long as they have the same argument list).



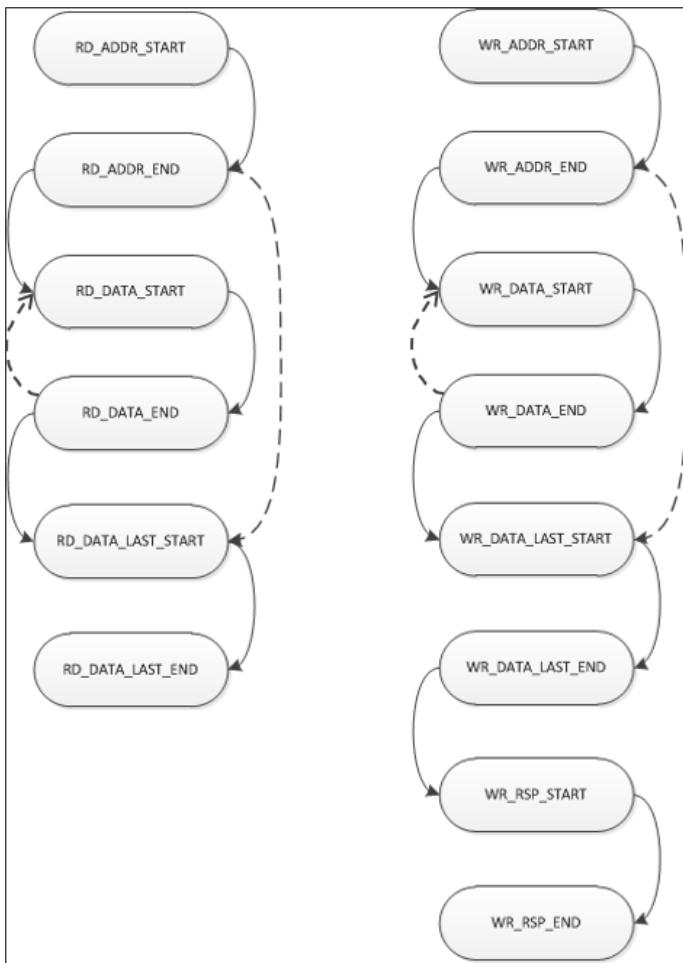
- An FT target should always register methods for both blocking and non-blocking transport calls. *Blocking to non blocking* conversion or *Non blocking to blocking* conversion done by the simple target socket will not take care of the mapping from GP to FT protocol.
- Multi pass-through sockets are not supported by Platform Architect/Virtualizer.

### 3.1.3 Port Adaptors

The SCML2 port adaptor objects provide a consistent interface to model device timing regardless of the underlying bus protocol. The initiator or target object that owns the port adaptor interacts with it using a simple API and optional callbacks that are registered for each protocol phase of interest.

Port adaptors invoke optional callbacks registered on protocol-agnostic events within the transaction life cycle.

**Figure 3-1 Port Adaptor Events**



Dashed transitions shown above are for payloads that have a single beat.

The event IDs map to protocol-specific states, as shown below.

**Table 3-1 Port Adaptor Event ID Mapping Protocol States**

Port Adaptor Event ID	TLM2 GP Phase	AXI Protocol State	AXI4Stream Protocol State	GFT Protocol State	CHI
CREDIT					LCREDI T
BUS_GRANT					CRESP
RD_ADDR_START	BEGIN_REQ	ARVALID		RD_CMD	REQ
RD_ADDR_END	END_REQ	ARREADY		RD_CMD_ACK	
RD_DATA_START		RVALID		RD_DATA	RDATA
RD_DATA_END		RREADY		RD_DATA_ACK	
RD_DATA_LAST_START	BEGIN_RESP	RVALID_LAST		RD_DATA_LAST	RDATA

Port Adaptor Event ID	TLM2 GP Phase	AXI Protocol State	AXI4Stream Protocol State	GFT Protocol State	CHI
RD_DATA_LAST_END	END_RESP	RREADY_LAST		RD_DATA_LAST_ACK	
WR_ADDR_START	BEGIN_REQ	AWVALID		WR_CMD	REQ
WR_ADDR_END	END_REQ	AWREADY		WR_CMD_ACK	
WR_DATA_START		WVALID	TVALID	WR_DATA	WDATA
WR_DATA_END		WREADY	TREADY	WR_DATA_ACK	
WR_DATA_LAST_START		WVALID_LAST	TVALID_LAST	WR_DATA_LAST	WDATA
WR_DATA_LAST_END		WREADY_LAST	TREADY_LAST	WR_DATA_LAST_ACK	
WR_RSP_START	BEGIN_RESP	BVALID			
WR_RSP_END	END_RESP	BREADY			
RSP_START					CRESP / SRESP
RSP_END					
SNOOP_ADDR_START					SNP
SNOOP_ADDR_END					
SNOOP_DATA_START					WDATA
SNOOP_DATA_END					
SNOOP_DATA_LAST_START					WDATA
SNOOP_DATA_LAST_END					
SNOOP_RSP_START					SRESP
SNOOP_RSP_END					

**Table 3-2 Port Adaptor Event ID Mapping Protocol States (Serial Protocols)**

Port Adaptor Event ID	PCIe / CXL.io (Initiator Socket)	PCIe / CXL.io (Target Socket)	CXL.mem (Initiator Socket)	CXL.mem (Target Socket)
WR_ADDR_START				
WR_DATA_LAST_START	PCIE_RX_REQ_WITH_DATA_TLP	PCIE_TX_REQ_WITH_DATA_TLP		CXL_MEM_M2S_REQ_WITH_DATA
WR_DATA_LAST_END				
WR_RSP_START	PCIE_RX_CPL_TLP	PCIE_TX_CPL_TLP	CXL_MEM_S2M_RESP	

<b>Port Adaptor Event ID</b>	<b>PCIe / CXL.io (Initiator Socket)</b>	<b>PCIe / CXL.io (Target Socket)</b>	<b>CXL.mem (Initiator Socket)</b>	<b>CXL.mem (Target Socket)</b>
WR_RSP_END				
RD_ADDR_START	PCIE_RX_REQ_TLP	PCIE_TX_REQ_TLP		CXL_MEM_M2S_REQ
RD_ADDR_END				
RD_DATA_START				
RD_DATA_LAST_START	PCIE_RX_CPL_WITH_DATA_TLP	PCIE_TX_CPL_WITH_DATA_TLP	CXL_MEM_S2M_RESPONSE_WITH_DATA	
RD_DATA_LAST_END				
RSP_START			CXL_MEM_S2M_RESPONSE	
RSP_END				
CREDIT	PCIE_RX_P_CREDIT_DLLP	PCIE_TX_P_CREDIT_DLLP		
	PCIE_RX_NP_CREDIT_DLLP	PCIE_TX_NP_CREDIT_DLLP		
	PCIE_RX_CPL_CREDIT_DLLP	PCIE_TX_CPL_CREDIT_DLLP		
TRANSMIT_STATUS	PCIE_RX_DLLP	PCIE_TX_DLLP		

Initiators and targets are free to register as many or as few callbacks as they wish. For example, a simple AXI FT target model could register a callback only for the ARVALID and AWVALID states to implement basic timing for read and write access to a SCML2 memory. The port adaptor will handle all of the AXI FT complexity including queuing transfers, protocol state transitions and timing annotation.

At each phase of the transaction, one of the following occurs:

- If no callback is registered, or a callback is registered and it returns TLM\_UPDATED, then the adaptor advances the protocol state and timing annotation internally.
- If a callback is registered and it returns TLM\_ACCEPTED, owner of the callback is responsible to explicitly advance the protocol state machine by invoking the update\_transaction API.

Port adaptors are constructed as shown below. The adaptor automatically looks up the `ft_protocol_tag` property of the socket to determine which protocol to use. If no such property exists, the adaptor defaults to `TLM2_GP`.

```
// create port adaptor and bind to socket
scml2::ft_target_socket<32> my_socket("my_socket");
scml2::target_port_adaptor *my_adaptor =
    scml2::target_port_adaptor::create("my_adaptor", &my_socket, &pClk);
```

Timing callbacks are then registered directly on the port adaptor, with an optional user-defined argument.

```
SCML2_REGISTER_TIMING_CBK(my_adaptor, scml2::RD_ADDR_END, this, onRD_ADDR_END);
SCML2_REGISTER_TIMING_CBK_WITH_ARG(my_adaptor, scml2::RD_DATA_START, this,
```

```
onRD_DATA_START, my_param);
```

A target port adaptor may be bound to an SCML2 memory as shown below.

```
// bind adaptor to memory  
scml2::memory my_memory("my_memory", 0x100);  
(*my_port_adapter)(my_memory);
```

In this case, timing callbacks may be registered on the memory object in a similar fashion to the normal storage/behavior callbacks.

```
scml2::set_timing_callback(my_memory, scml2::WR_ADDR_START, this, &onWR_ADDR_START);
```

Behavior callbacks may also be registered on the memory object as normal. The target port adaptor will automatically invoke the behavior callback at the appropriate point in the transaction; BEGIN\_REQ for TLM2\_GP protocols, and RD\_ADDR\_START and WR\_DATA\_LAST\_START events for AXI and GFT.

```
scml2::set_callback(my_memory, SCML2_CALLBACK(onRead), scml2::SELF_SYNCING);
```

Timing callbacks are always NEVER\_SYNCING. Behavior callbacks may be AUTO\_SYNCING, SELF\_SYNCING or NEVER\_SYNCING as normal. The port adaptor will automatically convert non-blocking transports to blocking if the SCML2 memory behavior callback is not NEVER\_SYNCING.

The SCML memory bound to the port adaptor must not mix NEVER\_SYNCING behavior callbacks with AUTO\_SYNCING and SELF\_SYNCING callbacks. For example, if there are registers accessed through the SCML memory that use AUTO or SELF\_SYNCING behavior callbacks, the SCML memory must also use an AUTO or SELF\_SYNCING callback.

More than one SCML memory may be bound to the target port adaptor, in which case a select callback should be registered to determine which memory is accessed by a given payload.

Port adaptors may be configured using named attributes and the set\_attribute API. This provides a basic level of timing configurability that may be suitable for simple devices. The timing attributes are only used when a clock has been bound to the adaptor, and when the registered callback for the relevant event returns tlm::TLM\_UPDATED, indicating that the port adaptor should advance the protocol state internally. If the registered callback returns tlm::TLM\_ACCEPTED, the user model is responsible for updating the timing annotation.

The table below lists the attribute names and default values for initiator port adaptors.

**Table 3-3 Attribute Names and Default Values for Initiator Port Adaptors**

Property Name	Description	Protocol				
		TLM2_GP	AXI	AXI4 Stream	GFT	CHI
invoke_timing_cbks	Controls whether timing callbacks are invoked (whether registered or not). Possible values are: <ul style="list-style-type: none"><li>• FT only (-1)</li><li>• Never (0)</li><li>• LT and FT (1)</li></ul>	-1	-1	-1	-1	

Property Name	Description	Protocol			
read_capacity	Maximum number of outstanding read and write transactions that may be issued on the initiator socket(AXI only). Additional transactions will be buffered by the adaptor and sent as soon as an outstanding transaction completes. Set to -1 to disable maximum transaction limits.		-1		
write_capacity			-1	-1	
total_capacity	read_capacity and write_capacity are ignored when total_capacity is non-zero.	1	0	0	1
rd_data_accept_cycles	Number of clock cycles to accept each read data beat.	0	0		0
wr_data_trigger_cycles	Number of clock cycles to be inserted before each write data beat.		0	0	0
wr_rsp_accept_cycles	Number of clock cycles to accept the write response.	0	0		
handle_tkeep	When set to a non-zero value, the port adaptor will automatically generate the tkeep array extension based on the bus width and transaction burst size. When set to 0, the user model is responsible for setting up the tkeep array extension (AXI4Stream only).			1	
dba_supported	When set to 1, port adaptor will use DBA (data beat array) extensions if supported by the target.		0		0
max_beats	Maximum number of beats in a transaction issued by the port adaptor. Transaction that exceed this value will be automatically split.		256		
auto_send_credits	Controls whether port adaptor automatically sends L-Credits to the target as soon as they available.				1
req_lcredits	(read-only) Number of request L-Credits currently available.				
chi_version	Specifies the CHI version to be used by the initiator.				B
src_id	Specifies the source id of the initiator (Request Node).				0

Property Name	Description	Protocol				
dct_int	Specifies whether Direct Cache Transfer (DCT) is to be used as hint by Request Node (RN). It should be set only when <code>chi_version</code> is E. Snoop transactions with DCT will be processed as normal Snoop transactions, if set.					0
wdata_lcredits	(read-only) Number of write data L-Credits currently available.					
sresp_lcredits	(read-only) Number of snoop response L-Credits currently available.					
max_lcredits	Specifies the maximum number of L-Credits sent at start of simulation.					2
try_dmi_for_b_transport	Optimize <code>b_transport()</code> calls by using DMI for these calls.  Possible values are: <ul style="list-style-type: none"><li>• <i>Optimization Enabled (1, Default)</i></li><li>• <i>Optimization Disabled (0)</i></li></ul>	1	1	1	1	1
write_data_interleaving	Set write data interleaving support on AXI Initiator Port Adaptor.  Possible values are: <ul style="list-style-type: none"><li>• <i>Interleaving disabled (0)</i></li><li>• <i>Interleaving enabled (1)</i></li><li>• <i>Legacy behavior (2, Default)</i></li></ul>		2			

The table below lists the attribute names and default values for target port adaptors.

**Table 3-4 Attribute Names and Default Values for Target Port Adaptors**

Property Name	Description	Protocol				
		TLM2_GP	AXI	AXI4 Stream	GFT	CHI
invoke_timing_cbks	Controls whether timing callbacks are invoked (whether registered or not). Possible values are: <ul style="list-style-type: none"><li>• FT only (-1)</li><li>• Never (0)</li><li>• LT and FT (1)</li></ul>	-1	-1	-1	-1	
invoke_behavior_cbks	Controls whether the behavior callback ( <code>b_transport</code> on bound mappable_if) is invoked by the port adaptor.	1	1	1	1	

Property Name	Description	Protocol			
read_capacity	Maximum number of outstanding read and write transactions that will be accepted by the target adaptor(AXI only). Additional transactions will be buffered and not accepted until an outstanding transaction completes. Set to -1 to disable maximum transaction limits.		-1		
write_capacity			-1	-1	
total_capacity	read_capacity and write_capacity are ignored when total_capacity is non-zero.	1	0	0	1
rd_cmd_accept_cycles	Number of clock cycles to accept a read address.	0	0		0
rd_data_trigger_cycles	Number of clock cycles to be inserted before each read data beat.		0		0
wr_cmd_accept_cycles	Number of clock cycles to accept a write address.	0	0		0
wr_data_accept_cycles	Number of clock cycles to accept each write data beat.		0		0
wr_rsp_trigger_cycles	Number of clock cycles to be inserted before write response.	0	0		
req_accept_cycles	Number of clock cycles to accept a bus request (GFT only).				0
collect_databeats	When set to non-zero value, the port adaptor will collect the data and byte_enables for each beat into a single buffer respectively which will be available at the start of the last write data beat (WR_DATA_LAST_START event). When set to 0, only the data and byte_enables of the last beat will be available (AXI4Stream only).			1	
dba_supported	When set to 1, port adaptor will use DBA (data beat array) extensions if supported by the target.		0		0
auto_behavior	Controls when the bound mappable_if::b_transport callback is invoked. For more details, see description of the target_port_adaptor bind to mappable_if method.	0	0	0	0

Property Name	Description	Protocol			
auto_send_credits	Controls whether port adaptor automatically sends L-Credits to the initiator as soon as they available.				1
snp_lcredits	(read-only) Number of snoop L-Credits currently available.				
rdata_lcredits	(read-only) Number of read data L-Credits currently available.				
cresp_lcredits	(read-only) Number of CRESP L-Credits currently available.				
chi_version	Specifies the CHI version to be used by the target.				B
dmt_enabled	Specifies whether Direct Memory Transfer (DMT) is enabled at the Slave Node (SN).				1
max_lcredits	Specifies the maximum number of L-Credits sent at start of simulation.				2

User models are free to use protocol-specific SCML2 transaction extensions, or custom extensions as required. The port adaptors also make use of the following generic extensions:

**Table 3-5 Extensions**

Extension	Description
scml2::trans_id_extension	This extension should be set by the initiator model before calling send_transaction. For protocols that support it, read and write data may be interleaved for payloads with different trans_id (AXI, AXI4Stream). If read or write data is provided for a payload by calling update_transaction, and another payload with the same trans_id has already started sending data, the new payload data will not be sent until all of the previous payload data phases have completed.  If not unique already, CHI initiator port adaptors will automatically override the trans_id_extension to a value not already in use by other active transactions.
scml2::burst_size_extension	This extension should be set by the initiator model before calling send_transaction. The burst_size, address, and data_length of the payload determine the number of beats.
scml2::can_accept_data_beat_array_extension	This extension is set by the initiator or target port adaptor during the address phase of the transaction. For protocols that support burst timing, the extension will be set to true by the port adaptor if the payload has more than one beat and no RD_DATA_END or WR_DATA_END callback has been registered. The user model may set the extension manually to force use of data beat array timing, prior to the WR_ADDR_START or RD_ADDR_END events respectively.

Extension	Description
scml2::data_beat_avail_extension	This array extension is set by sending port adaptor to indicate the clock cycle at which each data beat begins.
scml2::data_beat_used_extension	This array extension is set by the receiving port adaptor to indicate the clock cycle at which each data beat completes.
scml2::beat_index_extension	This extension is set by the initiator or target port adaptor before invoking a registered timing callback. The extension value indicates the index of the current data beat.

Initiators may set the size of the burst explicitly by setting scml2::burst\_size\_extension on the payload prior to calling send\_transaction. The burst size will not be changed by the port adaptor, but byte\_enables may be inserted if required to comply with the alignment requirements of the underlying protocol. If the burst\_size\_extension is not set, or set to 0, the initiator port adaptor will select an appropriate burst size based on the bus width and start address alignment.

### 3.1.3.1 Initiators

```
initiator_port_adaptor* create(const std::string& name, socket_type* socket,
                               sc_in<bool>* clk=NULL)
initiator_port_adaptor* create(socket_type* socket, sc_in<bool>* clk=NULL)
```

This static member function creates a new initiator port adaptor and binds it to the socket. You may supply a name for the new port adaptor object explicitly using the name argument or have the port adapter named automatically. The protocol for the adaptor is selected using the socket ft\_protocol\_tag property, or TLM2\_GP, if the property does not exist. If you specify a clock pointer using the optional clk argument, the clock port will be bound to the adaptor at the end of elaboration. The supported protocols are TLM2\_GP, AXI, GFT, AXI4Stream, and CHI.

```
void operator()(scml_clock_if* clk)
```

Binds a clock interface to the adaptor at end\_of\_elaboration. This is required to make use of the timing attributes, and will allow the adaptor to handle dynamic clock period changes.

```
template <class ObjT, typename FuncT>
void set_clock_changed_callback(ObjT* obj, FuncT func)
```

Sets callback to be invoked, if the clock period of the clock bound to the port adaptor changes.

```
sc_time clock_cycles_to_time(unsigned int cycles)
```

Convenience API to convert the given number of clock cycles to sc\_time using the bound clock interface.

```
std::string get_protocol()
```

Gets the protocol in use by the adaptor.

```
void set_attribute(const std::string& name, int value)
```

Sets a name-value attribute on the port adaptor. This is used to set protocol-specific parameters on the port adaptor.

```
int get_attribute(const std::string& name)
```

Gets the value of the named attribute.

```
void set_timing_callback(callback_event_enum event_id, timing_callback_base* cb)
```

Registers a callback function for a particular phase in the transaction life cycle. The callback function object can be created using `scml2::create_timing_callback` API. A convenient macro is also provided to simplify the syntax:

```
SCML2_REGISTER_TIMING_CBK(adaptor, event_id, obj, func)
```

The callback function should return `tlm::TLM_UPDATED` to indicate that the port adaptor should advance the protocol state internally, or `tlm::TLM_ACCEPTED` to stall the transfer until `update_transaction` is invoked with updated timing annotation and protocol state.

```
scml2::callback_event_enum get_event_id(unsigned int protocol_state)
```

Lookup callback event ID from protocol-specific state.

```
bool has_callback(scml2::callback_event_enum state_id)
```

Convenience API to check if there is any callback registered for a particular protocol phase.

```
unsigned int get_bus_width()
```

Gets the width of the bus in bytes.

```
unsigned int get_unique_trans_id()
```

Returns the smallest transaction ID that is not already in use by an outstanding transaction.

```
void enable_dmi()
```

Enables DMI for LT transfers. The port adaptor uses LT (`b_transport`) or FT (`nb_transport_fw`) depending on the value of the `scml2::timing_abstraction_level_switch`.

```
void disable_dmi()
```

Disables DMI for LT transfers. The port adaptor uses LT (`b_transport`) or FT (`nb_transport_fw`) depending on the value of the `scml2::timing_abstraction_level_switch`.

```
bool is_dmi_enabled()
```

Returns `true` if DMI is enabled. DMI is enabled by default.

```
tlm::tlm_generic_payload& alloc_and_init_trans(tlm::tlm_command cmd)
```

Allocates a new transaction and initializes some of the common fields. The caller is responsible for setting relevant fields such as `data_length` and `data_pointer` prior to invoking `send_transaction` or `b_transport`. Only transactions created using the `alloc_and_init_trans` API should be used with the port adaptor.

```
bool read(unsigned long long address, DT& data)
bool read(unsigned long long address, DT& data, sc_core::sc_time& t)
bool read(unsigned long long address, DT* data, unsigned int count)
bool read(unsigned long long address, DT* data, unsigned int count, sc_core::sc_time& t)
bool read(unsigned long long address, unsigned char* data, unsigned int count,
          sc_time& t)
```

Issues a `read` transaction and blocks until the transaction is complete. Returns `true` if the transaction response was successful. The type of transaction issued on the socket (blocking or non-blocking) depends on the `scml2::timing_abstraction_level` switch.

```
bool write(unsigned long long address, const DT& data)
bool write(unsigned long long address, const DT& data, sc_core::sc_time& t)
bool write(unsigned long long address, const DT* data, unsigned int count)
bool write(unsigned long long address, const DT* data, unsigned int count,
          sc_core::sc_time& t)
```

```
bool write(unsigned long long address, const unsigned char* data, unsigned int count,
          sc_time& t)
```

Issues a write transaction and blocks until the transaction is complete. Returns true if the transaction response was successful. The type of transaction issued on the socket (blocking or non-blocking) depends on the `scml2::timing_abstraction_level` switch.

```
unsigned int read_debug(unsigned long long address, DT& data)
unsigned int read_debug(unsigned long long address, DT* data, unsigned int count)
unsigned int read_debug(unsigned long long address, unsigned char* data, unsigned int
                      count)
```

Reads data via the socket `dbg_transport` interface. Returns the number of bytes read.

```
unsigned int write_debug(unsigned long long address, const DT& data)
unsigned int write_debug(unsigned long long address, const DT* data, unsigned int count)
unsigned int write_debug(unsigned long long address, const unsigned char* data, unsigned
                        int count)
```

Writes data via the socket `dbg_transport` interface. Returns the number of bytes written.

```
bool send_transaction(tlm::tlm_generic_payload& trans)
bool send_transaction(tlm::tlm_generic_payload& trans, const sc_time& delay)
```

Issues a non-blocking transaction at `sc_current_time` (+ `delay`) if possible, or queues it for sending if another transaction is currently in progress. Returns true unconditionally. Any number of transactions may be sent, and the port adaptor will forward them onto the socket one by one. This method may be called from within a timing callback function.

```
bool update_transaction(tlm::generic_payload& trans, sc_time& delay,
                        scml2::callback_event_enum event_id)
bool update_transaction(tlm::tlm_generic_payload& trans, sc_time& delay);
```

Attempts to advance the transaction state to the state `event_id` that is provided as method argument. If you skip the `event_id` argument you have to set the desired protocol state of the transaction payload explicitly before calling `update_transaction`. The port adaptor first verifies that the transition is valid, then advances the transaction state. Returns true if the transaction state was advanced successfully. This method may be called from within a timing callback function.



The `delay` argument may be modified by the port adaptor, so should be set explicitly before calling `update_transaction` again.

### 3.1.3.2 Targets

```
target_port_adaptor* create(const std::string& name, socket_type* socket,
                           sc_in<bool>* clk=NULL)
target_port_adaptor* create(socket_type* socket, sc_in<bool>* clk=NULL)
```

This static member function creates a new target port adaptor and binds it to the socket. You may supply a name for the new port adaptor object explicitly using the `name` argument or have the port adapter named automatically. The protocol for the adaptor is selected using the `socket_ft_protocol_tag` property, or `TLM2_GP`, if the property does not exist. If you specify a clock pointer using the optional `clk` argument, the clock port will be bound to the adaptor at the end of elaboration. The supported protocols are `TLM2_GP`, `AXI`, `GFT`, `AXI4Stream`, and `CHI`.

```
void operator()(scml_clock_if* clk)
```

Binds a clock interface to the adaptor at `end_of_elaboration`. This is required to make use of the timing attributes, and will allow the adaptor to handle dynamic clock period changes.

```

template <class ObjT, typename FuncT>
void set_clock_changed_callback(ObjT* obj, FuncT func)

```

Sets callback to be invoked, if the clock period of the clock bound to the port adaptor changes.

```

sc_time clock_cycles_to_time(unsigned int cycles)

```

Convenience API to convert the given number of clock cycles to sc\_time using the bound clock interface.

```

std::string get_protocol()

```

Gets the protocol in use by the adaptor.

```

void set_attribute(const std::string& name, int value)

```

Sets a name-value attribute on the port adaptor. This is used to set protocol-specific parameters on the port adaptor.

```

int get_attribute(const std::string& name)

```

Gets the value of the named attribute.

```

void set_timing_callback(callback_event_enum event_id, timing_callback_base* cb)

```

Registers a callback function for a particular phase in the transaction life cycle. The callback function object can be created using scml2::create\_timing\_callback API. A convenient macro is also provided to simplify the syntax:

```

SCML2_REGISTER_TIMING_CBK(adaptor, event_id, obj, func)

```

The callback function should return tlm::TLM\_UPDATED to indicate that the port adaptor should advance the protocol state internally, or tlm::TLM\_ACCEPTED to stall the transfer until update\_transaction is invoked with updated timing annotation and protocol state.

```

scml2::callback_event_enum get_event_id(unsigned int protocol_state)

```

Lookup callback event ID from protocol specific state.

```

bool has_callback(scml2::callback_event_enum state_id)

```

Convenience API to check if there is any callback registered for a particular protocol phase.

```

unsigned int get_bus_width()

```

Gets the width of the bus in bytes.

```

void operator()(mappable_if& dest)

```

Binds a scml2::mappable\_if (usually an SCML2 memory) to the port adaptor. The port adaptor will invoke the b\_transport API of the object at the appropriate phase in the transaction cycle. More than one mappable\_if object may be bound to the port adaptor, in which case, a select callback should be registered to route incoming transactions to the appropriate object.

The auto\_behavior attribute controls how the port adaptor and timing callbacks interact with the bound mappable\_if.

When auto\_behavior = 0: If a RD\_ADDR\_START or WR\_DATA\_LAST\_START timing callback is registered, the target model is responsible for invoking mappable\_if::b\_transport and advancing the protocol state to RD\_DATA\_LAST\_START or WR\_RSP\_START respectively. Returning TLM\_UPDATED from the timing callback will only advance the protocol state to RD\_ADDR\_END or WR\_DATA\_LAST\_END. If no such timing callback is registered, the port adaptor will automatically call the bound mappable\_if::b\_transport and advance the protocol state.

When `auto_behavior = 1`: The port adaptor will always invoke the bound `mappable_if::b_transport` automatically after `RD_ADDR_END/WR_DATA_LAST_END` state. This is the preferred modeling methodology. Returning `TLM_UPDATED` from the `RD_ADDR_START/WR_DATA_LAST_START` timing callback will advance the state to `RD_DATA_LAST_START/WR_RSP_START`, allowing the transaction to complete without further updates from the user model if desired.

```
void set_select_callback(memory_select_callback_base* cb)
```

Sets a callback to be invoked if there is more than one `mappable_if` bound to this port adaptor. The callback is invoked to determine which `mappable_if` should process an incoming transaction.

If a callback is registered for a particular event, the port adaptor ensures that it will be invoked, even if the target or initiator had skipped that phase. For example, an initiator bound to an AXI FT port adaptor has callbacks registered for `ARREADY`, `RREADY`, and `RREADY_LAST`.

- Initiator invokes `send_transaction()` for a four-beat burst transfer.
- Target returns `ARREADY`, initiator `ARREADY` callback is invoked.
- Target returns `RREADY_LAST`, port adaptor invokes `RREADY` callbacks for three phases, then `RREADY_LAST` for final phase.

### 3.1.3.3 AXI4 Stream Protocol

AXI4 Streaming initiators and targets have some additional complexity because of the streaming protocol semantics. User models that wish to take advantage of AXI4Stream-specific features such as `tkeep` and undefined length bursts should take a note of the following:

- The AXI4Stream initiator port adaptor will generate an error and exit the simulation if the user model attempts to send a read transaction.
- An AXI4Stream initiator does not need to know the complete burst size in advance. To support this use case, the initiator sets up the payload for the first beat, but sets `data_length` to 0 before invoking `send_transaction`. The initiator must register a `TREADY` timing callback in order to update the `data_ptr` contents for each beat before returning `TLM_UPDATED`, or invoking `update_transaction` with event `WR_DATA_START`. On the final beat, the initiator must call `update_transaction` with event `WR_DATA_LAST_START`.
- If the initiator knows the complete burst size in advance, it should set the payload `data_ptr` and `data_length` to the complete size before calling `send_transaction`. The port adaptor will take care of streaming the data correctly.
- In the same way as for other FT protocols, the AXI4Stream initiator can set the `scml2::burst_size_extension` before calling `send_transaction` to specify the burst size. If not set, the burst size defaults to the data bus width.
- By default, the port adaptor will setup the `tkeep` array extension, taking into account the burst size and bus width. To override this behavior and take control of the `tkeep` array for each beat, the initiator should call `set_attribute("handle_tkeep", 0)` before calling `send_transaction`.
- AXI4Stream target port adaptors can optionally collect all of the data and `byte_enables` for each beat into single buffers suitable for a standard SCML memory access on `TVALID_LAST`. This default behavior can be disabled by calling `set_attribute("collect_databeats", 0)` on the target port adaptor.

### 3.1.3.4 CHI

CHI is a credit-based protocol, where transfers may only be sent if the transmitter has received the appropriate L-Credit from the receiver. Port adaptors always track the number of L-Credits available for each channel at both the initiator and target. By default, port adaptors will send L-Credits to the transmitter as soon as they are available. This means at *simulation start*, two credits for each channel will be sent automatically. And, when an L-credit is consumed (for example, an CHI\_REQ received by the target), the port adaptor will automatically send it back to the transmitter on the following cycle. The number of credits of each type is available by calling `get_attribute()`.

The model may take control of issuing L-Credits instead by setting the `auto_send_credits` attribute on the port adaptor to 0. In this mode, the port adaptor will not send any L-Credits, but will still track the number of credits available at each channel and will not send a CHI\_REQ for example until a REQ L-Credit is available. The model may issue an L-Credit by allocating a payload, setting the protocol state and opcode extensions, and calling `send_transaction`.

A model can register a timing callback for the `scml2::CREDIT` event to be notified of new credit and credit returns. This is an advanced use-case and it is expected that most models will not need to register the callback or manually send L-Credits. Since the same timing callback is used on both receiver and transmitter side, the model must examine the `protocol_state` to determine (a) the type of L-Credit, and (b) whether it is a credit or a return. For an initiator for example, REQ, RDATA and SRESP are credits, but CRESP, WDATA, SNP are credit-returns.

```
SCML2_SET_TIMING_CBK(adaptor, scml2::CREDIT, this, & MyObject::HandleLCredit);
...
{ tlm::tlm_sync_enum MyObject::HandleLCredit(tlm::tlm_generic_payload& trans, sc_time& t)
{
    // get protocol state for custom handling
    GET_EXT_ATTR(&trans,
        scml2::chi_rnf_protocol_state_extension,
        scml2::protocol_state_enum,
        protocol_state);
    // handle credit
    switch (protocol_state) {
    ...
    }
    return tlm::TLM_ACCEPTED;
}
```

FT CHI protocol uses a single socket for both incoming and outgoing transactions. To support this, the `alloc_and_init()` and `send_transaction()` APIs are available for CHI target port adaptors to initiate L-Credit and Snoop transfers. The timing abstraction level switch is ignored for target `send_transaction`, and target initiated transactions are always sent using `nb_transport_bw`.

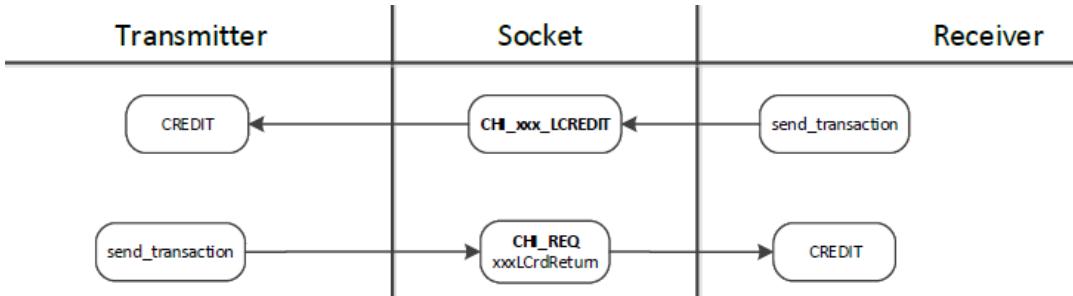
Because CHI supports so many different transaction types, a CHI-aware model must check the opcode extension inside the timing callback to determine the appropriate behavior.

- Initiator model is responsible for returning CompAck response (if it has set ExpCompAck) by calling `update_transaction RSP_START`.
- Initiator model may call `update_transaction RSP_START` to send the CompAck from inside `RD_ADDR_END` callback. In this case, the port adaptor will not send the CompAck until Comp [What does it stand for?] is received from target.
- If the initiator model calls `update_transaction` with `WR_DATA_START`, but the DBID response has not yet been received, the port adaptor will delay sending the write data until DBID has received from the target.
- If no `BUS_GRANT` timing callback has been registered by the initiator model, the port adaptor will keep track of any `RetryAck'd` transactions, and will automatically retry the oldest one when the `PCrdGrant` is received.

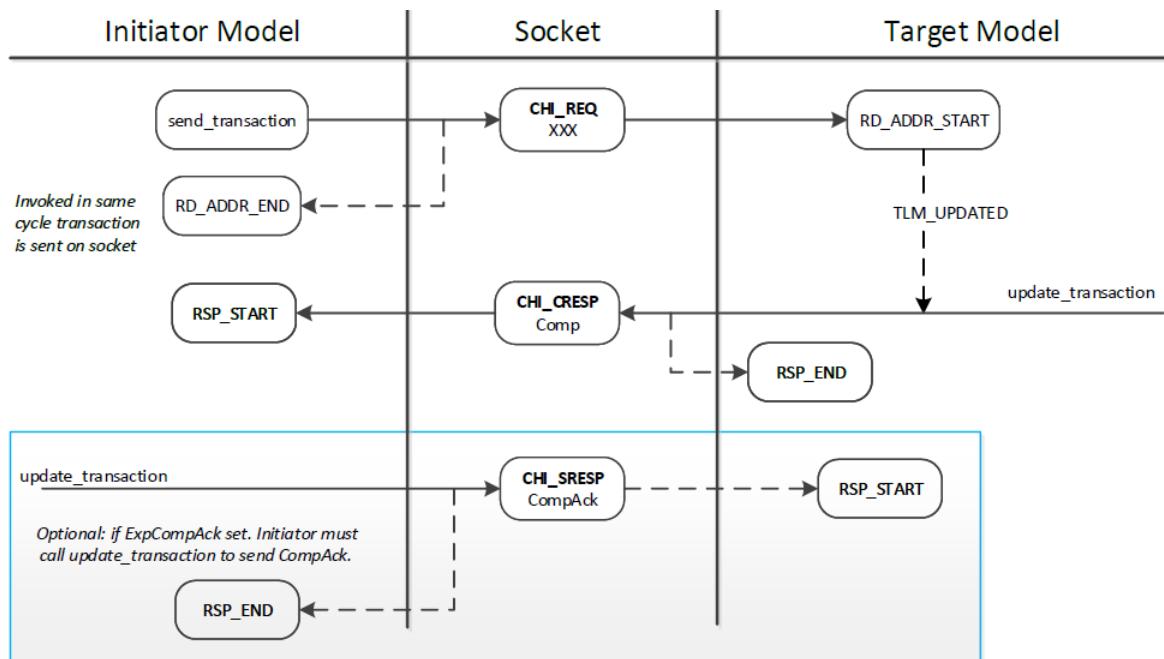
- If the initiator model has registered a BUS\_GRANT timing callback, the port adaptor will not track RetryAck'd transactions, and it is up to the model to retry the transaction or return the P-Credit.

The following figures shows the transaction flows for some basic transaction types:

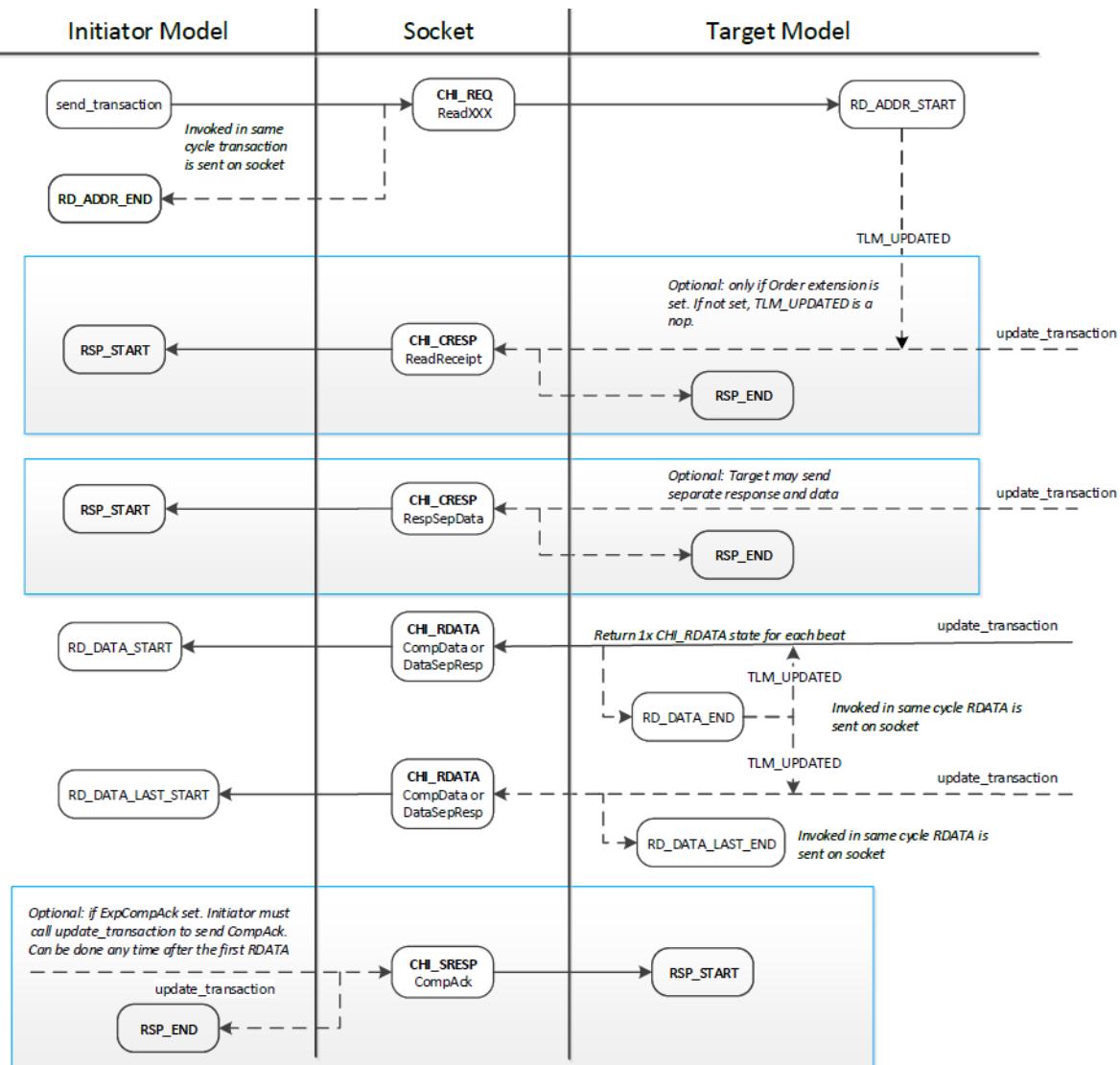
**Figure 3-2 Credit Send and Return Flow**



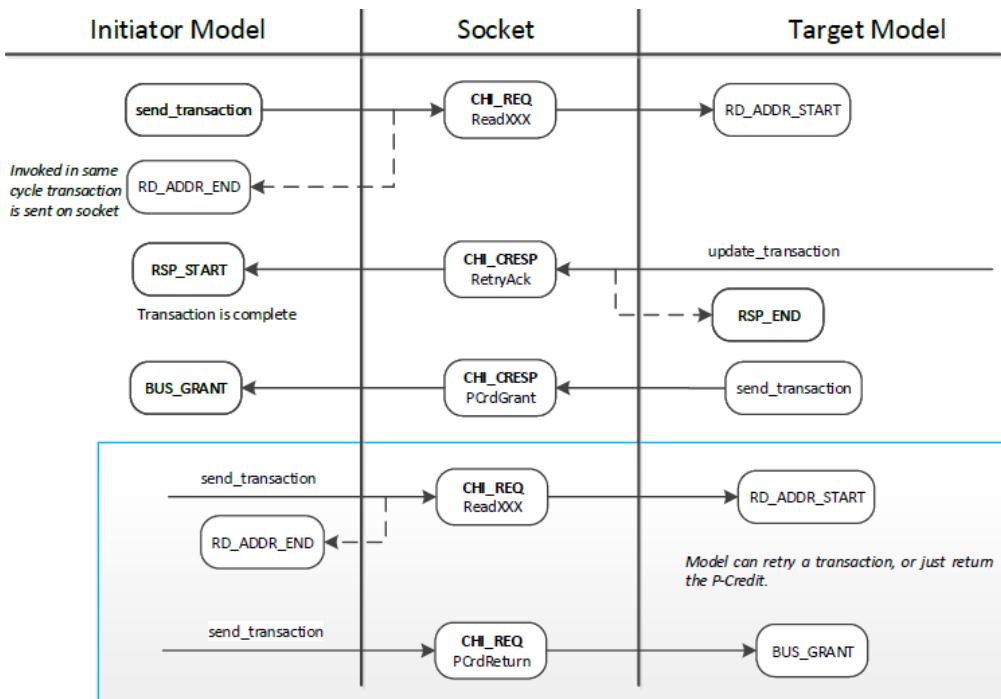
**Figure 3-3 Read Transaction Flow with no CompData**



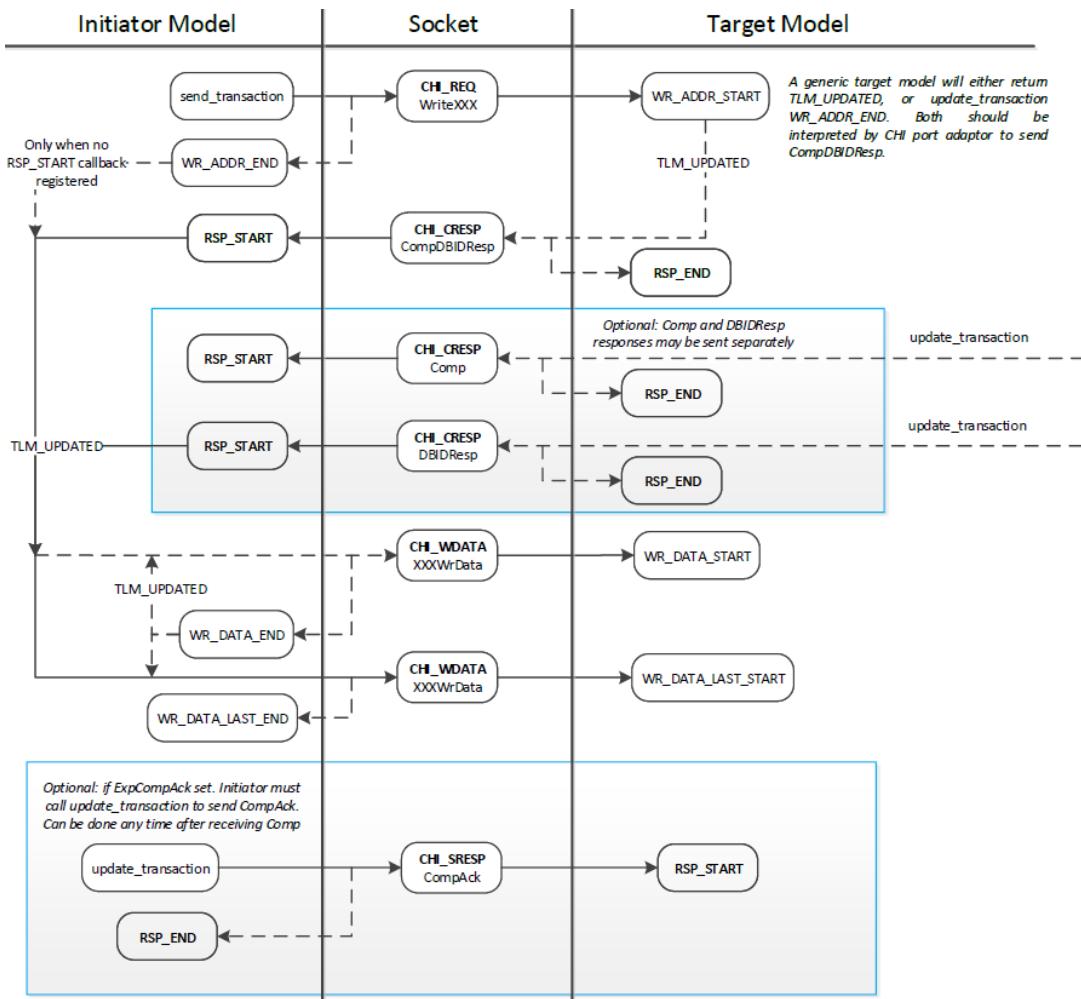
**Figure 3-4 Read Transaction Flow with CompData and Optional ReadReceipt**



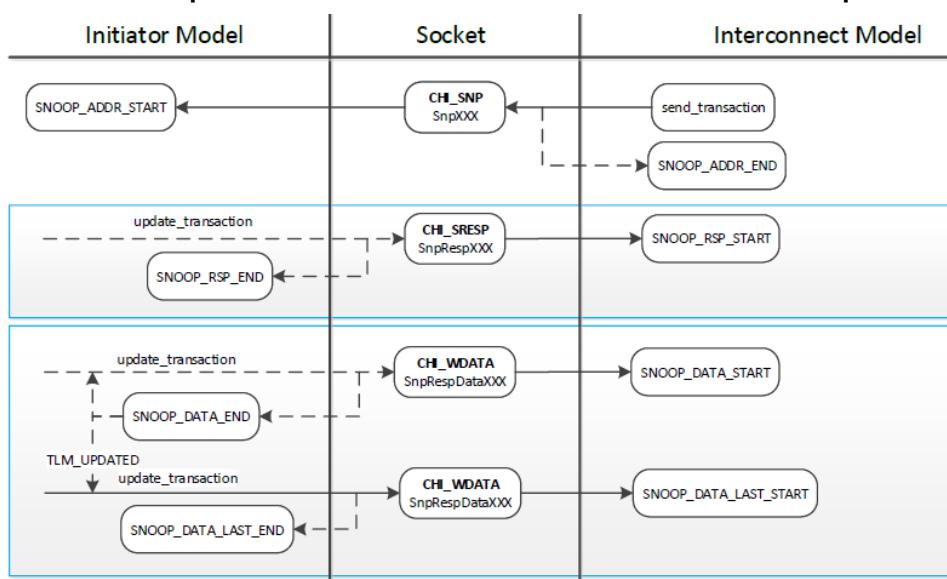
**Figure 3-5 Read Transaction Flow with Retry and Credit Return**



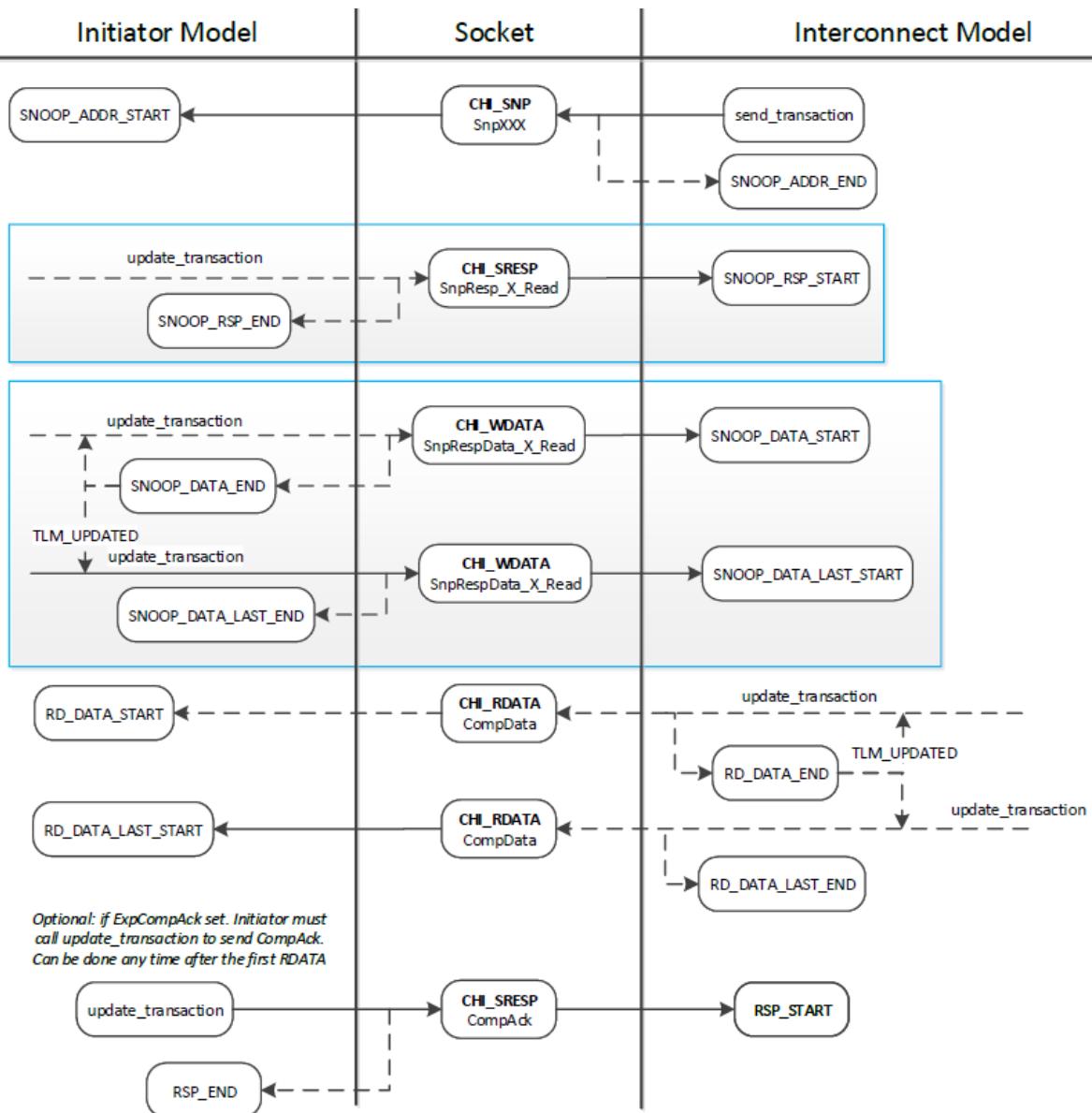
**Figure 3-6 Write Transaction Flow with Combined and Separate Comp and DBId Response**



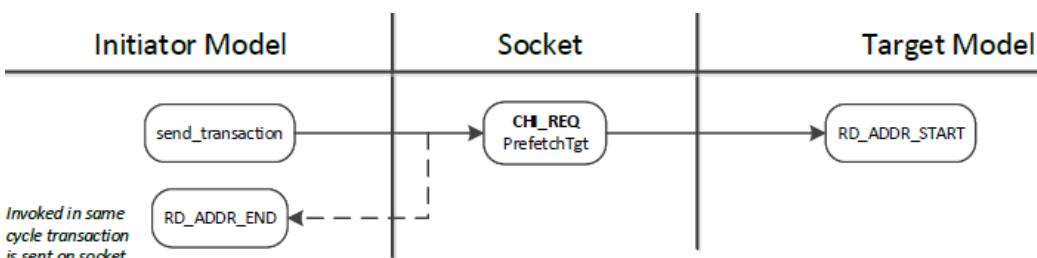
**Figure 3-7 Snoop Transaction Flow between Interconnect and Snooped Initiator**



**Figure 3-8 Snoop Transaction Flow between Interconnect, Snooped Initiator and Requesting Initiator**



**Figure 3-9 Prefetch Transaction Flow**



### 3.1.4 Protocol States

In the FT modeling API, the phases as found on the `nb_transport` calls for the TLM2.0 base protocol are implemented in a protocol-specific extension. This enables compatibility with the TLM2.0 base protocol. These phases are implemented as a protocol state extension, which has the same meaning as the TLM2.0 phases: they indicate timing points, represents the states in the protocol state machine and determine what attributes can be accessed or modified.

Phases typically come in pairs:

<b>TLM2-BP</b>	BEGIN_x	END_x
<b>AXI</b>	xVALID	xREADY
<b>AXI4Stream</b>	xVALID	xREADY
<b>GFT</b>	x	x_ACK

Phase pairs represent a 'transfer'; a basic information exchange between target and initiator. A transfer is initiated either by the initiator or by the target phase by using the first protocol state. The second is used by the receiving component to indicate the completion of the information exchange.

The timing of a transfer is important to consider; it defines the basic rules for delay modeling with phases.

FT API timing:

- The provider of information should implement the basic delay to make the data available.
- The receiver of information should implement the additional delay to accept the data
- For example in AXI:
  - Initiator can provide write data beat on every cycle, so minimal one cycle delay between consecutive transport calls to send WVALID.
  - Target can accept data immediately: 0 additional delay on return of the call.

Timing can be implemented in the following ways:

- Timing annotation:  
Initiate multiple transport calls from the same SystemC context, add initiation delay for each call.
- Explicit Synchronization:  
Use SCML2 FT modeling objects or SystemC synchronization mechanisms to initiate transport calls.

Timing annotation is the preferred approach in order to maximize model performance. Obviously, it is not always possible to use timing annotation. For example if a delay is dependent on multiple inputs, a model first will need to synchronize to make sure it can evaluate all inputs that might have come in at a certain time.



FT Initiators and FT Targets should always send `TLM_ACCEPTED` or `TLM_UPDATED`, as early transaction termination is not supported in FT modeling.

The *TLM2.0 payload* provides with a set of attributes that are common to most if not all memory mapped protocol interfaces. When creating a protocol definition for a real hardware protocol, these payload attributes are reused as much as possible, when protocol features match what is available in the TLM2.0 protocol then that attribute should be reused.

There are also a set of very commonly used attributes which can be used for a large set of hardware protocols. A generic FT extension is provided for these attributes. This avoids the need to provide with protocol conversion functions for these attributes.

The set of common extensions defined in the FT modeling style are available in the `scml2` namespace and are defined as follows:

- Transaction ID:

```
DECLARE_EXTENSION(trans_id_extension, unsigned int, 0);
```

This provides a transaction ID, which should be set by the initiator. The exact semantics and use of the ID is protocol specific.



The `transaction_id_extension` shall be set by the initiator, but may be overwritten by one or more interconnect components. This may be necessary, if the interconnect component wants to ensure that each initiator presents a unique `transaction_id` to the downstream components, for example as done in the AXI protocol. Once the `transaction_id` is overwritten in this way, the old value is lost (unless it was explicitly saved somewhere).

- Burst size:

```
DECLARE_EXTENSION(burst_size_extension, unsigned int, 0);
```

The burst size extension allows initiators to float narrow burst transfers. Default value 0 implies that burst size is equal to bus width.

- Wrap address

```
DECLARE_EXTENSION(wrap_addr_extension, unsigned long long, 0);
```

Indicates the start address for wrapping bursts, this is the address that will be issued first for the burst access. The address in the payload itself should refer to the begin address for the whole data array in order to remain consistent with TLM2.0 base protocol.

- Wrap Data

```
DECLARE_EXTENSION(wrap_data_extension, bool, false);
```

Is a flag for the target to indicate whether the wrap data is used.

### 3.1.5 Alignment in FT Protocols

FT protocols retain normal TLM2 semantics even for unaligned accesses; that is, the `transaction_address` specifies the address of the first byte to be transferred, and `data_length` always denotes the total number of bytes to be transferred.

The number of data phases to transfer the payload is a function of `address`, `data_length` and `burst_size_extension`, and will be protocol dependent. For example, AXI FT requires beats to be aligned to a beat boundary which can require an additional beat for unaligned accesses, as shown below.

```
num_beats = (trans.get_data_length() + burst_size - 1) / burst_size;
if ((trans.get_address() % burst_size) != 0)
    num_beats++;
```

The following examples show the bytes transferred during an unaligned access. The shaded cells indicate bytes that are not transferred, but there is no need to use `byte_enables` to align the start or end of the payload to a particular address boundary in FT.

**Figure 3-10 Examples of Unaligned Accesses on 32-bit AXI FT Bus**

CA AXI byte lanes				FT AXI bytes			
	31	16	15	0			
address	0x00	3	2	1	0	00,01,02,03	1st transfer
data_length	8	7	6	5	4	04,05,06,07	2nd transfer
burst_size	4						
address	0x1	3	2	1	0	01,02,03	1st transfer
data_length	8	7	6	5	4	04,05,06,07	2nd transfer
burst_size	4	B	A	9	8	08	3rd transfer
address	0x3	3	2	1	0	03	1st transfer
data_length	4	7	6	5	4	04,05,06	2nd transfer
burst_size	4						

**Figure 3-11 Examples of Unaligned Accesses on 64-bit AXI FT Bus**

CA AXI byte lanes								FT AXI bytes			
	63	32	31	0							
address	0x7	7	6	5	4	3	2	1	0	07	1st transfer
data_length	16	F	E	D	C	B	A	9	8	08,09,0A,0B	2nd transfer
burst_size	4	F	E	D	C	B	A	9	8	0C,0D,0E,0F	3rd transfer
		17	16	15	14	13	12	11	10	10,11,12,13	4th transfer
		17	16	15	14	13	12	11	10	14,15,16	5th transfer
address	0x1		7	6	5	4	3	2	1	0	01,02,03,04
data_length	4										1st transfer
burst_size	8										

## 3.2 Protocol Definitions

- FT GFT Protocol Definition
- FT AXI Protocol Definition
- FT ACE Protocol Definition
- FT AXI4 Stream Protocol Definition
- FT CHI Protocol Definition
- FT PCIe Protocol Definition
- FT CXL Protocol Definition

### 3.2.1 FT GFT Protocol Definition

SCML2 comes with a number of predefined protocol definitions. The Generic Fast Timed (GFT) protocol definition is created as an extended version of the TLM2 Base Protocol. It is a generic protocol in the sense that it is not created as an implementation of a specific hardware protocol, but rather as a set of protocol states and attributes that allow different implementations for an interconnect which therefore can mimic

different protocol properties. The protocol definition of GFT in that sense is more an extended set of basic protocol features as they can be found in real world hardware protocols. The GFT protocol has the following additional features on top of the TLM2 base protocol.

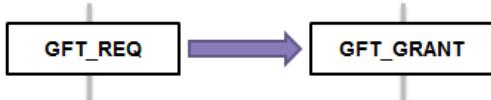
- support for burst data beat timing
- single channel transaction initiation

The GFT protocol definition is created based on the FT Modeling interfaces and adds a set of specific extensions for this protocol.

The GFT protocol defines the following different transfers which can be grouped in four types of transfers:

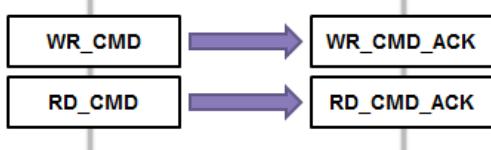
- **Request transfer:** Indicates the start of a transaction. With the request transfer, the initiator requests access to the bus. It is not necessary for pure point-to-point connections. Even when interacting with the bus, it is not required to start a transaction with the request transfer. It is not required to have any other attributes of the transaction set when starting the request transfer. A target or interconnect component should not look at any attributes at this point (also the address is not required to be set).

**Figure 3-12 Request Transfer**



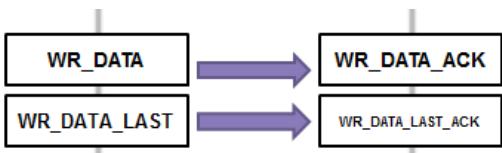
- **Command transfers:** These are an alternative starting point for a transaction. It is used by an initiator to setup transaction attributes. The address attribute should be available and all address decoding related attributes (burst\_type, lock\_type, access\_mode and access\_type) and the TLM2 base protocol attributes: tlm\_command and data\_length, it is not required to have the data array set. It is allowed to skip the request transfer and start the transaction with a command transfer.

**Figure 3-13 Command Transfer**



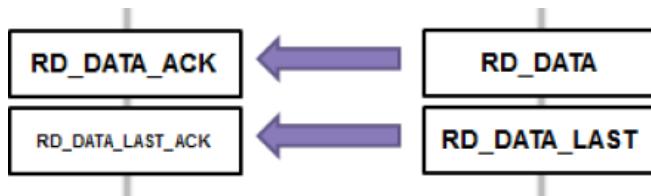
- **Write data transfers:** The write data transfers are used by the initiator to setup the write data beats. The target sets the response extension on the last transfer. It is possible to start the write data in parallel to the write command, but not earlier. When a set of write data transfers are started it is not allowed to interrupt them with write data transfers from another transaction. The number of write data transfers is determined by the TLM2.0 base protocol data\_length attribute which indicates the burst size. It is required that the last transfer is using the WR\_DATA\_LAST and WR\_DATA\_LAST\_ACK protocol state values. It is always required to complete the full set of beats of a burst, even if a decode error was issued by the interconnect.

**Figure 3-14 Write Data Transfer**



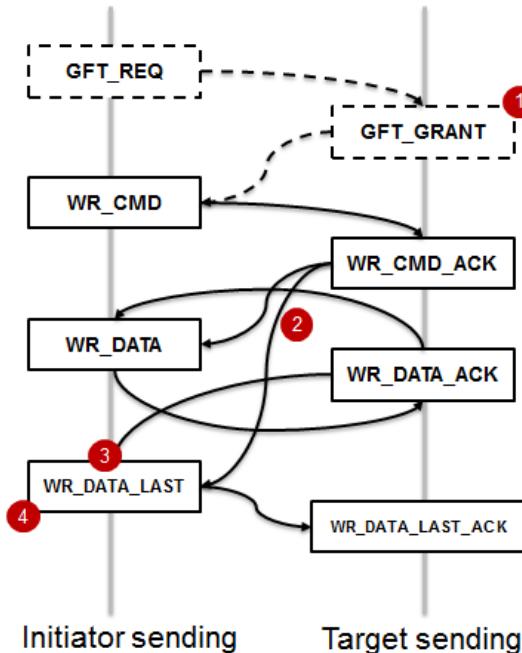
- Read data transfers: These transfers are initiated by the target to setup the read data values and read response extension. The read data transfers should always follow the read command and cannot be started in parallel to the command, although the first read data beat can come along with the command acknowledge. The response extension can be send along with any of the read data transfers, but it is required to complete all data beats of the transaction independent of the response status. The number of read data transfers is determined by the TLM2.0 base protocol data\_length attribute which indicates the burst size. It is required that the last transfer is using the RD\_DATA\_LAST and RD\_DATA\_LAST\_ACK protocol state values.

**Figure 3-15 Read Data Transfer**



The GFT protocol state machine uses the protocol states above to model all timing points of a generic protocol.

**Figure 3-16 GFT Protocol State Machine for Write Access**



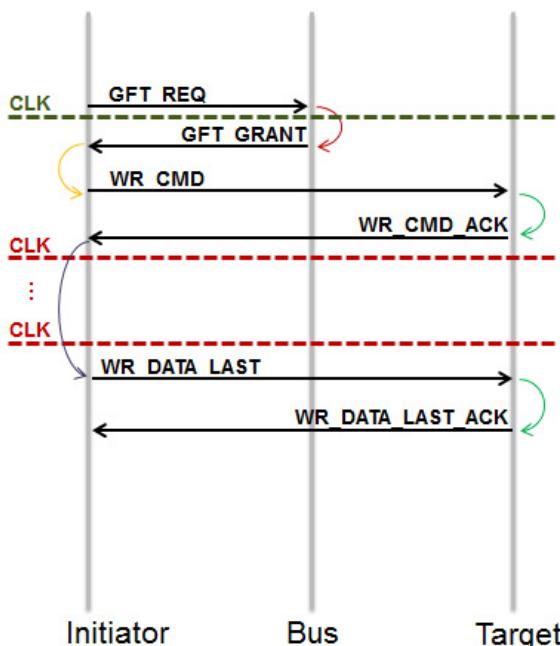
The protocol state machine for a write access looks as in the figure above. The figure shows the different protocol states that are used during a write transfer with all protocol states that are defined by the initiator on the left and all protocol states defined by the target on the right. The arrows give all permitted state transitions between the different protocol states. The dotted protocol states and arrow indicate the optional request transfer.

Notes on the protocol state machine:

- It is not required to implement the request phase. A target should implement a grant return on a request, but can simply return the grant protocol state if it is not interested in this transfer, as typically it will not be forwarded by the bus to the target.
- An Initiator can send a CMD, after receiving the GRANT for the REQ if the transaction was started through the request phase. The CMD phase can only be sent if the CMD channel is free, that is, if the previous CMD is acknowledged by the target.
- It is not allowed to send a WR\_DATA phase in parallel with a WR\_CMD, the WR\_DATA can only be sent after the WR\_CMD\_ACK is received.
- It is allowed for an initiator to skip the WR\_DATA transfer in case the burst length smaller than or equal to the data width of the socket. It is also allowed to skip WR\_DATA transfers and to jump immediately to the WR\_DATA\_LAST. The initiator should ensure that all write data is available in this case and that the timing annotation comprises of all setup delay for all data beats. In this case, the initiator assumes that target and interconnect can calculate the timing delay for the full transaction, and do not need additional SystemC synchronization to achieve this.
- Similarly, the initiator can skip certain write data transfers during the transaction, but it can only jump forward to the last transfer.
- It is required to issue WR\_DATA\_LAST.

An example of an implementation of the GFT write data transaction is shown below through a message sequence chart. It shows the initiator, bus and target and the different TLM2.0 interface method calls as they are done by these components. The clock boundaries show the required timing synchronization (dark green line) and timing annotation (red lines) for the transaction.

**Figure 3-17 GFT Write Data Transaction**



The transaction starts with the initiator issuing a GFT\_REQ on the forward non-blocking transport API. The bus requires a synchronization with SystemC to implement the bus arbitration (it needs to make sure it sees all incoming transactions that happen at the same time). The bus will send the GFT\_GRANT over the

backward transport path once the arbitration is completed (minimum one cycle delay). The initiator can respond with a WR\_CMD that it sends in the return path of the nb\_transport\_bw call from the bus. The bus forwards this call to the target over the forward non-blocking interface path. If the target can handle the transaction immediately, it can complete the command transfer on the return path to the bus. At that point, the bus will call the initiator with the information from the target. In the end, the bus is forwarding information from initiator to target and vice versa and all calls are happening without synchronization with SystemC. Timing annotation is used everywhere, except for the arbitration. In the above example, the transaction is one word long or the initiator is skipping all intermediate write data transfers.

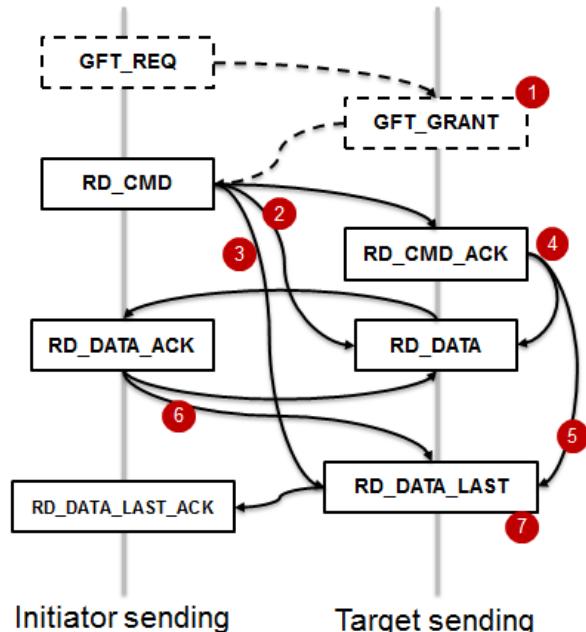
An important aspect of the protocol definition is to apply the FT modeling timing annotation rules to the protocol, to ensure that it is clear which component should take care of the transfer timing. The following table gives the overview of the timing annotation requirements for all protocol state transitions.

**Table 3-6 Timing Annotation Requirements**

Start	End	Who	Delay to be implemented	Minimum
GFT_REQ	GFT_GRANT	Bus	Arbitration delay.	1 clock cycle in case of bus, 0 otherwise
GFT_GRANT	WR_CMD	Initiator	Command setup time.	0
WR_CMD	WR_CMD_ACK	Target	Command accept time including address decoding.	0
WR_CMD_ACK	WR_DATA	Initiator	First beat data ready delay: delay between command accept return and first write data beat.	1
WR_CMD_ACK	WR_DATA_LAST	Initiator	Write data ready delay: delay between command acceptance and all data available, in case of burstlength is greater than 1, no assumption can be made on the timing of the other beats.	1 clock
WR_DATA	WR_DATA_ACK	Target	Write data beat accept delay, time to accept data coming from the initiator.	0
WR_DATA_ACK	WR_DATA	Initiator	Interbeat delay, time for the initiator to setup next beat of data.	1 clock cycle
WR_DATA_ACK	WR_DATA_LAST	Initiator	Final interbeat delay, time for the initiator to setup the remainder of the data.	1 clock cycle
WR_DATA_LAST	WR_DATA_LAST_ACK	Target	Final write data accept delay, time to accept data coming from initiator, includes time to setup response	0

In case of a read transaction, the protocol state machine looks as shown below. The figure shows the different protocol states that are used during a read transfer with all protocol states that are defined by the initiator on the left and all protocol states defined by the target on the right.

**Figure 3-18 Protocol States During Read Transaction**



The arrows give all permitted state transitions between the different protocol states. The dotted protocol states and arrow indicate the optional request transfer.

Notes on the protocol state machine:

1. It is not required to implement the request phase. A target should implement a grant return on a request, but can simply return the grant protocol state if it is not interested in this transfer, as typically it will not be forwarded by the bus to the target.
2. An Initiator can send a CMD, after receiving the GRANT for the REQ if the transaction was started through the request phase. The CMD phase can only be send if the CMD channel is free, that is, if the previous CMD is acknowledged by the target.
3. A target is allowed to skip the RD\_CMD\_ACK protocol state and respond to a RD\_CMD with a RD\_DATA protocol state in case it can start the read data transfers at the same time as the read command acknowledge. In this case, the end of the command transfer is implied, the initiator should assume that the read command transfer is completed and is allowed to start another one.
4. The same applies for RD\_DATA\_LAST in case the burstlength equals one word.
5. When a target returns a RD\_CMD\_ACK state to complete the read command transfer it is required to start the next transfer (RD\_DATA) from a different SystemC context. This is a TLM2.0 standard requirement. The new transfer should be started on the backward path and it is not allowed to call the TLM interfaces from each others context.
6. Same applies for RD\_DATA\_LAST in case burstlength equals one word.

7. It is allowed to skip read data transfers and jump forward to the last data beat of the transaction.

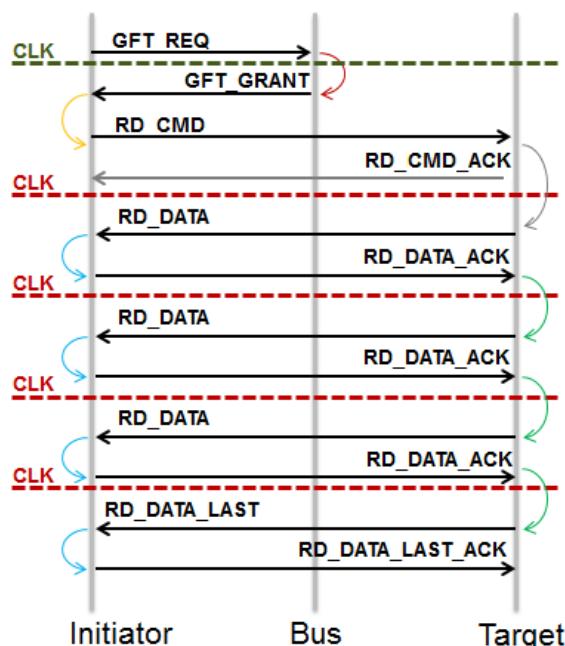


For read transactions there is the common `beat_timing_request_extension` which is used by the initiator to indicate whether it is interested to receive beat timing. A target is not obliged to honor the request.

8. It is always required to use the RD\_DATA\_LAST transfer.

An example of an implementation of the GFT read data transaction is shown below through a message sequence chart. It shows the initiator, bus and target and the different TLM2.0 interface method calls as they are done by these components. The clock boundaries show the required timing synchronization (dark line) and timing annotation (lighter lines) for the transaction.

**Figure 3-19 GFT Read Data Transaction**



As with the write transaction the initiator starts the transaction with a call on the forward non-blocking API. The bus needs a one cycle delay at least to implement the arbitration synchronization, but then calls the initiator on the backward path with a GFT\_GRANT protocol state. Hereafter, the initiator can continue with the read command that the bus simply forwards to the target. The target skips the acknowledge on the read command and jumps forward to start the read data transfers. Each read data transfer adds at least one clock cycle to the transaction timing. All further calls are done with timing annotation and again the bus ensures that all transaction calls are routed correctly between initiators and targets. The same transaction could be initiated with a read command immediately in which case the bus is required to forward the read command to the target after the arbitration delay, the target can add further delay before sending the command acknowledge back to the initiator.

The timing modeling rules for all possible protocol state transitions of a read transaction are listed in the table below:

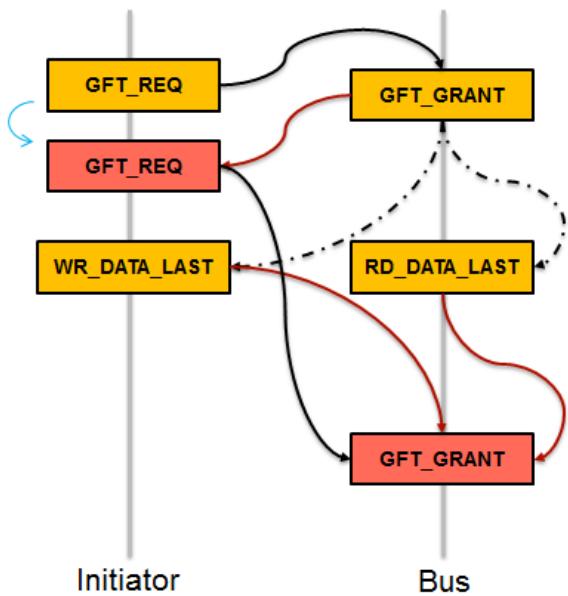
**Table 3-7 Timing Modeling Rules**

Start	End	Who	Delay to be Implemented	Minimum
GFT_REQ	GFT_GRANT	Bus	Arbitration delay	1 clock cycle in case of bus, 0 otherwise
GFT_GRANT	RD_CMD	Initiator	Command setup time	0
RD_CMD	RD_CMD_ACK	Target	Command accept time including address decoding.	0
RD_CMD	RD_DATA	Target	First beat data ready delay when command accept is zero.	1 clock cycle
RD_CMD	RD_DATA_LAST	Target	Transaction delay; total delay to handle the transaction after receiving a command in case no beat timing is available; no assumptions can be made for the other timing points.	1 clock cycle
RD_CMD_ACK	RD_DATA	Target	First beat data ready delay; delay between command accept return and first read data available.	1 clock cycle
RD_CMD_ACK	RD_DATA_LAST	Target	Data ready delay; total delay to provide with all data after returning command accept; no assumptions can be made about the other timing points.	1 clock cycle
RD_DATA	RD_DATA_ACK	Initiator	Read data accept delay; delay for the initiator to accept the incoming data.	0
RD_DATA_ACK	RD_DATA	Target	Inter beat delay; time for the target to setup the next data beat	1 clock cycle
RD_DATA_ACK	RD_DATA_LAST	Target	Final inter beat delay; time for the target to setup the remainder of the data.	1 clock cycle
RD_DATA_LAST	RD_DATA_LAST_ACK	Initiator	Final read data accept delay; delay for the initiator to accept the remaining incoming data.	0

The final set of rules of the GFT protocol definition relate to state transitions between different transactions. These rules define when an initiator is allowed to start a new transaction.

The GFT protocol can be used by an initiator and interconnect to model a shared bus communication. An initiator should wait until the request transfer is completed before initiating a new request. The interconnect or bus should enforce that only a single transaction is active at any time, the initiator can still start a new request transfer, but the interconnect will only complete the transfer once the previous transaction is completed. As such the bus is in control of the communication parallelism that is available in the system by controlling the timing points to complete the request transfer or the command transfer.

**Figure 3-20 GFT Protocol Used to Model Shared Bus Communication**



### 3.2.2 FT AXI Protocol Definition

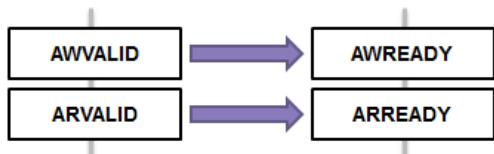
One of the most common protocols that are used in embedded SoC systems is the AXI protocol, as defined by Arm® in the AMBA® specification. Therefore, SCML2 also contains a protocol definition for this hardware protocol. The AXI protocol is a specific hardware protocol. Hence, the goal of the protocol definition is to provide with an accurate representation of the AXI features, both in functionality and timing. The AXI protocol extensions are defined to enable to create accurate AXI interfaces that support all AXI features. At the same time the AXI protocol also serves as the typical or generic interface to use when modeling communication where concurrent read and writes are used and where multiple transaction can be outstanding.

The FT AXI Modeling protocol definition is obviously closely related to the actual hardware protocol definition. As with the attribute definitions, the `protocol` state attribute is defined to match the AXI protocol timing points. The FT AXI protocol definition defines protocol states according to seven transfers, which represent the five hardware AXI protocol channels (two extra to indicate the last data transfer). They can be grouped into the following sets:

- Address transfers:

These represent the write address channel and read address channel of AXI, and correspond to the AWVALID, AWREADY and ARVALID, and ARREADY signals. For simplicity, the protocol states get the same name.

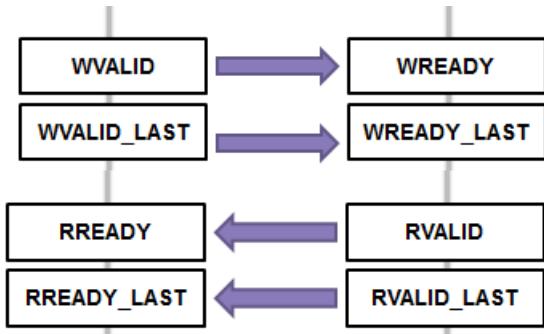
**Figure 3-21 Address Transfer**



- Data transfers:

These represent the write data and read data channels. Again, the protocol states have the same names as the AXI signals, except that the WLAST and RLAST signals are not represented by a protocol state, but the combination of an RVALID with RLAST and WVALID with WLAST is represented by the RVALID\_LAST and WVALID\_LAST protocol states, which are accompanied with RREADY\_LAST and WREADY\_LAST to define a separate transfer.

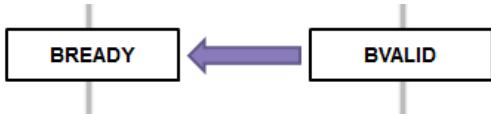
**Figure 3-22 Data Transfer**



- Write response transfers:

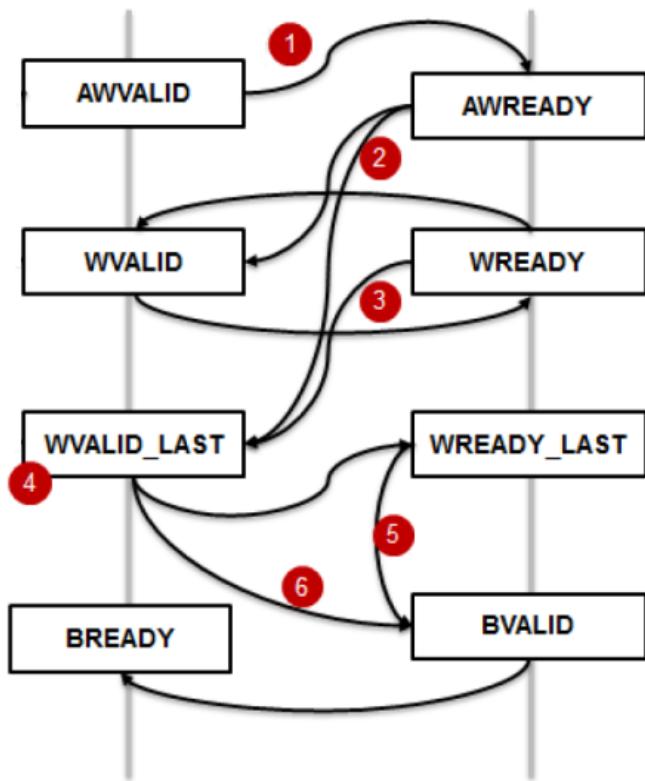
Represents the BVALID, BREADY signals used by the target to setup the response data.

**Figure 3-23 Write Response Transfer**



The AXI protocol state definition supports the hardware AXI protocol state machine, but since this is a modeling paradigm, there is additional flexibility in the protocol state changes to support accuracy for speed trade-offs or to allow to take short cuts in the protocol state machine for cases where the timing is very predictable and there is no need to go through every single state change as is required in hardware. The protocol state machine for a write access looks as shown below.

**Figure 3-24 Protocol State Machine for Write Access**



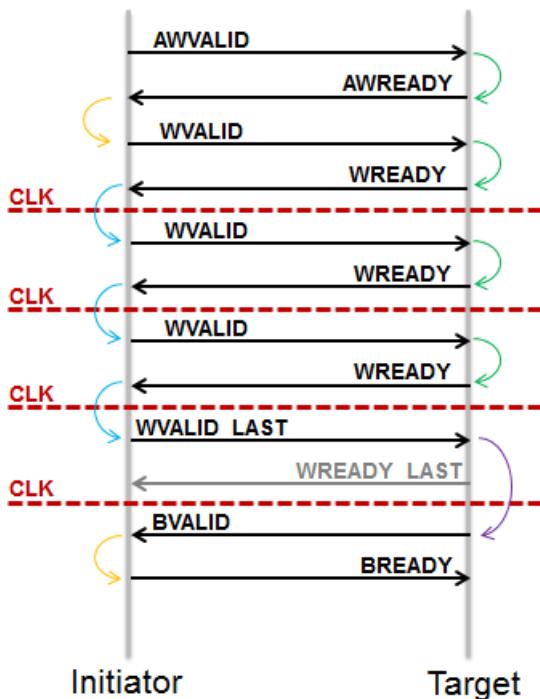
The above figure shows the different protocol states that are used during a write transfer with all protocol states that are defined by the initiator on the left and all protocol states defined by the target on the right. The arrows give all permitted state transitions between the different protocol states.

Notes on the protocol state machine:

1. An initiator should always start data transfer after the address phase is complete. Starting data transfer before starting address transfer is not allowed in FT semantics, as is theoretically possible in the hardware AXI protocol definition. However, the data transfer can start at the same simulation time as address transfer, provided the address transfer is already finished.
2. It is allowed to jump to WVALID\_LAST after completing the address transfer in case the length of the transaction is one word or in case there is no beat timing available or necessary (that is, the initiator sends the full data transaction at once and will account for the total delay of the write data setup).
3. The state transition from WREADY to WVALID\_LAST is necessary in case of the last data beat, but can also be used by the initiator to jump forward to the last beat. In this case, it should take into account the data setup time for the remaining data beats.
4. It is always required to use WVALID\_LAST.
5. When a target issues WREADY\_LAST on the last data beat, it will need to switch SystemC context in order to initiate the BVALID protocol state since it is not allowed in TLM2.0 to call a backward non-blocking API call in the context from the forward API calls and vice versa.
6. It is allowed to skip the WREADY\_LAST protocol state for a target, in case the WREADY\_LAST and BVALID protocol states happen at the same time point. This will avoid the context switch described above.

An example of an implementation of the AXI write data transaction is shown below through a message sequence chart.

**Figure 3-25 AXI Write Data Transaction**



It shows the initiator and target plus the different TLM2.0 interface method calls as they are done by these components. The clock boundaries show the required timing synchronization which can be implemented as explicit SystemC timing synchronization or simply through an increment of the timing annotation on the transport API calls.

The transaction starts with a call on the forward non-blocking interface by the initiator with the protocol state set to AWVALID, the target can complete the address transfer on the return of this call by updating the protocol state to AWREADY. The target only needs to add any additional timing it would need for address decoding, the immediate return results in a one cycle address transfer since the initiator can only start a new transaction on the next clock cycle. When the address transfer is completed the initiator should initiate another call on the forward non blocking interface to start the first write data transfer. It can choose to do this in the same cycle, this implements a case where address and data are made available in the same clock cycle by the initiator. Again, the target can respond with WREADY on the return of the transport API call. The next data beat can only be initiated in the next clock cycle, but it is possible to continue to use timing annotation on the transport APIs to model this. In this example, the target skips the WREADY\_LAST protocol state and returns with the protocol state set to BVALID indicating that WREADY\_LAST and BVALID happened at the same timing point. The initiator then completes the transaction by sending a BREADY, the target should not update the protocol state in this case.

An important aspect of the protocol definition is to apply the FT modeling timing annotation rules to the protocol, to ensure that it is clear which component should take care of the transfer timing. The following table gives the overview of the timing annotation requirements for all protocol state transitions.

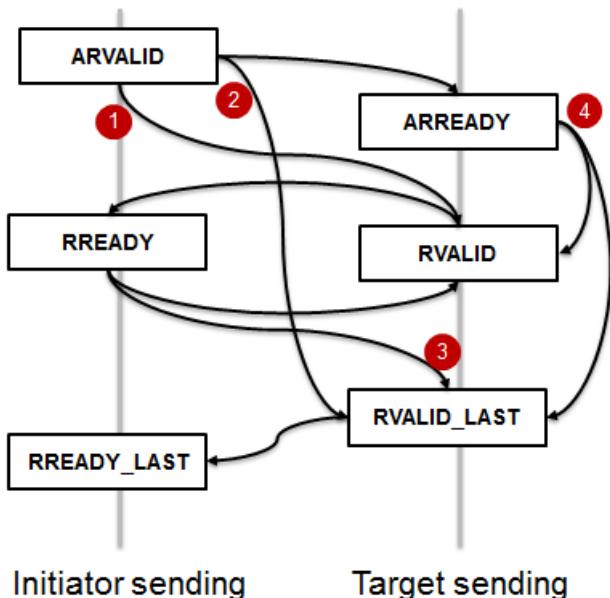
**Table 3-8 Timing Annotation Requirements**

Start	End	Who	Delay to be implemented	Minimum
AWVALID	AWREADY	target	Address response delay.	Can be 0 in case of pre-asserted address ready.
AWVALID	WVALID	initiator	First beat ready delay; delay to setup data independent of address return.	Is 0 if address and data are sent in the same cycle.
AWVALID	WVALID_LAST	initiator	data_ready delay: delay to setup all data independent of address return in case of first and only beat or when jumping forward to last beat.	One clock cycle, can be 0 in case burstlength equals 1.
AWREADY	WVALID	initiator	First beat ready delay; delay between address acceptance and first write data beat.	0
AWREADY	WVALID_LAST	initiator	data_ready delay: delay between address acceptance and all data available, in case of burstlength greater than 1, no assumptions on timing of other beats.	One clock cycle, can be 0 in case burstlength equals 1.
WVALID	WREADY	target	Write data beat accept delay, time to accept data coming from initiator.	0
WREADY	WVALID	initiator	Interbeat delay, time for initiator to setup next data beat.	One clock cycle
WREADY	WVALID_LAST	initiator	Final inter beat delay, time for initiator to setup remainder of the data.	One clock cycle
WVALID_LAST	WREADY_LAST	target	Write data accept delay: time to accept remaining data coming from initiator.	0

Start	End	Who	Delay to be implemented	Minimum
WVALID_LAST	BVALID	target	Write data accept delay plus response setup time, in case response setup time == 0.	0
WREADY_LAST	BVALID	target	Response setup time: delay for target to setup write response.	0
BVALID	BREADY	initiator	Response accept time: delay for initiator to accept write response.	0

In case of a read transaction, the protocol state machine looks as shown below.

**Figure 3-26 Protocol State Machine for Read Transaction**



The above figure shows the different protocol states that are used during a read transfer with all protocol states that are defined by the initiator on the left and all protocol states defined by the target on the right. The arrows give all permitted state transitions between the different protocol states.

Notes on the protocol state machine:

1. A target is allowed to advance the protocol state machine to RVALID and skip ARREADY. In this case, the end of the read address transfer is implied, the initiator should assume that the read command transfer is completed and is allowed to start another one in the same cycle as the RVALID. This avoids additional transport API calls and a SystemC context switch.
2. The same holds for RVALID\_LAST, in case the burst length of the transaction equals one word or when there are no intermediate timing points available from the target, or when the initiator is not requesting them (request\_beat\_timing\_extension).

3. The target is allowed to jump forward to RVALID\_LAST for the last data beat and it can skip data beats to get there. It is required to make sure that all read data is available in the data pointer.

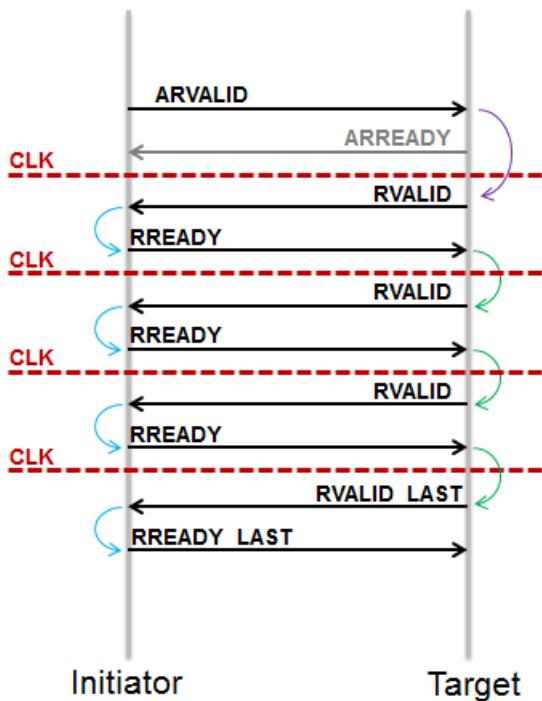


**Note** For read transactions, there is the common `beat_timing_request_extension` which is used by the initiator to indicate whether it is interested to receive beat timing. A target is not obliged to honor the request.

4. When a target issues a ARREADY to complete the read address transfer it is required to start the next transfer (RVALID or RVALID\_LAST) from a different SystemC context. This is a TLM2.0 standard requirement. The new transfer should be started on the backward path and it is not allowed to call the TLM interfaces from each others context.

An example of an implementation of the AXI read data transaction is shown below through a message sequence chart.

**Figure 3-27 AXI Read Data Transaction Implementation**



It shows the initiator and target plus the different TLM2.0 interface method calls as they are done by these components. The clock boundaries show the required timing synchronization which can be implemented as explicit SystemC timing synchronization or simply through an increment of the timing annotation on the transport API calls.

The initiator starts the transaction with the read address transfer start protocol state ARVALID. The target skips the ARREADY and returns with an RVALID on forward call that the initiator used to start the transaction. The target adds one cycle timing annotation to the call. This implies that the initiator can start the next transaction in the same cycle as the RVALID, but cannot assume anything about the actual arrival time of the ARREADY. If the timing annotation would be larger than one clock cycle, the initiator still cannot start a new transaction earlier than the RVALID. Obviously, the target should ensure that the read data is

available with the RVALID protocol state, the data for the later data beats can be added immediately or only with the individual data transfers.

The initiator completes the read data transfer by updating the protocol state to RREADY and sending it over a new forward transport call, it is allowed to do this from the same SystemC context and without timing annotation. The target again can use the return path of the call by the initiator to update the protocol state to start the next data transfer. This continues until all data beats are completed.

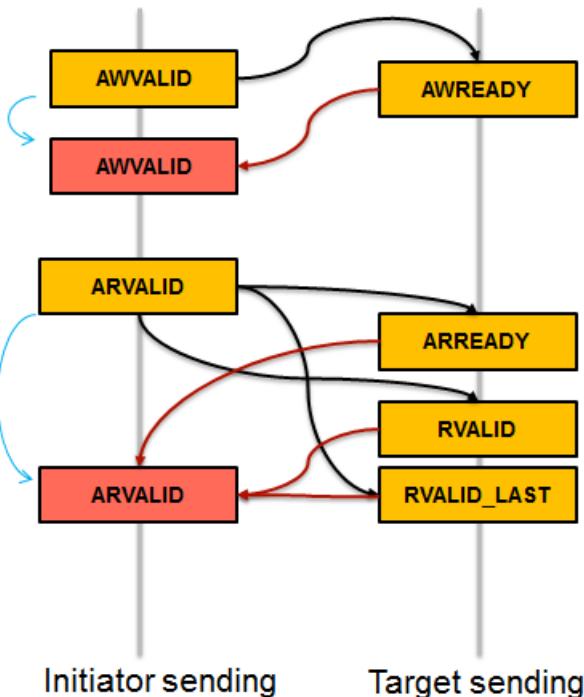
The timing modeling rules for all possible protocol state transitions of a read transaction are listed in the table below:

**Table 3-9 Timing Modeling Rules**

Start	End	Who	Delay to be Implemented	Minimum
ARVALID	ARREADY	Target	Address response delay.	Can be 0 in case of pre-asserted address ready.
ARVALID	RVALID	Target	First beat ready delay; delay to send RVALID in case address response delay is 0 (pre-asserted ARREADY).	One clock cycle.
ARVALID	RVALID_LAST	Target	data_ready delay: delay to send RVALID_LAST, total delay for transaction in case no beat timing is available no assumptions can be made for the other timing points (for example, ARREADY).	One clock cycle.
ARREADY	RVALID	Target	First beat ready delay: delay between accepting address and first data beat available.	One clock cycle.
ARREADY	RVALID_LAST	Target	First beat ready delay: in case of single beat transfer.	One clock cycle.
RVALID	RREADY	Initiator	Read data beat accept delay, time to accept data coming from target.	0
RREADY	RVALID	Target	Inter beat delay, time for the target to setup next data beat.	One clock cycle.
RREADY	RVALID_LAST	Target	Final inter beat delay, time for the target to setup remainder of the data.	One clock cycle.
RVALID_LAST	RREADY_LAST	Initiator	Final read data accept delay, time for the initiator to accept data coming from target.	0

The final set of rules of the AXI protocol definition relate to state transitions between different transactions. These rules define when an initiator is allowed to start a new transaction. The rules for the FT AXI protocol definition are the same as for the hardware AXI protocol definition.

**Figure 3-28 AXI Protocol Definition Rules**



An initiator is allowed to start read and write transactions concurrently via the address transfers. There is no dependency between the two.

- A new transaction can be started as soon as the corresponding address transfer is completed. This means as soon as AWREADY is received or as soon as ARREADY is received or implied by RVALID or RVALID\_LAST.
- It is possible to have multiple transactions in flight and a bus and target can control the number of read or write transactions that are outstanding using the address READY timing points effectively preventing an initiator from starting more transactions.
- There is a minimum of one clock cycle delay required for an initiator to start a new transaction of the same type.
- A new transaction should always be started on the forward transport API.



The target response does not necessarily come on the return path, but may be delayed and come via the backward path.

Since new transactions can be issued after AWREADY or ARREADY is received it is possible to have multiple read/write requests outstanding. The order in which these transactions are completed should follow AXI rules. In AXI ordering is governed by the transaction ID:

- Transactions with different ID have no ordering restrictions.
- Read or Write transactions with the same ID should complete in order.
- No ordering restrictions between reads and writes.

Another AXI feature is *Write Data interleaving*, which allows a target interface to accept partial data transfers for transactions coming with different IDs. This means that a target is not required to complete the full data

burst transfer for writes and that it can receive data transfers from other initiators intermixed with an already started transaction. This feature is fully under control of the target and does not imply any additional support from the initiator.

### 3.2.3 FT ACE Protocol Definition

Arm has defined ACE protocol to achieve the Cache Coherency in modern day's complex designs. The ACE protocol extends the AXI/AXI4 protocol and provides support for hardware coherent caches.

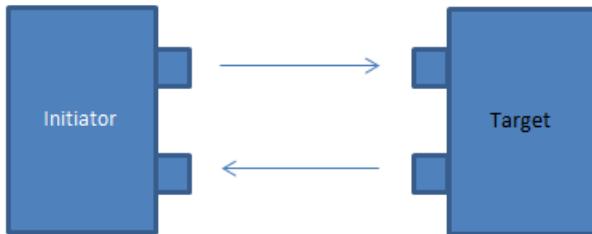
SCML2 also contains a protocol definition for this hardware protocol. The goal of the protocol definition is to provide with an accurate representation of the ACE features, both in functionality and timing. The ACE protocol extensions are defined to enable users to create accurate ACE interfaces that support all ACE protocol features. At the same time, the ACE protocol also serves as the typical or generic interface to use while modeling systems with Cache Coherency requirements.

The FT ACE protocol definition is closely related to the actual hardware protocol definition. ACE protocol has eight different channels out of which five channels are the same as in AXI protocol. There are three additional channels to allow an initiator in the system to snoop into the caches of the other initiators.

FT ACE protocol definition is also based on FT AXI protocol definition with additional semantics to model the ACE protocol.

FT ACE interface between initiator and target is a combination of two FT AXI sockets, as shown in the figure below. Outgoing socket from the initiator (or incoming socket on the target) is same as the FT AXI socket. In addition, there is an incoming socket on the initiator (or outgoing socket on the target) to model snoop channels/requests. The additional socket is also using FT AXI protocol.

**Figure 3-29 FT ACE Interface Between Initiator and Target**



Additional socket for the FT ACE protocol models Snoop Address (AC), Snoop Response (CR) and Snoop Data (CD) channels. As these channels are modeled using FT AXI, following is the mapping for the incoming interface on the initiator.

**Table 3-10 Mapping ACE Channels to FT AXI Channels**

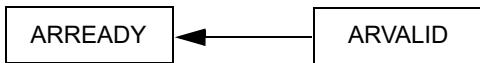
ACE Channel	Mapping to FT AXI Channel
AC	AR
CD	R
CR	R

This results in defining few new transfers in addition to the seven FT AXI transfers, as described in "[FT ACE API Definition](#)" on page 141.

- Snoop Address Transfers

These represent the Snoop Address channel of ACE, and correspond to the ACVALID and ACREADY signals. For simplicity, these are given the names as ARVALID and ARREADY protocol states.

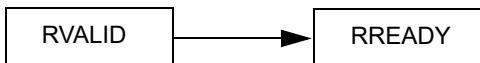
**Figure 3-30 Snoop Address Transfers**



- Snoop Data and Snoop Response Transfers

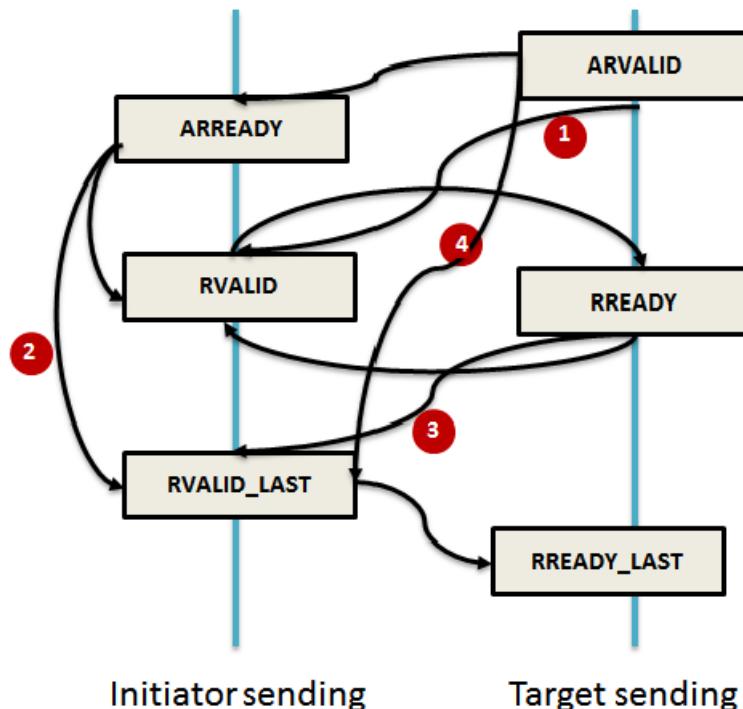
These represent the Snoop channel. Again, the protocol states have the names as RVALID and RREADY. Snoop Response channel is merged with the Snoop Data channel by passing CRRESP through RRESP.

**Figure 3-31 Snoop Data Transfers**



The protocol state machine for Snoop Access looks as shown below:

**Figure 3-32 Protocol State Machine for Snoop Access**



The above figure shows the different protocol states that are used during a snoop transfer with all protocol states that are defined by the initiator on the left and all protocol states defined by the target on the right.

The arrows give all permitted state transitions between the different protocol states.

Some important points to note about the protocol state machine are:

1. An initiator is allowed to advance the protocol state machine to RVALID and skip ARREADY. In this case, the end of the snoop address transfer is implied, the target should assume that the snoop command transfer is completed and is allowed to start another one in the same cycle as the RVALID. This avoids additional transport API calls and a SystemC context switch.

2. The same holds for RVALID\_LAST, in case the burst length of the transaction equals one word or when there are no intermediate timing points available from the target, or when the initiator is not requesting them.
3. The initiator is allowed to jump forward to RVALID\_LAST for the last data beat and it can skip data beats to get there. It is required to make sure that all read data is available in the data pointer.
4. When an initiator issues a ARREADY to complete the snoop address transfer, it is required to start the next transfer (RVALID or RVALID\_LAST) from a different SystemC context. This is a TLM2.0 standard requirement. The new transfer should be started on the backward path and it is not allowed to call the TLM interfaces from each other's context.
5. As all Snoop transactions do not require data to be transferred, initiator should jump to RVALID\_LAST with ace\_rsp\_pass\_data\_extension set to false.

ACE protocol has support for two additional acknowledge signals. These signals are used to indicate that an initiator has completed a `read` or `write` transaction. In case of FT AXI protocol, these signals are modeled implicitly. RACK signal is considered to be received when RREADY\_LAST is received by the target. Also, WACK signal is considered to be received when BREADY is received by the target.

### 3.2.4 FT AXI4 Stream Protocol Definition

The AXI4 Stream protocol is used as a standard interface to connect components that wish to exchange data. The interface can be used to connect a single initiator, that generates data to a single target that receives data. The protocol can also be used when connecting larger numbers of initiator and target components.

This hardware protocol uses the Fast Timed (FT) methodology. SCML2 also contains a protocol definition for this hardware protocol. The goal of the protocol definition is to provide with an accurate representation of the AXI4 Stream features, both in functionality and timing. The AXI4 Stream protocol extensions are defined to enable users to create accurate AXI4 Stream interfaces that support all AXI4 Stream protocol features.

- Initiator starts a new data transfer by setting protocol state as TVALID for a multi beat transfer and to TVALID\_LAST for a single beat transfer.
- Target on receiving the transfer, acknowledges with TREADY protocol state.
  - To mimic pre-asserted TREADY, target can send TREADY with zero cycle delay.
  - To mimic post-asserted TREADY, target can send TREADY with delay of one or more cycles.
- Last transfer of the packet is indicated by the TVALID\_LAST protocol state.
- For a single transfer packet/burst, TVALID\_LAST is set as protocol state and is acknowledged by TREADY\_LAST.
- Initiator can send next TVALID/TVALID\_LAST only one cycle after the TVALID/TREADY handshake is over.
  - TREADY with  $t=0$  means handshake over in the same cycle.
  - TREADY with  $t>0$  means handshake over at simulation time `sc_time_stamp(t=0)`.
- The initiator should set appropriate extensions indicating values for TID, TDEST and TUSER.
- The byte enable length and byte enable pointer follows the same rules, as specified in the [TLM2 Reference manual](#).

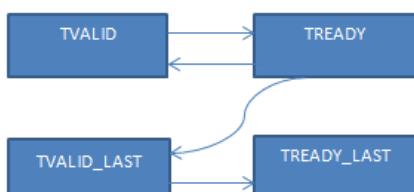
- The array extension `axi4_stream_tkeep_extension` should be set if any of the bytes in the data stream are NULL bytes. It is not required to be set, if none of the NULL bytes exists in the payload. In this case, the value of the array is assumed to be 0xFF for all bytes. This extension should be set and get using the above defined macros and their memory management is done by the extension infrastructure.
- State Skipping* is allowed, that is, initiator can send `TVALID_LAST` directly as first protocol state, after setting the complete data in the data pointer. In doing so, it should maintain an appropriate inter beat timing delay.
- When AXI4 Stream initiator is mapped to GP target, mapper acknowledges all `TVALID`s with `TREADY`'s. `TVALID_LAST` is mapped to `BEGIN_REQ` phase on `TLM2_GP`. `BEGIN_RESP` from GP is mapped to `TREADY_LAST` and send back to initiator.
- An AXI4 Stream initiator is not expected to know the number of beats in the complete burst in advance. Therefore, an AXI4 Stream initiator should always set the data length of the payload as the `BUS_WIDTH` which is also the size of each beat of AXI4 Stream packet.
- An AXI4 Stream initiator should also allocate number of bytes equal to `BUS_WIDTH` for its data pointer, byte enable pointer and `TKEEP` extension pointer. It is expected that an AXI4 Stream initiator will use the same data buffer for each subsequent beat.
- For sending narrow burst and AXI4 Stream, initiator would set the corresponding bytes in `TKEEP` extension array to be NULL.



Mapping from AXI to AXI4 Stream is not supported.

- When GP initiator is connected to AXI4 Stream target - `BEGIN_REQ` for WRITE request is mapped to `TVALID_LAST` on AXI4 Stream target. And `TREADY_LAST` is mapped to `BEGIN_RESP` on GP side.
- Appropriate channel timings would be maintained by the mapper.
- When GP initiator is connected to AXI4 Stream target- `BEGIN_REQ` for READ is responded by `BEGIN_RESP` from the mapper, with `tlm_response` set to `TLM_COMMAND_ERROR_RESPONSE`.
- When AXI4 Stream initiator is connected to GP or AXI target, each beat from initiator is mapped to single transaction on target. The mapper will convert the `BUS_WIDTH` on initiator side to data length and burst size attribute of the transaction on target side.

**Figure 3-33 Protocol State Machine for AXI4 Stream Protocol**



The following table lists the timing requirement for the protocol.

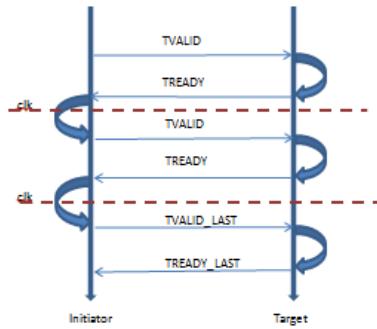
**Table 3-11 Timing Rules**

Start State	End State	Who	Delay to be Implemented	Minimum Delay
TVALID	TREADY	Target	Data Accept Delay	0 for pre-asserted Ready
TREADY	TVALID	Initiator	Interbeat Delay	1 cycle

Start State	End State	Who	Delay to be Implemented	Minimum Delay
TREADY	TVALID_LAST	Initiator	Interbeat Delay	1 cycle
TVALID_LAST	TREADY_LAST	Target	Data Accept Delay	0 cycle (pre-asserted)
TREADY_LAST	TVALID	Initiator	Inter Packet Delay	1 cycle

The following *Message Sequence* chart shows the flow for AXI4 Stream Initiator and Target when Target is giving pre-asserted TREADY.

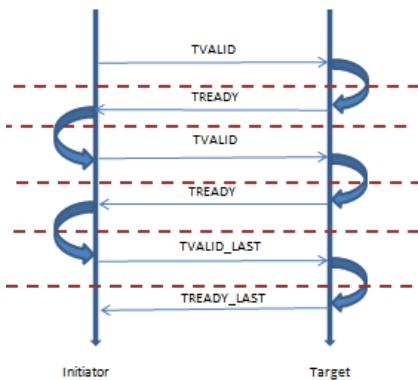
**Figure 3-34 Pre Asserted TREADY**



The red dotted line in the image indicates CLK cycle delay.

The following *Message Sequence* chart shows the flow for AXI4 Stream Initiator and Target when Target is giving post-asserted TREADY.

**Figure 3-35 Post Asserted TREADY**



- The red dotted line in the image indicates CLK cycle delay.
- The `clk` is only time and can be annotation, not necessarily an increase in simulation time.
- The number of clock cycles can be one or more.

### 3.2.5 FT CHI Protocol Definition

Arm has defined the CHI protocol to achieve Cache Coherency in modern day's complex designs. This is done keeping in mind, the ever increasing number of coherent initiators. CHI stands for *Coherent Hub Interface*; and targets the interface to the coherent hub, that is found in many SoCs.

SCML2 contains a protocol definition for this protocol. The goal of the protocol definition is to provide an accurate representation of the CHI features, both in functionality and timing.

The FT CHI protocol definition is closely related to the actual hardware protocol definition. FT CHI defines six different channels.

**Table 3-12 FT CHI Channel Mapping**

RN-F	RN-D	RN-I	SN-F / SN-I	FT CHI	Description
TXREQ	TXREQ	TXREQ		ChiTxReq	Outbound Request
RXSNP	RXSNP (DVM)		RXREQ	ChiRxReq	Inbound Request
TXDAT	TXDAT	TXDAT	TXDAT	ChiTxData	Outbound Data
RXDAT	RXDAT	RXDAT	RXDAT	ChiRxData	Inbound Data
TXRSP	TXRSP	TXRSP	TXRSP	ChiTxResp	Outbound Response
RXRSP	RXRSP	RXRSP		ChiRxResp	Inbound Response

Unlike FT ACE, FT CHI interface between two nodes is represented using a single socket. This single socket, is used for both inbound and outbound requests. As such, LT mode using TLM blocking transport interface is not supported.

FT CHI protocol definition defines the protocol states for six different transfers, which represent the six hardware CHI protocol channels. They can be grouped into the following sets:

- Address transfers

These represent read and write address transfers on ChiTxReq and ChiRxReq channels.

- Data transfers

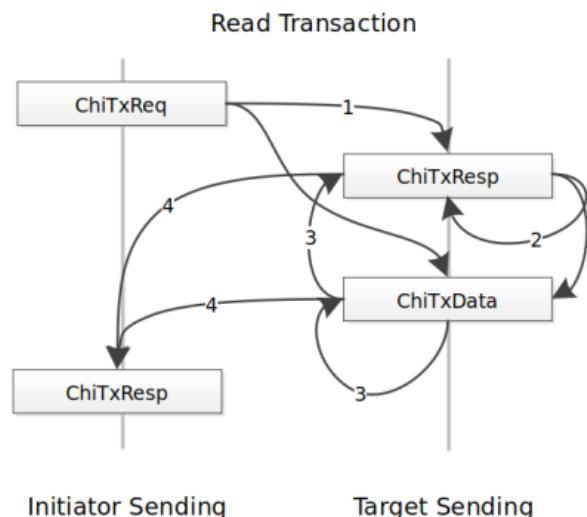
These represent read, write and snoop data transfer on ChiTxData and ChiRxData channels.

- Response transfers

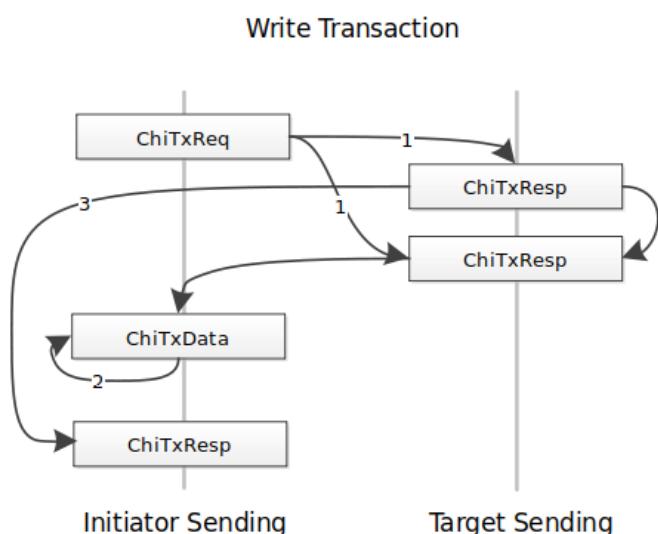
These represent read, write and snoop response transfers on ChiTxResp and ChiRxResp channels.

Protocol state machine for some basic transactions is illustrated below. These figures show different protocol states that are used during a transaction between two nodes. The arrows give all permitted state transitions between the different protocol states.

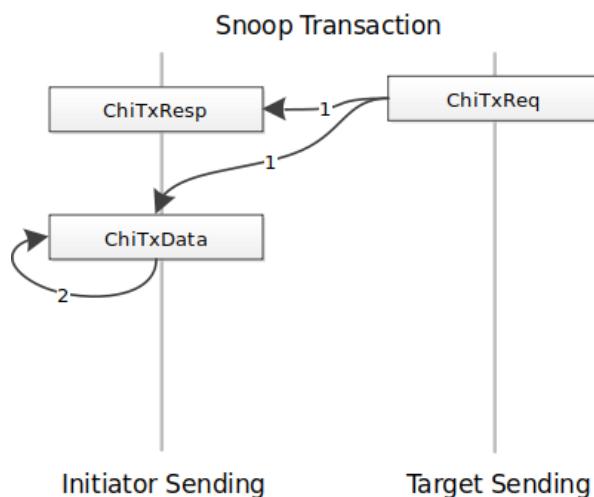
**Figure 3-36 Protocol State Machine for Read Transaction**



**Figure 3-37 Protocol State Machine for Write Transaction**



**Figure 3-38 Protocol State Machine for Snoop Transaction**



### 3.2.6 FT PCIe Protocol Definition

PCI Express (Peripheral Component Interconnect Express) is a high-speed serial computer expansion bus standard. It is the common motherboard interface for personal computers' graphics cards, hard disk drive host adapters, SSDs, Wi-Fi and Ethernet hardware connections.

PCIe has numerous improvements over the older standards, including higher maximum system bus throughput, lower Input/Output pin count and smaller physical footprint, better performance scaling for bus devices, and so on.

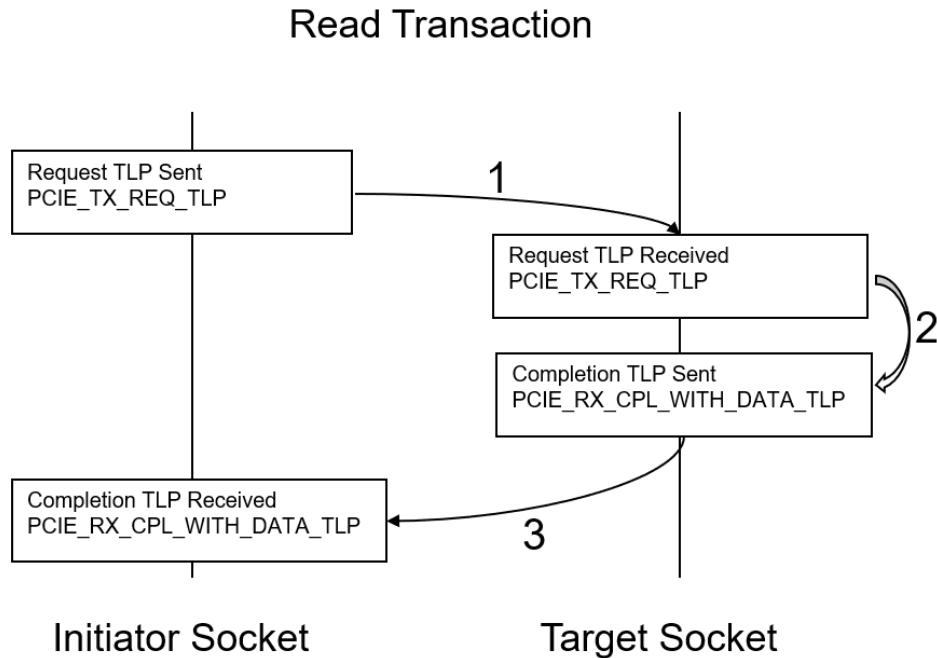
SCML2 contains a protocol definition for this protocol. The goal of the protocol definition is to provide an accurate representation of the PCIe features that can be used for performance studies and architectural exploration. All the three PCIe protocol stack layers (Transaction Layer, Data Link Layer and Physical Layer) are implemented in Port Adaptors.

Both the PCIe Tx and Rx interfaces are supported using a single socket pair that is, both the PCIe sockets as well as port adaptors support bidirectional operation

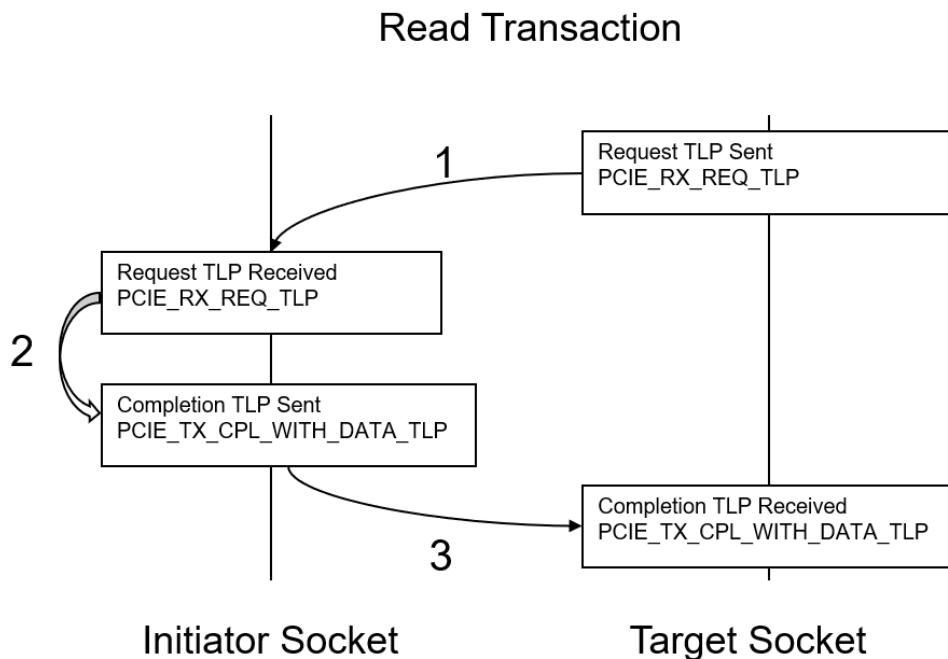


Tx and Rx are always from Initiator socket perspective. For example, a Tx packet always means an outgoing packet from Initiator socket and an Rx packet always means an incoming packet at Initiator socket. In other words, a Tx packet always means an incoming packet at Target socket and an Rx packet always means an outgoing packet from Target socket.

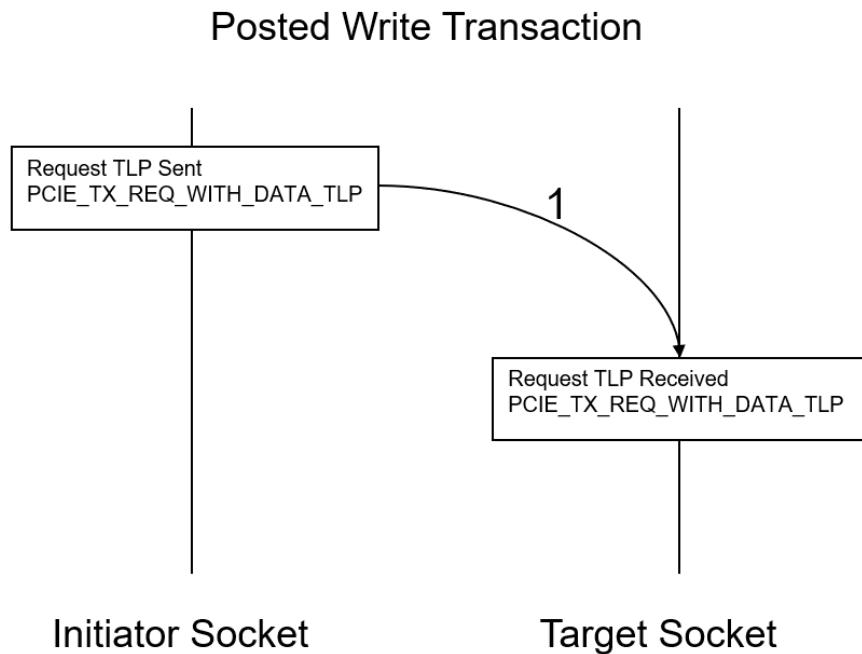
**Figure 3-39 Protocol State Machine for Memory Read Transaction from Initiator Socket (Root Complex)**



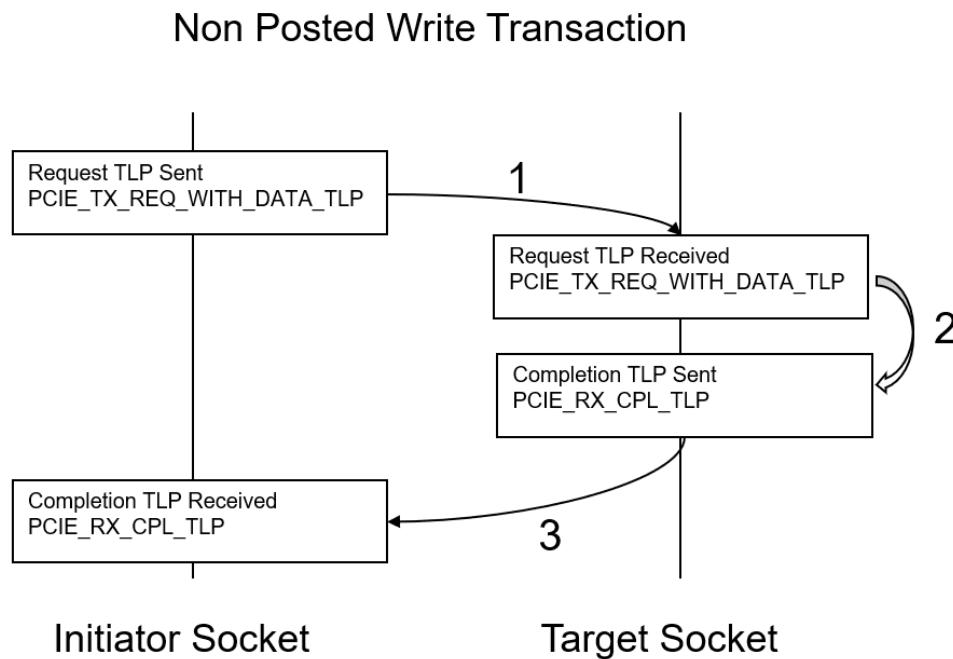
**Figure 3-40 Protocol State Machine for Memory Read Transaction from Target Socket (End Point)**



**Figure 3-41 Protocol State Machine for Memory Write Transaction from Initiator Socket (Root Complex)**



**Figure 3-42 Protocol State Machine for IO Write Transaction (Non Posted) from Initiator Socket (Root Complex)**



### 3.2.7 FT CXL Protocol Definition

Compute Express Link (CXL) is an open standard for high-speed CPU-to-device and CPU-to-memory connections, designed for high performance data centre computers.

CXL is built on the PCIe physical and electrical interface and includes PCIe-based block input/output protocol (CXL.io) and new cache-coherent protocols for accessing system memory (CXL.cache) and device memory (CXL.mem).

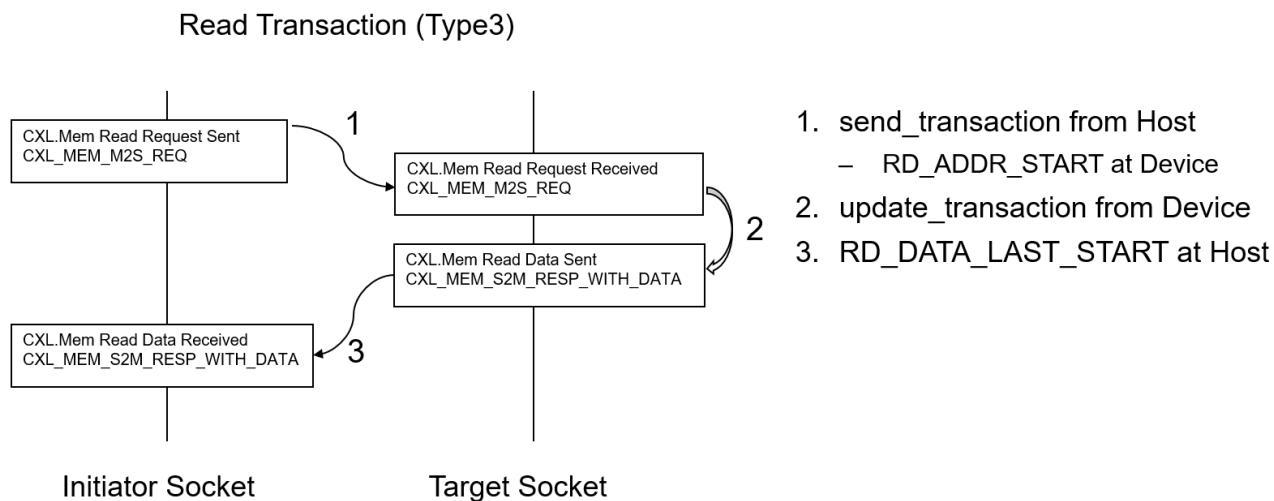
SCML2 contains a protocol definition for this protocol. The goal of the protocol definition is to provide an accurate representation of the CXL features that can be used for performance studies and architectural exploration. All the three CXL protocol stack layers (Transaction Layer, Data Link Layer and Physical Layer) are implemented in Port Adaptors.

Both the CXL Tx and CXL Rx interfaces are supported using a single socket pair, that is, both the CXL sockets as well as port adaptors support bidirectional operation.

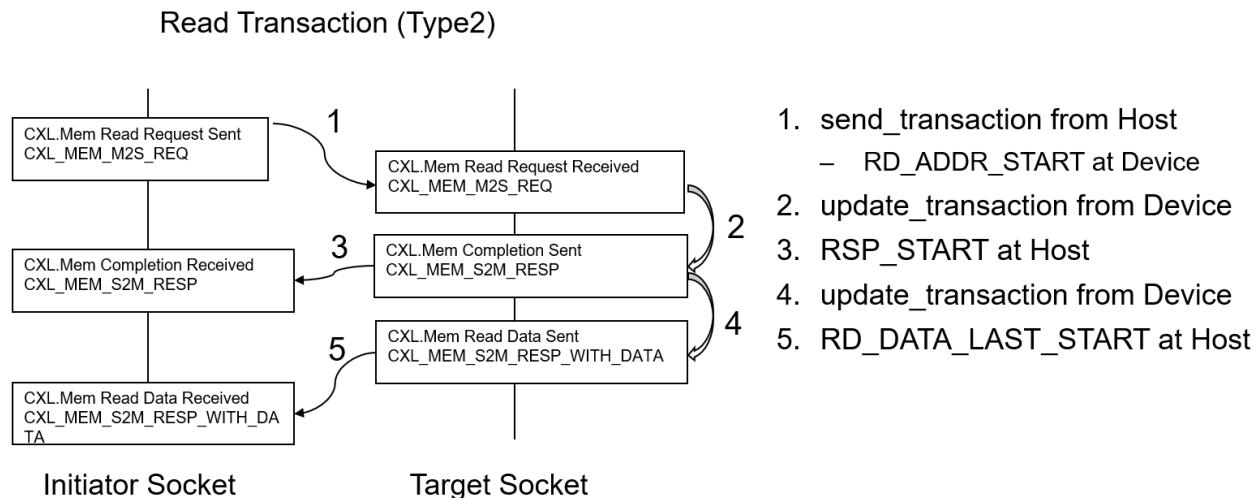


**Note** Tx and Rx are always from Initiator socket perspective. For example, a Tx packet refers to an outgoing packet from the Initiator socket, and an Rx packet refers to an incoming packet at the Initiator socket. In other words, a Tx packet refers to an incoming packet at the Target socket and an Rx packet refers to an outgoing packet from the Target socket.

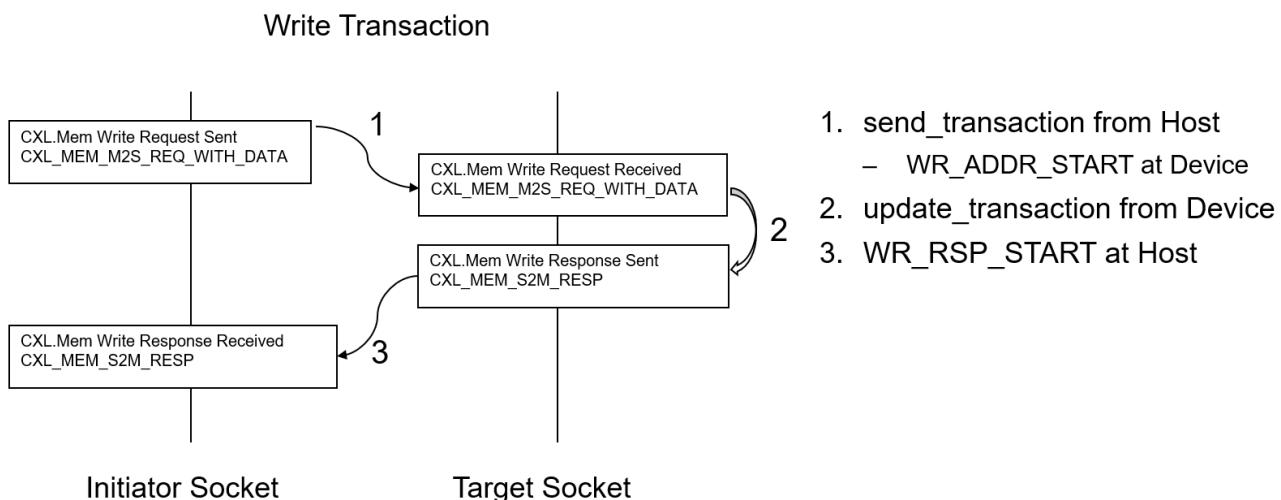
**Figure 3-43 Protocol State Machine for Memory Read Transaction (Type3) from Initiator Socket (Host)**



**Figure 3-44 Protocol State Machine for Memory Read Transaction (Type2) from Initiator Socket (Host)**

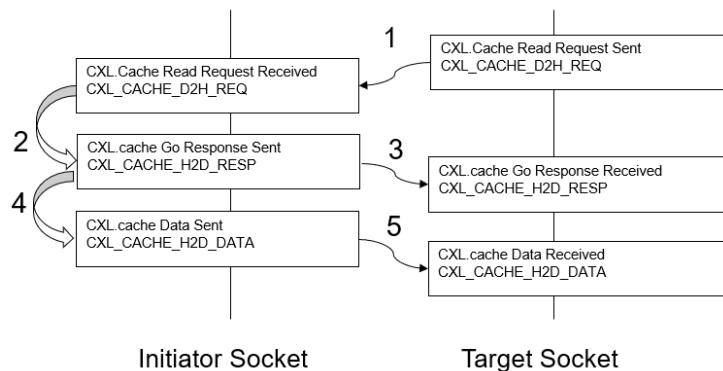


**Figure 3-45 Protocol State Machine for Memory Write Transaction from Initiator Socket (Host)**



**Figure 3-46 Protocol State Machine for CXL.cache Read Transaction to Host Memory (Type1/Type2) from Target Socket (Device)**

Read Transaction To Host Memory (Type1 / Type2)



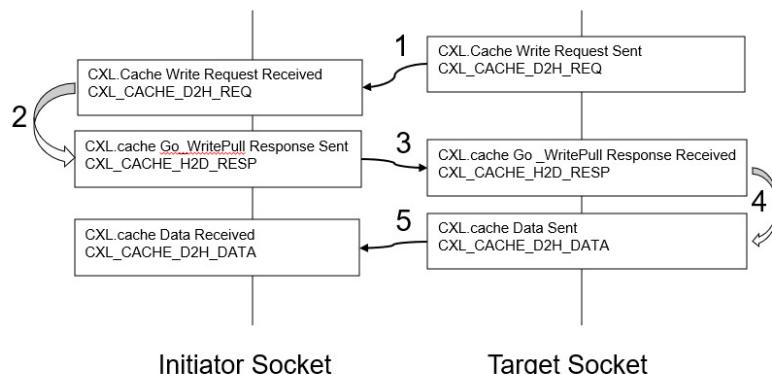
1. send\_transaction from Device
  - RD\_ADDR\_START at Host
2. update\_transaction from Host
3. RSP\_START at Device
4. update\_transaction from Host
5. RD\_DATA\_LAST\_START at Device

Note:

- RdCurr has no GO response
- RdOwnNoData has no Data response
- GO and Data can be received in any order

**Figure 3-47 Protocol State Machine for CXL.cache Write Transaction to Host Memory (Type1/Type2) from Target Socket (Device)**

Write Transaction To Host Memory (Type1 / Type2)



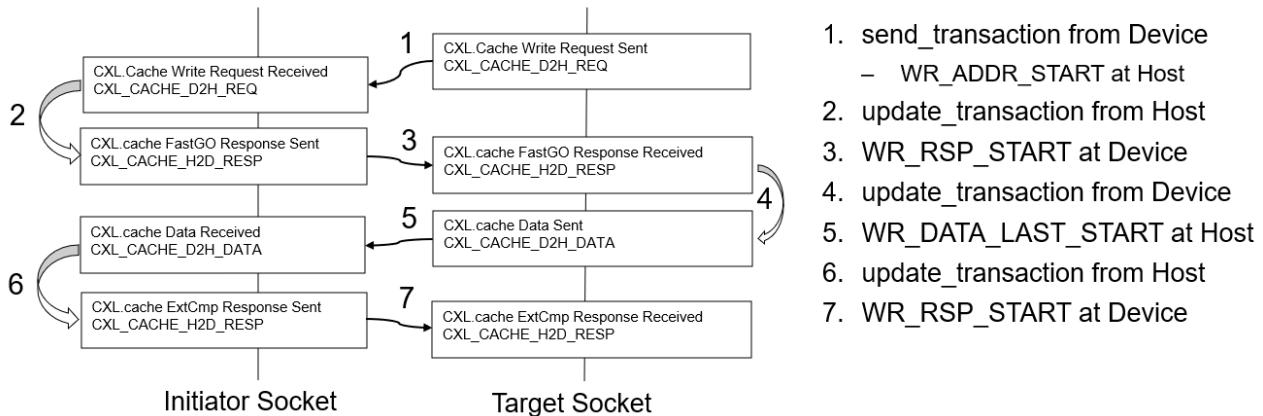
1. send\_transaction from Device
  - WR\_ADDR\_START at Host
2. update\_transaction from Host
3. WR\_RSP\_START at Device
4. update\_transaction from Device
5. WR\_DATA\_LAST\_START at Host

Note:

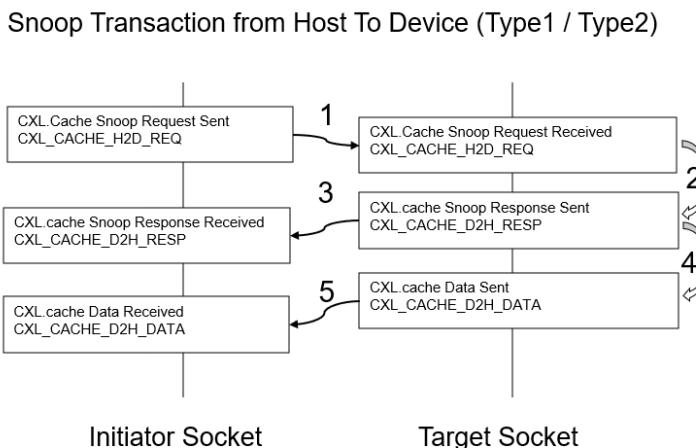
- CleanEvictNoData has only GO response and no Data
- CleanEvict can have GO\_WritePullDrop response

**Figure 3-48 Protocol State Machine for CXL.cache Write Transaction with ExtCmp to Host Memory (Type1/Type2) from Target Socket (Device)**

Write Transaction To Host Memory With ExtCmp (Type1 / Type2)



**Figure 3-49 Protocol State Machine for CXL.cache Snoop Transaction (Type1/Type2) from Initiator Socket (Host)**

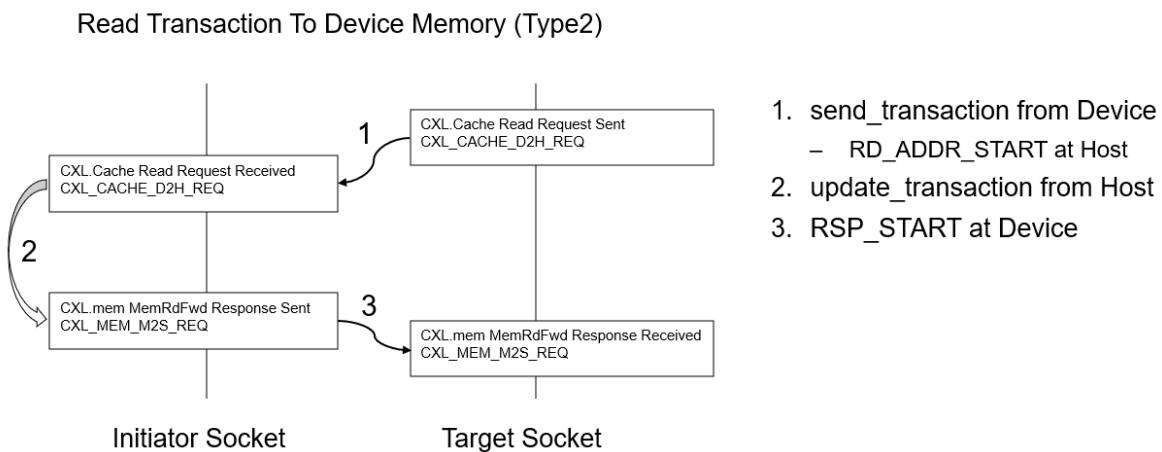


1. send\_transaction from Host
  - RD\_ADDR\_START at Device
2. update\_transaction from Device
3. RD\_RSP\_START at Host
4. update\_transaction from Device
5. RD\_DATA\_LAST\_START at Host

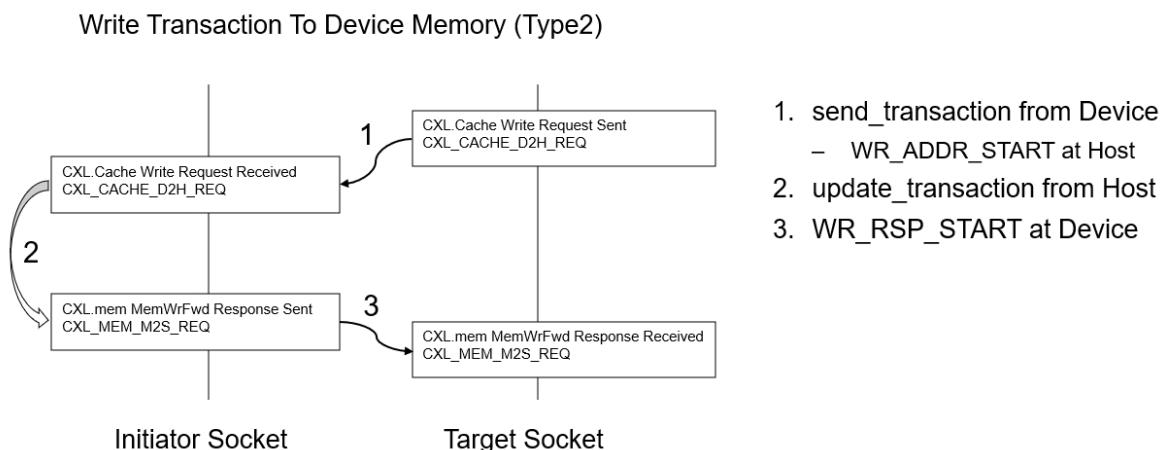
Note:

- Response and Data can be received in any order
- Data is only expected when response is Rsp\*Fwd\*

**Figure 3-50 Protocol State Machine for CXL.cache Read Transaction to Device Memory in Host Bias (Type2) from Target Socket (Device)**



**Figure 3-51 Protocol State Machine for CXL.cache Write Transaction to Device Memory in Host Bias (Type2) from Target Socket (Device)**



### 3.3 API Definitions

- [FT GFT API Definition](#)
- [FT AXI API Definition](#)
- [FT ACE API Definition](#)
- [FT AXI4 Stream API Definition](#)
- [FT CHI API Definition](#)
- [FT PCIe API Definition](#)
- [FT CXL API Definition](#)

#### 3.3.1 FT GFT API Definition

The basics of the FT GFT API are described in [SCML2 FT Modeling Interfaces](#). The attributes that do not fit with the TLM2 base protocol need to be added as ignorable extensions as well as the protocol timing points. GFT extends the common FT interfaces with the following extensions:

- `scml2::gft_protocol_state_extension`:

This is the payload extension that holds the protocol timing points for the GFT protocol it is used as follows:

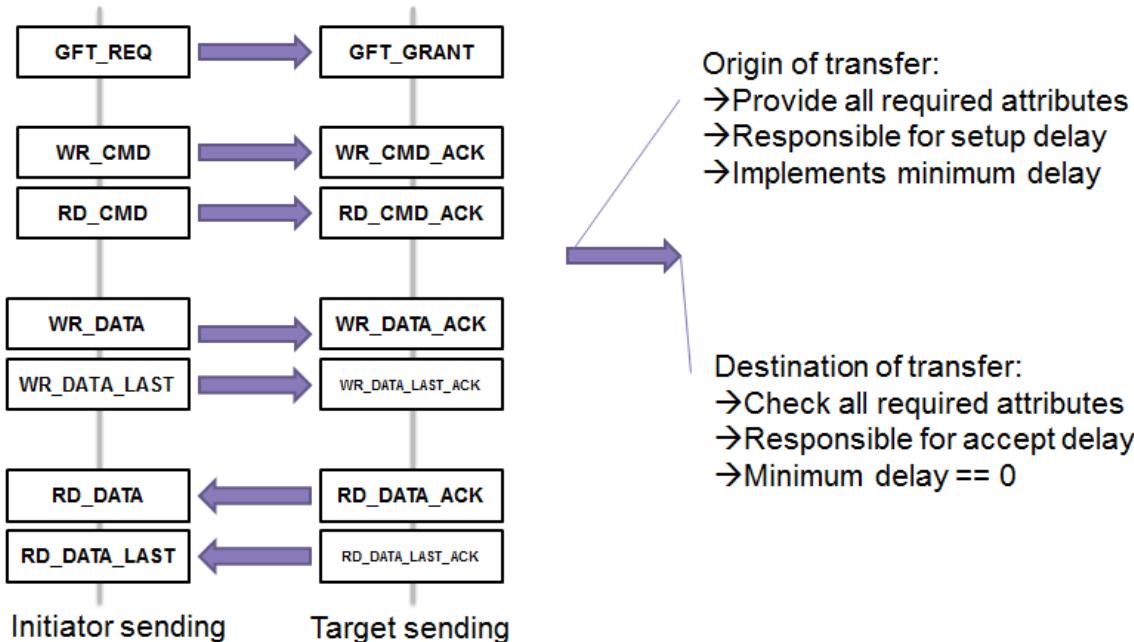
```
scml2::gft_protocol_state_enum protocol_state;
scml2::gft_protocol_state_extension *gft_state;
transaction.get_extension_attr(gft_state, protocol_state);
```

As socket that intends to communicate using the GFT protocol should indicate this via the `set_protocol` API:

```
P_socket.set_protocol<scml2::gft_protocol_state_extension>("GFT");
```

The GFT protocol defines the following protocol states, each referring to a protocol timing point. They are represented in pairs to indicate the transfers that they represent.

**Figure 3-52 GFT FT API**



- `scml2::gft_response_extension`:

The GFT protocol definition has an extended set of transaction responses, for example, to fit better with the features of protocols like AXI. Therefore, an extended response is provided. This implies that when using the GFT protocol definition, it is required to ignore the TLM2 base protocol response definition.

```
typedef enum { gftOK, gftEX_OK, gftSLV_ERR, gftDEC_ERR, gftSplit, gftRetry} gft_rsp_enum;

scml2::gft_rsp_enum response;
scml2::gft_response_extension*rsp_ext;
transaction.get_extension_attr(rsp_ext, response);
```

- `gftOK`: Indicates normal access has been successful, can indicate failure of an exclusive access. This is the default value for the extension.

- `gftEX_OK`: Indicates an exclusive access has been successful.
- `gftSLV_ERR`: Is used to indicate a target error, which means the transaction reached the target, but the target wishes to return an error condition to the initiator
- `gftDEC_ERR`: Indicates a decode error by an interconnect component, when there is no target at the transaction address.
- `gftSplit`: Indicates a split response by the target for the given transaction, that is, that it will take a while for this transaction to complete and that other initiators should be given priority. This implies that other transactions may come in on the same target while the current transaction is being processed.
- `gftRetry`: Indicates a retry response by the target, which is similar to a split but here the preference is that only initiators with higher priority can come in on the same target while the current transaction is being processed.
- `scml2::gft_burst_type_extension`:

The GFT protocol definition supports an additional burst type on top of what is available in TLM2 base protocol, this is a wrapping burst. For which the common wrap address and wrap data extensions can be used.



The TLM2.0 base protocol supports burst accesses, but does not really have a burst type definition.

The GFT protocol does not violate the TLM2.0 base protocol data organization rules. This means that functionally there is no difference between GFT and TLM2.0 base protocol concerning burst transactions. The `gft_burst_type_extension` is important to correctly model burst beat timing. The difference relates to the first data element that is returned for a wrapping burst versus an incremental burst.

```
typedef enum { gftFIXED, gftINCR, gftWRAP} gft_burst_type_enum;

scml2::gft_burst_type_enum burst_type;
scml2::gft_burst_type_extension*burst_ext;
transaction.get_extension_attr(burst_ext, burst_type);
```

- `gftFIXED`: Indicates burst with fixed address.
- `gftINCR`: Indicates incremental burst. This is the default value for the extension.
- `gftWRAP`: Is to be used for wrapping bursts.
- `scml2::gft_lock_type_extension`:

This GFT attribute indicates whether the transaction is a locking or exclusive access.

```
typedef enum { gftNORMAL, gftEXCL, gftLOCK} gft_lock_type_enum;

scml2::gft_lock_type_enum lock_type;
scml2::gft_lock_type_extension *lock_type_ext;
transaction.get_extension_attr(lock_type_ext, lock_type);
```

- `gftNORMAL`: indicates a normal transaction as defined in the TLM2.0 base protocol. This is the default value for this extension.
- `gftEXCL`: indicates an exclusive read access, which means that the target cannot accept any transactions by another initiator until the initiator has done a write to this address.

- `gftLOCK`: indicates a locked access is initiated by the initiator which implies the interconnect bus ensure that only that initiator has access to the target until the initiator unlocks the access through a second access to the same location.

- `scml2::gft_access_mode_extension`:

Is the GFT extension that indicates what the security attributes of the transaction are.

```
typedef enum { gftNORM, gftPRIV, gftNORM_NON_SECURE, gftPRIV_NON_SECURE}
gft_access_mode_enum;

scml2::gft_access_mode_enum access_mode;
scml2::gft_access_mode_extension *access_mode_ext;
transaction.get_extension_attr(access_mode_ext, access_mode);
```

- `gftNORM`: Indicates a normal TLM2.0 base protocol transaction. This is the default value of the extension.
- `gftPRIV`: Indicates a privileged access, which can be used to indicate the processing mode that issued the transaction, can be used to restrict access to certain parts of the system.
- `gftNORM_NON_SECURE`: Indicates normal non-secure access. The effect of this on the handling of the transaction is system specific.
- `gftPRIV_NON_SECURE`: Indicates a privileged non-secure access. The effect of this on the handling of the transaction is system specific.
- `scml2::gft_access_type_extension`:

Is the GFT extension that indicates whether the access is a data or instruction access. The default value is `gftDATA_ACCESS`.

```
typedef enum { gftDATA_ACCESS, gftINST_ACCESS} gft_access_type_enum;

scml2::gft_access_type_enum access_type;
scml2::gft_access_type_extension *access_type_ext;
transaction.get_extension_attr(access_type_ext, access_type);
```

The GFT protocol definition is built according to the FT Modeling approach, so this means that the extensions defined for the GFT protocol are not ignorable within the component. The GFT protocol definition is built on top of the TLM2.0 base protocol so the TLM2.0 base protocol attributes are always available. However, the GFT protocol defines certain extensions that overrule the TLM2.0 base protocol. Therefore, the following rules apply with regards to the TLM2.0 base protocol attributes for a GFT transaction:

- `Burst_type`: burst type should be taken from the `burst_type` enum and not according to the TLM2 base protocol rules. The burst type attribute should be valid with the `WR_CMD` and `RD_CMD` protocol states, the same applies for lock, `access_mode` and `access_type`
- `tlm_response_status`: this attribute should not be used, it is required to use the `gft_rsp_enum`. The response should be valid with the `WR_DATA_LAST_ACK` protocol state for writes, and with any of the read data transfers for read transactions. There is only one error response allowed for the whole transaction in case of writes, for reads multiple responses are allowed.
- `Byte_enables` are not supported in a GFT transaction, so the byte enable attributes should be ignored.
- It is not allowed to use `IGNORE_COMMAND`.
- `Streaming_width` shall be ignored.

The conversion between the GFT protocol definition and the TLM2.0 base protocol needs to take care of these attribute semantics.

### 3.3.2 FT AXI API Definition

The FT AXI API definition is built on top of the TLM2.0 base protocol just like any other protocol in the FT modeling approach. The FT AXI protocol definition adds the following extensions to the TLM2.0 base protocol:

- scml2::axi\_protocol\_state\_extension:

This is the payload extension that holds the protocol timing points for the AXI protocol it is used as follows:

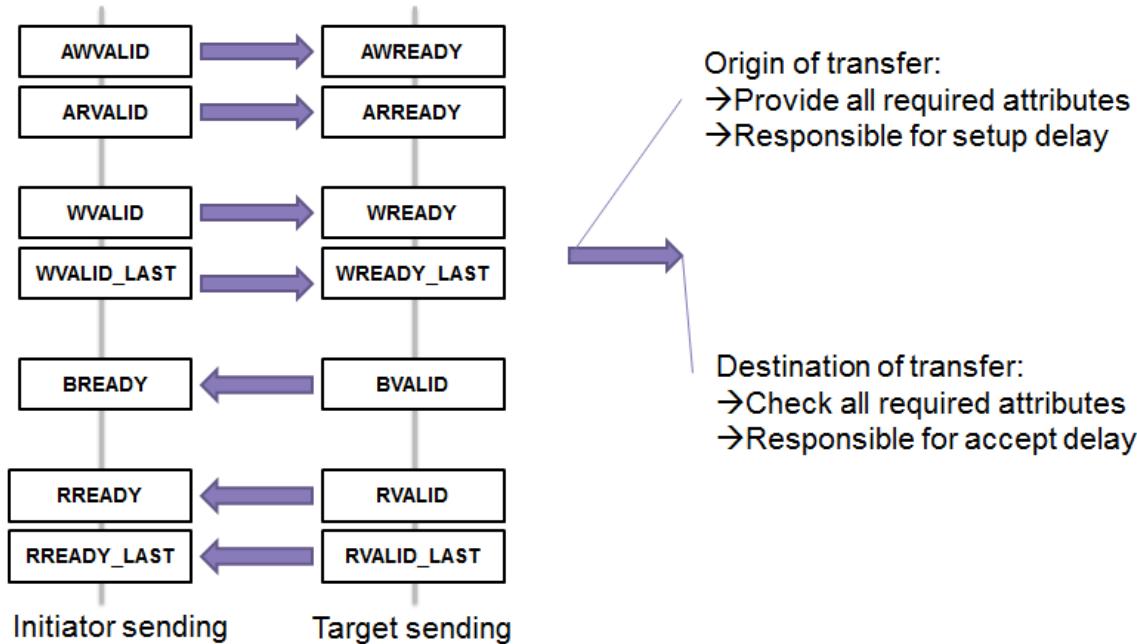
```
scml2::axi_protocol_state_enum protocol_state;
scml2::axi_protocol_state_extension *axi_state;
transaction.get_extension_attr(axi_state, protocol_state);
```

As socket that intends to communicate using the AXI protocol should indicate this via the set\_protocol API:

```
P_socket.set_protocol<scml2::axi_protocol_state_extension>("AXI");
```

The AXI protocol defines the protocol states, each referring to a protocol timing point, as shown below. They are represented in pairs to indicate the transfers that they represent.

**Figure 3-53 AXI Protocol Definition**



- scml2::axi\_response\_extension:

This attribute models the RRESP[1:0] and BRESP[1:0] signals of the AXI protocol. Since these have additional semantics compared to the TLM2.0 base protocol response attribute, there is a specific AXI

attribute for responses. The values correspond to the encodings of the AXI response signals and follow the AXI rules:

- One response for the entire burst in case of writes.
- In case of a read transaction, the target can give different responses for the transfers in the burst.
- All data beats need to be executed independent of the response status.
- The default value is `axiOK`.

The definition of the attribute values in relation to the AXI definition is given in the following table.

**Table 3-13 AXI Definition Attribute Values**

<b>axi_rsp_enum</b>	<b>RRESP[1:0] BRESP[1:0]</b>	<b>Response</b>	<b>Meaning</b>
axiOK	b00	OKAY	Indicates that a normal access has been successful, can indicate failure of an exclusive access. This is the default value for the extension.
axiEX_OK	b01	EXOKAY	Indicates an exclusive access has been successful.
axiSLV_ERR	b10	SLVERR	Is used to indicate a target error, which means the transaction reached the target, but the target wishes to return an error condition to the initiator.
axiDEC_ERR	b11	DECERR	Indicates a decode error by an interconnect component, when there is no target at the transaction address.

- `scm12::axi_burst_type_extension`:

The burst type extension represents the `ARBURST[1:0]` and `AWBURST[1:0]` signals of the AXI protocol. Since AXI defines an additional burst type than is supported via the TLM2.0 base protocol, the specific extension is required. The default value is `axiFIXED`.

```
typedef enum { axiFIXED, axiINCR, axiWRAP, axiBURST_ERR} axi_burst_type_enum;

scm12::axi_burst_type_enum burst_type;
scm12::axi_burst_type_extension*burst_ext;
transaction.get_extension_attr(burst_ext, burst_type);
```

The definition of the attribute values in relation to the AXI definition is given below:

**Table 3-14 Attribute Values**

<b>axi_burst_type_enum</b>	<b>ARBURST[1:0] AWBURST[1:0]</b>	<b>Burst Type</b>	<b>Description</b>	<b>Access</b>
axiFIXED	b00	FIXED	Fixed Address burst	FIFO-type
axiINCR	b01	INCR	Incremental address burst	Normal sequential memory

<b>axi_burst_type_enum</b>	<b>ARBURST[1:0] AWBURST[1:0]</b>	<b>Burst Type</b>	<b>Description</b>	<b>Access</b>
axiWRAP	b10	WRAP	Incrementing address bursts wrapping to a lower address at the wrap boundary	Cache line
axiBURST_ERR	b11	Reserved	-	-

- scml2::axi\_lock\_type\_extension:

The lock type extension is added to represent the atomic access support in AXI and represents the values for the signals ARLOCK [1:0] and AWLOCK [1:0].

```
typedef enum { axiNORMAL, axiEXCL, axiLOCK, axiLOCK_ERR} axi_lock_type_enum;

scml2::axi_lock_type_enum lock_type;
scml2::axi_lock_type_extension *lock_type_ext;
transaction.get_extension_attr(lock_type_ext, lock_type);
```

The definition of the attribute values in relation to the AXI definition is given below:

**Table 3-15 Attribute Values**

<b>axi_lock_type_enum</b>	<b>ARLOCK[1:0] AWLOCK[1:0]</b>	<b>Access Type</b>
axiNORMAL	b00	Normal access
axiEXCL	b01	Exclusive access
axiLOCK	b10	Locked access
axiLOCK_ERR	b11	Reserved

- scml2::axi\_access\_mode\_extension:

Is an extension to represent the AWPROT [1:0] and ARPROT [1:0] signals, so only the signals related to the protection unit. The default value is axiNORM.

```
typedef enum { axiNORM, axiPRIV, axiNORM_NON_SECURE, axiPRIV_NON_SECURE}
axi_access_mode_enum;

scml2::axi_access_mode_enum access_mode;
scml2::axi_access_mode_extension *access_mode_ext;
transaction.get_extension_attr(access_mode_ext, access_mode);
```

The definition of the attribute values in relation to the AXI definition is given below:

**Table 3-16 Attribute Values**

<b>axi_access_mode_enum</b>	<b>ARPROT[1:0] AWPROT[1:0]</b>	<b>Protection Level</b>
axiNORM	b00	Normal access

<b>axi_access_mode_enum</b>	<b>ARPROT[1:0] AWPROT[1:0]</b>	<b>Protection Level</b>
axiPRIV	b01	Privileged access
axiNORM_NON_SECURE	b10	Non secure normal access
axiPRIV_NON_SECURE	b11	Non secure privileged access

- scml2::axi\_access\_type\_extension:

This attribute represents ARPROT [2] and AWPROT [2] which indicate whether the transaction represents a data or instruction access. It has been separated out in order to allow for easier conversion to other protocols that also support access type information, but not necessarily combined with the protection information as exists in AXI. The default value is axiDATA\_ACCESS.

```
typedef enum { axiDATA_ACCESS, axiINST_ACCESS} axi_access_type_enum;

scml2::axi_access_type_enum access_type;
scml2::axi_access_type_extension *access_type_ext;
transaction.get_extension_attr(access_type_ext, access_type);
```

The definition of the attribute values in relation to the AXI definition is given below:

**Table 3-17 Attribute Values**

<b>axi_access_type_enum</b>	<b>ARPROT[2] AWPROT[2]</b>	<b>Access Type</b>
axiDATA_ACCESS	b0	Data access
axiINST_ACCESS	b1	Instruction access

- Cache support:

The AXI protocol defines cache information signals through ARCACHE [3 : 0] and AWCACHE [3 : 0]. These signals are represented in the FT AXI protocol definition through separate boolean extensions. Notice the amba\_ prefix for these extensions which allows to reuse them for other AMBA protocol definitions.

```
DECLARE_EXTENSION(amba_cacheable_extension, bool, false);
DECLARE_EXTENSION(amba_bufferable_extension, bool, false);
DECLARE_EXTENSION(amba_cache_wr_alloc_extension, bool, false);
DECLARE_EXTENSION(amba_cache_rd_alloc_extension, bool, false);
```

The definition of the attribute values in relation to the AXI definition is given below:

amba_bufferable_extension	ARCACHE [0] and AWCACHE [0]	Bufferable bit (B)
amba_cacheable_extension	ARCACHE [1] and AWCACHE [1]	Cacheable bit(C)
amba_cache_rd_alloc_extension	ARCACHE [2] and AWCACHE [2]	Read Allocate (RA)
amba_cache_wr_alloc_extension	ARCACHE [3] and AWCACHE [3]	Write Allocate (WA)

- User signals:

The AXI specification and also the FT AXI definition supports user extensions. These are user-defined signals that are available in the FT AXI definition as `sc_dt::sc_bignum<1024>` signals.

```
DECLARE_EXTENSION(amba_aw_user_signal_extension, sc_dt::sc_bignum<1024>, 0);
DECLARE_EXTENSION(amba_ar_user_signal_extension, sc_dt::sc_bignum<1024>, 0);
DECLARE_EXTENSION(amba_w_user_signal_extension, sc_dt::sc_bignum<1024>, 0);
DECLARE_EXTENSION(amba_r_user_signal_extension, sc_dt::sc_bignum<1024>, 0);
DECLARE_EXTENSION(amba_b_user_signal_extension, sc_dt::sc_bignum<1024>, 0);
```

- `scml2::axi_qos_extension`:

The AXI specification and also the FT AXI definition supports QoS signals. These signals are quality of service signals that are available in the FT AXI definition as `unsigned int` signals.

```
DECLARE_EXTENSION(axi_qos_extension, unsigned int, 0);
```

These represent ARQOS [3 : 0] and AWQOS [3 : 0] signals in AXI protocol. A default value of 0 indicates that the interface is not participating in any QoS scheme.



This signal is implemented only in AXI4.

- `scml2::axi_region_extension`:

The AXI specification and also the FT AXI definition supports Region signals. These are to support multiple regions that are available in the FT AXI definition as `unsigned int` signals.

```
DECLARE_EXTENSION(axi_region_extension, unsigned int, 0);
```

These represent ARREGION [3 : 0] and AWREGION [3 : 0] signals in AXI protocol.

The 4-bit region identifier can be used to uniquely identify up to sixteen different regions. The region identifier can provide a decode of higher order address bits. The region identifier must remain constant within any 4 kilobyte address space.



This signal is implemented only in AXI4.

- `scml2::axi_snoop_extension`

The AXI specification and also the FT AXI definition support Snoop signals. These are to support different kind of snoop transactions that are available in the FT AXI.

```
DECLARE_EXTENSION(axi_snoop_extension, axi_snoop_enum, axiInvalidSnoop);
```

In FT AXI, enum values have been defined which represent a combination of Barrier, Domain and Snoop Signals. This is done to ease modeling of AXI4 devices.

The definition of the attribute values in relation to the AXI definition is given below:

**Table 3-18 AXI Definition Attribute Values**

axi_snoop_enum	AxBAR [0]	AxDOMAIN[1:0]	ARSNOOP[3:0] / AWSNOOP[2:0]	Snoop Request Type
axiInvalidSnoop	-	-	-	Initialization Value
axiReadNoSnoop	0b0	0b00/0b11	ARSNOOP = 0b0000	Read No Snoop
axiReadOnce	0b0	0b01/0b10	ARSNOOP = 0b0000	Read Once
axiReadShared	0b0	0b01/0b10	ARSNOOP = 0b0001	Read Shared
axiReadClean	0b0	0b01/0b10	ARSNOOP = 0b0010	Read Clean
axiReadNotShareDirty	0b0	0b01/0b10	ARSNOOP = 0b0011	Read Not Shared Dirty
axiReadUnique	0b0	0b01/0b10	ARSNOOP = 0b0111	Read Unique
axiCleanUnique	0b0	0b01/0b10	ARSNOOP = 0b1011	Clean Unique
axiMakeUnique	0b0	0b01/0b10	ARSNOOP = 0b1100	Make Unique
axiCleanShared	0b0	0b00/0b01/0b10	ARSNOOP = 0b1000	Clean Shared
axiCleanInvalid	0b0	0b00/0b01/0b10	ARSNOOP = 0b1001	Clean Invalid
axiMakeInvalid	0b0	0b00/0b01/0b10	ARSNOOP = 0b1101	Make Invalid
axiDVM_Complete	0b0	0b01/0b10	ARSNOOP = 0b1110	DVM Complete
axiDVM_Message	0b0	0b10/0b10	ARSNOOP = 0b1111	DVM Message
axiWriteNoSnoop	0b0	0b00/0b11	AWSNOOP = 0b000	Write No Snoop
axiWriteUnique	0b0	0b01/0b10	AWSNOOP = 0b000	Write Unique
axiWriteLineUnique	0b0	0b01/0b10	AWSNOOP = 0b001	Write Line Unique
axiWriteClean	0b0	0b00/0b01/0b10	AWSNOOP = 0b010	Write Clean
axiWriteBack	0b0	0b00/0b01/0b10	AWSNOOP = 0b011	Write Back
axiEvict	0b0	0b01/0b10	AWSNOOP = 0b100	Evict
axiWriteEvict	0b0	0b00/0b01/0b10	AWSNOOP = 0b101	WriteEvict

- scml2::axi\_domain\_extension

The AXI specification and also the FT AXI definition support Domain signals. These indicate the shareability domain of a read/write transaction.

```
DECLARE_EXTENSION(axi_domain_extension, axi_domain_enum, axiNon_Shareable);
```

These represent ARDOMAIN[1:0] and AWDOMAIN[1:0] signals in AXI protocol.

The definition of the attribute values in relation to the AXI definition is given below.

**Table 3-19 AXI Definition Attribute Values**

axi_domain_enum	AxDOMAIN[1:0]	Domain Type
axiNon_Shareable	0b00	Non-Shareable
axiInner_Shareable	0b01	Inner Shareable
axiOuter_Shareable	0b10	Outer Shareable
axiSystem	0b11	System

- scml2::axi\_barrier\_extension

The AXI specification and also the FT AXI definition support Barrier signals. These indicate a read/write barrier transaction.

```
DECLARE_EXTENSION(axi_barrier_extension, axi_barrier_enum, axiNormal);
```

These represent ARBAR [1:0] and AWBAR [1:0] signals in AXI protocol.

The definition of the attribute values in relation to the AXI definition is given below.

**Table 3-20 AXI Definition Attribute Values**

axi_barrier_enum	AxBAR[1:0]	Barrier Type
axiNormal	0b00	Normal access, respecting barriers
axiMemory	0b01	Memory Barrier
axiIgnore	0b10	Normal access, ignoring barriers
axiSynchronizing	0b11	Synchronization barrier

- scml2::axi\_wr\_unique\_extension

The AXI specification and also the FT AXI definition support AWUNIQUE signals. These can be used to improve the operation of lower levels of cache hierarchy, such as an L3 or system level cache.

```
DECLARE_EXTENSION(axi_wr_unique_extension, bool, false);
```

All payload attributes should be created by the initiator at the start of the transaction (AWVALID or ARVALID) to make sure that all extensions that will be used are available. All AXI attributes should be set by the initiator at the start of the transaction, except:

- axi\_rsp\_enum: Should be set by the target. It can be set once for writes (with the BVALID phase) and can be set with each data beat (RVALID) for reads.
- The content pointed to by the data\_ptr can be modified during the transaction:
  - By the initiator to make write data available per write data beat.
  - By the target to make read data available per read data beat.
  - A target should not use the data\_ptr ahead of the first WVALID in case of write transactions.

The AXI protocol definition is built according to the FT Modeling approach, so this means that the extensions defined for the AXI protocol are not ignorable within the component. The AXI protocol definition is built on top of the TLM2.0 base protocol so the TLM2.0 base protocol attributes are always available. However, the AXI protocol defines certain extensions that overrule the TLM2.0 base protocol. Therefore, the following rules apply with regards to the TLM2.0 base protocol attributes for a AXI transaction:

1. **Burst\_type**: The burst type should be taken from the `burst_type` enum and not according to the TLM2-GP rules. The `burst_type` attribute should be valid with the `AWVALID` and `ARVALID` protocol states, the same applies for `lock`, `access_mode` and `access_type`.
2. **tlm\_response\_status**: This attribute should not be used, it is required to use the `axi_rsp_enum`. The response should be valid with the `BVALID` protocol state, or with any of the read data transfers. There is only one error response allowed for the whole transaction in case of writes, for reads multiple responses are allowed.
3. **Data\_length** attribute is limited to the permitted values in AXI: it represents `AWLEN`/`ARLEN` as follows:  
`data_length = AWLEN + 1;`
4. **Byte\_enables** represent the `WSTRB` note, which for AXI byte enables cannot be wider than the `BUSWIDTH` template parameter on the sockets.
5. It is not allowed to use `IGNORE_COMMAND`.
6. **Streaming\_width** shall be ignored.
7. The AXI signals `AWSIZE` and `ARSIZE` are represented by the common extension `burst_size_extension`, as follows:

```
ARSIZE = log2(burst_size_extension == 0? BUSWIDTH/8 : burst_size_extension)
```

The values are limited to the allowed values in AXI (1, 2, 4, 8, 16, 32, 64 and 128).

8. The AXI signals `AWID`, `ARID`, `WID`, `RID`, `BID` are represented by the common extension `trans_id_extension`.
9. The common extension `wrap_addr_extension` represents the start address for a wrapping burst in AXI, the payload address will always refer to the lowest address in the transaction as per the TLM2 standard.

### 3.3.3 FT ACE API Definition

The FT ACE API definition is built on top of the FT AXI protocol and includes all the APIs as defined by FT AXI protocol.

The FT ACE protocol definition adds the following extensions to the FT AXI protocol:

- `scml2::ace_response_extension`

This extension models the `RRESP[3:2]` and `CRRESP[3:2]` signals of the ACE protocol. These additional read/snoop response bits provide information for shareable read transaction.

In case of a read transaction, the `ace_response_extension` must have constant value for all data beats in the burst.

The definition of the attribute values in relation to the ACE definition is given in the following table.

**Table 3-21 ACE Definition Attribute Values**

ace_response_enum	RRESP[3:2]/C RRESP[3:2]	Response	Meaning
aceRspNotSharedClean	b00	Not Shared & Clean	Indicates that initiator does not take the responsibility to ensure that the cache line is written to the main memory. Also, it indicates that the line is the only cached copy.
aceRspNotSharedDirty	b01	Not Shared & Dirty	Indicates that cache line is dirty and initiator takes the responsibility to ensure that the cache line is written to the main memory. Also, it indicates that the line is the only cached copy.
aceRspSharedClean	b10	Shared & Clean	Indicates that initiator does not take the responsibility to ensure that the cache line is written to the main memory. Also, it indicates that another copy of the data might be held in another cache.
aceRspSharedDirty	b11	Shared & Dirty	Indicates that cache line is dirty and initiator takes the responsibility to ensure that the cache line is written to the main memory. Also, it indicates that another copy of the data might be held in another cache.

- scml2::ace\_rsp\_pass\_data\_extension

This extension models the CRRESP[0] signal of the ACE protocol. It indicates whether the snoop request will have data transfer or not.

```
DECLARE_EXTENSION(ace_rsp_pass_data_extension, bool, false);
```

When set to true, it indicates that a full cache line of data will be provided on the Snoop Data channel.

- scml2::ace\_rsp\_was\_unique\_extension

This extension models the CRRESP[4] signal of the ACE protocol. It provides information on the unique/shared state of the cache line before the snoop request.

```
DECLARE_EXTENSION(ace_rsp_was_unique_extension, bool, false);
```

When set to true, it indicates that the cache line was in unique state before snoop request.



FT AXI protocol has no semantics to indicate CRRESP[1] bit. This is because the bit represents hardware errors which are not modeled in modeling world.

### 3.3.4 FT AXI4 Stream API Definition

The FT AXI4 Stream API definition is built on top of the TLM2.0 base protocol just like any other protocol in the FT modeling approach. The FT AXI4 Stream protocol definition adds the following new extensions to the TLM2.0 base protocol.

- `scml2::axi4_stream_protocol_state_extension`

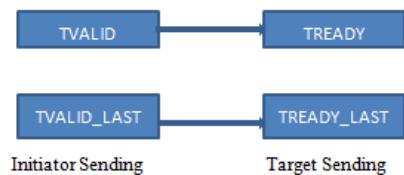
This is the payload extension that holds the protocol timing points for the AXI4 Stream protocol.

A socket that intends to communicate using the AXI4 Stream protocol should indicate this via the `set_protocol` API:

```
P_socket.set_protocol<scml2::axi4_stream_protocol_state_extension>("AXI4-Stream");
```

The AXI4 Stream protocol defines the protocol states, each referring to a protocol timing point, as shown below. They are shown in pairs to indicate the transfers that they represent.

**Figure 3-54 FT AXI4 Stream API Definition**



- `scml2::axi4_stream_tkeep_extension`

```
DECLARE_ARRAY_EXTENSION(axi4_stream_tkeep_extension, unsigned char);
```

The value of this extension is `unsigned char*` array with length equal to number of valid bytes in the array. This array corresponds to the TKEEP signal of the AXI4 Stream interface and gives information on the NULL bytes in the data stream. The semantics for this extension follow the same rules that are defined for byte enable array in [TLM2 Reference manual](#). The following rules are important to note:

- The elements in the `tkeep_extension` array shall be interpreted as follows. A value of 0 at index *i* shall indicate that byte at index *i* in data stream is NULL bytes, and a value of `0xff` shall indicate that the corresponding byte is either Positional or Data byte depending on value of `byte_enable` array at the same index. The meaning of all other values shall be undefined. The value `0xff` has been chosen so that the `tkeep` extension array can be used directly as a mask.
- The `tkeep` extension mask may be defined by a small pattern applied repeatedly or by a large pattern covering the whole data array.
- If the length of `tkeep` extension is 0, then it implies that the value of `tkeep` array is assumed to be `0xff` for all the data bytes, that is, there are no NULL bytes in the stream.

This extension will be added to the TLM2 payload using the following semantics:

```
SET_ARRAY_EXT_ATTR(trans, scml2::axi4_stream_tkeep_extension, unsigned char,
                    tkeep_array_ptr, datalength );
```

Where:

<code>trans</code>	Is the FT payload.
<code>tkeep_array_ptr</code>	Returns the pointer to start of extension.

data_length	Is the length of the array in bytes.
-------------	--------------------------------------

- The macro will add the `axi4_stream_tkeep_extension` to the payload, if it does not exist already.
- This gets automatically resized in case, the extension existed already and the original valid count was less than the count specified.

To retrieve the value of the array, use the following macro:

```
GET_ARRAY_EXT_ATTR(trans, scml2:: axi4_stream_tkeep_extension, unsigned char,
                   tkeep_array_ptr, num_entries);
```

- The number of valid entries gets stored in `num_entries`.
- The start of the array is stored in `tkeep_array_ptr` variable which is of type `unsigned char*`.
- `scml2::axi4_stream_tuser_extension`

This extension contains value of TUSER signal on an AXI4 Stream interface. The TUSER signal contains the user sideband information that needs to be sent with the data stream.

This is not a new extension, but a `typedef` for `scml2:: amba_wdata_usr_sig_extension`.

- `scml2::axi4_stream_tdest_extension`

```
DECLARE_EXTENSION(axi4_stream_tdest_extension, unsigned, 0);
```

This extension can be used to specify TDEST signal on an AXI4 Stream interface.

In addition to the above new extensions, AXI4 Stream protocol uses the following extensions that are already defined in FT Modeling.

- `scml2::trans_id_extension`

```
DECLARE_EXTENSION(transaction_id_extension, unsigned int, 0);
```

This extension contains value of TID signal on an AXI4 Stream interface. The TID signal specifies the data stream identifier that indicates different streams.

TDATA and TSTRB signals of the AXI4 Stream protocol are modeled using the existing semantics of data pointer and byte enable pointer in TLM2 payload.

TDEST signal is modeled by address field of the TLM Generic Payload. The following table summarizes all the signals of the AXI4 Stream protocol and their corresponding payload semantics.

**Table 3-22 AXI4 Stream Protocol Signals and Payload Semantics**

AXI4 Stream Interface Signal Names	FT Modeling Semantics
TVALID	Value on <code>axi4_stream_protocol_state_extension</code> .
TREADY	Value on <code>axi4_stream_protocol_state_extension</code> .
TDATA	Data pointer on TLM2 Payload.
TSTRB	Byte Enable pointer on TLM2 Payload one-to-one correspondence with Data pointer.

<b>AXI4 Stream Interface Signal Names</b>	<b>FT Modeling Semantics</b>
TKEEP	Value on <code>scml2::axi4_stream_tkeep_extension</code> .
TLAST	Value on <code>axi4_stream_protocol_state_extension</code> .
TID	Value on <code>scml2::transaction_id_extension</code> .
TDEST	Address field of TLM2 payload or <code>axi4_stream_tdest_extension</code> .
TUSER	Value on <code>scml2::amba_wdata_usr_sig_extension</code> .

### 3.3.5 FT CHI API Definition

The FT CHI API definition is built on top of the TLM2.0 base protocol just like any other protocol in the FT modeling approach. The FT CHI protocol definition adds the following extensions to the TLM2.0 base protocol:

- `scml2::chi_rnf_protocol_state_extension`

This is the payload extension, that holds the protocol timing points for the CHI protocol. Since CHI RN-F includes all the CHI channels, this extension can be used to model timing points for RN-D, RN-I, SN-F and SN-I.

- `scml2::chi_allow_retry_extension`

This extension models the AllowRetry CHI protocol signal.

- `scml2::chi_req_opcode_extension`

This extension models the Opcode CHI protocol signal on Request channel.

- `scml2::chi_snp_opcode_extension`

This extension models the Opcode CHI protocol signal on Snoop request channel.

- `scml2::chi_data_opcode_extension`

This extension models the Opcode CHI protocol signal on Data channel.

- `scml2::chi_resp_opcode_extension`

This extension models the Opcode CHI protocol signal on Response channel.

- `scml2::chi_atomic_opcode_extension`

This extension models the sub opcodes for atomic load and store transactions.

- `scml2::chi_data_pull_extension`

This extension models the DataPull CHI protocol signal.

- `scml2::chi_data_source_extension`

This extension models the DataSource CHI protocol signal.

- `scml2::chi_dbid_extension`

This extension models the DBID CHI protocol signal.

- `scml2::chi_do_not_data_pull_extension`

This extension models the DoNotDataPull CHI protocol signal.

- `scml2::chi_do_not_go_to_sd_extension`  
This extension models the DoNotGoToSD CHI protocol signal.
- `scml2::chi_endian_extension`  
This extension models the Endian CHI protocol signal.
- `scml2::chi_exclusive_extension`  
This extension models the Exclusive CHI protocol signal.
- `scml2::chi_exp_comp_ack_extension`  
This extension models the ExpCompAck CHI protocol signal.
- `scml2::chi_fwd_nid_extension`  
This extension models the FwdNID CHI protocol signal.
- `scml2::chi_fwd_state_extension`  
This extension models the FwdState CHI protocol signal.
- `scml2::chi_fwd_txnid_extension`  
This extension models the FwdTxnId CHI protocol signal.
- `scml2::chi_home_nid_extension`  
This extension models the HomeNid CHI protocol signal.
- `scml2::chi_likely_shared_extension`  
This extension models the LikelyShared CHI protocol signal.
- `scml2::chi_lpid_extension`  
This extension models the LPID CHI protocol signal.
- `scml2::chi_memattr_early_write_ack_extension`  
This extension models the CHI MemAttr[0] protocol signal.
- `scml2::chi_memattr_device_extension`  
This extension models the MemAttr[1] CHI protocol signal.
- `scml2::chi_memattr_cacheable_extension`  
This extension models the MemAttr[2] CHI protocol signal.
- `scml2::chi_memattr_allocate_hint_extension`  
This extension models the MemAttr[3] CHI protocol signal.
- `scml2::chi_ccid_extension`  
This extension models the CCID CHI protocol signal.
- `scml2::chi_dataid_extension`  
This extension models the DataID CHI protocol signal.
- `scml2::chi_data_check_extension`  
This extension models the DataCheck CHI protocol signal.
- `scml2::chi_poison_extension`  
This extension models the Poison CHI protocol signal.

- `scml2::chi_non_secure_extension`  
This extension models the NS CHI protocol signal.
- `scml2::chi_order_extension`  
This extension models the Order CHI protocol signal.
- `scml2::chi_p_crd_type_extension`  
This extension models the PCrdType CHI protocol signal.
- `scml2::chi_qos_extension`  
This extension models the QoS CHI protocol signal.
- `scml2::chi_resp_extension`  
This extension models the Resp CHI protocol signal.
- `scml2::chi_resp_err_extension`  
This extension models the RespErr CHI protocol signal.
- `scml2::chi_data_resp_extension`  
This extension models the Resp CHI protocol signal on Data channel.
- `scml2::chi_data_resp_err_extension`  
This extension models the RespErr CHI protocol signal on Data channel.
- `scml2::chi_ret_to_src_extension`  
This extension models the RetToSrc CHI protocol signal.
- `scml2::chi_return_nid_extension`  
This extension models the ReturnNID CHI protocol signal.
- `scml2::chi_rsvdc_extension`  
This extension models the RSVDC CHI protocol signal.
- `scml2::chi_return_txnid_extension`  
This extension models the ReturnTxnID CHI protocol signal.
- `scml2::chi_snoop_me_extension`  
This extension models the SnoopMe CHI protocol signal.
- `scml2::chi.snp_attr_extension`  
This extension models the SnpAttr CHI protocol signal.
- `scml2::chi_srcid_extension`  
This extension models the SrcID CHI protocol signal.
- `scml2::chi_stash_lpid_extension`  
This extension models the StashLPID CHI protocol signal.
- `scml2::chi_stash_lpid_valid_extension`  
This extension models the StashLPIDValid CHI protocol signal.
- `scml2::chi_stash_nid_extension`  
This extension models the StashNID CHI protocol signal.

- `scml2::chi_stash_nid_valid_extension`  
This extension models the `StashNIDValid` CHI protocol signal.
- `scml2::chi_req_tgtid_extension`  
This extension models the `TgtID` CHI protocol signal on Request channel.
- `scml2::chi_data_tgtid_extension`  
This extension models the `TgtID` CHI protocol signal on Data channel.
- `scml2::chi_resp_tgtid_extension`  
This extension models the `TgtID` CHI protocol signal on Response channel.
- `scml2::chi_req_txnid_extension`  
This extension models the `TxnID` CHI protocol signal on Request and Snoop channel.
- `scml2::chi_resp_txnid_extension`  
This extension models the `TxnID` CHI protocol signal on Response channel.
- `scml2::chi_data_txnid_extension`  
This extension models the `TxnID` CHI protocol signal on Data channel.
- `scml2::chi_vmid_extension`  
This extension models the `VMIDExt` CHI protocol signal.
- `scml2::chi_deep_extension`  
This extension models the `Deep` CHI protocol signal.
- `scml2::chi_pgroup_id_extension`  
This extension models the `PGroupId` CHI protocol signal.
- `scml2::chi_mpam_extension`  
This extension models the `MPAM` CHI protocol signal.
- `scml2::chi_resp_cbusy_extension`  
This extension models the `CBusy` CHI protocol signal on Response channel.
- `scml2::chi_data_cbusy_extension`  
This extension models the `CBusy` CHI protocol signal on Data channel.
- `scml2::chi_slc_rep_hint_extension`  
This extension models the `SLCRepHint` CHI protocol signal.
- `scml2::chi_do_dwt_extension`  
This extension models the `DoDWT` CHI protocol signal.
- `scml2::chi_stash_group_id_extension`  
This extension models the `StashGroupId` CHI protocol signal.
- `scml2::chi_tag_group_id_extension`  
This extension models the `TagGroupId` CHI protocol signal.
- `scml2::chi_req_tag_op_extension`  
This extension models the `TagOp` CHI protocol signal on Request channel.

- `scml2::chi_data_tag_op_extension`  
This extension models the TagOp CHI protocol signal on Data channel.
- `scml2::chi_resp_tag_op_extension`  
This extension models the TagOp CHI protocol signal on Response channel.
- `scml2::chi_tag_extension`  
This extension models the Tag CHI protocol signal.
- `scml2::chi_tag_update_extension`  
This extension models the TU CHI protocol signal.
- `scml2::chi_trace_tag_extension`  
This extension models the TraceTag CHI protocol signal.

### 3.3.6 FT PCIe API Definition

The FT PCIe API definition is built on top of the TLM2.0 base protocol just like any other protocol in the FT modeling approach. The FT PCIe protocol definition adds the following extensions to the TLM2.0 base protocol.

- `pcie_req_tlp_extension`  
This extension models the contents of PCIe Request TLP.
- `pcie_cpl_tlp_extension`  
This extension models the contents of PCIe Completion TLP.
- `pcie_credit_dllp_extension`  
This extension models the contents of PCIe Credit DLLP.
- `pcie_dllp_extension`  
This extension models the contents of all other DLLPs, except Credit DLLP.
- `pcie_protocol_state_extension`  
This extension holds the protocol timing points for FT PCIe protocol.

Contents of these extensions are defined in

`$(SNPS_VP_HOME)/common/include/scml2_tlm2/snps_tlm2_extensions/snps_tlm2_pcie_protocol_extension.h`

### 3.3.7 FT CXL API Definition

The FT CXL API definition is built on top of the TLM2.0 base protocol just like any other protocol in the FT modeling approach. The FT CXL protocol definition adds the following extensions to the TLM2.0 base protocol.

- `cxl_mem_m2s_req_extension`  
This extension models the contents of CXL.mem request message (read/write request to CXL memory).
- `cxl_mem_s2m_ndr_extension`  
This extension models the contents of CXL.mem non-data response message (write response from CXL memory).
- `cxl_mem_s2m_drs_extension`

This extension models the contents of CXL.mem data response message (read data from CXL memory).

Contents of these extensions are defined in

`$(SNPS_VP_HOME)/common/include/scml2_tlm2/snps_tlm2_extensions/snps_tlm2_cxl_protocol_extension.h`

## 3.4 Protocol Checker

This chapter describes:

- [Introduction](#)
- [Features](#)
- [Input Requirements](#)
- [Getting Started](#)

### 3.4.1 Introduction

Protocol checker is a utility added in extended SCML2 initiator and target socket to enable verification of FT protocol semantics in the TLM2 peripherals. It supports the following protocols:

- TLM2 FT AXI
- TLM2 FT GFT
- TLM2 GP

### 3.4.2 Features

This section describes the various protocol violations reported by Protocol Checker. To detect a violation, Protocol Checker applies the standard TLM2 FT and TLM2 GP protocol rules for each protocol, and reports if any of the rules is violated.

An invalid state transition by the initiator or target is reported as State Transition violation. For valid state transitions, see “[FT GFT Protocol Definition](#)” on page 99 and “[FT AXI API Definition](#)” on page 134. For TLM2 GP, phase transition is checked. For example, in the FT AXI protocol ARREADY to BVALID is an invalid transition and is reported as State Transition violation by Protocol Checker, as shown below.

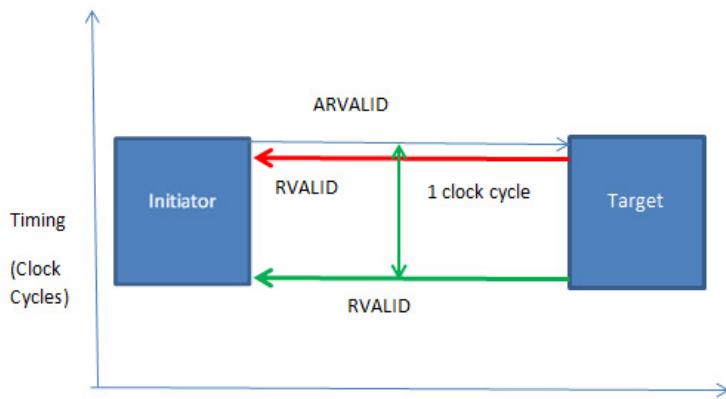
**Figure 3-55 State Transition ARREADY — BVALID is Invalid**



The Protocol Checker reports State Transition Timing violation if minimum delay requirement between any valid state transitions is not met. To validate the timing, it applies Timing annotation rules as mentioned in table 3-6, table 3-8 and table 3-9. This violation is not reported for TLM2 GP as the base protocol does not have any rules regarding phase transition timing.

The Protocol Checker reports State Transition Timing violation, as the minimum requirement of one clock delay between ARVALID and RVALID is not met. This is indicated by red signal in the figure below. The green signal indicates call at delay of one clock cycle, as required by the protocol definition.

**Figure 3-56 State Transition Timing Violation**



Checking for channel availability is a universal feature of FT modeling that needs to be verified for every protocol. The channel owner, who initiates access to that channel, is responsible to ensure that the channel is free before you invoke a new access on the channel. If the channel is accessed before it is free, the Protocol Checker reports Channel violation.

### 3.4.3 Input Requirements

To report State Transition Timing violations, Protocol Checker requires clock period at which initiators and targets operate. For this, it relies on the extended scml2 sockets that have an additional user API, `set_clock`. This API should be called by each initiator and target at the end of elaboration to specify the clock period at which they will operate.



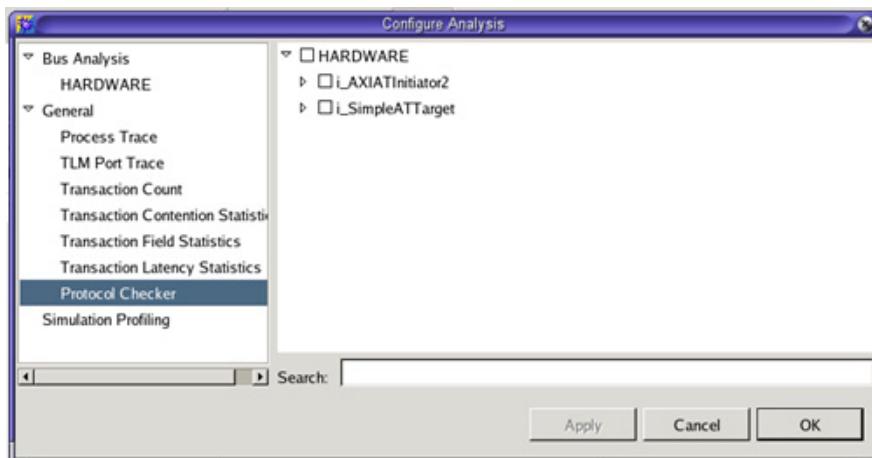
If the user model does not call `set_clock` to set the clock period, the Protocol Checker does not report timing violations.

### 3.4.4 Getting Started

#### To enable or disable the Protocol Checker:

- 1 From the VP Explorer menu bar, select *Analysis > Configure*.  
The Configure Analysis dialog box appears.

**Figure 3-57 Configure Analysis Dialog Box — Enabling Analysis**



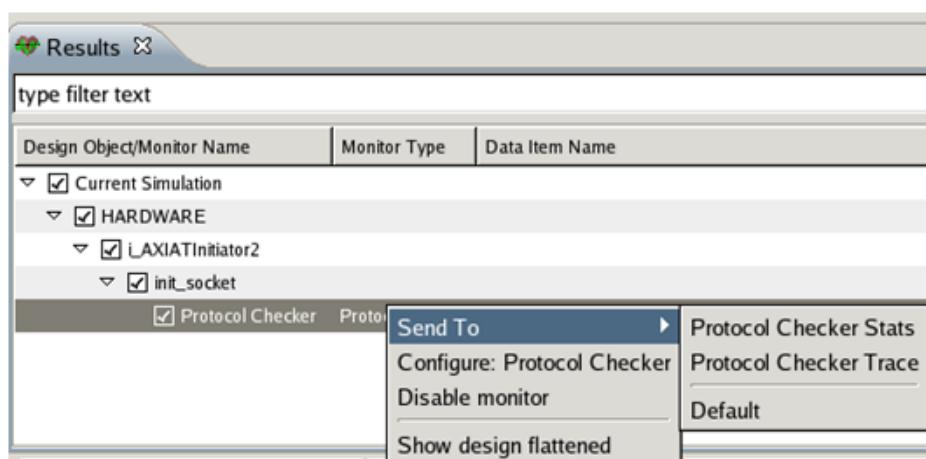
- 2 Expand the *General* node and select *Protocol Checker*.
- 3 Toggle the checkboxes on right-hand side panel to enable or disable the Protocol Checker.

**To view the results of your analysis, after the simulation suspends or completes:**

- 1 Open the *Results* view and right-click on the Protocol Checker item for your model.
- 2 From the context menu, select *Send To > Protocol Checker Trace/Stats*.

Where, *Protocol Checker Stats* is the statistic data that depicts the number of violations and *Protocol Checker Trace* is the trace data that shows the event corresponding to the time of violation.

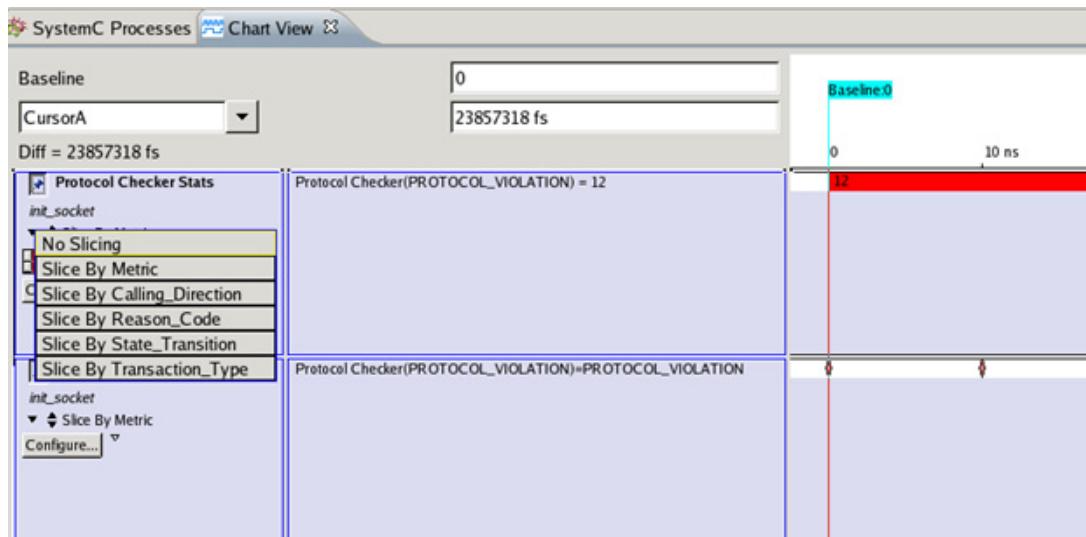
**Figure 3-58 Viewing Results**



This section provides example of the slicing parameters that you can use to understand the violations.

- Metric
- Calling\_Direction
- Reason\_Code
- State\_Transition
- Transaction\_Type

**Figure 3-59 Bifurcating Results**



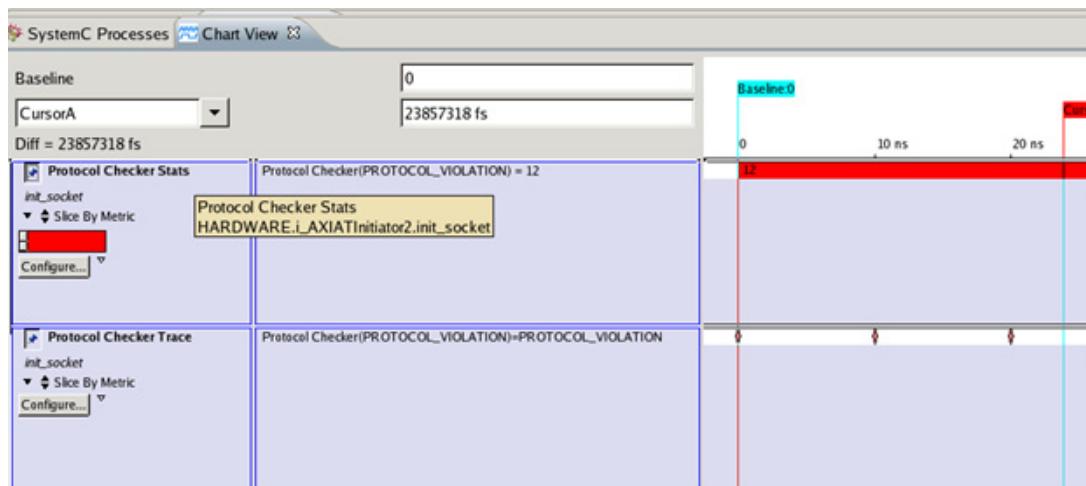
#### 3.4.4.1 Metric

Default view that shows all the existing violations in a single view.

For example:

Protocol Checker (PROTOCOL\_VIOLATION)=12 depicts, protocol violation occurred 12 times during the simulation, as shown below.

**Figure 3-60 Metric Slicing**



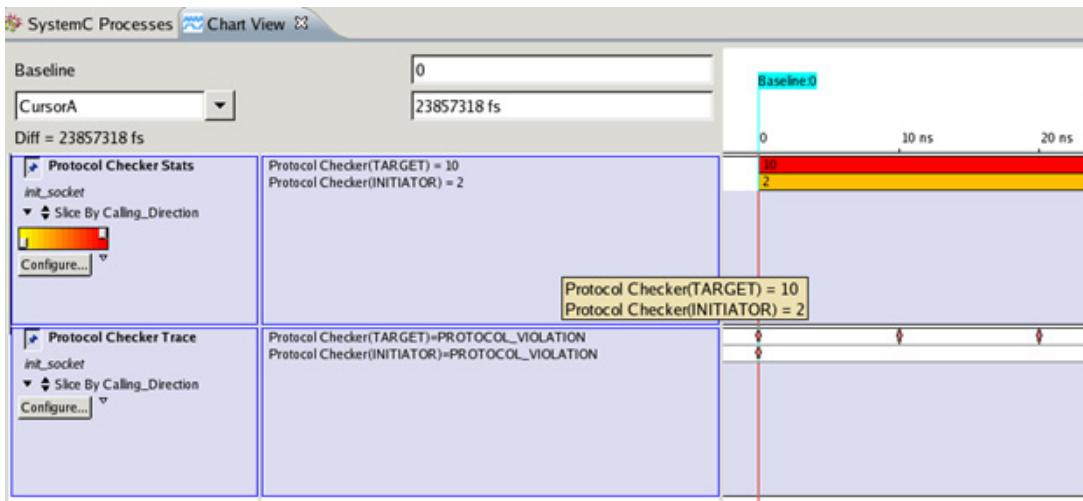
#### 3.4.4.2 Calling\_Direction

You can bifurcate results on calling direction, or the originator of violations.

For example:

Protocol Checker (Target) = 10 depicts, target is responsible for these violations, as shown figure below.

**Figure 3-61 Calling\_Direction Slicing**



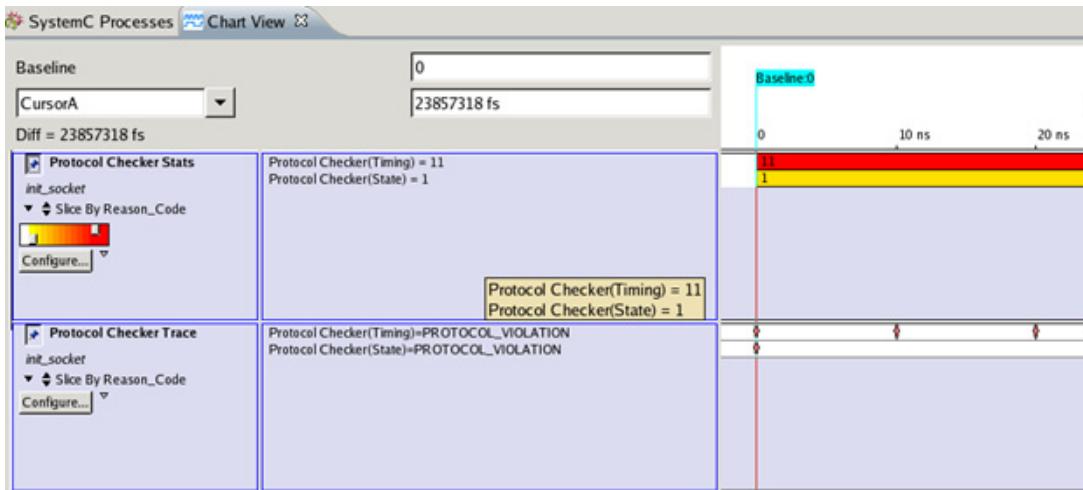
#### 3.4.4.3 Reason\_Code

Specifies the reason of violation.

For example:

Protocol Checker (Timing) depicts, timing violation has occurred, as shown below.

**Figure 3-62 Reason\_Code Slicing**



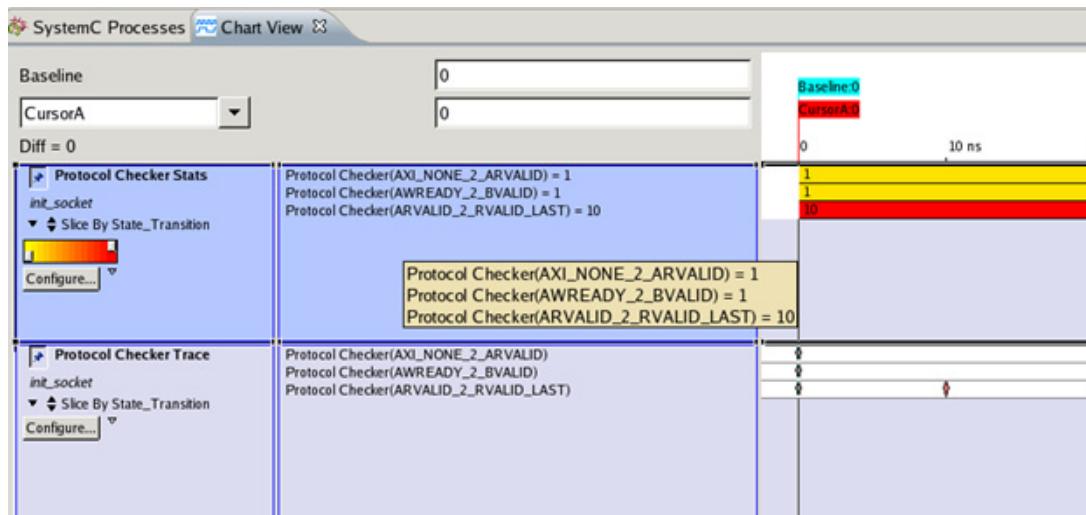
#### 3.4.4.4 State\_Transition

Specifies the state transition associated with the reported violation.

For example:

Protocol Checker (ARVALID\_2\_RVALID\_LAST) depicts, violation occurred while going from ARVALID to RVALID\_LAST, as shown below.

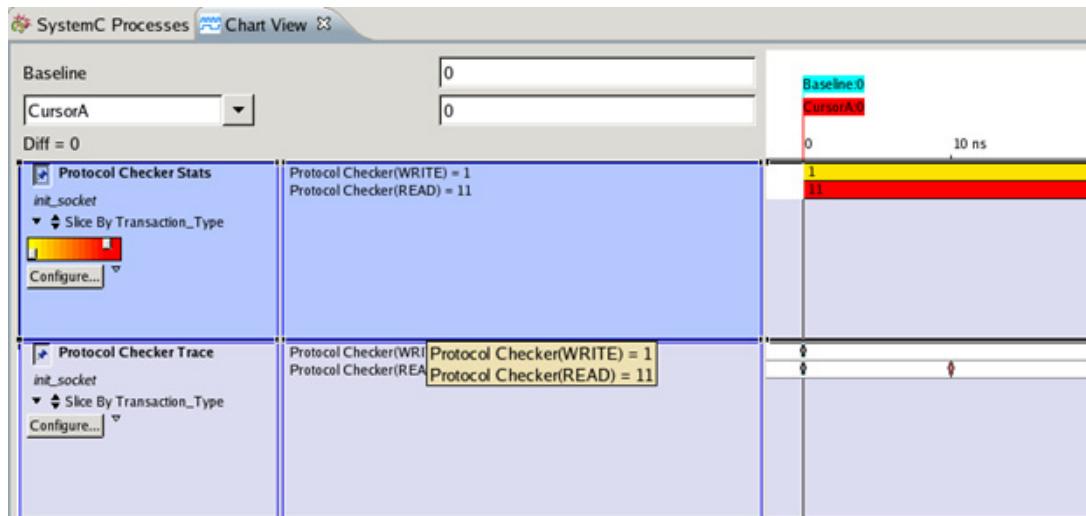
**Figure 3-63 State\_Transition Slicing**



#### 3.4.4.5 Transaction\_Type

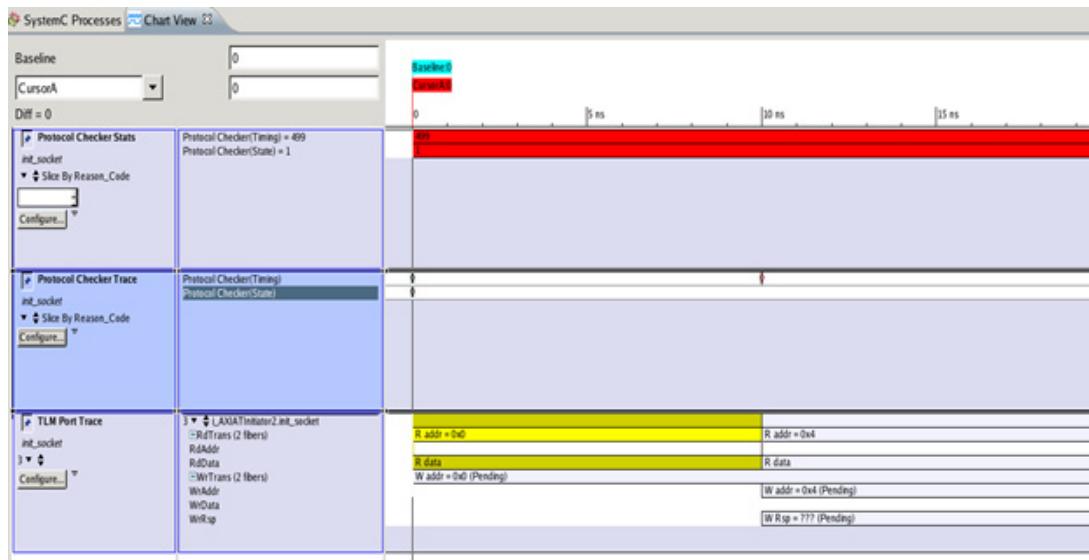
Bifurcates data on the type of transaction. Data is separated for READ and WRITE transactions, as shown below.

**Figure 3-64 Transaction\_Type Slicing**



You can also enable TLM port tracing to get the associated transaction for a particular violation, as shown below.

**Figure 3-65 Interpolating with TLM Port Trace**

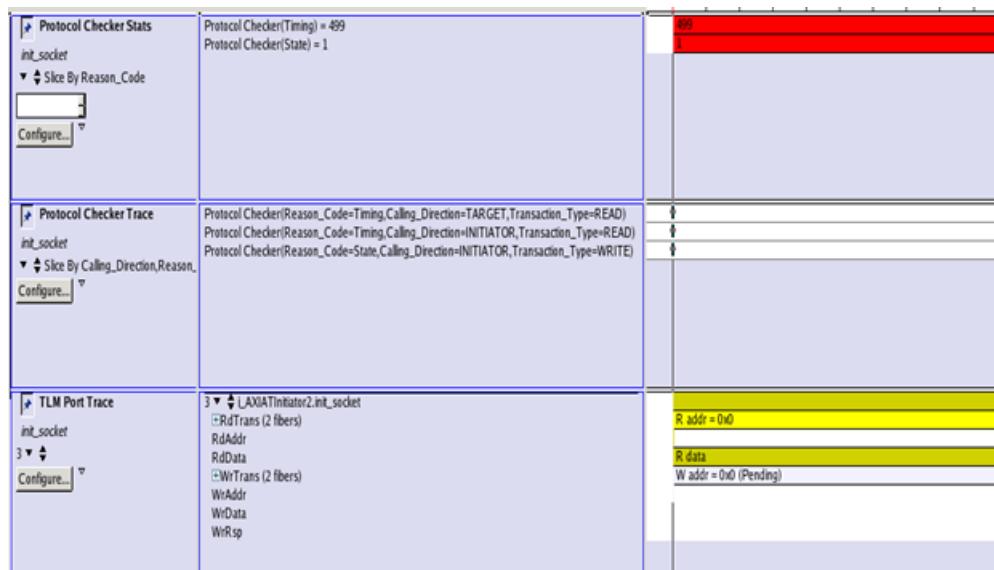


You can use multiple slicing options simultaneously to slice the data, an example is shown below.

#### To slice the data using Reason\_Code, Transaction\_Type, Calling\_Direction:

- 1 In Chart view, click on *Configure* on the left hand side, as shown below.

**Figure 3-66 Example of Multiple Slicing**



- 2 Set the required slicing parameters to true.

For example:

```
Protocol Checker (Reason_code=Timing,Calling_Direction=
Target,Transaction_Type=Read) depicts, timing violation by target for read transaction.
```

If you interpolate it with TLM port trace, RData starts in parallel with Raddr which causes timing violation. This is because, the protocol demands a minimum of clock cycle delay between the two.

# Chapter 4

## Clock Objects

This chapter describes the clock objects.

- [Overview](#)
- [Clocks and Reset](#)
- [Modeling Objects for Clocks \(Clock Objects\)](#)
- [Base Classes](#)
- [Modeling Objects for Base Classes \(Modeling Objects\)](#)
- [Convenience Classes](#)
- [Modeling Objects for Convenience Classes \(Convenience Objects\)](#)
- [Code Example](#)



If the following error message is issued when simulating:

```
ERROR: Error in scml_clock 'divider': the attached master clock 'master' is no  
scml_clock!
```

or

```
Error in get_scml_clock 'divider': the driving clock port must be bound to a channel  
that implements scml_clock_if!
```

You must use an object of type `scml_clock` for the clock source object and export it using an `sc_export<sc_signal_inout_if<bool>>` export, as shown below.

```
sc_export<sc_signal_inout_if<bool>> p_CLK; // clock output port  
scml_clock m_clkObject; // clock source object
```

For details, see `GIPL_CLK` example provided in the [Generic IP Library](#) manual, available with the [Synopsys Model Library documentation](#).

## 4.1 Overview

The following table summarizes the clock objects.

**Table 4-1 Clock Objects**

Modeling Object	Summary
<code>scml_clock</code>	It implements <code>sc_clock_if</code> . It is an optimized version of <code>sc_clock</code> .
<code>scml_divided_clock</code>	It is a clock derived from another clock by multiplying the start time and/or the period with specified integer factors.
<code>scml_clock_gate</code>	It is a module which takes a clock and an enable signal as inputs and produces a gated clock as output.
<code>scml_clock_counter</code>	It has to be attached to a clock and is used to get the number of clock edges that have happened in a certain period.

Modeling Object	Summary
<code>scml2::clocked_module</code>	It is the base class for modules that want to receive SCML clock tick callbacks.
<code>scml2::clocked_timer</code>	It is a modeling object that provides a timer callback mechanism based on an SCML clock.
<code>scml2::clocked_callback</code>	It is a convenience class that forwards a clock tick callback to any member function of a module without the need to inherit from the <code>clocked_module</code> base class.
<code>scml2::clocked_event</code>	It is a convenience class that allows a SystemC method or thread to wait until a certain clock tick happens.
<code>scml2::clocked_peq_container</code>	It is a modeling object for TLM2 FT models using the non-blocking APIs. It buffers payload arriving in the model, like multiple outstanding transactions, possibly coming with different timing annotations from different initiators.
<code>scml2::clocked_peq</code>	It is a modeling object similar to the <code>clocked_peq_container</code> that can trigger a callback whenever an element from the payload buffer becomes available.

## 4.2 Clocks and Reset

- `scml_clock`
- `scml_divided_clock`
- Dynamic Clock Parameter Change and Reset

### 4.2.1 `scml_clock`

An `scml_clock` object implements `sc_clock_if`. It is an optimized version of an `sc_clock`. When nothing is sensitive to the events of the clock, no events notifications are scheduled.

A number of extensions come with an `scml_clock` that are not available for an `sc_clock`:

- changing the period of a clock
- disabling/enabling a clock
- receiving a callback after a specified number of clock ticks

Objects of type `scml_clock` can be constructed using one of the following constructors:

```
scml_clock(const char* name,
           const sc_core::sc_time& period,
           double dutyCycle=0.5,
           const sc_core::sc_time& startTime=sc_core::SC_ZERO_TIME,
           bool posedgeFirst=true);

scml_clock(const char* name,
           double periodV,
           sc_core::sc_time_unit periodTu,
           double dutyCycle=0.5);

scml_clock(const char* name,
           double periodV,
```

```

sc_core::sc_time_unit periodTu,
double dutyCycle,
double startTtimeV,
sc_core::sc_time_unit startTimeTu,
bool posedgeFirst=true);

```

where:

<i>name</i>	Specifies a name for the clock object.
<i>period</i>	Specifies the clock period.
<i>periodV</i>	Specifies the value of the clock period.
<i>periodTu</i>	Specifies the time unit for the clock period.
<i>dutyCycle</i>	Specifies the duty cycle of the clock object.
<i>startTime</i>	Specifies the time of the first clock edge.
<i>startTimeV</i>	Specifies the value of the time of the first clock edge.
<i>startTimeTu</i>	Specifies the time unit for the time of the first clock edge.
<i>posedgeFirst</i>	Specifies if the first edge will be a posedge or a negedge.

The following functions are available to set/get properties of the clock object:

```
const char* name() const;
```

Returns the name of the clock.

```
bool is_master() const;
```

Returns true for clock sources and false for divided (class scml\_divided\_clock) or gated clocks (class scml\_clock\_gate).

```
sc_core::sc_time get_period() const;
void set_period(const sc_core::sc_time &t);
```

Gets and sets the clock period. Setting the period is only valid for clock sources. It can be set (changed) at any time. The new value will be returned by `get_period()` after the next update phase of the SystemC kernel.

```
double get_duty_cycle() const;
void set_duty_cycle(double d);
```

Gets and sets the duty cycle. The value provided to `set_duty_cycle()` must be larger 0.0 and smaller 1.0.

```
sc_core::sc_time& get_start_time() const;
void set_start_time(const sc_core::sc_time& t);
```

Gets or sets the start time of the clock. Immediately after changing the period or enabling the clock, it returns the start time of the next period, that is the first period following the new properties. Setting the start time is only valid for clock sources. It must only be called before the initialization phase of the SystemC kernel.

```
bool get_posedge_first() const;
void set_posedge_first(bool posedgeFirst);
```

Gets and sets the `posedge_first` property of the clock. It must only be called before the initialization phase of the SystemC kernel.

```
double get_period_multiplier() const;  
void set_period_multiplier(double m);
```

Gets and sets the period multiplier of the clock. Setting the period multiplier is only valid for divided clocks. It can be set at any time. The new value will be returned by `get_period_multiplier()` after the next update phase of the SystemC kernel.

```
void enable();  
void disable();
```

Enables (that is, makes active) or disables (that is, makes inactive) the clock. When an `scml_clock` is disabled, the output is 0 for a normal clock, and 1 for an inverted clock (starting with a negative edge, that is, `get_posedge_first()` returning `false`).

```
bool disabled();
```

Tests whether the clock is disabled.

```
bool running();
```

Tests whether the clock is running. For a clock source, this is equivalent to `!disabled()`. For a divided clock or gated clock, it also considers the running state of the clock sources. That is, a divided clock or gated clock is running if it is enabled and the clock source is running.

The `scml_clock` provides a notification mechanism for changes of the parameters `period` and `enabled`. An observer of clock parameter changes must inherit the base class `scml_clock_observer` and implement the method `handle_clock_parameters_updated()`. When the period of a clock is changed, or a clock is enabled or disabled, the new parameters become active with the next update phase. During the update, the clock calls `handle_clock_parameters_updated()` for all registered observers. If multiple parameters of the clock are changed within the same cycle, then `handle_clock_parameters_updated()` is only called once.

```
void register_observer(scml_clock_observer* o);  
void unregister_observer(scml_clock_observer* o);
```

Registers or unregisters a clock observer with the clock.

The `scml_clock` is the basic modeling object for the clocked modeling style of the SCML fast timed (FT) modeling style, see “[Introduction to SCML FT Modeling](#)” on page 237. It provides an ease of use mechanism to register callbacks with clock boundaries. A *clock boundary* is defined as the beginning of a period. For a normal clock, it corresponds to the positive edge; for an inverted clock (`get_posedge_first()` returning `false`) to a negative edge. Within this clock callback API, the clock boundary is called a *clock tick*.

Clock tick callbacks are called from the context of an SystemC method, that is internal to the clock. The call happens during the evaluation phase of the first delta cycle of the SystemC time, that corresponds to the clock edge.



SystemC processes that are sensitive to the clock edge event, will only be activated in the second delta cycle.

The main user interface of the clock tick mechanism is provided by the class `clocked_module`, see “[scml2::clocked\\_module](#)” on page 167. Within an `scml_clock`, the following functions are related to clock ticks:

```
bool check_at_tick() const;
```

Checks whether there is a clock tick at the current SystemC time. The result is independent from the current delta cycle. That means, `check_at_tick()` returns `true` already in the first delta cycle that corresponds to a clock tick, although processes that are sensitive to the clock edge event are only activated in the second delta cycle.

```
sc_core::sc_time get_next_edge_offset(bool pos_edge) const;
```

Returns the offset (as SystemC time) to the next positive or negative edge of the clock, depending on the value of the argument `pos_edge`.

```
unsigned long long get_tick_count() const;
```

Queries the tick count of the clock. This corresponds to the number of clock ticks, since the start of the simulation. The return value takes all phases when the clock was disabled, as well as, all changes of the clock period into account.

This function has been optimized for speed when called in a synchronous way, that is from a tick callback. An unsynchronized call is more expensive, but only the first time for a given SystemC time, due to internal caching of the result.

```
sc_dt::uint64 get_clock_count() const;
```

Returns the same value as `get_tick_count()`. This function is provided for backward compatibility.

```
unsigned long long get_tick_count(const sc_core::sc_time& delay) const;
```

Returns the tick count of the clock for the future SystemC time `sc_time_stamp() + delay`. For example, if called with `delay=SC_ZERO_TIME`, it returns the current tick count, which is equivalent to `get_tick_count()`.

```
sc_core::sc_time get_tick_time(long long clock_ticks_to_skip) const;
```

Gets the SystemC time for the clock tick that happens after `clock_ticks_to_skip` ticks from now.

For example if called with `clock_ticks_to_skip=0`, it returns the time of the next tick.

```
void get_next_tick_data(const sc_core::sc_time& delay, unsigned long long& count,
sc_core::sc_time& time) const;
```

Gets the clock count and SystemC time for the next clock tick after a given future SystemC time. The future point in time is given by the `sc_time` argument `delay`, which is interpreted relative to the current SystemC time `sc_time_stamp()`. If called synchronized (the future point in time corresponds to a clock tick), it returns the data for the next following clock tick.

Several events are available for clock objects. They can be accessed using the following functions:

```
const sc_event& value_changed_event() const;
const sc_event& posedge_event() const;
const sc_event& negedge_event() const;

bool event();
bool posedge() const;
bool negedge() const;
```

Boolean functions to test whether a certain event occurred.

For tracing purposes, a reference to the current value can be obtained:

```
const bool & get_data_ref() const;
```



The `scml_clock` cannot be optimized when tracing is enabled. Enabling tracing will disable clock optimizations.

## 4.2.2 scml\_divided\_clock

A divided clock is a special version of an `scml_clock` that is derived from another clock by multiplying the start time and/or the period with specified integer factors. In case both multipliers are 1, a local mirror of the clock is obtained. An advantage of such a mirror of a clock is that it can be enabled and disabled locally.

Objects of type `scml_divided_clock` can be constructed using one of the following constructors:

```
scml_divided_clock(const char * name,
                    sc_in<bool> & clk,
                    unsigned int periodMultiplier=1,
                    unsigned int startMultiplier=0);

scml_divided_clock(const char * name,
                    scml_clock_if & clk,
                    unsigned int periodMultiplier=1,
                    unsigned int startMultiplier=0);

scml_divided_clock(const char * name,
                    unsigned int periodMultiplier=1,
                    unsigned int startMultiplier=0);
```

where:

<code>name</code>	Specifies a name for the clock object.
<code>clk</code>	Specifies the clock from which this object is derived.
<code>periodMultiplier</code>	Specifies the factor by which the period is multiplied.
<code>startMultiplier</code>	Specifies the factor by which the start time is multiplied.

The default values are such that a clone of the incoming clock is obtained.

The following functions are provided to connect the divided clock to its input clock.

```
void bind(sc_in<bool> & );
void bind(scml_clock_if & );
void operator()(sc_in<bool> & );
void operator()(scml_clock_if & );
```

In addition to the API of an `scml_clock`, the following functions are available to set properties:

```
void set_divider(unsigned int div);
unsigned int get_divider() const;
```

Changes the clock period of a divided clock in multiples of the original clock period. The *original clock period* is the parent's clock period multiplied by the period multiplier constructor argument. For example, if the parent clock period is  $p$  and the divided clock period multiplier constructor argument is 2, the original period is two times  $p$ .

`set_divider(4)` indicates that the new period is four times the original period, that is  $8p$ .

The majority of methods of an `scml_divided_clock` behave the same as with an `scml_clock`, with the following exceptions:

```
bool is_master() const;
```

Always returns `false`.

```
bool disabled();  
Tests whether the clock is disabled by the disable()/enable().
```

```
bool running();  
Tests whether the clock is running. This is the case if it is not disabled, and the clock source is running.
```

```
double get_duty_cycle() const;  
void set_duty_cycle(double d);
```

Gets and sets the duty cycle of the divided clock. By default, a divided clock inherits the duty cycle from its parent clock. Any change of the duty cycle of the clock source will also become visible on the divided clock. This behavior changes after the duty cycle of the divided clock has been set explicitly by a call `set_duty_cycle()`. After such a call, the divided clock maintains its own value for the duty cycle and the value of the clock source is no longer used. The value provided to `set_duty_cycle()` must be larger than 0.0 and smaller than 1.0.

The following functions to set properties are not supported by a divided clock and must not be called:

- `set_start_time(const sc_core::sc_time&)`
- `set_period(const sc_core::sc_time&)`
- `set_posedge_first(bool)`

### 4.2.3 Dynamic Clock Parameter Change and Reset

This section describes how dynamic clock parameter changes and dynamic reset can be handled in the FTM coding style.

With FTM coding style, it can be difficult to assure correct model behavior on dynamic clock parameter changes like dynamically changing the clock period or disabling and enabling it. The main root cause for the difficulties is *temporal decoupling*, but also pending transactions can be hard to handle.

To enable easier handling of dynamic clock parameter changes, class `scml_clock_parameter_change_control_if` provides an extension to the `scml_clock` API which allows driving each model into a state where clock parameter changes can safely be done before the change is actually executed. The solution is based on a handshaking mechanism which blocks the execution of the clock parameter change request until all registered clients signaled readiness.

#### 4.2.3.1 Flow

The general dynamic clock parameter change flow is the following:

1. On clock-parameter-change-request, the clock notifies all registered clients about a new clock-parameter-change-request.
2. The clients prepare for clock parameter change. Usually this means returning from time traveling and waiting for pending transactions. A model should no longer initiate any transactions or write to external pins after it received a clock-parameter-change-request-notification.

When the client reaches the state where it can safely handle clock parameter changes, it signals readiness to the clock and waits for the actual clock-parameter-changed-notification. A client can also immediately signal readiness.

3. When all clients signaled readiness, the actual clock parameter change is executed in the following update phase. The clock will send out a notification to all clock observers.
4. The clients continue running, usually waiting for the first clock tick after the notification.

### 4.2.3.2 Model State Requirements

Before signaling readiness for a clock parameter change, a model needs to meet certain requirements:

- The model should not be time travelling (temporal decoupling).
- The model is not allowed to have any pending transactions. This rule enables much easier handling of clock parameter changes in network and target models.
- No SystemC process owned by the model is allowed to be suspended by a wait call in a different model. With FTM coding style this should only be the case for pending transactions, which anyway need to be handled.
- The model specific state has to allow clock parameter changes.



**Note** If all models in a virtual prototype follow these rules, pure target models usually need not take part in the handshaking mechanism. They should natively allow clock parameter changes as soon as all other models signal readiness.

### 4.2.3.3 Architecture

The `scml_clock_parameter_change_control_if` is an extension to the `scml_clock_if`. For details, see “[scml\\_clock](#)” on page 158. To support dynamic parameter changes, a clock needs to implement both interfaces. Usually this is done by creating a wrapper around an `scml_clock` which implements `scml_clock_parameter_change_control_if`.

The clock parameter change control needs to be bound to a `scml_clock` so that it can be found by the clients.

Client models need to implement both `scml_clock_parameter_change_control_if::client` and `scml_clock_observer` to take part in the dynamic clock parameter change handshaking mechanism.

Clients have to check if a connected clock has an implementation of `scml_clock_parameter_change_control_if` bound to it and register to it.

### 4.2.3.4 API

#### Binding

```
void scml2::bind_scml_clock_parameter_change_control(  
    scml_clock_if* clk, scml_clock_parameter_change_control_if* d)
```

Binds an instance of a `scml_clock_parameter_change_control_if` implementation `d` to clock `clk`. Only one instance can be bound at a time.

```
void scml2::unbind_scml_clock_parameter_change_control(scml_clock_if* clk)
```

Unbinds any `scml_clock_parameter_change_control_if` from clock `clk`.

#### Fetch Clock Parameter Control Pointer

```
scml_clock_parameter_change_control_if*  
scml2::get_scml_clock_parameter_change_control(scml_clock_if* clk)
```

Gets pointer to `scml_clock_parameter_change_control_if` bound to clock `clk`. Returns 0 if no `scml_clock_parameter_change_control_if` is bound to this clock.

```
scml_clock_parameter_change_control_if* scml2::get_scml_clock_parameter_change_control(  
    sc_core::sc_in<bool>& clk_in_port)
```

Convenience function: Gets pointer to `scml_clock_parameter_change_control_if` bound to the clock signal connected to port `clk_in_port`. Returns 0 if no `scml_clock_parameter_change_control_if` is bound to this clock signal.

### **scml\_clock\_parameter\_change\_control\_if**

```
virtual void register_client(client* c)
```

Registers client `c` to clock parameter change control.

```
virtual void signal_ready_for_parameter_change(client* c)
```

To be called by client `c` to signal readiness to clock parameter change control.

### **scml\_clock\_parameter\_change\_control\_if::client**

```
virtual void handle_clock_parameter_change_request(  
    scml_clock_parameter_change_control_if*)
```

Called by clock parameter change control to notify the client a clock parameter change was requested.  
The client should start preparing for clock parameter change on this notification.

The requirements for dynamic reset are even stricter than for dynamic clock parameter change. Reset has to be executed in each model in a Virtual Prototype at exactly the same time to assure functional correctness.

To achieve this, the proposal is to have a long enough reset pulse that every model in a Virtual Prototype has enough time to drive itself into a state where it can safely execute the reset. There is no handshaking involved, when a model is unable to reach this state in time it shall terminate with an error message.

#### **4.2.3.5 Flow**

1. On reset assert, all models stop regular execution as soon as possible and prepare for reset. When a state is reached where reset can safely be executed, they shall block until reset de-assert.

A model should no longer initiate any transactions or write to external pins after reset assert.

2. On reset de-assert, the actual reset is executed.

If a model is still in a state where reset cannot be executed, it shall throw an error and terminate execution. The simulation has to be restarted with a longer reset pulse.

#### **4.2.3.6 Model State Requirements**

The model state where reset can safely be executed is defined by the following rules:

- The model is not time travelling.
- The model does not have any pending transactions.
- No SystemC process owned by the model is suspended by a wait call in a different model.
- The model specific state allows executing reset.



**Note** If all models in a Virtual Prototype follow these rules, pure target models usually do not need extra adaptation. They natively meet the requirements.

#### **4.2.3.7 Architecture**

No extra interfaces are involved in the dynamic reset flow. The only requirement is a reset generator model which supports adjusting the reset pulse length.

## 4.3 Modeling Objects for Clocks (Clock Objects)

- [scml\\_clock\\_gate](#)
- [scml\\_clock\\_counter](#)

### 4.3.1 scml\_clock\_gate

`scml_clock_gate` is a module which takes a clock and an enable signal as input and produces a gated clock as output. If the enable signal is `true`, the output clock equals the input clock. If the enable signal is `false`, the output clock is disabled.

A clock gate has a clock and an enable input port:

```
sc_in<bool> clk;  
sc_in<bool> en;
```

The following constructor is available:

```
scml_clock_gate(sc_module_name name);
```

where *name* specifies a name for the clock object.

This constructor is explicit.

The `scml_clock_gate` modeling object implements the interface `scml_clock_if`. It can, therefore, be used like an `sc_clock` or an `scml_clock` object. For an API reference, see “[scml\\_clock](#)” on page 158.

The majority of methods of an `scml_clock_gate` behaves the same as with an `scml_clock`, with the following exceptions:

```
bool is_master() const;
```

Always returns `false`.

```
bool disabled();
```

Tests whether the clock is disabled by the `en` input port.

```
bool running();
```

Tests whether the clock is running. This is the case if it is enabled by the `en` input port, and the clock source is running.

The following functions are not supported by a clock gate and must not be called:

- `enable()`
- `disable()`
- `set_duty_cycle(double)`
- `set_start_time(const sc_core::sc_time&)`
- `set_period(const sc_core::sc_time&)`
- `set_posedge_first(bool)`
- `set_period_multiplier(double)`

### 4.3.2 scml\_clock\_counter

An object of type `scml_clock_counter` needs to be attached to a clock. It is used to get the number of clock cycles that have happened in a certain period. The value of the counter is incremented at every clock cycle. Its initial value is 0.

The following type definitions are available:

```
typedef sc_dt::uint64 data_type;
```

Objects of type `scml_clock_counter` can be constructed using one of the following constructors:

```
scml_clock_counter(const char * name,  
                    scml_clock_if & clk);  
scml_clock_counter(const char * name);
```

where:

<code>name</code>	Specifies a name for the clock object.
<code>clk</code>	Specifies the clock that should be used to determine this counter value.

The single-argument constructor is explicit.

The counter can be manipulated by means of the following sets of functions:

```
data_type get_count() const;  
void set_count(data_type var);
```

Gets and sets the counter.

```
data_type read() const;  
void write(const data_type var);
```

Read and write function.

```
operator const data_type() const;  
data_type operator=(const data_type var);
```

Accesses the `scml_clock_counter` as a variable.

The following functions are provided to connect the clock counter to its input clock.

```
void bind(scml_clock_if &);  
void operator()(scml_clock_if &);
```

## 4.4 Base Classes

- [scml2::clocked\\_module](#)

### 4.4.1 scml2::clocked\_module

The class `clocked_module` is the lower level user API for SCML clock tick callbacks. The intended use is that a module can receive clock tick callbacks by inheriting from the `scml2::clocked_module` class and by implementing the virtual method `handle_clock_tick`.

A `clocked_module` has a fixed association with one SCML clock interface, it is not allowed to change the clock used by the `clocked_module` at runtime.

A clock tick callback can be requested by the method `request_clock_tick` callback. If there is already a callback request pending, the earlier one of the two will be maintained. It is possible to re-trigger or cancel a pending request.

Objects of type `scml2::clocked_module` can be constructed using one of the following constructors:

```
clocked_module(scml_clock_if* clock=0);  
clocked_module(sc_core::sc_in<bool>& p);
```

where:

<i>clock</i>	Specifies the SCML clock object that is associated with the clocked module.
<i>p</i>	Specifies the input port through which the clocked module is bound to its associated SCML clock.

Clocked modules can be constructed and destructed at any time during a simulation run. If a clocked module is destructed and there is still a pending request for a clock tick callback for that object, it will automatically be canceled.

The following functions are provided to connect the clocked module to an SCML clock.

```
void set_clock(scml_clock_if* clock);
```

Associates the `clocked_module` with the provided SCML clock.

```
void set_clock(sc_core::sc_in<bool>& p);
```

Associates the `clocked_module` with the SCML clock, which is bound to the provided input port.

```
scml_clock_if* get_clock() const;
```

Returns the associated SCML clock.

```
void request_clock_trigger(long long clock_ticks_to_skip);
```

Schedules a callback (that is, call to function `handle_clock_tick`) for a future clock tick. The argument `clock_ticks_to_skip` defines the number of ticks that shall be skipped from now. This function can be called at any SystemC time.

If called while a previous clock tick callback request is still pending, the earlier one of the two will be maintained. That is, the new request is ignored if it targets the same or a later clock tick. If it targets an earlier clock tick, then the previous request is canceled, and the request is scheduled for the new earlier tick.

If a clock trigger was requested, then the requested/scheduled clock tick can be retrieved by a call to method `get_scheduled_clock_tick()`.

For example, after a call of `request_clock_trigger(0)`, the clock tick count returned by `get_scheduled_clock_tick()` will be one larger than the current clock tick count of the associated SCML clock.

```
void request_clock_trigger(const sc_core::sc_time& delay);
```

Schedules a callback (that is, call to function `handle_clock_tick`) for a future clock tick after the time defined by the argument `delay`. The argument `delay` is a SystemC time that is interpreted relative to the current SystemC time `sc_time_stamp()`. This method will usually be used by TLM2 models with temporal decoupling, in order to synchronize to the next clock tick after the given local time argument.

If called while a previous clock tick callback request is still pending, the earlier one of the two will be maintained. That is, the new request is ignored if it targets the same or a later clock tick. If it targets an earlier clock tick, then the previous request is canceled and the request is scheduled for the new earlier tick.

```
virtual void handle_clock_tick()=0;
```

When a requested clock tick callback expires, the SCML clock calls the method `handle_clock_tick()`. The call happens from within the context of an `SC_METHOD` that is owned by

the clock object. This is an abstract method that shall be implemented by the user in the derived class in order to provide the functionality that shall be executed on a clock tick callback.



**Caution** You must not call `next_trigger()` from within a clock tick callback, since this will break the SCML clock mechanism and will result in a fatal misbehavior of the clock system.

```
bool is_clock_trigger_requested() const;
```

Returns true if a clock tick callback was requested and is still pending.

```
void cancel_clock_trigger();
```

Cancels a pending clock tick callback request. It does nothing if no clock tick callback request is currently pending. During the processing of a clock tick (that is within a `handle_clock_tick()` call), it is not permitted to cancel a clock tick callback request that was scheduled for the currently processed clock tick.

```
unsigned long long get_scheduled_clock_tick() const;
```

In case of a pending clock tick callback, this method returns the clock tick for which the tick callback was requested/scheduled. If no clock tick callback is pending, then the returned value is undefined.

## 4.5 Modeling Objects for Base Classes (Modeling Objects)

- [scml2::clocked\\_timer](#)

### 4.5.1 scml2::clocked\_timer

This is a modeling object that provides a timer callback mechanism based on an SCML clock. The `clocked_timer` object is based on the `clocked_module` object and is similar to the `clocked_callback` object. The timer can be started to expire after a number of clock ticks. When it expires, it calls the callback that is registered with the object. The timer can be configured to expire once, multiple times or to run forever. It is possible to stop and resume the timer.

The following type definition is available:

```
enum eState { eS_Idle=0, eS_Running, eS_Stopped };
```

Objects of type `clocked_timer` can be constructed using the following constructor:

```
clocked_timer(const std::string& name, scml_clock_if* clock=0);
```

where:

<code>name</code>	Specifies a name for the clocked timer object.
<code>clock</code>	Specifies the SCML clock object that is associated with the clocked timer object.

Clocked timer objects can be constructed and destructed at any time during a simulation run

The following functions are provided to connect the clocked timer to an SCML clock.

```
void set_clock(scml_clock_if* clock);
```

Associates the `clocked_timer` with the provided SCML clock.

```
void set_clock(sc_core::sc_in<bool>& p);
```

Associates the `clocked_timer` with the SCML clock, which is bound to the provided input port.

```
scml_clock_if* get_clock() const;
```

Returns the associated SCML clock.

Registering a callback for alarm notifications is typically done using the following macro:

```
m_clk_timer.set_callback(SCML2_CLOCKED_CALLBACK(clock_cb));  
void start(long long ticks_per_period, long long shot_count=-1);
```

Starts the timer. It will expire after `ticks_per_period` ticks. For example, a timer started with `start(1)` will expire at the next clock tick. With `ticks_per_period=0`, the timer does not start.

The argument `shot_count` specifies how many alarm callbacks will be triggered by the timer with a distance of `ticks_per_period` ticks. With `shot_count=1`, the timer expires once. With `shot_count=-1` (default), the timer runs forever or until it is stopped by calling `stop()`. With `shot_count=0` the timer does not start.

If `start()` is called while the timer is running, it will first be stopped before it is restarted. If a running timer is started with `ticks_per_period=0` or `shot_count=0`, the timer is stopped and left in the idle state, so that it is not possible to resume it at a later time.

```
void start(const sc_core::sc_time& period, long long shot_count=-1);
```

Starts the timer. This is a convenience method comparable to `start(long long, long long)`. The period in clock ticks is derived by the `sc_time` argument `period`, by rounding up to the next higher multiple of the current clock period. For more details, see the description of the method, `start(long long, long long)`

```
void stop();
```

Stops a running timer. The timer is left in a state that allows it to be resumed at a later time. It has no effect, if called on an idle or already stopped timer.

```
bool resume_stopped();
```

Resume a stopped timer. Only timers that have been stopped by `stop()` can be resumed. In that case, the timer continues to run with the status at which it was stopped. If the timer was stopped in the middle of a period, than the timer finishes the already started period. For example, a timer started at clock tick 100 with `period=10` and `shot_count=3`, stopped at time 115 (after one alarm and another half period), and resumed at time 200, will expire at the times 205 and 215.

It returns `true` on success. It returns `false`, if it is called on an idle or already running timer.

```
unsigned long long get_counter_value();
```

Returns the value of the internal counter of the timer. It counts the number of clock ticks since the last start or restart.

```
bool get_counter_value(const sc_core::sc_time delay, unsigned long long& value);
```

Tries to retrieve the value of the internal counter of the timer for the future time `sc_time_stamp() + DELAY`. For example, if called with `DELAY=SC_ZERO_TIME`, it delivers the same value as `get_counter_value()`. The value can only be delivered if the provided delay is shorter than the duration until the timer fires next, in which case the function returns `true` and writes the counter value into the reference argument `VALUE`. If delay exceeds the next fire time, the function returns `false` and does not write into `VALUE`. The function will also fail and return `false` in case it is called from a timer callback.

```
eState get_state();
```

Returns the state of the timer. In order to avoid dependency on scheduling order, the state at the beginning of the current SystemC time is returned. A change to the state will only become visible after

the SystemC time advances. If called from the timer callback, it returns the state before the current clock cycle.

```
bool is_active();
```

Returns true, if the timer is running. In order to avoid dependency on scheduling order, the activity state at the beginning of the current SystemC time is returned. A change to the state will only become visible after the SystemC time advances. If called from the timer callback, it returns the state before the current clock cycle.

```
long long get_remaining_shot_count();
```

Returns the number of remaining shots. In order to avoid dependency on scheduling order, the activity state at the beginning of the current SystemC time is returned. A change to the state will only become visible after the SystemC time advances. If called from the timer callback, it returns the value before the current clock cycle.

## 4.6 Convenience Classes

- [scml2::clocked\\_callback](#)
- [scml2::clocked\\_event](#)

### 4.6.1 scml2::clocked\_callback

The class `scml2::clocked_callback` is a convenience class that forwards a clock tick callback to any member function of a module without the need to inherit from the `clocked_module` base class. This makes it easier to have multiple clock callbacks within a single module. Since the callback is received via a `clocked_callback`, it involves an internal method redirection which is slightly slower than a callback received via inheriting the module directly from `clocked_module`.

Objects of type `clocked_callback` can be constructed using the following constructor:

```
clocked_callback(const std::string& name, scml_clock_if* clock=0)
```

where:

<code>name</code>	Specifies a name for the clocked callback object.
<code>clock</code>	Specifies the SCML clock object that is associated with the clocked callback object.

Clocked callbacks can be constructed and destructed at any time during a simulation run. If a clocked callback is destructed and there is still a pending callback request for that object, it will automatically be canceled.

The following functions are provided to connect the clocked callback to an SCML clock.

```
void set_clock(scml_clock_if* clock);
```

Associates the `clocked_callback` with the provided SCML clock.

```
void set_clock(sc_core::sc_in<bool>& p);
```

Associates the `clocked_callback` with the SCML clock, which is bound to the provided input port.

```
scml_clock_if* get_clock() const;
```

Returns the associated SCML clock.

Registering a method to be called as clock callback is typically done using the following macro:

```
m_clk_cb.set_callback(SCML2_CLOCKED_CALLBACK(clock_cb));
```

```
void request_trigger(long long clock_ticks_to_skip);
```

Schedules a callback for a future clock tick. The argument `clock_ticks_to_skip` defines the number of ticks that shall be skipped from now. This function can be called at any SystemC time.

If called while a previous callback request is still pending, the earlier one of the two will be maintained. That is, the new request is ignored if it targets the same or a later clock tick. If it targets an earlier clock tick, then the previous request is canceled, and the request is scheduled for the new earlier tick.

If a callback was requested, then the requested/scheduled clock tick can be retrieved by a call to method `get_scheduled_clock_tick()`.

For example, after a call of `request_trigger(0)`, the clock tick count returned by `get_scheduled_clock_tick()` will be one larger than the current clock tick count of the associated SCML clock.

```
void request_trigger(const sc_core::sc_time& delay);
```

Schedules a callback for a future clock tick after the time defined by the argument `delay`. The argument `delay` is a SystemC time that is interpreted relative to the current SystemC time `sc_time_stamp()`. This method will usually be used by TLM2 models with temporal decoupling in order to synchronize to the next clock tick after the given local time argument.

If called while a previous callback request is still pending, the earlier one of the two will be maintained. That is, the new request is ignored if it targets the same or a later clock tick. If it targets an earlier clock tick, then the previous request is canceled, and the request is scheduled for the new earlier tick.

```
bool is_trigger_requested() const;
```

Returns `true` if a callback was requested and is still pending.

```
void cancel_trigger();
```

Cancels a pending callback request. It does nothing, if no callback request is currently pending. During the processing of a callback, it is not permitted to cancel a callback request that was scheduled for the currently processed clock tick.

In certain cases, it can also be useful to create a one-shot callback. Here, the idea is to have a class method to be called once on a clock tick. For this purpose, a set of global functions are provided that create a `clocked_callback` and trigger a request after a certain number of ticks or a certain SystemC delay. This can be used for example, to trigger a certain behavior after a register access.

```
template <class MOD_TYPE, typename FUNCPTR_TYPE>
void request_clocked_method_callback(
    scml_clock_if* clock, long long clock_ticks_to_skip,
    MOD_TYPE* mod, FUNCPTR_TYPE func
);
template <class MOD_TYPE, typename FUNCPTR_TYPE>
void request_clocked_method_callback(
    sc_core::sc_in<bool>& clock_port, long long clock_ticks_to_skip,
    MOD_TYPE* mod, FUNCPTR_TYPE func
);
template <class MOD_TYPE, typename FUNCPTR_TYPE>
void request_clocked_method_callback(
    scml_clock_if* clock, const sc_core::sc_time& delay,
    MOD_TYPE* mod, FUNCPTR_TYPE func
);
template <class MOD_TYPE, typename FUNCPTR_TYPE>
void request_clocked_method_callback(
```

```
sc_core::sc_in<bool>& clock_port, const sc_core::sc_time& delay,  
MOD_TYPE* mod, FUNCPTR_TYPE func
```

#### 4.6.2 scml2::clocked\_event

An `scml2::clocked_event` is a convenience class that allows a SystemC method or thread to wait until a certain clock tick happens. The `clocked_event` object has the same set of APIs as a `clocked_module`, but also provides a `wait()` API, so that a thread can wait for a number of clock cycles or for the first clock tick after a certain delay. It has a `wait_for_trigger()` API that halts execution, until a certain clock tick specified by `requested_trigger` (similar to the `clocked_module`) happens. This object also has a `next_trigger()` API to mimic dynamic sensitivity of the methods. In this case, the trigger will be aligned with clock edges.

Objects of type `clocked_event` can be constructed using one of the following constructors:

```
clocked_event(const std::string& name, scml_clock_if* clock=0);  
clocked_event(const std::string& name, sc_core::sc_in<bool>& p);
```

where:

<code>name</code>	Specifies a name for the clocked event object.
<code>clock</code>	Specifies the SCML clock object that is associated with the clocked event object.
<code>p</code>	Specifies the input port through which the clocked event is bound to its associated SCML clock.

Clocked events can be constructed and destructed at any time during a simulation run. If a clocked callback is destructed and there is still a trigger pending, it will automatically be canceled. If a thread or a method is waiting for a clock tick, it will never wake up or be activated again.

The following functions are provided to connect the clocked event to an SCML clock.

```
void set_clock(scml_clock_if* clock);
```

Associates the `clocked_event` with the provided SCML clock.

```
void set_clock(sc_core::sc_in<bool>& p);
```

Associates the `clocked_event` with the SCML clock, which is bound to the provided input port.

```
scml_clock_if* get_clock() const;
```

Returns the associated SCML clock.

```
void request_trigger(long long clock_ticks_to_skip);
```

Schedules a trigger for a future clock tick, which will wake up a SystemC thread (which is called `wait()`), or which re-activates a SystemC methods (which is called `next_trigger()`). The argument `clock_ticks_to_skip` defines the number of ticks that shall be skipped from now. It can be called at any SystemC time.

If called while a previous trigger request is still pending, the earlier one of the two will be maintained. That is, the new request is ignored if it targets the same or a later clock tick. If it targets an earlier clock tick, then the previous request is canceled, and the request is scheduled for the new earlier tick.

If a trigger was requested, then the requested/scheduled clock tick can be retrieved by a call to method `get_scheduled_clock_tick()`.

For example, after a call of `request_clock_trigger(0)`, the clock tick count returned by `get_scheduled_clock_tick()` will be one larger than the current clock tick count of the associated SCML clock.

```
void request_trigger(sc_core::sc_time delay);
```

Schedules a trigger for a future clock tick after the time defined by the argument `delay`. The argument `delay` is a SystemC time that is interpreted relative to the current SystemC time `sc_time_stamp()`. This method will usually be used by TLM2 models with temporal decoupling, in order to synchronize to the next clock tick after the given local time argument.

If called while a previous trigger request is still pending, the earlier one of the two will be maintained. That is, the new request is ignored if it targets the same or a later clock tick. If it targets an earlier clock tick, then the previous request is canceled, and the request is scheduled for the new earlier tick.

```
bool is_trigger_requested() const;
```

Returns true if a trigger was requested and is still pending.

```
void cancel_trigger();
```

Cancels a pending trigger request. It does nothing if no trigger is currently pending. During the processing of a clock tick (that is from a callback, or a `handle_clock_tick()` call), it is not permitted to cancel a trigger request that was scheduled for the currently processed clock tick.

```
void waitfor_trigger();
```

Blocks the calling SystemC thread, until a clocked trigger happens. The clocked trigger is requested by a call to the method `request_trigger(ticks/delay)`. The blocked thread wakes up when the requested clock tick happens. It must only be called from the context of a SystemC thread.

```
void wait(long long clock_ticks_to_skip);
```

Blocks the calling SystemC thread, until the specified clock tick happened. The clock tick is specified by the argument `clock_ticks_to_skip`, which defines the number of ticks that shall be skipped from now. It can be called at any SystemC time, but only from the context of a SystemC thread.

It is possible to wake up the thread at an earlier time by calling `request_trigger(ticks/delay)` with an argument that specifies an earlier clock tick. In order to wake up the thread at a later time, it is first necessary to cancel the pending trigger with a call to `cancel_trigger()`.

```
void wait(sc_core::sc_time delay);
```

Blocks the calling SystemC thread, until the specified clock tick happened. The clock tick is specified by the argument `delay`, which is a SystemC time that is interpreted relative to the current SystemC time `sc_time_stamp()`. This method will usually be used by TLM2 models with temporal decoupling in order to synchronize to the next clock tick after the given local time argument. It can be called at any SystemC time, but only from the context of a SystemC thread.

It is possible to wake up the thread at an earlier time by calling `request_trigger(ticks/delay)` with an argument that specifies an earlier clock tick. In order to wake up the thread at a later time, it is first necessary to cancel the pending trigger with a call to `cancel_trigger()`.

```
void next_trigger();
```

Temporarily overrides the static sensitivity list of the calling SystemC method. The method will be reactivated when a certain clock tick happens. With this version of the `next_trigger()` function, the triggering clock tick is left undefined. It must be scheduled by calling `request_trigger(ticks/delay)`.

```
void next_trigger(long long clock_ticks_to_skip);
```

Temporary overrides the static sensitivity list of the calling SystemC method. The method will be reactivated when the specified clock tick happens. The clock tick is specified by the argument `clock_ticks_to_skip`, which defines the number of ticks that shall be skipped from now. It can be called at any SystemC time, but only from the context of a SystemC method.

It is possible to reactivate the method at an earlier time by calling `request_trigger(ticks/delay)` with an argument that specifies an earlier clock tick. In order to wake up the method at a later time, it is first necessary to cancel the pending trigger with a call to `cancel_trigger()`.

```
void next_trigger(sc_core::sc_time delay);
```

Temporary overrides the static sensitivity list of the calling SystemC method. The method will be reactivated when the specified clock tick happens. The clock tick is specified by the argument `delay`, which is a SystemC time that is interpreted relative to the current SystemC time `sc_time_stamp()`. This method will usually be used by TLM2 models with temporal decoupling in order to synchronize to the next clock tick after the given local time argument. It can be called at any SystemC time, but only from the context of a SystemC method.

It is possible to reactivate the method at an earlier time by calling `request_trigger(ticks/delay)` with an argument that specifies an earlier clock tick. In order to wake up the method at a later time, it is first necessary to cancel the pending trigger with a call to `cancel_trigger()`.

## 4.7 Modeling Objects for Convenience Classes (Convenience Objects)

- [scml2::clocked\\_peq\\_container](#)
- [scml2::clocked\\_peq](#)

### 4.7.1 scml2::clocked\_peq\_container

A typical problem in TLM2 FT models using the non-blocking APIs is that it can be required to manage multiple outstanding transactions, possibly coming with different timing annotations from different initiators. The `clocked_peq_container` is a convenience object to help in this case. It can be used to buffer payloads arriving in the model. Payloads are pushed into the container tagged with an arrival stamp (clock tick counts to be precise). The container behaves like a list of payload that is sorted by the arrival stamp. There is an API to iterate over the payloads in the buffer, the iterator will only provide with those payloads for which the arrival time is in the past (compared to the current SystemC time when iterating). In this way, this object helps to comply with the basic FT clocked modeling rules. Because the timestamp is stored based on tick counts, it is possible to adjust arrival times when clock parameters change (period/enable/disable).

The API of the container conforms to the STL and provides iteration into the forward direction of the list (from old to new payloads). There is a constant and a non-constant version of the iterator. The constant iterator provides a faster iteration mechanism. The non-constant iterator allows removing the payload it is pointing to from the list.

The `scml2::clocked_peq_container` class is templated with the underlying type of the payload:

```
template <typename PAYLOAD> class clocked_peq_container;
```

The following type definitions for iterators are available:

```
typedef clocked_peq_container_iterator<PAYLOAD> iterator;
typedef clocked_peq_container_const_iterator<PAYLOAD> const_iterator;
```

Objects of type `clocked_peq_container` can be constructed using one of the following constructors:

```
clocked_peq_container(scml_clock_if* clock=0);
```

```
clocked_peq_container(sc_core::sc_in<bool>& p);
```

where:

<i>clock</i>	Specifies the SCML clock object that is associated with the clocked PEQ container object.
<i>p</i>	Specifies the input port through which the clocked PEQ container is bound to its associated SCML clock.

The following functions are provided to connect the clocked PEQ container to an SCML clock.

```
void set_clock(scml_clock_if* clock);
```

Associates the `clocked_peq_container` with the provided SCML clock.

```
void set_clock(sc_core::sc_in<bool>& p);
```

Associates the `clocked_peq_container` with the SCML clock, which is bound to the provided input port.

```
scml_clock_if* get_clock() const;
```

Returns the associated SCML clock.

```
bool notify(PAYLOAD* payload, const sc_core::sc_time& arrival_local_time);
```

Pushes a payload into the container that was received by a temporal decoupled transport at local time `arrival_local_time`. The local time argument is given as SystemC time. It is converted to the clock tick count by rounding down to the last clock tick.

```
bool notify(
    PAYLOAD* payload, const sc_core::sc_time& arrival_local_time,
    long long delay_ticks
);
```

Pushes a payload into the container that was received by a temporal decoupled transport. The local time is defined by two arguments, the SystemC time `arrival_local_time` and the additional number of clock ticks `delay_ticks`. The internal arrival clock tick count is calculated by first converting `arrival_local_time` to the clock tick count by rounding down to the last clock tick, and then incrementing it by `delay_ticks`.

```
bool notify(void* payload, unsigned long long clock_ticks_to_skip);
```

Pushes a payload into the container that was received by a temporal decoupled transport at the local time that is `clock_ticks_to_skip` clock ticks in the future.

```
bool notifyAt(void* payload, unsigned long long arrival_tick_count);
```

Pushes a payload into the container that was received by a temporal decoupled transport at the local time represented by the clock tick count `arrival_tick_count`.

```
bool is_empty() const;
```

Returns true if the container is empty. This only considers visible payload (`timestamp < get_clock()->get_tick_count()`).

```
const_iterator begin() const;
```

Returns a constant iterator that points to the beginning of the internal sorted list. This only considers visible payload (`timestamp < get_clock()->get_tick_count()`).

```
iterator begin();
```

Returns a non-constant iterator that points to the beginning of the internal sorted list. This only considers visible payload (`timestamp < get_clock()->get_tick_count()`).

```
const_iterator end() const;
```

Returns a constant iterator that points to the end of the internal sorted list. This only considers visible payload (`timestamp < get_clock() -> get_tick_count()`).

```
iterator end();
```

Returns a non-constant iterator that points to the end of the internal sorted list. This only considers visible payload (`timestamp < get_clock() -> get_tick_count()`).

```
PAYLOAD* get_next();
```

Extracts the first element from the internal sorted list. It returns a null-pointer if the list is empty. This only considers visible payload (`timestamp < get_clock() -> get_tick_count()`).

```
void remove(const iterator& pos);
```

Removes the payload at the position, which is given by the iterator argument `pos`. The payload itself is not deleted.

This operation invalidates all the iterators, which are currently pointing into the container. Managing the iterators is the user's responsibility.

```
bool remove(PAYLOAD* payload);
```

Removes the payload given by the `payload` argument. The payload itself is not deleted. This only considers visible payload (`timestamp < get_clock() -> get_tick_count()`). It returns `true` if the payload was found and removed. `false` is returned if the payload was not found. If the same payload is contained multiple times, than only the first occurrence is removed.

If this operation succeeds, it invalidates all iterators which are currently pointing into the container. Managing the iterators is the user's responsibility.

```
bool has_more_events() const;
```

Returns `true` if more payload is available in the container. In contrast to the method `is_empty()`, it considers not only the visible, but all payload. This method is usually used, if all visible/past payload has been processed/removed, in order to check, if more processing has to be done at a future time.

```
unsigned long long get_next_event_arrival_tick() const;}
```

Returns the arrival clock tick of the next payload in the sorted list of the container. This method must not be called if `has_more_events()` returned `false`.



This method does not only consider the visible, but all payload. This method is usually used when all current processing has been done, in order to retrieve the time for which more processing shall be scheduled.

An object of class `clocked_peq_container_iterator` is a pointer into a `clocked_peq_container` that allows iterating in forward direction over the sorted list of the container. This non-constant version of iterators can be used to remove that payload from the container, which it is currently pointing to. Iterating is slightly slower than with the constant version `clocked_peq_container_const_iterator`. Iterators are usually generated by the `begin()` and `end()` method of a clocked PEQ container.

The `scml2::clocked_peq_container_iterator` class is templated with the underlying type of the payload:

```
template <typename PAYLOAD> class clocked_peq_container_iterator;
```

The API of the iterator conforms to the STL and provides the following methods:

```
PAYLOAD* operator*() const;
```

Returns the payload pointed to by the iterator.

```
PAYLOAD* operator->() const;
```

Returns the payload pointed to by the iterator.

```
self& operator++();
```

Moves the iterator forward to the next entry in the container. It returns an iterator, that points to the new position.

```
self operator++(int);
```

Moves the iterator forward to the next entry in the container. It returns an iterator, that points to the old position.

```
bool operator==(const self& x) const;
```

Compares the iterator to the other iterator *x*. It returns `true`, if both iterators are pointing to the same position.

```
bool operator!=(const self& x) const;
```

Compares the iterator to the other iterator *x*. It returns `true`, if both iterators are pointing to different positions.

An object of class `clocked_peq_container_const_iterator` is a pointer into a `clocked_peq_container` that allows iterating in forward direction over the sorted list of the container. This constant iterator cannot be used to remove that payload from the container, which it is currently pointing to. Iterating is slightly faster than with the non-constant version `clocked_peq_container_iterator`. Iterators are usually generated by the `begin()` and `end()` method of a container.

The `scml2::clocked_peq_container_const_iterator` class is templated with the underlying type of the payload:

```
template <typename PAYLOAD> class clocked_peq_container_const_iterator;
```

The API of the iterator conforms to the STL and provides the following methods:

```
const PAYLOAD* operator*() const;
```

Returns the payload pointed to by the iterator.

```
const PAYLOAD* operator->() const;
```

Returns the payload pointed to by the iterator.

```
self& operator++();
```

Moves the iterator forward to the next entry in the container. It returns an iterator, that points to the new position.

```
self operator++(int);
```

Moves the iterator forward to the next entry in the container. It returns an iterator, that points to the old position.

```
bool operator==(const self& x) const;
```

Compares the iterator to the other iterator *x*. It returns `true`, if both iterators are pointing to the same position.

```
bool operator!=(const self& x) const;
```

Compares the iterator to the other iterator *x*. It returns `true`, if both iterators are pointing to different positions.

#### 4.7.2 scml2::clocked\_peq

This is a modeling object similar to the `clocked_peq` container. It has a set of additional features:

- It allows registering a callback, which will be triggered whenever an element from the payload buffer becomes available.
- The callback will be called when the SystemC time has moved forward to the point of the arrival/notified time, that got stored with the payload in the buffer.
- It is possible to block the callback triggers. When blocked, the callback will not be triggered, until it is unblocked.
- When the callback is unblocked and the buffer contains a payload with arrival time that is in the past, the callback will be triggered at the next clock edge following the unblock time.

The `scml2::clocked_peq` class is templated with the underlying type of the payload:

```
template <typename PAYLOAD> class clocked_peq;
```

The following type definitions for iterators are available:

```
typedef clocked_peq_iterator<PAYLOAD> iterator;
typedef clocked_peq_const_iterator<PAYLOAD> const_iterator;
```

Objects of type `clocked_peq` can be constructed using one of the following constructors:

```
clocked_peq(scml_clock_if* clock=0);
clocked_peq(sc_core::sc_in<bool>& p);
```

where:

<code>clock</code>	Specifies the SCML clock object that is associated with the clocked PEQ container object.
<code>p</code>	Specifies the input port through which the clocked PEQ container is bound to its associated SCML clock.

The following functions are provided to connect the clocked PEQ to an SCML clock.

```
void set_clock(scml_clock_if* clock);
```

Associates the `clocked_peq` with the provided SCML clock.

```
void set_clock(sc_core::sc_in<bool>& p);
```

Associates the `clocked_peq` with the SCML clock, which is bound to the provided input port.

```
scml_clock_if* get_clock() const;
```

Returns the associated SCML clock.

Registering a callback for notifications of new payload is typically done using the following macro:

```
m_clk_peq.set_callback(SCML2_CLOCKED_CALLBACK(clock_cb));
```

```
bool notify(PAYLOAD* payload, const sc_core::sc_time& arrival_local_time);
```

Pushes a payload into the PEQ that was received by a temporal decoupled transport at local time `arrival_local_time`. The local time argument is given as SystemC time. It is converted to the clock tick count by rounding down to the last clock tick.

```
bool notify(PAYLOAD* payload, const sc_core::sc_time& arrival_local_time, long long delay_ticks);
```

Pushes a payload into the PEQ that was received by a temporal decoupled transport. The local time is defined by two arguments, the SystemC time `arrival_local_time` and the additional number of clock ticks `delay_ticks`. The internal arrival clock tick count is calculated by first converting `arrival_local_time` to the clock tick count by rounding down to the last clock tick, and then incrementing it by `delay_ticks`.

```
bool notify(void* payload, unsigned long long clock_ticks_to_skip);
```

Pushes a payload into the PEQ that was received by a temporal decoupled transport at the local time that is, `clock_ticks_to_skip` clock ticks in the future.

```
bool notifyAt(void* payload, unsigned long long arrival_tick_count);
```

Pushes a payload into the PEQ that was received by a temporal decoupled transport at the local time represented by the clock tick count `arrival_tick_count`.

```
bool is_empty() const;
```

Returns `true` if the PEQ is empty. This only considers visible payload (`timestamp < get_clock() -> get_tick_count()`).

```
const_iterator begin() const;
```

Returns a constant iterator that points to the beginning of the internal sorted list. This only considers visible payload (`timestamp < get_clock() -> get_tick_count()`).

```
iterator begin();
```

Returns a non-constant iterator that points to the beginning of the internal sorted list. This only considers visible payload (`timestamp < get_clock() -> get_tick_count()`).

```
const_iterator end() const;
```

Returns a constant iterator that points to the end of the internal sorted list. This only considers visible payload (`timestamp < get_clock() -> get_tick_count()`).

```
iterator end();
```

Returns a non-constant iterator that points to the end of the internal sorted list. This only considers visible payload (`timestamp < get_clock() -> get_tick_count()`).

```
PAYLOAD* get_next();
```

Extracts the first element from the internal sorted list. It returns a null pointer if the list was empty. This only considers visible payload (`timestamp < get_clock() -> get_tick_count()`).

```
void remove(const iterator& pos);
```

Removes the payload at position which is given by the iterator argument `pos`. The payload itself is not deleted.

This operation invalidates all the iterators which are currently pointing into the container. Managing the iterators is the user's responsibility.

```
bool remove(PAYLOAD* payload);
```

Removes the payload given by the `payload` argument. The payload itself is not deleted. This only considers visible payload (`timestamp < get_clock() -> get_tick_count()`). It returns `true` if the payload was found and removed. `False` is returned if the payload was not found. If the same payload is contained multiple times, than only the first occurrence is removed.

If this operation succeeds, it invalidates all those iterators, which are currently pointing into the container. Managing the iterators is the user's responsibility.

```
bool has_more_events() const;
```

Returns `true` if more payload is available in the container. In contrast to the method `is_empty()`, this considers not only the visible, but all payload. This method is usually used, if all visible/past payload has been processed/removed. In order to learn, if more processing has to be done at a future time, even if no new payload is pushed into the container.; an unblocked PEQ schedules another callback notification for the future.

```
unsigned long long get_next_event_arrival_tick() const;
```

Returns the arrival clock tick of the next payload in the PEQ. This is the time tick at which an unblocked PEQ will send the next callback. This method must not be called if `has_more_events()` returned `false!`



This method does not only consider the visible, but all payload. This method is usually used when all current processing has been done, in order to retrieve the time, when the next callback will notify to continue with more processing.

```
void block();
```

Blocks the sending of callbacks for the next payload arrival time.

```
void unblock(const sc_core::sc_time& local_time_before_clock_tick);
```

Schedules to unblock the PEQ after a SystemC time interval of `local_time_before_clock_tick`. If at the time of unblocking, the PEQ contains a payload with an arrival time that lays in the past, then a callback is sent at the next clock edge following the unblock time.

For example: Calling `unblock(SC_ZERO_TIME)` on a blocked PEQ, that contains a visible payload (arrival time in the past) will schedule a callback for the next clock tick.

```
void unblock(const sc_core::sc_time& local_time_before_clock_tick, long long delay_ticks);
```

Schedules to unblock the PEQ after a SystemC time interval of `local_time_before_clock_tick` plus `delay_ticks` clock ticks. For more details, see the method `unblock(const sc_core::sc_time&)`.

```
void unblock(unsigned long long clock_ticks_to_skip);
```

Schedules to unblock the PEQ after `clock_ticks_to_skip` clock ticks. For more details, see the method `unblock(const sc_core::sc_time&)`.

For example: Calling `unblock(0)` on a blocked PEQ that contains a visible payload (arrival time in the past) will schedule a callback for the next clock tick.

```
void unblockAt(unsigned long long tick_count_for_unblock);
```

Schedules to unblock the PEQ for the time given by the clock tick count `tick_count_for_unblock`. If `tick_count_for_unblock` lays in the past, then the PEQ is unblocked at the next tick count. For more details, see the method `unblock(const sc_core::sc_time&)`.

For example: Calling `unblockAt(get_clock()->get_tick_count())` on a blocked PEQ that contains a visible payload (arrival time in the past) will schedule a callback for the next clock tick.

An object of class `clocked_peq_iterator` is a pointer into a `clocked_peq` that allows iterating in forward direction over the sorted list of the PEQ. This non-constant version of iterators can be used to remove the payload it is currently pointing to from the PEQ. Iterating is slightly slower than with the constant version `clocked_peq_const_iterator`. Iterators are usually generated by the `begin()` and `end()` method of a clocked PEQ.

The `scml2::clocked_peq_iterator` class is templated with the underlying type of the payload:

```
template <typename PAYLOAD> class clocked_peq_iterator;
```

The API of the iterator conforms to the STL and provides the following methods:

```
PAYLOAD* operator*() const;
```

Returns the payload pointed to by the iterator.

```
PAYLOAD* operator->() const;
```

Returns the payload pointed to by the iterator.

```
self& operator++();
```

Moves the pointer forward to the next entry in the PEQ. It returns an iterator that points to the new position.

```
self operator++(int);
```

Moves the pointer forward to the next entry in the PEQ. It returns an iterator that points to the old position.

```
bool operator==(const self& x) const;
```

Compares the iterator to the other iterator x. It returns true if both iterators are pointing to the same position.

```
bool operator!=(const self& x) const;
```

Compares the iterator to the other iterator x. It returns true if both iterators are pointing to different positions.

An object of class `clocked_peq_const_iterator` is a pointer into a `clocked_peq` that allows iterating in forward direction over the sorted list of the PEQ. This constant iterator cannot be used to remove the payload it is currently pointing to from the PEQ. Iterating is slightly faster than with the non-constant version `clocked_peq_iterator`. Iterators are usually generated by the `begin()` and `end()` method of a container.

The `scml2::clocked_peq_const_iterator` class is templated with the underlying type of the payload:

```
template <typename PAYLOAD> class clocked_peq_const_iterator;
```

The API of the iterator conforms to the STL and provides the following methods:

```
const PAYLOAD* operator*() const;
```

Returns the payload pointed to by the iterator.

```
const PAYLOAD* operator->() const;
```

Return the payload pointed to by the iterator.

```
self& operator++();
```

Moves the pointer forward to the next entry in the PEQ. It returns an iterator that points to the new position.

```
self operator++(int);
```

Moves the pointer forward to the next entry in the PEQ. It returns an iterator that points to the old position.

```
bool operator==(const self& x) const;
```

Compares the iterator to the other iterator x. It returns true if both iterators are pointing to the same position.

```
bool operator!=(const self& x) const;
```

Compares the iterator to the other iterator x. It returns true if both iterators are pointing to different positions.

## 4.8 Code Example

This section describes how a programmable clock peripheral can be implemented based on scml\_clock.

- [Programmable Clock Peripherals](#)

### 4.8.1 Programmable Clock Peripherals

Programmable clock peripherals and timers can be coded easily by using clock tick callbacks. This is achieved by deriving the Timer from scml2::clocked\_module and implementing handle\_clock\_tick():

```
SC_MODULE Timer : private scml2::clocked_module {
public:
    sc_in<bool> clk_in;
    [...]
private:
    [...]
    virtual void end_of_elaboration() {
        set_clock(clk_in);
    }
    virtual void handle_clock_tick();
};
```

The functionality of the peripheral is implemented in the callback function attached to a memory-mapped register, modeled as an object of type scml\_memory.

For example, consider a module with the following data members:

```
scml_memory <unsigned int> CURRENT_VALUE_REG;
scml_memory <unsigned int> END_VALUE_REG;
unsigned long long mTimerStartTickCount;
```

Two call-back functions are registered with the END\_VALUE\_REG memory-mapped register:

```
MEMORY_REGISTER_READ(CURRENT_VALUE_REG, f_read_curr_value);
MEMORY_REGISTER_WRITE(END_VALUE_REG, f_write_end_value);
```

The implementation of the write callback writes a new value to END\_VALUE\_REG memory-mapped register and restarts the timer:

```
void f_write_end_value(unsigned int new_value, unsigned int, unsigned int) {
    END_VALUE_REG=new_value;
    cancel_clock_trigger();
    mTimerStartTickCount=get_clock()->get_tick_count();
    request_clock_trigger(END_VALUE_REG - 1);
}
```

The actual counter value of the timer is only calculated on demand. The implementation of the read callback takes the current tick count of the driving clock and subtracts the tick count from when the timer started:

```
unsigned int f_read_curr_value(unsigned int, unsigned int) {
    return (unsigned int) (get_clock()->get_tick_count() - mTimerStartTickCount);
}
```



# Chapter 5

## Pulse TLM Modeling

The Pulse protocol is a TLM representation of a bool signal to model clocks, Pulse Width Modulation (PWM) signals or any other regular or recurring Boolean signal in a system. Other than the default SystemC representation using `sc_signal<bool>`, the pulse signal adds information so that it is possible to avoid individual signal value change events for signals with a regular repeating pulse shape. The result is better simulation performance. It is like the `scml_clock_if` in that it represents a repeating signal with a repeat frequency, start time, period, and active/idle width, but it differs from the `scml_clock_if` in that it also allows for irregular pulse shapes. The latter prevents that modelers need to fall back to the default SystemC `sc_signal` whenever a model may generate irregular or unpredictable signals in certain cases.

The basis for Pulse TLM modeling is the pulse interface and the derived pulse protocol and pulse modeling objects.

- The pulse protocol is a TLM2 standards-based set of sockets and TLM interfaces built using the pulse interface. The pulse protocol engine provides basic access to the pulse interface features or allows to forward the pulse interface to reuse objects for further processing.
- The pulse modeling objects provide a basic set of building blocks to create PWM generators, timers, and clock dividers.

This chapter describes:

- [The Pulse Interface](#)

### 5.1 The Pulse Interface

The pulse interface is a generic modeling interface used by building blocks as well as for model interfaces. It is the basic interface shared by both types of components for exchanging information. The pulse interface provides with pulse information for a Boolean signal:

- It indicates whether the pulse is: occurs once, runs for a predefined number of periods, or continuous running.
- It provides with pulse shaping information: active level, start time, period, active width, and idle width.
- The pulse shaping information is static. Any change to the shape of the pulse implies the definition of a new pulse shape with an updated start time, and so on. With every change, a pulse generator should notify its observers that a new pulse definition is available.

The pulse interface consists of:

- An interface called by the pulse generator to notify pulse users of any changes in pulse shape information for predictable pulses. In case of unpredictable pulses, a pulse generator will use the forward interface to indicate a new pulse edge occurred.
- An interface used by the pulse user to fetch the pulse shape information, get information about future events for the pulse, and register itself as a pulse observer.

The pulse interface provides with three interface styles:

- When the pulse is predictable and static, a target of the pulse can calculate future events for the pulse using the pulse shape information. So, rather than counting pulse edges, a target can calculate itself at which timepoint the N'th edge will occur using the start time, period, active and idle width parameters. This interfacing style is the best for simulation performance.
- When the pulse is predictable but has a semi-variable period, it is no longer possible for the target to calculate future events based on the pulse period and so on, since these may change. However, since the pulse is still predictable (the changes to the pulse shape are predictable and known to the generator) it is possible for the target to request any future timepoint from the pulse generator. This interfacing style is still good for simulation performance. The APIs to request future timepoints also applies when the pulse is predictable and static.
- When the pulse shape is not predictable or varies, the pulse generator should call the observer APIs at each activation and idle event. Since this coding style will generate simulation activity for every edge this is less optimal for simulation performance.

A pulse generator should try to use the most optimal interfacing style at any point in time, but may need to fall back to the less optimized style when needed. This change of style can happen during runtime, for example, with a new configuration of the generator. The receiving side of a pulse should support both interfacing styles. The target of a pulse interface cannot make any assumptions about the style used at a certain point in time by a pulse generator.

- [The Pulse Definition Interface](#)
- [The Pulse Observer Interface](#)

### 5.1.1 The Pulse Definition Interface

Components or building blocks that define pulses should implement the `pulse_if` interface. The definition of the pulse interface is as follows:

```
class pulse_if {
public:
    virtual ~pulse_if() {}
    virtual bool is_enabled() =0;
    virtual bool get_active_value() = 0;
    virtual int get_run_mode() =0;
    virtual bool is_predictable() =0;
    virtual bool has_fixed_period() = 0;
    virtual sc_core::sc_time get_period() =0;
    virtual sc_core::sc_time get_start_time() =0;
    virtual sc_core::sc_time get_active_width() =0;
    virtual sc_core::sc_time get_idle_width() =0;
    virtual unsigned long long get_pulse_count() =0;
    virtual sc_core::sc_time get_active_time(unsigned long long activations_to_skip=0) =0;
    virtual sc_core::sc_time get_idle_time(unsigned long long idles_to_skip=0) =0;
    virtual bool is_active() = 0;
    virtual bool register_observer(pulse_observer*) =0;
    virtual bool unregister_observer(pulse_observer*) =0;
};
```

The APIs are explained below:

- `bool is_enabled();`
- `bool get_active_value();`
- `bool is_predictable();`
- `int get_run_mode();`
- `bool has_fixed_period();`

- `sc_core::sc_time get_start_time();`
- `sc_core::sc_time get_period();`
- `sc_core::sc_time get_active_width();`
- `sc_core::sc_time get_idle_width();`
- `unsigned long long get_pulse_count();`
- `sc_core::sc_time get_active_time(unsigned long long activations_to_skip=0);`
- `sc_core::sc_time get_idle_time(unsigned long long idles_to_skip=0);`
- `bool is_active();`
- `bool register_observer(pulse_observer*); bool unregister_observer(pulse_observer*);`

#### **5.1.1.1      `bool is_enabled();`**

The return value of this API indicates whether the pulse is running or deactivated. For a PWM-based pulse, this setting also controls whether the counter is active. This setting overrules the run mode value (see below).



By default and at simulation start, pulses are disabled (unlike clocks which are running).

---

#### **5.1.1.2      `bool get_active_value();`**

This interface indicates the intended Boolean value for the active phase of the pulse. Pulse generators should use this setting when they associate a certain bool value with the active state of the pulse and must forward that further in the design. Pulse receivers can decide to ignore or overwrite this setting.

#### **5.1.1.3      `bool is_predictable();`**

The return value of this API indicates whether the pulse is predictable. When an observer of a pulse receives a parameter `updated` call, it should use this API to check what interface style of the pulse to use. A pulse is predictable, if it is possible to calculate the timepoint for all edges for the pulse from the start time, period, active width, and idle width values.



By default and at simulation start, pulses are predictable.

---

#### **5.1.1.4      `int get_run_mode();`**

This API indicates the predicted number of periods this pulse will emit. Allowed values are:

<b>Allowed Values</b>	<b>Description</b>
-1	Indicates a continuously running pulse with no end time.
N (>0)	A fixed number of pulses, each 1 period long (1 for one-shot).
0	Indicates a disabled pulse.

The run mode should be ignored for a disabled pulse (`enabled()` returns `false`). It is OK to have `run mode == -1` when `is_enabled` is `false`. It is an error to have a predictable pulse enabled and `run mode` set to 0. This setting does not apply to unpredictable pulses.

### **5.1.1.5      `bool has_fixed_period();`**

Indicates whether the pulse has a fixed or variable period. When the pulse has a fixed period and is predictable, then a pulse observer can calculate future events using period, active width, and idle width. A predictable pulse that does not have a fixed period should use the `get_active_time` and `get_idle_time` APIs. Predictable pulses with a fixed period are like clocks and derived pulses can also be predictable. Pulses derived from a predictable pulse with a variable period are no longer predictable themselves.

### **5.1.1.6      `sc_core::sc_time get_start_time();`**

This API gets the starting point for the pulse. Returns the SystemC kernel time at which point the first pulse starts. A pulse starts with an active period and is followed by the idle period. Pulses cannot start with an idle period; in that case, the start time should be delayed.

When receiving a parameter `updated` call, start time can be in the future or in the past:

- The start time is in the future in case there is a ramp-up period for the pulse to start.
- The start time is in the past for example, for a PWM generator in case there is a clock or frequency change in the middle of the pulse. In such a case, the start time should be recalculated to ensure that future events of the current pulse still happen at the expected timepoint. This may require start time to be in the past.

Start time only applies for predictable pulses with a fixed period.

### **5.1.1.7      `sc_core::sc_time get_period();`**

Returns the period of the pulse, this is the total length of a single pulse (the sum of the time the pulse is active and the time the pulse is idle). The period only applies to predictable pulses with a fixed period.

### **5.1.1.8      `sc_core::sc_time get_active_width();`**

Returns the width of the pulse (the time it is active). It is possible for the active width to be `SC_ZERO_TIME`, for example, in case of a 0% duty cycle. Active width only applies to predictable pulses with a fixed period.

### **5.1.1.9      `sc_core::sc_time get_idle_width();`**

Returns the width of the inactive part of the pulse. It is possible for the idle width to be `SC_ZERO_TIME`, for example, in case of 100% duty cycle. Idle width only applies to predictable pulses with a fixed period.

### **5.1.1.10     `unsigned long long get_pulse_count();`**

Returns the number of pulses that have occurred on this `pulse_if`. It is up to the pulse generator implementation to decide when to (re-)start the pulse count. The pulse count should continue after a parameter update even if that implies a start time that does not correspond to the first counted pulse. The pulse count should be available for predictable as well as unpredictable pulses.

### **5.1.1.11     `sc_core::sc_time get_active_time(unsigned long long activations_to_skip=0);`**

Returns the SystemC time for a future activation edge. The return value is the absolute timepoint, that is, not relative to the current SystemC time. Returns the SystemC time for the activation (start of next active time) that happens after `activations_to_skip` activations from now. When called with `activations_to_skip` set to 0 (default), it should return the activation time of the current pulse, which could be in the past! If it is not possible to predict the activation edge the API call should return `SC_ZERO_TIME`. This API can be used for predictable pulses with fixed and variable periods, but not for non-predictable pulses.

### **5.1.1.12    `sc_core::sc_time get_idle_time(unsigned long long idles_to_skip=0);`**

Returns the SystemC time for a future idle edge. The return value is the absolute timepoint, that is, not relative to the current SystemC time. Returns the SystemC time for the switch to idle that happens after `idles_to_skip` idle states from now. When called with `idles_to_skip=0` (default), it should return the time for the idle edge of the idle edge for the current pulse, which could be in the past! If it is not possible to predict the idle edge the API call should return `SC_ZERO_TIME`. This API can be used for predictable pulses with fixed and variable periods, but it not for non-predictable pulses.

### **5.1.1.13    `bool is_active();`**

Should return true when the pulse is active for the current SystemC time. This API can be used for predictable as well as unpredictable pulses.

### **5.1.1.14    `bool register_observer(pulse_observer*);`               `bool unregister_observer(pulse_observer*);`**

These APIs register or deregister pulse observers. A pulse generator should notify all its observers whenever there is a change in the pulse shaping information for predictable pulses, and on each active an idle edge for unpredictable pulses.

## **5.1.2    The Pulse Observer Interface**

The `pulse_observer` interface should be implemented by components or building blocks that receive pulses. Pulse generators should call the observer APIs to notify the pulse receivers of changes or events. The definition of the pulse observer interface is as follows:

```
class pulse_observer {
public:
    virtual ~pulse_observer() {}
    virtual void handle_pulse_parameters_updated(scml2::objects::pulse_if*) =0;
    virtual void handle_pulse_activation(scml2::objects::pulse_if*) =0;
    virtual void handle_pulse_idle(scml2::objects::pulse_if*) =0;
    virtual void handle_pulse_deleted(scml2::objects::pulse_if*) =0;
};
```

The APIs are explained below:

- `void handle_pulse_parameters_updated(scml2::objects::pulse_if*);`
- `void handle_pulse_activation(scml2::objects::pulse_if*);`
- `void handle_pulse_idle(scml2::objects::pulse_if*);`
- `void handle_pulse_deleted(scml2::objects::pulse_if*);`

### **5.1.2.1    `void handle_pulse_parameters_updated(scml2::objects::pulse_if*);`**

Pulse generators should call this API to notify the pulse observers of any changes to the pulse definition. This is whenever the pulse is enabled or disabled or when the predictability of the pulse is changing. Pulse generators should call the API whenever any parameter of a predictable pulse is changing (active level, start time, period, active/idle width). The `pulse_if*` argument allows to observe multiple pulse interfaces from a single implementation.

Rules to consider for a parameter update:

- A parameter update results in a new pulse definition, that is, users of the pulse information should disregard all previous information immediately (no transition effects). A series of pulses can be generated as a series of one-shot pulses or pulses with a run mode set larger than 0.
- On a pulse update for all future events related to that pulse or derived from that pulse definition should be canceled.
- It is allowed for an update to define a new pulse with start time in the past. Periods in the past should be ignored. An activation edge in the past is possible as transitional effect of a clock frequency update when moving to a slower clock.
- It is allowed to disable a pulse at any time, it is up to the receiving model to define the consequences related to that (signal resets and so on.)

#### **5.1.2.2      `void handle_pulse_activation(scml2::objects::pulse_if*);`**

Pulse generators should use this to notify the pulse observers of a pulse activation event for unpredictable pulses.

Rules to consider for a pulse activation call:

- A pulse generator should call this API when the active part of a pulse starts.
- It can be called from a method process (that is, no `wait()` call is allowed in the implementation).
- It is not allowed for predictable pulses.
- It is not allowed for disabled pulses.
- When a pulse activation call is received, all pulse shaping information of that pulse definition should be ignored (as with unpredictable pulses in general).

#### **5.1.2.3      `void handle_pulse_idle(scml2::objects::pulse_if*);`**

Pulse generators should use this to notify the pulse observers that the pulse is entering the idle phase for unpredictable pulses.

Rules to consider for a pulse idle call:

- A pulse generator should call this API when the idle part of a pulse starts.
- It can be called from a method process (that is, no `wait()` call is allowed in the implementation).
- It is not allowed for predictable pulses.
- It is not allowed for disabled pulses.
- When a pulse activation call is received all pulse shaping information of that pulse definition should be ignored (as with unpredictable pulses in general).

#### **5.1.2.4      `void handle_pulse_deleted(scml2::objects::pulse_if*);`**

Pulse generators should use this to notify the pulse observers of the deletion of the `pulse_if`. It should be used to ensure a proper cleanup at the end of a simulation. Local copies of the `pulse_if` pointer should be ignored and the pulse observer should no longer try to access any information from this pulse after this call.

# Chapter 6

## Modeling Utilities

This chapter describes:

- Port Utilities
- Commands
- Parameters
- Reporting
- FastTrack

### 6.1 Port Utilities

This section describes:

- `dmi_handler`
- `initiator_socket`
- Pin Callback Functions
- Utility APIs

#### 6.1.1 `dmi_handler`

`scml2::dmi_handler` is a convenience object that takes care of the DMI pointer requests and book keeping. The object is targeting protocols for which the DMI handling is the same as for the TLM2 base protocol. It may be required to create a dedicated DMI handler for other protocol definitions, for example, when address decoding can be influenced by other attributes like security and so on. The DMI handler is initialized by giving it an interface that it should use to initiate TLM2 transport calls. The model itself should then use the read and write APIs of the DMI handler to start a transaction. The DMI handler will first check if it does not have a pointer available to provide the read or write data, if not it will try to request a DMI pointer and if that fails it will forward the transaction of the TLM2 interface. The DMI handler also has APIs to control whether DMI accesses will be attempted.

The include file of the `dmi_handler` objects is `scml2/dmi_handler.h`.

The following sections describe:

- Methods that are available to configure the `dmi_handler` object:
- The access methods of the `dmi_handler` object.

The following methods are available to configure the `dmi_handler` object:

```
void set_interface(tlm::tlm_fw_direct_mem_if<tlm::tlm_generic_payload*>* ifs)
```

Sets the forward DMI. This interface is used to request the DMI pointers.

```
bool is_dmi_enabled()
```

Returns true if DMI accesses are allowed for the object, false otherwise. DMI is enabled by default.

```
void enable_dmi()
void disable_dmi()
```

Enables/disables DMI accesses for the object.

The `dmi_handler` object has the following access methods:

```
bool read(unsigned long long address,
          unsigned char* data,
          unsigned int dataLength,
          const unsigned char* byteEnables,
          unsigned int byteEnableLength,
          sc_core::sc_time& t)
bool write(unsigned long long address,
           const unsigned char* data,
           unsigned int dataLength,
           const unsigned char* byteEnables,
           unsigned int byteEnableLength,
           sc_core::sc_time& t)
bool read(unsigned long long address,
          unsigned char* data,
          unsigned int dataLength,
          sc_core::sc_time& t)
bool write(unsigned long long address,
           const unsigned char* data,
           unsigned int dataLength,
           sc_core::sc_time& t)
```

Try to do a DMI access. If a DMI access is not possible or if the access does not fit into one DMI range, `false` is returned. Otherwise the data is copied and `true` is returned. The `t` argument is incremented with the read or write latency, respectively.

```
bool read_debug(unsigned long long address,
                unsigned char* data,
                unsigned int dataLength)
bool write_debug(unsigned long long address,
                const unsigned char* data,
                unsigned int dataLength)
```

Tries to do a DMI access. If a DMI access is not possible or if the access does not fit into one DMI range, `false` is returned. Otherwise the data is copied and `true` is returned.

```
bool transport(tlm::tlm_generic_payload& trans, sc_core::sc_time& t)
bool transport_debug(tlm::tlm_generic_payload& trans)
```

Try to do a DMI access. If a DMI access is not possible or if the access does not fit into one DMI range, `false` is returned. Otherwise the data is copied and `true` is returned.

```
void invalidate_direct_mem_ptr(sc_dt::uint64 startRange,
                               sc_dt::uint64 endRange)
```

Must be called when the DMI pointers have to be invalidated.

### 6.1.2 initiator\_socket

`scml2::initiator_socket` is a convenience TLM2 initiator socket specifically targeting the LT coding style. It contains a `dmi_handler` to manage DMI accesses and it implements the `mappable_if` so that this socket can be used as a destinations for a router. The `initiator_socket` also gets a reference to a `quantum_keeper` that it will use to maintain timing annotation whenever the `dmi_handler` does an

access of the forward transport interface of the socket. Transactions are initiated in the same way as for the `dmi_handler`; that is, via read and write APIs.

The `initiator_socket` object implements the `mappable_if` object, which means that it can be the destination for a mapped range of a `router` object.

The include file of the `initiator_socket` objects is `scml2/initiator_socket.h`.

The `initiator_socket` class is templated with the `BUSWIDTH`:

```
template <unsigned int BUSWIDTH> class initiator_socket
```

The following methods are available to configure the `initiator_socket` object:

```
template <typename T>
void set_quantumkeeper(T& quantumKeeper)
```

Sets the quantum keeper the socket should use. The registered class must implement the following methods (see the section on `tlm_quantumkeeper` in the *IEEE Std 1666 TLM-2.0 Language Reference Manual*):

```
void inc(const sc_core::sc_time& t)
void set(const sc_core::sc_time& t)
bool need_sync() const
void sync()
sc_core::sc_time get_local_time() const
```

If a quantum keeper is set, the socket will pass the local time when doing a bus access and increment the local time when the timing annotation was incremented by the DMI access or bus access. If needed (`need_sync()` returns `true`), the socket will synchronize the quantum keeper after incrementing the local time.

If no quantum keeper is set, `sc_core::SC_ZERO_TIME` will be passed and `wait()` will be called if the timing annotation was incremented.

```
void set_endianness(tlm::tlm_endianness endianness)
```

Sets the endianness of the initiator mode. If the endianness is different from the host endianness, the socket converts the address and data before doing the access.

```
bool is_dmi_enabled()
```

Returns `true` if DMI accesses are allowed for the object, `false` otherwise. DMI is enabled by default.

```
void enable_dmi()
void disable_dmi()
```

Enables/disables DMI accesses for the object.

The following access methods are available:

```
template <typename DT>
bool read(unsigned long long address, DT& data)
template <typename DT>
bool write(unsigned long long address, const DT& data)
```

Access methods to do single-word or subword accesses. The data passed must be in arithmetic format (host endianness). If the endianness of the socket is different from the host endianness, the address and data are converted before doing the access. First a DMI access is done. If this fails, a bus access (`b_transport()`) is done. If this access fails with an error response, `false` is returned, otherwise `true` is returned. If a quantum keeper is set, the local time is passed with the bus access and the local

time of the quantum keeper is incremented with the returned timing annotation. If no quantum keeper is set, SC\_ZERO\_TIME is passed and wait() is called if the timing annotation was incremented.

```
template <typename DT>
bool read(unsigned long long address, DT* data, unsigned int count)
template <typename DT>
bool write(unsigned long long address, const DT* data, unsigned int count)
```

Access methods for burst accesses. The passed data pointer should contain an array of words in arithmetic format (host endianness). If the endianness of the socket is different from the host endianness, the address and data are converted before doing the access. First a DMI access is done. If this fails, a bus access (b\_transport()) is done. If this access fails with an error response, false is returned; otherwise true is returned. If a quantum keeper is set, the local time is passed with the bus access and the local time of the quantum keeper is incremented with the returned timing annotation. If no quantum keeper is set, SC\_ZERO\_TIME is passed and wait() is called if the timing annotation was incremented.

```
template <typename DT>
bool read(unsigned long long address, DT& data, sc_core::sc_time& t)
template <typename DT>
bool write(unsigned long long address, const DT& data, sc_core::sc_time& t)
```

Access methods to do single-word or subword accesses. The data passed must be in arithmetic format (host endianness). If the endianness of the socket is different from the host endianness, the address and data are converted before doing the access. First a DMI access is done. If this fails, a bus access (b\_transport()) is done. If this access fails with an error response, false is returned, otherwise true is returned.

The time argument is passed with the b\_transport() call. If a quantum keeper was set in the socket, it will be ignored.

```
template <typename DT>
bool read(unsigned long long address, DT* data, unsigned int count,
          sc_core::sc_time& t)
template <typename DT>
bool write(unsigned long long address, const DT* data, unsigned int count)
```

Access methods for burst accesses. The passed data pointer should contain an array of words in arithmetic format (host endianness). If the endianness of the socket is different from the host endianness, the address and data are converted before doing the access. First a DMI access is done. If this fails, a bus access (b\_transport()) is done. If this access fails with an error response, false is returned; otherwise true is returned. The time argument is passed with the b\_transport() call. If a quantum keeper was set in the socket, it will be ignored.

```
template <typename DT>
bool read_debug(unsigned long long address, DT& data)
template <typename DT>
bool write_debug(unsigned long long address, const DT& data)
```

Access methods to do single-word or subword debug accesses. The data passed must be in arithmetic format (host endianness). If the endianness of the socket is different from the host endianness, the address and data are converted before doing the access. First a DMI access is done. If this fails, a bus access (b\_transport()) is done. If the debug bus access did not succeed, false is returned; otherwise true is returned.

```
template <typename DT>
bool read_debug(unsigned long long address, DT* data, unsigned int count)
template <typename DT>
bool write_debug(unsigned long long address, const DT* data, unsigned int count)
```

Access methods to do burst debug accesses. The passed data pointer should contain an array of words in arithmetic format (host endianness). If the endianness of the socket is different from the host endianness, the address and data are converted before doing the access. First a DMI access is done. If this fails, a bus access (`b_transport()`) is done. If the debug bus access did not succeed, `false` is returned; otherwise `true` is returned.

```
void b_transport(tlm::tlm_generic_payload& trans, sc_core::sc_time& t)
unsigned int transport_dbg(tlm::tlm_generic_payload& trans)
tlm::tlm_sync_enum nb_transport_fw(tlm::tlm_generic_payload& trans,
                                  tlm::tlm_phase& phase, sc_core::sc_time& t)
bool get_direct_mem_ptr(tlm::tlm_generic_payload& trans, tlm::tlm_dmi& dmiData)
```

TLM2 access methods. First a DMI access is tried. If this fails, a bus access is done. No endianness conversions are done; the passed transaction should already be in the correct format.

The following methods are available to register or unregister a backward path interface to the `initiator_socket`. For more information, see the *Accellera IEEE 1666 LRM Language Reference Manual*.

```
void register_bw_direct_mem_if(tlm::tlm_bw_direct_mem_if* bwInterface)
void unregister_bw_direct_mem_if(tlm::tlm_bw_direct_mem_if* bwInterface)
```

Register or unregister `tlm::tlm_bw_direct_mem_if` to the `initiator_socket`. The `invalidate_direct_mem_ptr` method of all registered interfaces will be called in case the `invalidate_direct_mem_ptr` call is done on the backward path of the `initiator_sockets`. Multiple interfaces can be registered. In such cases, the call will be forwarded to all registered interfaces.

```
typedef tlm::tlm_bw_nonblocking_transport_if<tlm::tlm_generic_payload, tlm::tlm_phase>
BwTransportIf
virtual void register_bw_transport_if(BwTransportIf* bwInterface)
virtual void unregister_bw_transport_if(BwTransportIf* bwInterface)
```

Register or unregister `tlm::tlm_bw_nonblocking_transport_if` to the `initiator_socket`. The `nb_transport_bw` method of the registered interface will be called in case the `nb_transport_bw` call is done on the backward path of the `initiator_sockets`. Only one `tlm::tlm_bw_nonblocking_transport_if` can be registered to the `initiator_socket`.

### 6.1.3 Pin Callback Functions

The following functions are available for registering user callbacks on changes of input pins:

- The convenience functions for registering user callbacks on pins:

```
set_change_callback(pin, object, callback);
set_change_callback(pin, object, callback, tag);
set_posedge_callback(pin, object, callback)
set_posedge_callback(pin, object, callback, tag)
set_negedge_callback(pin, object, callback);
set_negedge_callback(pin, object, callback, tag);
```

where:

pin	Specifies the pin of type <code>sc_in&lt;T&gt;</code> . For <code>set_posedge_callback</code> and <code>set_negedge_callback</code> , the pin has to be of type <code>sc_in&lt;bool&gt;</code> .
object	Is a pointer to the class containing the callback method.

callback	Is a pointer to a member function of the object class. It must have one of the following signatures: <code>void changeCallback()</code> <code>void changeCallback(int tag)</code>
tag	Is an user-provided integer that is passed to the callback.



- The SCML2\_CALLBACK macro can be used as a convenience macro for registering a member function as a callback.
- The Pin Callback functions can only be registered **before** end of elaboration, so for example in the module's constructor, but **not** in its `end_of_elaboration()` method.

#### 6.1.4 Utility APIs

As part of the SCML2 library, there are some tool interfaces available. The important ones to be taken into account while creating component models are listed below.

- `scml2::register_reset_trigger`:

This is an API that is to be used in a component that has a reset output signal. The API allows a software debugger or a simulation debugger like VP Explorer to trigger the reset behavior of the component, so that you can develop a debugging and testing script that includes resetting part or all of the system.

```
void scml2::register_reset_trigger(const std::string& name,
                                    SCML2_CALLBACK(func), const std::string& description);
```

##### Arguments:

<i>name</i>	Is the logical reset domain name for the reset that will be generated. This is the name for the domain as it will be used in the debugger. The specific use of this domain name is specific to each debugger integration. For more information, check the debugger integration manual. It is allowed to have multiple reset generators with the same domain name. When a debugger triggers a reset domain, all trigger functions that are registered with that domain will be executed.
<i>func</i>	Is the member function of module that will be called by the debugger and which should implement a reset trigger.
<i>description</i>	Is a string to describe the usage of the reset trigger.

Define the function that will be triggered as follows:

```
int func (const std::string& arg);
```

- The function should return an integer value where 0 means success and all other values indicate an error. The specific meaning of the return value is left to the designer. The return value will be passed to the debugger so that the right action can be taken.
- The string argument is intended to pass arguments from the debugger command into the function driving the reset, so that the reset generator can be configured for a specific behavior. It is up to the reset generator to decide what to do with the argument, it is required that the reset generator has a default behavior so that it can work with an empty argument string.

- When a reset domain contains multiple reset generators they will all get called with the same argument string. The return value to the debugger will be 0 if all calls were successful, else the return value will be determined by the last failing trigger call.

## 6.2 Commands

This section describes:

- [scml\\_command\\_processor](#)
- [scml\\_loader](#)

### 6.2.1 scml\_command\_processor

`scml_command_processor` is an SCML object that provides the link between an interactive debugger and the simulation. It allows the debugger to execute commands in the simulation, for example, to switch the operating mode of a component or to generate data analysis about the execution and state of the model. The command processor operates local to a specific component in a model and can manage multiple commands each with their own set of arguments.

For more information on issuing user commands, see "About Commands" in the [VP Explorer Tcl Interface Reference Manual](#), and "[Calling SCML User Commands](#)" in the [SystemC Shell Manual](#).

`scml_command_processor` can only be used via the following macros:

- `SCML_COMMAND_PROCESSOR`

This macro is used to indicate which method should be called when the external debugger sends a command to the object. The macro takes exactly one argument: the name of the command handler method. The return type of this method should be `std::string` and it should accept one parameter of type `const std::vector<std::string> &`, which contains the command and its arguments, if any.

It is guaranteed that the command handler method is only called for commands that have been declared using the `SCML_ADD_COMMAND` macro and for which the number of arguments lie within the bounds declared by the `SCML_ADD_COMMAND` macro.

The string that is returned by the command handler method is displayed by the debugger.

- `SCML_ADD_COMMAND`

This macro is used to declare which commands can be handled by the object. The macro takes the following parameters:

- `std::string name` specifies the name of the command.
- `unsigned int minParam` specifies the minimum number of parameters the command needs.
- `unsigned int maxParam` specifies the maximum number of parameters the command accepts.
- `std::string synopsis` is a short description of the command and its arguments.
- `std::string description` is an elaborate description of the command and its arguments.



**Caution** Long-running SCML commands (execution time > 1 second) can cause issues with interactive debuggers! Use `SCML_COMMAND_YIELD` to prevent long delays in other debugger requests.

- `SCML_COMMAND_YIELD`

SCML commands block other debugger requests during their complete execution time. To avoid unresponsive or even erroneous behavior in debuggers connected to the simulation, long-running commands must call SCML\_COMMAND\_YIELD once a second, to give other debugger requests a chance to get executed.

SCML\_COMMAND\_YIELD macro takes no parameters.

The following commands are available in the ::scsh namespace to access scml\_command\_processor objects:

**Table 6-1 Commands to Access scml\_command\_processor Objects**

Scsh Command	VP Explorer Command	Description
list_commands	help or <i>instance help</i>	When no arguments are given, the help command lists all global command followed by the commands available only on the current instance context.
help_command <i>command</i>	help <i>command</i> or <i>instance help command</i>	When the name of a command is given, the help command provides help on just that command.
execute_command <i>instance command args</i>	<i>command args</i> or <i>instance command args</i>	If the current context is to some instance, then you can call that instance's commands directly.

The VP Explorer command-line interface supports the notion of a current instance context to simplify what users need to type. Hence, the following approaches are shown in the above table for each command:

- Where the current context is already set to an instance so the command acts on that instance directly, and
- Where the instance's name is the first part of the command to call that instance's command without changing the context.

To set the current instance context, enter the name of an instance. You can enter its full hierarchical name, and in most cases, a convenient shortcut is provided that you can also enter the instance name without any hierarchy since that also uniquely identifies the instance. For example:

```
Tcl> i_MyCore1
i_MyCore1> help
```

In the above example, entering the instance name changed the current context to that instance and so the prompt changed. Then, the subsequent help command will show both the global commands and the commands on that instance i\_MyCore1.

The following section shows a code example.



**Note** The command handler method is called from a debugger. When scml\_memory or scml\_router objects are accessed from this handler, the readDebug() / writeDebug() methods must be used. Using read() / write() or put() / get() from this handler will give undefined behavior.

```

SC_MODULE(MyModule) {
    SC_CTOR(MyModule) {
        // Whenever a debugger sends a command to this object, we want the
        // handleCommand method to be invoked.
        SCML_COMMAND_PROCESSOR(handleCommand);

        // This object understands two commands: 'status' and 'load'.
        SCML_ADD_COMMAND("status", 0, 0, "Reports that the status of this device",
                         "This command will report the status of the A, B and C "
                         "registers");
        SCML_ADD_COMMAND("load", 1, 2, "load <filename> [format]",
                         "This command loads the given file onto this device."
                         "If no format is supplied, the ELF format is assumed."
                         "Other supported formats are 'raw' and 'coff'.");
    }

    std::string handleCommand(const std::vector<std::string>& cmd)
    {
        if (cmd[0] == "status") {
            return getStatus();
        } else if (cmd[0] == "load") {
            std::string filename = cmd[1];
            std::string format = "ELF";
            if (cmd.size() == 2) format = cmd[2];
            if (load(filename, format)) {
                return "OK";
            } else {
                return "Could not load " + filename;
            }
        } else {
            // This should never happen!
            assert(false);
            return "ERROR: unknown command " + cmd[0];
        }
    }

    std::string getStatus() const
    {
        ...
    }

    bool load(std::string filename, std::string format)\n
    {
        ...
    };
}

```

## 6.2.2 scml\_loader

The `scml_loader` class is an abstract base class that can be used to add image loading capabilities to your `sc_module`. It only has one pure virtual method, `loadImage`, which has the following signature:

```
bool loadImage(const std::string& fileName, long long offset,
               const std::string& type)
```

where:

<code>fileName</code>	Specifies the relative or absolute path to the image that should be loaded.
<code>offset</code>	Specifies the offset that should be added to all addresses in the image to obtain the absolute addresses at which the image should be loaded. If your module does not support relative loading, your implementation should check that <code>offset</code> is 0 and if not, return <code>false</code> .
<code>type</code>	Specifies the file format that is used by the image. If this is an empty string and your module supports multiple file formats, your implementation should try to find out the file format by itself. If your module does not support multiple formats, your implementation should verify that <code>type</code> is an empty string and if not, return <code>false</code> . Where applicable, the "raw" type should also be supported for loading plain binary files. The possible values of the <code>type</code> parameter may be freely chosen.

Your implementation should return `true` if the image was loaded successfully and `false` otherwise.

`loadImage()` can be invoked in several ways:

- By loading the image through the VP Explorer. For more information, see "[Loading Software Images to the Simulation](#)" in *VP Explorer User Guide*.
- By means of the `--cwr_load` or the `--cwr_imagemap` command-line arguments. The following section describes these arguments.

Instantiating an `scml_loader` object in a simulation automatically adds the following command-line arguments:



**Note** The `--cwr_load` and `--cwr_imagemap` command-line arguments only work on targets that have this loading facility explicitly enabled.

- `--cwr_imagemap imageMappingFile` can be used to invoke the `loadImage()` method of an `scml_loader` object. The `imageMappingFile` should contain lines of the form `moduleName imageFile offset`, separated by newlines, where:

<code>moduleName</code>	Is the hierarchical name of the <code>sc_module</code> that instantiates the <code>scml_loader</code> object.
<code>imageFile</code>	Is the relative or absolute path name to the file you want to load.
<code>offset</code>	Is the offset at which you want the image to be loaded.

- `--cwr_load moduleName, imageFile[, offset]` can be used to invoke the `loadImage()` method of an `scml_loader` object, where:

<code>moduleName</code>	Is the hierarchical name of the <code>sc_module</code> that instantiates the <code>scml_loader</code> object.
<code>imageFile</code>	Is the relative or absolute path name to the file you want to load.
<code>offset</code>	Is the offset at which you want the image to be loaded. Specifying the offset is optional. If no offset is specified, offset 0 is assumed.

## 6.3 Parameters

This section describes:

- [scml\\_property](#)
- [scml\\_property\\_registry](#)
- [scml\\_property\\_server\\_if](#)
- [scml\\_simple\\_property\\_server](#)

### 6.3.1 scml\_property

`scml_property` is an SCML object used for model configuration, it can hold a value of type `int`, `bool`, `double` or `string`. The value of a `scml_property` is loaded during elaboration and is intended to provide the link between platform authoring tools and the simulation. It provides an easy to use and very flexible configuration mechanism for models. Typically, configuration parameters are stored in an VP Config which is generated by the authoring tool (Platform Creator) and is loaded by VP Explorer at the start of the simulation. Without recompiling or editing the model, it is then possible to change the configuration of the simulation simply by editing or modifying the XML file.

The scml\_property classes can be used inside SystemC modules and will automatically read their value from the XML file exported by Platform Creator.

The following scml\_property classes are available:

```
scml_property<int>
scml_property<unsigned int>
scml_property<double>
scml_property<bool>
scml_property<std::string>
scml_property<long long>
scml_property<unsigned long long>
```

The scml\_property class is templated with the underlying value type. The following type definitions are available to support generic programming:

```
typedef T value_type;
typedef scml_property_base<value_type> this_type;
typedef this_type* this_pointer_type;
typedef this_type& this_reference_type;
```

The following constructors are available:

```
scml_property(const ::std::string& name);
scml_property(const ::std::string& name, T defaultValue);
```

where:

<i>name</i>	Specifies the name of the property. The name of the property is used together with the hierarchical SystemC name of the module to access the value of the property in the XML file.
<i>defaultValue</i>	Specifies the default value of the property. This default value is only used if the property is not found in the XML file.

The following assignment operators are available:

```
this_reference_type operator=(const scml_property<T>&);
this_reference_type operator=(value_type);
```

The following arithmetic assignment operators are available and behave as defined for the underlying value type:

```
this_reference_type operator += (value_type);
this_reference_type operator -= (value_type);
this_reference_type operator /= (value_type);
this_reference_type operator *= (value_type);
this_reference_type operator %= (value_type);
this_reference_type operator ^= (value_type);
this_reference_type operator &= (value_type);
this_reference_type operator |= (value_type);
this_reference_type operator <<= (value_type);
this_reference_type operator >>= (value_type);
```

A property object can be converted to the underlying value type:

```

operator T() const;

std::string getName() const;
    Returns the name of the scml_property.

std::string getType() const;
    Returns the type of the scml_property. The type can be one of the following strings: int,
    unsigned int, bool, double, string, unsigned long long, long long.

```

```

class mymodule : public sc_module
{
public:
    SC_HAS_PROCESS(mymodule);

    mymodule(sc_module_name name)
        : sc_module(name),
          intProp("intProp"),
          boolProp("boolProp"),
          doubleProp("doubleProp"),
          stringProp("stringProp")
    {
        SC_THREAD(my_thread);
    }

    scml_property<int> intProp;
    scml_property<bool> boolProp;
    scml_property<double> doubleProp;
    scml_property<string> stringProp;

    void my_thread () {
        cout << "mymodule: Int: " << intProp
        << " Bool: " << boolProp
        << " double: " << doubleProp
        << " and string: " << stringProp
        << endl;
    }
};

```

This module has a property of each of the possible types. Each of the properties is part of the initialization list of the constructor. The properties automatically get their value upon construction of the module. They can be used as their value types anywhere in the module.

Platform Creator exports the XML file containing all the properties of all the modules in the system. For Platform Creator to know that a certain module has properties, these properties need to be added to the respective module. Platform Creator can recognize the SCML properties automatically, or you can add them manually. For details on how to create SCML properties, see "SCML Properties" in the *Platform Creator Reference Manual*, which is included in the Platform Architect documentation.

### 6.3.2 scml\_property\_registry

The `scml_property` classes can only be used inside SystemC modules. Using `scml_property` objects does not require any knowledge of the `scml_property_registry`. Objects of type `scml_property` get their values automatically.

Two mechanisms are available to load values in properties:

- An XML file exported by Platform Creator
- A custom property server

The XML file exported by Platform Creator contains all the parameters of the system. Besides MODULE parameters (which are the `scml_property` objects of a module), Platform Creator also exports CONSTRUCTOR, PORT, and PROTOCOL parameters.

`scml_property_registry` offers an API to read the values of these parameters.

The following enumeration type is available:

```
enum PropertyType {  
    GLOBAL,  
    CONSTRUCTOR,  
    MODULE,  
    PORT,  
    PROTOCOL  
};
```

This enumeration type indicates the kind of parameter you want to access:

- GLOBAL is reserved for internal usage.
- CONSTRUCTOR indicates a constructor argument of a module.
- MODULE indicates a module parameter.
- PORT indicates a port parameter.
- PROTOCOL indicates a protocol parameter.

The scml\_property\_registry class is a singleton class. A reference to the instance of the class can be obtained by calling the static inst() function:

```
static scml_property_registry& inst();
```

The following functions are available for getting the values of a certain parameter:

```
int getIntProperty(PropertyType type, const std::string& schierName,  
                  const std::string& name);  
bool getBoolProperty(PropertyType type, const std::string& schierName,  
                     const std::string& name);  
std::string getStringProperty(PropertyType type,  
                               const std::string& schierName,  
                               const std::string& name);  
double getDoubleProperty(PropertyType type, const std::string& schierName,  
                        const std::string& name);
```

These functions all take the same parameters:

- PropertyType type specifies the type of property you want to access.
- const std::string& schierName specifies the hierarchical SystemC name of the sc\_object that contains the parameter. In case of a PORT or PROTOCOL property, this is the hierarchical SystemC name of the port. In case of a MODULE or CONSTRUCTOR parameter, this is the hierarchical name of the module.
- const std::string& name specifies the name of the property whose value you want to get.

```
bool setCustomPropertyServer(scml_property_server_if *);
```

Sets the custom property server.



This function must be called before any property that depends on it (to get its value) is constructed. For more information about the scml\_property\_server\_if class, see “scml\_property\_server\_if” on page 204.

```
std::vector<std::string> getPropertyNames( const std::string& schierName);
```

Queries the list of names of the available parameters.

```
scml_property_registry::PropertyType getPropertyType( const std::string& scHierName,  
const std::string& propName);
```

Queries the type of property.

```
class myport : public sc_port<my_interface>  
{  
public:  
    myport(const string& name)  
        : sc_port<my_interface>(name.c_str())  
    {  
        // In a port, we can get our parameters by using the propertyAPI  
        intParam = scml_property_registry::inst().getIntProperty  
            (scml_property_registry::PROTOCOL, sc_object::name(), "intParam");  
    }  
  
    unsigned int intParam;  
};
```

### 6.3.3 scml\_property\_server\_if

This class defines the interface a property server should implement.

```
virtual long long getIntProperty(const std::string & name);  
virtual unsigned long long getUIntProperty(const std::string & name);  
virtual bool getBoolProperty(const std::string & name);  
virtual std::string getStringProperty(const std::string & name);  
virtual double getDoubleProperty(const std::string & name);
```

A property server should override these interface functions. It needs to provide the value of the property whose name is provided as the argument.

Default implementations are available. They return 0, false, or the empty string depending on the type.

This example shows how to implement a custom property server, based on STL maps.

```
class exampleCustomPropertyServer : public scml_property_server_if {  
public:  
    exampleCustomPropertyServer() { this->load(); }  
    virtual ~exampleCustomPropertyServer() {}  
  
public:  
    // scml_property_server_if  
    virtual long long getIntProperty(const std::string & name);  
    virtual unsigned long long getUIntProperty(const std::string & name);  
    virtual bool getBoolProperty(const std::string & name);  
    virtual std::string getStringProperty(const std::string & name);  
    virtual double getDoubleProperty(const std::string & name);  
  
private:  
    // disable  
    exampleCustomPropertyServer & operator= (const exampleCustomPropertyServer  
                                              &);  
    exampleCustomPropertyServer(const exampleCustomPropertyServer &);  
  
private:
```

```

void load();

private:
    // data members
    map<string, long long> mName2longLong;
    map<string, unsigned long long> mName2unsignedLongLong;
    map<string, bool> mName2bool;
    map<string, string> mName2string;
    map<string, double> mName2double;
};

void
exampleCustomPropertyServer::load()
{
    mName2string[ "HARDWARE.module1.myString" ] = "the string";
    mName2string[ "HARDWARE.module2.sub.myString" ] = "the string";

    mName2double[ "HARDWARE.module1.myDouble" ] = 1.234;
    mName2double[ "HARDWARE.module2.sub.myDouble" ] = 1.234;

    mName2bool[ "HARDWARE.module1.myBool" ] = true;
    mName2bool[ "HARDWARE.module2.sub.myBool" ] = true;

    mName2longLong[ "HARDWARE.module1.myInt" ] = 30;
    mName2longLong[ "HARDWARE.module2.sub.myInt" ] = -33;
}

long long
exampleCustomPropertyServer::getIntProperty(const std::string & name)
{
    const long long r = mName2longLong[ name ];
    return r;
}

unsigned long long
exampleCustomPropertyServer::getUIntProperty(const std::string & name)
{
    const unsigned long long r = mName2unsignedLongLong[ name ];
    return r;
}

bool
exampleCustomPropertyServer::getBoolProperty(const std::string & name)
{
    const bool r = mName2bool[ name ];
    return r;
}

std::string
exampleCustomPropertyServer::getStringProperty(const std::string & name)
{
    const string r = mName2string[ name ];
    return r;
}

```

```

double
exampleCustomPropertyServer::getDoubleProperty(const std::string & name)
{
    const double r = mName2double[ name ];
    return r;
}

```

### 6.3.4 scml\_simple\_property\_server

This class defines an example property server that implements the [scml\\_property\\_server\\_if](#) interface.

```
bool load(const std::string& fileName);
```

Loads the properties from the file. Returns true if the load succeeds, false if an error occurred.

```

virtual long long getIntProperty(const std::string& name);
virtual unsigned long long getUIntProperty(const std::string& name);
virtual bool getBoolProperty(const std::string& name);
virtual std::string getStringProperty(const std::string& name);
virtual double getDoubleProperty(const std::string& name);

```

Returns the value of the property. If the property is not found, a warning message is printed and a default value is returned.

The property files have the following syntax:

```

file ::= {line}*
line ::= typeLine | valueLine | commentLine
typeLine ::= '[int]' | '[uint]' | '[bool]' | '[string]' | '[double]'
valueLine ::= name ':' value
commentLine ::= '#' string

```

where:

<i>name</i>	Specifies the hierarchical name of the property.
<i>value</i>	Specifies the value for the property.

Properties that appear before the first typeLine in the file are treated as int properties.

The following code shows an example of a property file.

```

intproperty: -1
:uint
property1 : 0
property2 : 1234
:string
property3 : This is a string property

```

## 6.4 Reporting

**DEPRECATED:** The `scml2_logging` stream output has been deprecated in Product Version M-2016.12 and redirected to the *FastTrack* infrastructure. However, you can temporarily enable the old behavior by using the environment variable `SNPS_VP_SLS_ENABLE_SCML2_STREAM_LOGGERS`.

This section describes:

- [status](#)
- [stream](#)
- [severity](#)

### 6.4.1 status

`scml2::status` is a simple object that maintains a string value and is used mostly to enable debugging and analysis features. The value of the object can be visualized in tracing views or be used for watchpoints and so on.

The `status` object is a very simple object that holds a status value in string format. It can be used as the base of other higher level modeling objects or it can be used to enable debugging and analysis for a module.

The include file of the `status` object is `scml2/status.h`.

The following sections describe:

- Constructors are available for the `status` object.
- Properties of the `status` object.

The following constructors are available for the `status` object.

```
explicit status(const std::string& name)
```

Creates a new `status` object with the specified name.

The following are the properties of the `status` object.

```
std::string get_name() const
```

Returns the name of the `status` object.

```
void set_status(const std::string& status)
```

Sets the new value of the `status` object.

```
const std::string& get_status() const
```

Returns the current value of the `status` object.

```
const std::string& get_description() const
```

Returns the description for the `status` object.

```
void set_description(const std::string&)
```

Sets the description for the `status` object.

### 6.4.2 stream

`scml2::stream` is a convenience object for logging. It allows a stream like formatting syntax to be used to log messages from models. The `stream` object will send the messages to backend logger objects for processing (for example, sending it to `std::cout`). The `stream` has a name and associated severity which

allows to control the messages that are being logged, for example, to restrict them to a certain hierarchy in the design and or a severity level. There are predefined severity levels for error, warning, debug, note, and so on.

The include file for this object is `scml2/stream.h`.

The following sections describe:

- Constructor available for the `stream` object.
- Properties of the `stream` object.

The following constructor is available for the `stream` object.

```
stream(const std::string& name, const severity& severity)
```

Creates a new stream with the specified name and severity level. For information on severity, see “severity” on page 209.

```
stream(const severity& severity)
```

Creates a new stream with the specified severity level. For information on severity, see “severity” on page 209. The name of the stream will be the name of the current `sc_module`.

The following are the properties of the `stream` object.

```
std::string get_name() const
```

Returns the name of the `stream` object.

```
const severity& get_severity() const
```

Returns the `severity` object of the `stream` object.

```
bool is_enabled() const
```

Returns `true` if the `stream` object is enabled, or returns `false` otherwise. The `stream` object will be enabled in case at least one back-end object requests output from this stream.

All methods that are defined on `std::ostream` are also defined on the `scml2::stream` object. A `stream` object can be used as a replacement of an `std::ostream` object like `std::cerr` or `std::cout`. The `scml2::stream` object will send the output to the back-end logger objects only when the stream is flushed. This is done when `std::endl` or `std::flush` is written to the stream.



`SCML2_LOG` can be used as a convenience macro while checking if a `stream` object is enabled.

For example:

```
SCML2_LOG(myStream) << "Debug output for myStream" << std::endl;
```

In case the stream is disabled, the macro will evaluate to one boolean check. There will be no performance impact caused by example, the operator `<<` or the implicit conversion operators in the debug output.

For performance reasons, the `SCML2_LOG` macro should always be used, or the `is_enabled` flag should be checked before sending output to the stream.

Similarly, the `SCML2_LOG_ASSERT` macro conditionally writes to a stream if its argument evaluates to `false`.

For example:

```
SCML2_LOG_ASSERT(value == 0x1234, mStream) << "Value can not be 0x1234" << std::endl;
```

### 6.4.3 severity

The severity object holds a severity name and value. Each stream object has an associated severity object. For information on the stream object, see “[stream](#)” on page 207.

Lower severity level values mean a higher severity.

The include file for this object is scml2/severity.h.

The following sections describe:

- Constructor is available for the severity object.
- Available properties of the severity object.
- Pre-defined severity levels.

The following constructor is available for the severity object.

```
severity(const std::string& name, unsigned int level)
```

Creates a new severity object with the specified name and severity level value.

The following are the available properties of the severity object.

```
const std::string& get_name() const
```

Returns the name of the severity object.

```
unsigned int get_level() const
```

Returns the severity level value of the severity object.

The following severity levels are pre-defined by the logging library:

- internal\_error (5)
- error (10)
- warning (100)
- note (1000)
- debug (10000)

The severity object has the following static methods to create the predefined severity objects:

```
static severity internal_error()
static severity error()
static severity warning()
static severity note()
static severity debug()
```

## 6.5 FastTrack

FastTrack allows the modeler to trigger messages from within peripheral models. The following distinct severities are defined:

- Error
- Warning
- Info

Each severity has distinct categories. For a complete listing of the categories, see “[FastTrack Categories](#)” on [page 212](#). For information on how to use the FastTrack messages from an end-user perspective, see “[FastTrack Messages](#)” of the *VP Explorer User Guide*.

The SCML2 modeling library provides macros to communicate a message to the FastTrack infrastructure, which will then present the information to the end user.

From the modeler perspective, the information that needs to be passed is limited to the severity, the category, and a user-defined message.

Under the hood, the FastTrack implementation will attempt to collect and present the following additional information (*tags*) which can be presented to the end user:

1. The simulation time at which the message was triggered.
2. The instance name of the object from which the message originated.
3. The instance name of the processor core that triggered the memory access during which the message was triggered.
4. The program counter of the fore mentioned processor.
5. If debug information is available for the software that is running on the processor:
  - a. The function name
  - b. The file and line number of the embedded software source code associated with the program counter
6. If an OS kit is active:
  - a. The name of the Software Context of the processor at the moment the memory access was executed.

The information presented to the end user can assist both in flagging embedded software errors or modeling problems that are detected in peripheral models, and in debugging the problem that is identified.

The modeler is strongly encouraged to flag FastTrack errors or warnings from model code in those cases where the model would be able to detect fault conditions or potentially wrong programming actions.

Adding FastTrack logging messages to the models can aid both the modeler and the embedded software engineer in debugging embedded software or model problems when functional issues are identified during simulation.



**Note** Besides the messages that are explicitly inserted in model code by the modeler, SCML2 modeling objects trigger FastTrack messages when certain events occur (*implicit messages*). For a complete list of events that are logged by SCML2 modeling objects, see “[Implicit FastTrack Messages](#)” on [page 216](#).

The subsequent section outlines the API available to the modeler. For a detailed description on how the collected information can be presented to and processed by the end user, see *VP Explorer User Guide* and *Virtualizer Studio User Guide*.

This section describes:

- [FastTrack API](#)
- [FastTrack Categories](#)
- [Implicit FastTrack Messages](#)
- [Suppressing FastTrack Messages](#)

### 6.5.1 FastTrack API

The *FastTrack* feature is exposed to the modeler via the SCML2 convenience macros, which are defined in the header:

## scml2/tagged\_message\_macros.h

Convenience macros for use in `sc_modules` (where the implicit 'this' pointer refers to an `sc_module`) are as follows:

- `SCML2_INFO` (info category)
- `SCML2_WARNING` (warning category)
- `SCML2_ERROR` (error category)
- `SCML2_MODEL_INTERNAL` (model internal category)

Convenience macros for use outside of `sc_modules` are as follows:

- `SCML2_INFO_TO` (module, info category)
- `SCML2_WARNING_TO` (module, warning category)
- `SCML2_ERROR_TO` (module, error category)
- `SCML2_MODEL_INTERNAL_TO` (model internal category)

where 'module' is a pointer to an `sc_module`. The FastTrack infrastructure will identify the message that is sent as originating from the given `sc_module`.

For a full description of info/warning/error categories, see ["FastTrack Categories" on page 212](#).

The macros presented above can be treated as stream objects, making the addition of FastTrack messages to the existing or new models very easy and low effort.

For example:

```
SCML2_INFO(FUNCTIONAL_LOG)
<< "Initiating DMA transfer on channel " << mChannelID"
<< std::endl;
```

where the category `FUNCTIONAL_LOG` indicates to the end user that this message conveys information about the high-level behavior of the model.

An additional convenience macro `SCML2_ASSERT` will check a given condition, and in case the condition evaluates to `false`:

- A FastTrack error of category `FATAL_ERROR` will be raised.
- The simulation will be stopped.



**Note** As the simulation is stopped using `sc_core::sc_stop`, the simulation execution will not immediately halt, but still continue until the next simulation sync point. Any code following the `FATAL_ERROR` needs to support this.

This allows the modeler to check, report and exit on conditions which might lead to unstable or catastrophic behavior in the model.

For example,

```
SCML2_ASSERT(all_is_fine == true) << "Something very bad has happened" << std::endl;
```



- The message content should be limited to a single line, and needs to be terminated with a `std::endl` character.
- Only one line should be sent to the `stream` object represented by the macro. For example, one should not write:

```
SCML2_INFO(FUNCTIONAL_LOG)
<< "Message 1" << std::endl
<< "Message 2" << std::endl;
```

- There is no need to include a timestamp or object name to the message, these will be added automatically.

## 6.5.2 FastTrack Categories

Based on the severity, the available FastTrack categories are Error, Warning, Info, Model Internal.

Each category has an enum value for use in the API while modeling, and a name which will be shown to the user when the message occurs.

The following categories are available for use with the Error severity:

**Table 6-2 Error Categories**

Category Enum	Category Name	Meaning
GENERIC_ERROR	Generic Error	Generic error raised by the model. Used as fall-back category for messages that are triggered by legacy SCML Stream or VRE logging infrastructure that is redirected to FastTrack.
SCML_INVALID_API_USAGE	SCML Invalid API Usage	Internal use by SCML2 modeling objects only. Points out wrong use of the SCML2 API.
ACCESS_WRITE_RESERVED_VALUE	Access Write Reserved Value	A register or bitfield is written with a reserved value.
ACCESS_WRITE_UNDEFINED_VALUE	Access Write Undefined Value	A register or bitfield is written with an undefined value.
CONFIGURATION_ERROR	Configuration Error	A parameter or combination of parameters on a peripheral model is set to an invalid value or combination of values.
ACCESS_PERMISSION_CHECK_FAIL	Access Permission Check Fail	A memory access to a register/bitfield or memory is not allowed. For example, due to insufficient privileges or due to the current configuration of the peripheral.
ACCESS_WRITE_TO_READ_ONLY	Access Write To Read Only	An attempt to write a read-only memory location is done.

Category Enum	Category Name	Meaning
ACCESS_READ_FROM_WRITE_ONLY	Access Read From Write Only	An attempt to read from a write-only memory location is done.
ACCESS_UNMAPPED_ADDRESS	Access Unmapped Address	An attempt to access from an unallocated memory location is done.
FUNCTIONAL_ERROR	Functional Error	A peripheral model can use this to flag unexpected behavior in the model.
FILE_NOT_FOUND	File Not Found	A file is not found.
FILE_FORMAT_ERROR	File Format Error	A file has the wrong format or cannot be parsed.
STATE_ILLEGAL_TRANSITION	State Illegal Transition	A peripheral model can use this to flag an illegal state transition in a state machine.
STATE_ILLEGAL	State Illegal	A peripheral model can use this to flag reaching an illegal state in a state machine.
BUFFER_UNDERFLOW	Buffer Underflow	A peripheral model can use this to flag a buffer underflow, that is, an attempt to read from a buffer which no longer has sufficient active elements.
BUFFER_OVERFLOW	Buffer Overflow	A peripheral model can use this to flag a buffer overflow, that is, an attempt to write to a buffer that is already full.
UNDEFINED_ERROR	Undefined Error	An error which does not fit any of the other predefined categories.
PROGRAMMING_ERROR	Programming Error	A peripheral model can use this to flag a generic programming error, that is, it is being accessed by software in a way that violates the specification.
FATAL_ERROR	Fatal Error	Flags a fatal error condition, and will terminate the simulation in a clean way.

The following categories are available for use with the Warning severity:

**Table 6-3    Warning Categories**

Category Enum	Category Name	Meaning
GENERIC_WARNING	Generic Warning	General warning raised by the model. Used as fall-back category for messages that are triggered by legacy SCML Stream or VRE logging infrastructure that is redirected to FastTrack.
SCML_IGNORED_CALL	SCML Ignored Call	Internal use by SCML2 modeling objects only. Points out an API call without any further effect (for example, making the same call twice, or canceling an event which was not scheduled or already canceled).
FEATURE_NOT_MODELED	Feature Not Modeled	A peripheral model can use this to indicate that a given feature that is being exercised is not modeled and not planned to be modeled due to for example, the abstraction level.
FEATURE_TBD	Feature TBD	A peripheral model can use this to indicate that a given feature that is being exercised is not modeled yet.
FILE_SW_IMAGE_OVERWRITE	File SW Image Overwrite	During image loading, a previously written location gets overwritten.
ACCESS_IGNORED	Access Ignored	An access to a memory location is ignored or has no effect.
ACCESS_INVALID_PERMISSION	Access Invalid Permission	A warning raised when an access to a memory object is allowed, but is for example, ignored due to mismatching access permission settings.
UNDEFINED_WARNING	Undefined Warning	A warning which does not fit any of the other predefined categories.
CONFIGURATION_WARNING	Configuration Warning	Used to flag setting of parameters in a way that is not advised.
PROGRAMMING_WARNING	Programming Warning	A peripheral model can use this to flag a generic programming warning, that is, it is being accessed by software in a way that does not meet recommendations outlined in the specification.

The following categories are available for use with the `Info` severity:

**Table 6-4 Info Categories**

Category Enum	Category Name	Meaning
GENERIC_INFO	Generic Info	Generic information raised by the model. Used as fallback category for messages that are triggered by legacy SCML Stream or VRE logging infrastructure that is redirected to FastTrack.
FILE_OPEN	File Open	Logs opening of a file.
FILE_CLOSE	File Close	Logs closing of a file.
FUNCTIONAL_LOG	Functional Log	Logs high-level functional behavior of a peripheral model, as an aid during debugging. The level should be such that it makes sense to a software developer.
FUNCTIONAL_LOG_VERBOSE	Functional Log Verbose	Logs detailed functional behavior of a peripheral model, as an aid during debugging. The level should be such that it makes sense to a software developer.
FUNCTIONAL_LOG_INTERNAL	Functional Log Internal	Logs internal behavior of a peripheral model, as an aid during (mainly) model debugging. This level of logging can expose implementation details of the peripheral model.
STATE_TRANSITION	State Transition	Logs a state transition of a state machine.
CONFIGURATION_INFO	Configuration Info	Logs parameter values/changes in configuration via for example, Tcl calls.
SOFTWARE_LOG	Software Log	Used for messages that are emitted directly by software running on a simulated processor, for example, using <i>SemiHosting</i> .

The following categories are available for use with the Model Internal severity.

**Table 6-5 Model Internal Categories**

Category Enum	Category Name	Meaning
SCML_CALLBACK_ENTRY	SCML Callback Entry	Internal use by SCML2 modeling objects only. Logs calling of a registered callback by an SCML modeling object.
SCML_CALLBACK_EXIT	SCML Callback Exit	Internal use by SCML2 modeling objects only. Logs returning from a registered callback by an SCML modeling object.
LEGACY_LOGGING	Legacy Logging	Logs emitted by legacy logging infrastructure like SCML stream.
LEVEL0	Internal Level 0	Lowest level log (least detailed) emitted by model code to aid model debugging.
LEVEL1	Internal Level 1	Emitted by model code to aid model debugging.
LEVEL2	Internal Level 2	Emitted by model code to aid model debugging.
LEVEL3	Internal Level 3	Emitted by model code to aid model debugging.

Category Enum	Category Name	Meaning
LEVEL4	Internal Level 4	Emitted by model code to aid model debugging.
LEVEL5	Internal Level 5	Emitted by model code to aid model debugging.
LEVEL6	Internal Level 6	Highest level (most detailed) log emitted by model code to aid model debugging.

### 6.5.3 Implicit FastTrack Messages

Implicit FastTrack messages are messages that are being emitted by SCML2 modeling objects. The subsequent sections outline which messages are being emitted, so the modeler is aware and does not add similar explicit messages in model code.



**Note** Messages that are emitted because of wrong API usage or configuration of SCML2 objects are not listed here.

The following table lists the implicit FastTrack Error messages.

**Table 6-6 Implicit FastTrack Error Messages**

Event	Category	Message
Calling SCML2_ASSERT(true)	FATAL_ERROR	<i>Message passed to SCML2_ASSERT</i>

The following table lists the implicit FastTrack Warning messages.

**Table 6-7 Implicit FastTrack Warning Messages**

Event	Category	Message
memory_disallow_access_callback with disallow	ACCESS_INVALID_PERMISSION	[read/write] access denied at address [address] on [object]
memory_disallow_access_callback ignored access	ACCESS_IGNORED	[read/write] access ignored at address [address] on [object]
bitfield_disallow_read_access_callback ignored access	ACCESS_IGNORED	Read access ignored on [object]
bitfield_disallow_write_access_callback ignored access	ACCESS_IGNORED	Write access ignored on [object]
bitfield_disallow_read_access_callback with disallow	ACCESS_INVALID_PERMISSION	Read access denied on [object]
bitfield_disallow_write_access_callback with disallow	ACCESS_INVALID_PERMISSION	Write access denied on [object]
Enabling tracing on an scml_clock object	UNDEFINED_WARNING	Warning in scml_clock [object]: disabling optimizations, clock cannot be optimized when tracing

The following table lists the implicit FastTrack Model Internal messages.

**Table 6-8 Implicit FastTask Info Message**

Event	Category	Message
Invoking a write callback on any scml_memory location or register	SCML_CALLBACK_ENTRY	<i>[callback function name] ([object name]): write to [address]: [data]</i>
Returning from a write callback on any scml_memory location or register	SCML_CALLBACK_EXIT	<i>[callback function name] ([object name])</i>
Invoking a read callback on any scml_memory location or register	SCML_CALLBACK_ENTRY	<i>[callback function name] ([object name])</i>
Returning from a read callback on any scml_memory location or register	SCML_CALLBACK_EXIT	<i>[callback function name] ([object name]): read from [address]: [data]</i>
Invoking a write callback on any scml_bitfield	SCML_CALLBACK_ENTRY	<i>[callback function name] ([object name]): write [value] with mask [bitmask]</i>
Returning from a write callback on an scml_bitfield	SCML_CALLBACK_EXIT	<i>[callback function name] ([object name])</i>
Invoking a read callback on an scml_bitfield	SCML_CALLBACK_ENTRY	<i>[callback function name] ([object name])</i>
Returning from a read callback on an scml_bitfield	SCML_CALLBACK_EXIT	<i>[callback function name] ([object name]): read [value] with mask [bitmask]</i>

#### 6.5.4 Suppressing FastTrack Messages

Suppressing a message means that it will not:

- be shown on standard output,
- be recorded to the analysis database, and
- trigger a FastTrack breakpoint.

To suppress specific FastTrack messages, place a comma-separated value (CSV) file called `suppressions.csv` in the simulation working directory.

The format of the CSV file is as follows:

- Each line represents one suppression rule.
- Each line consists out of the following comma-separated values, which reflect the values of attributes of the message that is to be suppressed. For a description of the meaning of the attributes, see the beginning of “[FastTrack](#)” on page 209. The attributes should be listed in the order as outlined in the following table:

**Table 6-9 Sequential Order of the Attributes**

Attribute	Optional (Y/N)	Regular Expression Support	Notes
<i>Simulation Time</i>	Y	N	unit is <i>ps</i> (pico seconds). This can be any one of the following: <ul style="list-style-type: none"><li>• a specific timestamp.</li><li>• a single time range, two numerical values separated with a – character.</li></ul>
<i>Severity</i>	N	N	This can be any one of the following: <ul style="list-style-type: none"><li>• error</li><li>• warning</li><li>• info</li><li>• model internal</li></ul>
<i>Instance Name</i>	Y	Y	
<i>Category</i>	Y	N	The possible values are contained in tables 5.2/5.3/5.4. Do not use the enumerated values for the categories, but the category names as they appear in the <i>Details</i> view of the FastTrack event trace in Virtualizer Studio / VP Explorer.
<i>Message</i>	Y	Y	
<i>Core Name</i>	Y	Y	
<i>Program Counter</i>	Y	N	This can be any one of the following: <ul style="list-style-type: none"><li>• a specific program counter value.</li><li>• a single program counter range, two numerical values separated with a – character.</li></ul>
<i>Software Function</i>	Y	Y	
<i>Software Context</i>	Y	Y	
<i>Software File Line Info</i>	Y	Y	



- Fields that are left empty are treated as wildcards, meaning that the value of this specific field is not considered when determining when a message should be suppressed or not.
- Fields that support regular expressions will only be treated as such if prefixed with the string `regex:`.
- If a range is specified for a numerical field.
  - The lower value should be specified first.
  - The upper value is inclusive.

The following lines extracted from a CSV file illustrates the above:

- To suppress all messages of severity *error*, emitted by HARDWARE.DISPLAY if the program counter of the core that did the access that triggered the message is between 0x210 and 0x214 (inclusive):

```
,error,HARDWARE.DISPLAY,,,0x210-0x214,,,
```

- To suppress all messages of severity *info*, emitted by HARDWARE.DISPLAY, if the message contains the string write:

```
,info,HARDWARE.DISPLAY,,regex:.*write.*,,,,,
```

- To suppress all messages of severity *info*, emitted by any module between 0 and 999999 ps:

```
0-999999,info,regex:.*,,,,,,,
```

- To suppress all messages of severity *info*, category State Transition, emitted by any module:

```
,info,regex:.*,State Transition,,,,,,
```



# Chapter 7

## Functional Coverage

---

This chapter describes:

- [Coverage Semantics](#)
- [SCML Functional Coverage Reference](#)
- [Examples](#)

The *functional coverage* solution part of SCML is heavily dependent on the work done in SystemVerilog on the same topic. In that standard, functional coverage is defined as the user-specified coverage to tie the verification environment to the design intent or functionality. It is in contrast to *code coverage* that can be automatically extracted from the design code. The same coverage issues exist in a TLM model: when looking at the code and function coverage, there is no way to know whether all possible features have been exercised, for example, whether all enumerated values for a bitfield have been covered. This example highlights a second problem: the code that contains the bitfield behavior is not part of the model, many functions of a TLM model are hidden in the simulation libraries through the use of SystemC and SCML modeling objects.

The following questions may come up while testing the model:

- Do the tests exercise all the required functionality in the model?
- Is enough testing being done? How to know that everything is done?

To answer these questions, a modeler can use the SCML *coverage objects* to create a *coverage model* that defines what functionality is expected to be tested. These coverage objects are used by the simulation engine to create a coverage database, from which the coverage report generator can create an HTML report with the coverage results. A Coverage model links the specification to the implementation. It determines what to cover, for example: what register accesses are important, which values for bitfields are relevant, and what interfaces should be tested. The coverage model exists next to the actual model and monitors the test activity versus the coverage specification and reports coverage metrics.

We define three states for the Coverage report. Initially, the Coverage model is unreviewed, leading to some uncovered and unreviewed coverage objects. At this stage, the Coverage report is labeled as *Failed*. After an initial review phase, all coverage objects are either covered or reviewed, at which point the coverage report will be labeled as *In Progress*. Finally, after all to-be-covered coverage objects are covered, the coverage report will indicate a *Pass*.

### 7.1 Coverage Semantics

This section describes:

- [Functional Coverage Constructs](#)
- [Functional Coverage Exemptions](#)
- [Functional Coverage Calculation](#)

### 7.1.1 Functional Coverage Constructs

The SCML *coverage semantics* are derived from the SystemVerilog functional coverage standard. In this standard, the following constructs are defined that are reused in SCML:

- Covergroup: The covergroup construct encapsulates the specification of a coverage model. It defines a set of coverage points and defines the sampling event for coverage points (when to measure).
- Coverage point: A coverage point specifies an expression or variable that is to be covered. A coverage point includes a set of bins associated with the sampled values, which get calculated to a coverage percent.
- Coverbin: A coverbin defines how coverage is to be counted and which values should be covered. A coverage point is covered when all its bins are covered.

Taking these semantics from SystemVerilog to SystemC TLM modeling with SCML, a number of modifications are required:

- Covergroup: In a TLM model, sampling via a covergroup does not make much sense. TLM transactions, signal events, and other activities are not as synchronized as in an RTL model. Therefore, sampling is not done via the covergroup, but is done for each coverage point separately. Moreover, the sampling is implicit for the predefined coverage points; a modeler does not need to care about sampling points.
- Coverage point: A predefined set of coverage points is made available with specific, implicit sampling behavior and attributes for the SystemC and SCML objects. Custom coverage points can be added by the user through a function coverage point where the function determines what should be validated for coverage. The sampling of function coverage points is user defined.
- Coverbin: In SCML, coverbins are not limited to values that should be evaluated before the start of the simulation like in SystemVerilog. In SCML, it is possible to define a coverbin through a function that defines whether a bin is covered or not. This allows to define coverbins for those objects, for which no predefined evaluation is possible.
- Exemptions: As the TLM model functions at a higher abstraction level, not all details of lower-level abstractions are required to be modeled. Additionally, some models are not required to be functionally complete. To indicate these differences in expected coverage between the model specification and the model implementations, exemptions can be put on specific coverage objects.

### 7.1.2 Functional Coverage Exemptions

TLM models often allow for a higher level of abstraction than described in the model specification. As such, the model specification will contain some aspects that are irrelevant for the TLM model. Additionally, a TLM model might not require the full functional capability of the model specification.

To address these deviations from the model specification, a Coverage model allows for exemptions to put on coverage objects to indicate that some deviation from the model specification is (temporarily) expected.

The following exemptions are currently supported:

- *Unmodeled Nonfunctional*

The coverage object is related to some model functionality which is nonfunctional at the required level of abstraction.

- *Logically Reserved*

The coverage object is related to a model object which logically is *reserved*, although the model specification does not explicitly state it as such.

- *Unmodeled Functional*

The coverage object is related to some model functionality which is not part of the functional scope of the TLM model.

- *Tested Externally*

The coverage object is related to some model functionality which is not part of the coverage scope of the TLM model.

- *Untested Safe*

The coverage object is related to some model functionality which is not part of the critical functional scope of the TLM model.

- *Untested Unsafe*

The coverage object is related to some model functionality which is part of the critical functional scope of the TLM model, but currently uncovered.

If any coverage object has this exemption associated with it, the coverage report will not report a *Pass*. This exemption is meant to facilitate the coverage report to transition from a *Failed* state to an *In Progress* state.

### 7.1.3 Functional Coverage Calculation

The report presents two coverage metrics: *Reviewed Coverage* and *Tested Coverage*.

*Reviewed Coverage* gives an indication of the level of coverage for the collection of coverage objects except those that have an exemption (including the exemption *Untested Unsafe*). Once this metric achieves full coverage, the functional coverage report will have the *In Progress* state.

*Tested Coverage* gives an indication of the level of coverage for the collection of coverage objects except those that have an exemption (excluding the exemption *Untested Unsafe*). Once this metric achieves full coverage, the functional coverage report will have the *Pass* state.

Functional coverage is calculated as follows:

$$C_g = (\sum_i W_i \times C_i) / (\sum_i W_i)$$

where:

- $i$  is  $\Sigma$  / set of coverage items (coverage points) defined in the coverage group.
- $W_i$  is the weight associated with item  $i$  (see [set\\_weight](#)).
- $C_i$  is the coverage of item  $i$ .

A coverpoint typically refers to a *feature* to be covered. It is a metric for the coverage of a feature. The overall coverage number of a covergroup is representing how well all features have been covered. Each feature can be more or less covered. The coverage of a coverpoint is calculated as:

$$C_i = |\text{bins}_{\text{covered}}| / |\text{bins}|$$

where:

- $|\text{bins}|$  is the cardinality of the set of bins defined.
- $|\text{bins}_{\text{covered}}|$  is the cardinality of the covered bins: the subset of all (defined) bins that are covered.

A coverbin determines how a feature should be measured. Only when all bins are covered, the feature is considered covered.

## 7.2 SCML Functional Coverage Reference

This section describes:

- [Covergroup](#)
- [Coverage Point Base Class](#)
- [Storage Coverage Points](#)
- [Parameter and Status Coverage Points](#)
- [Clock Coverage Point](#)
- [Signal Port Coverage Point](#)
- [TLM Socket Coverage Point](#)
- [Generic Function Coverage Point](#)
- [Coverbins](#)
- [Default Bin](#)

### 7.2.1 Covergroup

A covergroup is the owner of coverpoints. All derived classes should create coverpoints in their constructor. The covergroup collects all the coverage data and is used to write the coverage database at the end of the test run.

The covergroup can be constructed with the following constructor:

```
scml2::cov::covergroup::covergroup(const std::string& covergroup_name,  
                                     const std::string& test_name,  
                                     const sc_core::sc_module* module);
```

where:

<i>covergroup_name</i>	Specifies the name for the covergroup. It is used by the report generator to refer to the covergroup it is reporting for.
<i>test_name</i>	When a covergroup is used for different test runs, a different test name should be used for each test run. The <i>test_name</i> is used as base name for the log file in which the covergroup will store the coverage results. The generated file will be <code>./functional_coverage/test_name.log</code> and can be passed to the report generator.
<i>module</i>	The <i>module</i> is used as reference for error reporting. All messages reported by the coverage infrastructure in this covergroup will be reported against this module.

To serialize all the gathered data, the covergroup has a `write_log()` API. This can be called once, before any object related to the coverage collection is destructed.

```
scml2::cov::covergroup::write_log();
```

### 7.2.2 Coverage Point Base Class

There is no need to use the coverpoint base class in a cover model, it is only listed here to describe the generic properties of all coverpoints.

The coverage point supports the following methods:

```
void disable();
```

Disables a coverage point. No coverage metrics will be collected in this case.

```
void enable();
```

Undoes the `disable()` call.

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

Adds a comment to the coverage point. The comment is added to the log file and ends up in the HTML report.

```
void set_weight(float w);
```

Specifies the weight for this coverage point relative to the other coverage points, while calculating the overall coverage. By default, the weight is set to 1.

```
void iff(const std::function<bool ()>&);
```

Specifies a condition that should be satisfied, so that the sampling is recorded. The condition is specified via a function that is evaluated on sampling, typically a C++11 lambda function can be used.

```
void set_max_value(const DT& max_value);
```

Sets a maximum value for the object. This API should be used whenever the datatype of a SystemC or SCML object allows for more values than that can be represented in the object. For example, a 3-bit bitfield has a `max_value` seven, even though we use an `unsigned int` to represent the bitfield.

```
void set_auto_bin_max(size_t auto_bin_max);
```

Generates an automatic bin for the coverage point. This API is used when no bin is created for a coverage point. In that case, the infrastructure will automatically generate bins for the coverage point. The number of bins created is controlled via the `auto_bin_max` value. Automatic bin creation will create a bin for the full value range (0 to `max_value`) and configure it as an array of size `auto_bin_max`. This will split the value range for the coverage point equally, over as many bins as specified by the `auto_bin_max` value. The default value for `auto_bin_max` is 1. If `auto_bin_max` is set to 0, no bins will be generated.

```
bin_type& bins(const std::string& name,  
                const std::string& description = "");
```

Adds coverbins to the coverage point.

```
default_bin_type& default_bin(const std::string& name = "default_bin",  
                               const std::string& description = "");
```

Creates a default bin for the coverage point. For more information on default bins, see “[Default Bin](#)” on page 230.

```
void clear_bins();
```

Removes all bins that were added so far, that is, it clears the complete coverage point. This API is useful to overwrite the default bins generated by TLM Creator.

### 7.2.3 Storage Coverage Points

A specific coverage point type is available for each of the SCML2 memory objects:

- `scml2::cov::reg<T>`
- `scml2::cov::bitfield<T>`
- `scml2::cov::memory<T>`

- `scml2::cov::memory_alias<T>`

These are specialized coverage points for all the SCML2 memory objects. These coverage points have implicit sampling on a transaction access, or a regular `put()` call (or assignment) to the memory object. The sampling can be controlled via the `access_type` attribute of these coverage points.

The constructors for the storage coverage points are:

- `reg(scml2::reg<T>& r, const std::string& name);`
- `bitfield(scml2::bitfield<T>& r, const std::string& name);`
- `memory(scml2::memory<T>& r, const std::string& name);`
- `memory_alias(scml2::memory_alias<T>& r, const std::string& name);`

When constructing a storage coverage point, a reference to the original SCML2 storage object should be passed. The typename of the coverage point in the `scml2::cov` namespace should be the same as the typename of the SCML2 memory object in the `scml2::` namespace.



**Note** The name of the storage coverpoint is used by the coverage report generator to create a hierarchical representation of the storage objects. The SystemC hierarchy naming convention is used (using `.`), so `m.reg.A.ENABLE` represents three levels of hierarchy. This is important while adding coverage points to the storage objects already generated by TLM Creator, and to ensure that the extra coverage points end up in their logical place in the coverage report.

The storage coverage points support the following extra method:

```
void access(access_type at);
```

This API can be used to set the `access_type` attribute for the coverage point. This attribute is used to restrict the sampling to specific accesses to the storage object.

The `access_type` can be one of the following values:

- `TRANSACTION_READS`
- `TRANSACTION_WRITES`
- `WRITES`
- `ALL_WRITES`
- `ALL_WRITES_AND_TRANSACTION_READS`
- `WRITES_AND_TRANSACTION_READS`
- `ALL_TRANSACTION_ACCESSES`

The above values of `access_type` are a concatenation of the terms below. For example, `WRITES_AND_TRANSACTION_READS` value covers what is specified for `WRITES` and `TRANSACTION_READS` terms below.

- `TRANSACTION_READS`: samples transactions that return read data on a `read` command (no check whether value is the one stored).
- `TRANSACTION_WRITES`: samples transactions with a `write` command to the object (no check whether the value is actually stored).
- `WRITES`: checks for values actually written into the storage (`put()` calls).

The default `access_type` is `WRITES_AND_TRANSACTION_READS`. This covers those values that are read via a transaction, and all written values.



**Note** Typically `ALL_WRITES` will count accesses twice, once as the transaction comes in and once when the value is stored in the storage object.

For coverage points associated with `scml2::memory` and `scml2::memory_alias`, the default behavior is that the whole range of memory is covered. This means that while sampling, there is no restriction on the location to which a value is written. So, when a bin is created to cover the value `0x10`, the coverage bin will be covered as soon as this value is written to the memory, independent of the location it is written in.

The coverpoints for `memory` (`scml2::cov::memory<T>`) and `memory_alias` (`scml2::cov::memory_alias<T>`) have the following extra APIs:

```
void indices(unsigned long long start, unsigned long long end);
```

Restricts the sampling to a specific range in the storage object, specified by the `start` and `end` byte address.

```
void set_interface_and_offset(const std::string& _interface, size_t offset);
```

Guides the report generator. It enables the report generator to group memories connected to the same interface together, and to sort them according to their `offset`.

## 7.2.4 Parameter and Status Coverage Points

There are specialized coverage points for `scml_property` (`scml2::cov::scml_property<T>`) and `status` (`scml2::cov::status`) objects:

The constructor for these coverage points is:

```
scml_property(const ::scml_property<T>& prop,
              const std::string& name);
```

The template type of the `scml_property` coverpoint should be the same as for the `scml_property` itself. The `scml_property` coverpoint is sampled at initialization of the simulation. Since coverage is about values assigned to the object monitored by the coverage point, there is no check whether the `scml_property` is used. When an `scml_property` coverage point does not have any bins, it is automatically disabled. There is no need to explicitly call the `disable()` call.

```
status(const scml2::status& st, const std::string& name);
```

The `status` coverage point is sampled each time a value is assigned to the `scml2::status` object.

## 7.2.5 Clock Coverage Point

For the coverage of clocks inputs on a module, there is a specialized coverage point:

```
scml2::cov::scml_clock
```

This coverage point has the following constructor:

```
scml_clock(const sc_core::sc_in<bool>& in, const std::string& name);
```

The coverage point refers to the clock input signal on a module, but the coverage monitoring is done on the `scml_clock` instance that is at the source of this clock input signal. The clock coverage point is sampled whenever `set_period()` is called on the clock source. This works for both `sc_clocks`, as well as the clock objects in SCML2.

The clock coverage point has one extra attribute: the time unit in which the values in the coverbins are defined. The time unit attribute can be defined via the `set_time_unit` API:

```
void set_time_unit(sc_core::sc_time_unit tu);
```

The default time unit is `sc_core::SC_NS`.

## 7.2.6 Signal Port Coverage Point

To measure coverage on signal ports of a module, the following coverage points are available:

- `scml2::cov::sc_in<T>`
- `scml2::cov::sc_out<T>`
- `scml2::cov::sc_inout<T>`

with constructors:

- `sc_in(const sc_core::sc_in<T>& in, const std::string& name);`
- `sc_out(const sc_core::sc_out<T>& in, const std::string& name);`
- `sc_inout(const sc_core::sc_inout<T>& in, const std::string& name);`

These are specialized coverpoints for signal inputs and outputs of a module. The template type of the coverpoint should be same as the template type on the port of the module. All types supported by the SystemC signal ports are supported here as well. Sampling is done whenever there is a value change event on the signal associated with the port.

## 7.2.7 TLM Socket Coverage Point

For TLM sockets, there is a specialized coverage point:

```
scml2::cov::socket<T>
```

with constructor:

```
socket(<T>& s, const std::string& name);
```

The template type of the coverpoint should be same as the type of the port. At the moment, only `tlm::tlm_target_socket<BUSWIDTH>` and `tlm::tlm_initiator_socket<BUSWIDTH>` are supported (for a generic payload). Sampling is done whenever a `b_transport` interface is called on the socket, no other interface APIs will be sampled. The `default_bin` will check for any `b_transport` call on the socket, no checks on content are done. There is also no way to use the value bins for a socket coverage point. To customize the bin for a socket coverage point, a custom `evaluator` method is needed. For more information, see “[Coverbins](#)” on page 229.

## 7.2.8 Generic Function Coverage Point

It is possible to extend the SCML coverage solution by using the function coverage point:

```
scml2::cov::function<T>
```

This coverage point allows to create coverage points for features that are not related to any SystemC or SCML object. The type of the function coverage point refers to the values that will be measured for coverage.

The constructor for function coverage point is:

```
function(const std::function<T()>& f, const std::string& name);
```

When constructing the coverage point, a function is specified as `std::function<T()>&f`. This function should return a value of the coverage point type. This function will be called whenever the coverage point is sampled and should return the value that should be evaluated for coverage.

A sample function is:

```
T sample();
```

The function coverage point is the only coverage point without any implicit sampling, so it is required that the `sample()` function is called during the test, or is called by the object for which the function coverage point is created.

## 7.2.9 Coverbins

A coverbin is owned by a single coverage point and counts how many times a coverage point monitored a specific set of values.

A coverbin is always created via a coverage point:

```
auto& binA = cov_pt.bins("bin_name", "bins_description");
```

It can be immediately fully specified:

```
cov_pt.bins("bin_name").values({32,64,128,256}).array(4);
```

Bins are added to the coverage point via the `bins` API. This creates a new bin with the specified name and returns a reference for further refinement. The second argument is a description which will be recorded in the coverage database and will be added to the HTML report by the report generator.

There is no need to use or know the type of the `bins` objects, simply use the C++11 `auto` keyword to leave that to the compiler to figure out.

Since the `bins` API returns a reference and the same is true for all APIs that further refine the bind definition, it is possible to write the full bin definition in a single C++ statement (as shown in the [example](#)).

The coverbin objects have a set of APIs to further define the bin:

```
bin_type& illegal();
```

Indicates that the values specified in this bin are illegal, they are not considered in the coverage calculation of the coverage point. A runtime message will be printed through FastTrack, indicating when an illegal bin has been covered.

```
bin_type& iff(const std::function<bool(void)> &f);
```

Specifies a condition that needs to be satisfied in order to record the coverage sampling. The `iff` function takes a *functor* (or C++11 *lambda function*) that is evaluated each time the `sample` method for the coverage point is used. The function should return `true` if the value is to be recorded, and `false` if it is to be ignored.

```
bin_type& array(unsigned long long &array_size);
```

Splits the values that are specified for the coverbin into an array of size `array_size`. The values are equally spread over the bins, according to the order in which they were added. Values that were added

multiple times are kept as is, in the order they were added. The array call should be the last definition refinement on the bin. It is not possible to add more values after an array call has been made.

The following APIs are available to define the values for the coverbin:

```
bin_type& range(const value_type& start, const value_type& end);
```

Adds a range of values to the coverbin, defined from the *start* and up to and including the *end* value.

```
bin_type& value(const value_type& v);
```

Adds a single value to the coverbin.

```
bin_type& values(const std::initializer_list<value_type>& vs);
```

Adds a set of values to the coverbin. It uses an initializer list to specify the values for readability.

```
bin_type& evaluator(const std::function<bool(const value_type&)> &f);
```

Specifies a function that should be called when the coverage point is sampled, to determine whether the value that is sampled is part of the bin. The function takes one argument: the value that is being sampled. The boolean return value of the *evaluator* function should indicate whether the bin is covered.

The SCML coverage library provides a couple of utility evaluators that can be used in the *evaluator* API of a coverbin:

```
scml2::cov::bit(unsigned idx);
```

This function will return `true` when the bit with index *idx* is set in the value to be covered.

```
scml2::cov::bits(const std::initializer_list<value_type> idxs);
```

This function will return `true` when one of the bits is set in the value to be covered. The bits are defined via an initializer list.

## 7.2.10 Default Bin

A default bin can be created for a coverage point:

```
auto& defaultbin = cov_pt.default_bin("name", "comment");
```

When there are bins created for the coverage point, the default bin will represent all remaining values that are not covered by any other bin. The default bin is not present by default. It needs to be created using the *default\_bin* API on the coverage point. This feature is available for all coverage point types.

- The *default\_bin* optionally takes a name and a comment for the bin (the default name is *default\_bin*).
- There is one default bin for each coverage point.

The default bin has limited support for the coverage bin APIs:

- It is not possible to define values for the default bin.
- It is not possible to create an array for the default bin.

## 7.3 Examples

<code>cov_A.bins("A").values({32,64,128,256});</code>	Creates one bin for four values. As soon as any of these values is sampled, the bin is covered.
<code>cov_B.bins("B").values({32,64,128,256}).array(4);</code>	Creates four bins for four values, each bin contains one value. The coverpoint is fully covered when all four values are sampled.
<code>cov_C.bins("C").range(0, 255).array(8); cov_C.default_bin().illegal();</code>	A set of bins for the value range 0...255 split in eight equally sized bins. All other values are illegal.
<code>auto &amp;Q = this-&gt;B.bins("Q"); Q.value(32).range(3,9).values({4,7,10,20}).array(4);</code>	Creates four bins with values: (32, 3, 4), (5, 6, 7), (8, 9, 4), and (7, 10, 20).  NOTE: The order in which values are added to a bin is important.
<code>this-&gt;A.disable(); this-&gt;A.ENABLE.bins("all values").values({0,1}).array(2);</code>	Disables the coverage for the register A and creates two bins for the single bit bitfield ENABLE.
<code>this-&gt;C.access(scml2::cov::TRANSACTION_READS); this-&gt;C.iff([&amp;t](){return t.A.ENABLE == 1; });</code>	Only cover transaction reads for coverpoint C and adds a condition, so that sampling is only done when enable is set ( <i>t</i> is the model).
<code>this-&gt;P3.bins("allowed_values").values({"AXI", "GFT"}); this-&gt;P3.default_bin().illegal();</code>	Defines bins for the string parameter coverpoint via an initializer list and makes all other values illegal.
<code>this-&gt;A_VAL_writes.bins("S").evaluator([&amp;t](const unsigned int &amp;value) { return (value != 0 &amp;&amp; value == t.B + t.C);});</code>	A bin defined through an evaluator function that checks whether the sampled value is the sum of two other registers in the design.
<code>auto &amp;T = this-&gt;bus_reg_in.bins("transaction check"); T.evaluator([&amp;t](const tlm::tlm_generic_payload &amp;trans) {return trans.get_address() == 0x8;});</code>	A bin on a socket coverpoint checking whether a transaction on address 0x8 was received.
<code>this-&gt;A.ERRCODE.bins("FATAL ERROR").evaluator(scml2::cov::bit&lt;unsigned int&gt;(0));</code>	Creates a bin that checks whether bit 0 in the ERRCODE bitfield of register A is set.
<code>this-&gt;Q_ERRCODE.bins("Minor ERRORS").evaluator(scml2::cov::bits&lt;unsigned int&gt;({0,1,2,3,4}));</code>	Creates a single bin that checks whether bits 0,1,2,3 and 4 are set in register Q_ERRCODE.

```
this->UICCMDARG1.disable_all();  
this->UICCMDARG1.enable();
```

Disables the coverpoints for all bitfields in the UICCMDARG1 register. The disable\_all call also disables the register itself, therefore it is re-enabled via an enable call.

# Chapter 8

## Modeling Guidelines

This chapter describes:

- Requirements for a Virtual Prototype Model
- Virtual Prototype Model Content
- Introduction to SCML FT Modeling
- The SCML Modeling Guidelines for LT
- Synchronization and Modeling for Speed
- Getting Started

### 8.1 Requirements for a Virtual Prototype Model

Virtual Prototype models are developed in order to provide software developers and integration engineers with an abstract model of the system. The goal is to enable them to create application, middleware, and/or driver software, to optimize software performance and to validate and optimized system and software architectures. The key requirements for a virtual prototype model to enable this use case are (taken from the TLM2.0 requirements specification):

- Running real unmodified software

It is important that the object code as it will be compiled for the final system can be executed on the virtual prototype. This implies the use of Instruction-Set Simulators (ISSes) for the processors for which software will be developed.

- Simulation speed

A virtual prototype is a model of a design that will be executed on a host machine. It is important that the virtual prototype can execute software at a speed that is as close as possible to real time. This for example means that it should be possible to boot an OS in a few seconds in order to support driver software development. At the same time, there is a trade-off between simulation speed and temporal accuracy, which implies that for use cases that require a higher level of timing accuracy there will be a speed penalty, although also here the goal should be to achieve the highest possible simulation speed.

- Register accurate

In order to run embedded software correctly, the memory and memory-mapped register layout and content should be modeled.

- Functionally complete

All consequences of the software interaction with the rest of the system should be modeled. This implies how software interfacing with memory-mapped registers influences their content or the content of other memory-mapped registers. It also implies to model the influence of interrupt signals and other sideband signals that have an effect on the execution of software.

- Loosely Timed (LT)

Timing in a virtual prototype is intended to simplify the synchronization of hardware components with software. Timing information is not an indication of timing accuracy for the overall operation of the system. In an LT model, timer interrupts fire roughly at the expected time to successfully boot OS. In general it is important to have an indication of the speed of the hardware interactions with software (through interrupts, timers, and so on) and of how fast register content is updated. However, it is not required to have the exact timing for each and every event in the system in order to enable software development.

- Approximately Timed (AT):

For the software optimization and architecture analysis use cases, the Loosely timed abstraction does not provide with sufficient temporal accuracy. In this case, it is required to add more timing detail to the models. This implies that timing details of the processor need to be modeled in the instruction set simulator, including its memory subsystem (caches, pre-fetch operations, and so on). The goal is that the resulting system provides with enough detail to derive reasonably accurate performance data to decide on optimization strategies, resource mapping and memory architectures.

- Debugging and analysis

The virtual prototype should provide hooks to attach embedded software debuggers, and tools to perform software analysis for the design.

- Performance information

It should be possible to derive reasonably accurate performance data from simulation to enable software performance profiling and optimization. For this, timing annotation information may need to be improved, which may lead to additional functionality of the system that needs to be modeled. For example, the caches and memory controllers of the system should now be modeled more accurately.

- Configuration

Due to the possible speed difference when enabling performance information gathering, a runtime switch is required that enables these additional features.

## 8.2 Virtual Prototype Model Content

When assembling a virtual prototype, it is important to meet the requirements listed in “[Requirements for a Virtual Prototype Model](#)” on page 233. An actual embedded system usually consists of many components, not all of which need to be modeled in a virtual prototype. Some components have a direct relation to the execution of software, whereas others may have no relation to software. Only the components that are important to execute software correctly need to be modeled. Components like built-in self test, analog-digital converters, voltage regulators, on-chip debug interfaces, protocol converters, arbitration units, clock control, and any other block should not be modeled if they do not impact the functionally correct execution of software. A virtual prototype typically contains components of the following types:

- Processor cores

To run the object code of the actual software, ISSes are used, wrapped into a module with sockets for the data communication, interrupt signals, and integrated into the multitasking kernel of SystemC.

Processors that run software which is not part of the current development can be replaced with an abstract functional model of the software. Such a model contains the algorithm with timing annotation and explicit socket accesses to model the data communication with the rest of the system. When creating a virtual prototype, there is no need to model processors. Processor models are typically made available by the processor vendors.

- Memory and Interconnect Hierarchy IP

Communication between different components in a system can be very complex, but depending on the use case it may not be necessary to model all the details of the system interconnect. Instead communication could be limited to reflect the memory-map decoding of the actual system. The data transactions in the model should include all information necessary for correct software execution. For example, information about secure accesses, protection, exclusive access should be modeled. TLM2.0 has provided generic interconnect models and has set the basis for other organizations to develop standard interfaces for industrial interconnect components. For software optimization and exploration use cases, the interconnect model should have sufficient timing detail to allow modeling of the protocol specific timing implications of data and instruction exchanges. Even more accurate models of the interconnect are needed when the focus of the architecture exploration focuses on the interconnect itself, these models should be reusable for the software centric use cases, but are generally replaced by their more abstract versions.

- Memory IP

Obviously, the key component for software execution is to have a memory model. The memory subsystem does not have to include the full behavior of caches and memory controllers. It is possible to limit their function to have the control registers for these components modeled and leave the behavior out. However, if performance analysis of the software is required, then the functional behavior of caches and the memory subsystem should be present to be able to see the timing impact of the different data accesses by software. With SCML, generic memory models are trivial; so no special effort is required for these.

- Memory-mapped components

For virtual prototypes, these components are very important: They allow the functional verification of the embedded software. Internal memory-mapped registers and the behavioral consequences of a register access should be modeled. Model creation for memory-mapped components is one of the key topics of this modeling guidelines manual.

- External interfaces

USB, serial ports, Ethernet, audio and video IO, Camera, Firewire, SIM card and so on are key elements of many system on chip designs. For the virtual prototype development, they are partly memory mapped components, that is, the register interface of these components is important to enable embedded software development. However, in order to test the functionality of these components, the virtual prototype needs to be extended with a model of the 'external world' as seen by the system it is modeling. This external world can be modeled as real world IO by using similar capabilities in the host (for example, USB) allowing the platform to forward the communication to the host, alternatively virtual IO can be used where for example, the file system on the host is used to mimic things like MMC/SD/SDIO, HDMI, SATA.

- Communication components

This refers to DMA, communication bridges between subsystems, shared memory components, accelerators, and so on. These components are important for the part of their behavior that changes the content of memories and memory-mapped registers or how these components can be accessed. As they have an impact on the execution of software, these components need to be part of virtual prototype models.

- Platform synchronization

These components are required for scheduling and real-time software functionality and deal with synchronization in the system. Usually, these components are software configurable; for this part of their behavior they are not different from any other memory-mapped components. The difference is in

the behavior of these components, where it is important to model the correct timing and synchronization of the system. These components are also addressed in this manual.

- Data processing

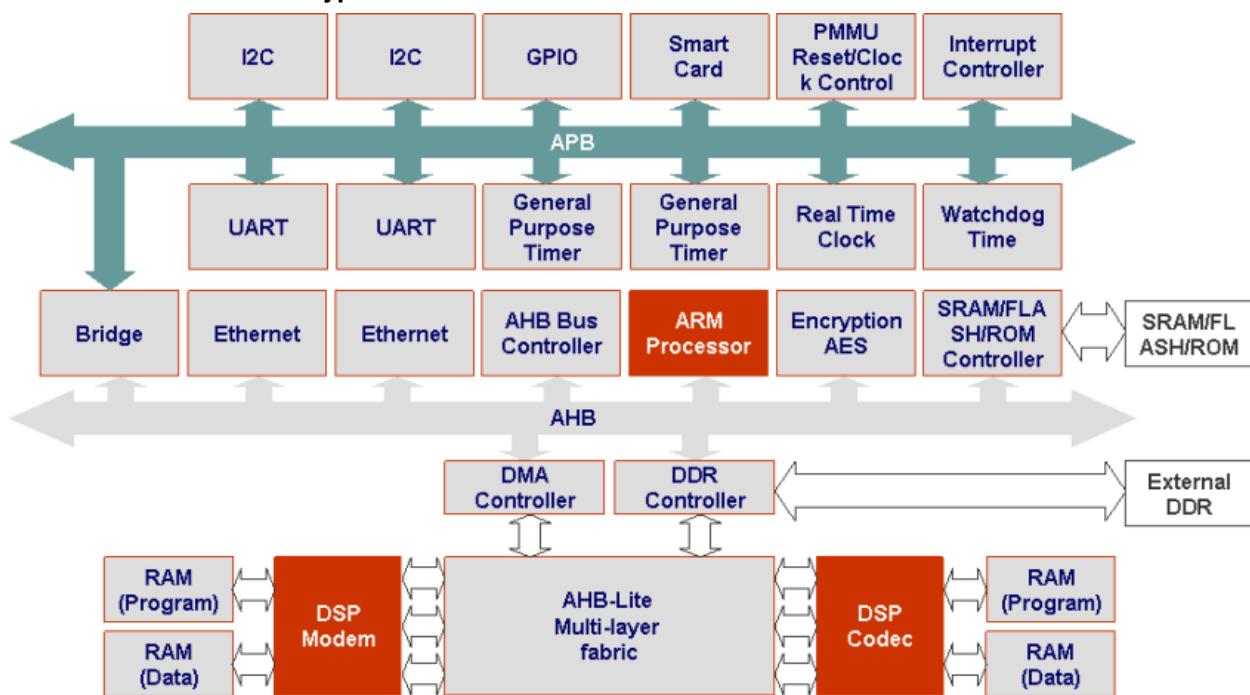
These are all the blocks that can perform certain functionality related to the processing of data in parallel to the execution of software, but that are tightly controlled by it. For the virtual prototype use cases the exact implementation of the processing algorithm may be of lesser importance, the focus of the models for these components should be on the functionality they provide, their register interface and the mechanisms they use to exchange data with the embedded software. Hence, it is possible to develop these models reusing generic data processing libraries as they exist for the host and wrap them with register and data exchange interface models to include them in the virtual prototype.

- Subsystems

As systems get more and more complex a key question when creating a virtual prototype is whether all components and subsystems need to be modeled for a certain use case or not. In general common sense should drive the decision when to stop. If the goal of the virtual prototype is to enable software development on the main application processor of the system, then it may not be necessary to model the wireless modem subsystem, or to have that limited to its data interface.

The following figure shows an example of a virtual prototype.

**Figure 8-1 A Virtual Prototype**



The purpose of SCML is to enable efficient modeling of virtual prototype components. This is achieved by encapsulating certain aspects of the TLM2.0 modeling standard with a generic implementation covering the most common uses of TLM2.0. SCML also provides a number of additional modeling objects that provide model-to-tool interactivity and a set of reusable timing objects.

## 8.3 Introduction to SCML FT Modeling

The purpose of SCML is to enable efficient modeling of virtual prototype components. This is achieved by encapsulating certain aspects of the TLM2.0 modeling standard with a generic implementation covering the most common uses of TLM2.0. SCML also provides a number of additional modeling objects that provide model-to-tool interactivity and a set of reusable timing objects.

The aim of this section is to provide the context for SCML modeling by explaining the background for system-level modeling, TLM, and SCML.

- [SystemC Transaction-Level Modeling](#)
- [Use Cases](#)
- [Fast Timed Modeling \(FTM\) Coding Style](#)
- [Modeling Concepts](#)
- [Creating SCML FT Models](#)

### 8.3.1 SystemC Transaction-Level Modeling

The background to SCML modeling stems from the desire to create an abstract model for an embedded systems design or an MPSOC so that software engineers, system architects, and verification engineers can start working on the design ahead of the actual system, prototype, or RTL design. For this purpose, SystemC has been developed. SystemC is a C++ library containing a set of classes to model system components and their communication interfaces, plus a co-operative multitasking environment to model concurrent activity in a system. Executing the SystemC model allows to simulate the embedded design or MPSOC.

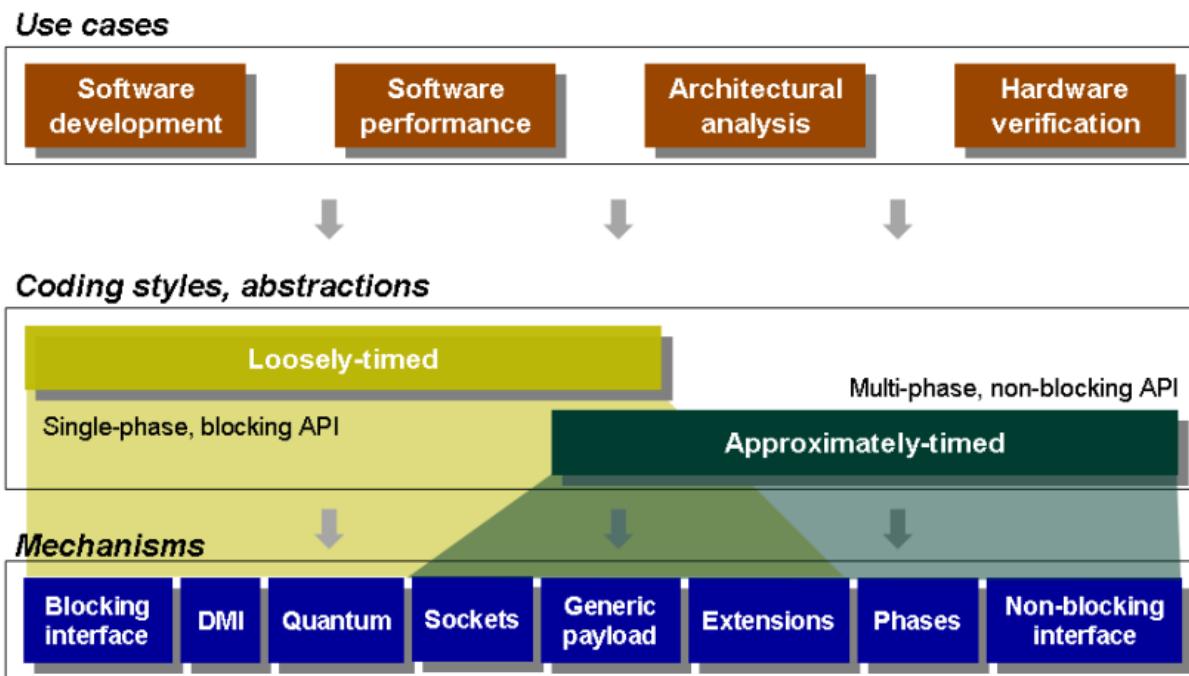
Next to this basic functionality, a Transaction Level-Modeling (TLM) library was developed. This library supports a modeling style where the abstraction for the communication interfaces between system components is not based on the individual wires or wire vectors but on function calls between components and a payload representing the full semantics of the communication interface. This results in a reduction of the number of synchronization points between component models, which improves the overall simulation speed. TLM2.0 provides a standard set of APIs and payload constructs to create memory-map-based TLM models.

There are many ways of using SystemC and TLM2.0 to build abstract models for an embedded design or MPSOC: Software engineers, system architects, and verification engineers have different requirements for the abstract model they need in order to perform their design task; and there are different approaches to create a model according to these requirements. Therefore, a standardized modeling style is required. Such a modeling style is based on use cases, coding style, and modeling objects and interfaces.

- Use cases identify the purpose for building a model and define what a user wants to achieve with the platform. They define the requirements for accuracy, speed, visibility, tool interaction, and so on.
- This manual focuses on the use of SystemC, TLM2.0, and SCML for the creation of virtual prototype models.
- SystemC, TLM2.0, and SCML provide the C++ modeling objects: the mechanisms, interfaces, and semantics to create these models.

The following figure - taken from the TLM2.0 introduction presentation - shows the relation between use cases, coding style, and TLM2.0 modeling objects.

**Figure 8-2 Relation Between Use Cases, Coding Style, and TLM2.0 Modeling Objects**



### 8.3.2 Use Cases

When creating a model for an embedded system, the first question that should be asked is what the use case for the model will be. This will determine which specific type of modeling style should be followed. SCML supports a modeling style to create platform models that supports several use cases.

- Embedded software development use case

In this use case, a virtual prototype is created that models (part of) an embedded system containing one or more processors, possibly running one or multiple OSes to allow the development of applications, middleware, or software drivers. This use case also covers certain aspects of the software performance use case shown in figure 8-2.

Such a model should provide enough details of the hardware system so that it is possible to validate the functional correctness of the software being developed.

Another key aspect of a virtual prototype model is that it should provide an efficient use model for the software developer. This means that, although it is a model for the system hardware, it should allow interactive software debugging and have enough debug and analysis visibility for software development. In a virtual prototype model, hardware functionality can be limited to what is visible or important for the software development; this means that behavior and timing of the different hardware components should be present but not detailed. For a virtual prototype model, it is important that the following aspects are covered:

- The memory and memory-mapped register layout and content.
- How software interaction with memory-mapped registers influences their content or the content of other memory-mapped registers.
- It is also important to have an indication of the frequency of the hardware interactions with software (through interrupts, timers, and so on) and how fast register content is updated.

Virtual prototypes are developed in order to have a model available ahead of the actual hardware. It is important that virtual prototypes can be created quickly and can be maintained to stay in sync with the hardware development. Virtual prototypes are an alternative to the actual hardware since they provide much more control and visibility into the operation of the system.

- Software optimization and evaluation use case:

In this use case, the virtual prototype model of an embedded system that is described for the previous use case is further extended to provide performance data of software execution so that decisions can be made with regards to software optimization and the evaluation of performance constraints. The virtual prototype model needs to be extended with additional timing details to support this use case. This affects the processor model as well as the memory subsystem plus any other timing critical components. It also implies that the model is further enhanced to contain all necessary functional detail to make these performance metrics relevant, for example, the processor pipeline needs to be modeled, as well as instruction and data-caches, any cache coherency strategies that are employed by the system and so on.

On the other hand, this use case does not require the same level of interactive software debugging and analysis as for the development use case. This allows to relax the simulation speed performance requirements, or in different words: since the additional detail does not allow to maintain the same level of simulation performance, it is better to configure the virtual prototype model so that this detail can be left out whenever the focus is on software development, and only have these details activated for batch processing.

For this extended virtual prototype, it is important to cover the following aspects on top of what is listed for the embedded software development use case:

- Instruction execution timing, including the effects of the processor pipelining on the progress of software execution.
- Processor memory interface including caches, cache coherency strategies, pre-fetch buffers and other processor architecture aspects that are added to optimize the software execution performance.
- Interconnect and memory timing: it is important to see the impact of the memory hierarchy on the software execution, as well as the impact of resource contention in case of a multiprocessor system.
- Architecture exploration and HW-SW validation use cases:

These use cases are closely related to the one described for software optimization and evaluation. The difference is that they rely on an even more accurate representation of the overall platform. For example the Architecture exploration use case may use a more accurate representation of the memory subsystem in order to evaluate individual configuration trade-offs for the interconnect architecture and memory controllers, the same holds for cache sizing and other architecture parameters. Basically the requirements for this use case is to have additional flexibility in the model and possibly some additional timing accuracy, but for all practical purposes the modeling style and guidelines for these use cases coincide with the requirements for the software optimization and evaluation use case and will not be treated separately in this document.

Other use cases for SystemC are (these are not addressed further in this manual):

- Functional specification

In this use case, SystemC is used to create a model of a system to validate and/or analyze the interaction of various functional components of the system before they are mapped to a hardware implementation. SystemC provides a co-operative multitasking environment that allows modeling the synchronization of various system components. The focus of this use case is on the synchronization points of the different functional components of a system and (possibly) the relative timing of these components. In

most cases, timing is left out as these models would be used to determine the functionality of a system. SCML and TLM2.0 provide little help for this use case.

- Architecture analysis

In this use case, a model for a system is created to explore different alternative implementations for the system. The focus is either on the dimensioning and optimization of the interconnect and memory subsystem, or on the mapping of functional components on different processing elements and the consequences that has on the interconnect and memory configuration. For this purpose, an accurate model of the interconnect is required so that the impact of different configurations and mappings can be analyzed. This implies that more timing accuracy (typically cycle-accurate models) as well as all behavior of the interconnect and memory subsystem should be modeled. Still it is possible to use abstract models for the initiator and target models in the system, possibly reuse some of the models created for a virtual prototype. SCML and TLM2.0 are key to the development of such a platform model.

- Hardware verification

The goal of this use case is to verify the actual RTL for a subsystem or a system component in the wider context of the embedded system. The SystemC models typically represent processor components or traffic generators that provide the test vectors for the part of the system that is tested. The execution of the subsystem or system component is done through an RTL simulator, emulator, or FPGA board. For this use case, models from the virtual prototype for software development can be reused.

Each of the use cases require to build a specialized model of the platform that is targeted. When creating the models for the individual components of the platform, it is tempting to create a dedicated model, specially crafted for the current design task. However, this should not be generalized. The speed and accuracy requirements of the use cases drive the need for specially crafted interconnect models and to a lesser extend the need for specialized models for memory controllers, processors, DMA, and so on. Many initiator and target models can be developed so that the model or parts of it can be reused between design tasks. Alternatively, a reuse-based modeling methodology also implies that the coding style allows extending a model when reusing it for a design task where more detail is required. Or even more: to create a model that can be configured to have more or less details depending on the design task at hand. Overall SCML targets a modeling style that allows to configure component models to the design task at hand or even to support runtime configurability so that the simulation can be tuned to provide the right level of accuracy for each piece of software individually.

### 8.3.3 Fast Timed Modeling (FTM) Coding Style

Once the use case is known, it is possible to apply SCML to create a model for this use case. For maximal efficiency, a common approach needs to be defined to apply SCML to the use case. A coding style defines a standard approach to use the mechanisms provided by the modeling libraries in such a way that the requirements of the use case are met as efficient as possible. The *IEEE Std 1666 TLM-2.0 Language Reference Manual* identifies the following coding styles:

- Loosely Timed (LT): a coding style that aims to maximize the simulation speed of a model providing synchronization at the level of the different embedded software threads that are being executed in a system.
- Approximately Timed (AT): a coding style that focuses on the timing of the data transactions between different initiators in a system, by providing multiple timing points for each transaction.

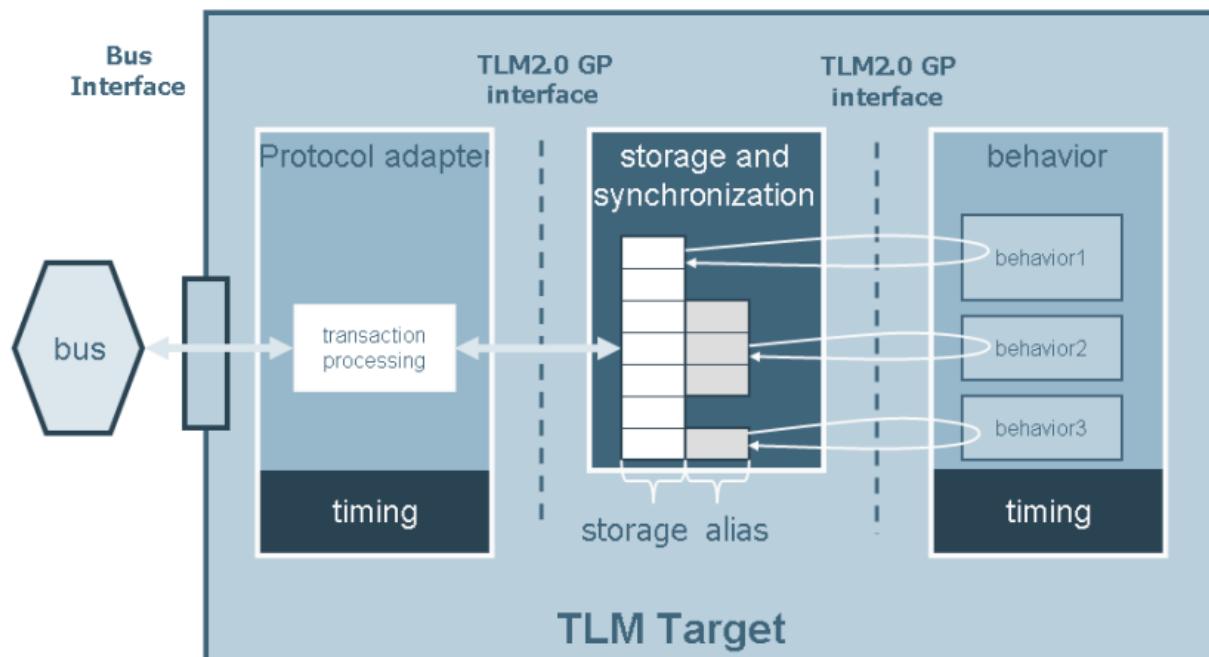
LT and AT are described as coding styles, but obviously they also relate to different abstraction levels, typically an LT coding style will lead to a higher level of abstraction with less timing details, while the AT coding style can be used to create very detailed and accurate models.

The FTM coding style is the name that is used for the SCML coding style that supports both the LT and AT styles. It is defined so that the software optimization and evaluation use case as well as the architecture exploration and HW-SW validation use cases can be addressed. This coding style is based on the TLM2.0 requirement that every model interface always has to support both the LT and AT interface styles. In essence this makes the LT and AT coding style *sub-styles* of the FT modeling style. Originally in the TLM2.0 standard it is assumed that while every component is required to support both interface styles it will only code against one coding style and hence very likely support only one abstraction level. The FT modeling style allows both styles to coexist next to each other for those components that are critical to the overall accuracy and speed of the system. A component modeled in the FT modeling style will in most cases be coded following the LT coding style, when more accuracy might be needed the FT modeling style allows to extend this with an AT style so that a single model can support all use cases.

SCML further refines this with the principle of separating communication, behavior and timing to create a coding style that supports all use cases. It also provides with modeling objects that provide TLM2.0 standard compatible extensions that support the timing accuracy requirements of the use cases while maintaining the interoperability features of the standard.

SCML adds modeling objects that provide an implementation of the key TLM2.0 semantics but also take care of handling the speed and visibility requirements of the use case. When developing the model of a platform component, it is important to create the model so that each of these aspects is an independent element, or in other words: that the model is decomposed into orthogonal properties. The following figure illustrates this coding style. Each element in the figure represents a modeling object or a piece of user code for a target component model.

**Figure 8-3 Coding Style**



The following explains the separation of communication, behavior, and timing in detail:

1. Separating the interface to the interconnect model from the actual behavior of the component. As mentioned in ["Use Cases" on page 238](#), interconnect models typically target a specific use case, and come with their own specialized refinement for the communication interfaces. However, for the actual behavior of the component, a generic TLM2.0 memory-mapped bus interface can be used. The standard

defines a generic protocol which is independent of any actual interconnect implementation. The standard provides an extension mechanism to define proprietary protocols based on TLM2.0. The interface between the extended protocol and the generic TLM2.0 protocol used by the SCML storage objects is done by a protocol adaptation layer.

2. The actual behavior of the component can be separated into a storage and synchronization layer (the internal registers or buffers) and the pure functional behavior of the model (the algorithm or state machine of the component). The storage and synchronization layer communicates through the generic TLM2.0 interface to the protocol adaptation layer and provides an entry point to the component model.
3. Finally, the different needs in timing accuracy can be addressed by separating the code that models the timing of the component from the pure functional behavior. This can be achieved in one of the following ways (in increasing order of effort and accuracy that can be obtained):
  - a. Annotating timing in the storage and synchronization layer
  - b. Implementing explicit timing annotation in the functional behavior based on the state component
  - c. Using a specialized protocol adapter layer that implements explicit timing annotation based on the internal state of the component

SCML supports this separation by providing modeling objects for each of these layers.

The FT Modeling style is based on the TLM2.0 APIs. The modeling style defines a set of model interfaces that are compatible with the TLM2.0 Base Protocol. The standard interfaces are enhanced by adding extensions to model HW-protocol-specific attributes and supports additional timing points for increased temporal accuracy. The FT modeling style also defines a set of modeling rules to ensure a consistent modeling style that provides ease of modeling and ensures a consistent, deterministic behavior. The whole modeling style is setup such that the approach is scalable to support any number of HW protocols without losing interoperability between components that use different protocols or are using them at a different abstraction level (LT versus AT coding style).

A first key aspect of the FT modeling style is that it ensures a consistent and deterministic behavior. This aspect is built on the following modeling rules which are defined for the FT modeling style:

1. Rule-1: All inputs (for example, events, signals or interface method calls) are annotated with the time at which they are received:

```
input_received_time = sc_time_stamp() + local_time
```

where, *local\_time* is the temporally decoupled time of the model (or the timing annotation of the TLM interface calls).

2. Rule-2: All model state changes are only dependent on inputs that were received at:

```
"time < input_received_time";
```

that is. do not look at inputs received at the current time.

3. Rule-3: Outputs (that is, events, signals and interface method calls) can be driven with any time annotation (possibly zero) as a result of a state change.

These are the basic rules for any component, interface or modeling object of the FT Modeling style.

The second key aspect of the FT modeling style is that it maintains interoperability between components of different abstraction level or sub-style (AT or LT coding styles) and also components that are coded for different HW protocols. This is enabled by the fact that the FT modeling interfaces are defined as an extension to the TLM2.0 base protocol. The reuse of the TLM2.0 standard allows one modeling style for both the architecture exploration and software development use cases which maximizes model reuse. It also offers high flexibility in order to choose trade-offs between speed and accuracy at run-time. Most of all it

reduces the need for transactors, it is always possible to fall back to the TLM2.0 base protocol semantics and the FT modeling style objects provide with automated protocol conversion when connecting models coded for a different HW protocol, the modeling objects also provide with abstraction level conversion when connecting models coded for a different modeling substyle (LT versus AT)

### 8.3.4 Modeling Concepts

SystemC, TLM2.0, and SCML provide a wide range of APIs and modeling objects. For a detailed overview of these libraries, see the manuals mentioned in “[References](#)” on page 12. To get started with SCML modeling for virtual prototypes, it suffices to understand the following concepts that are represented by the library elements:

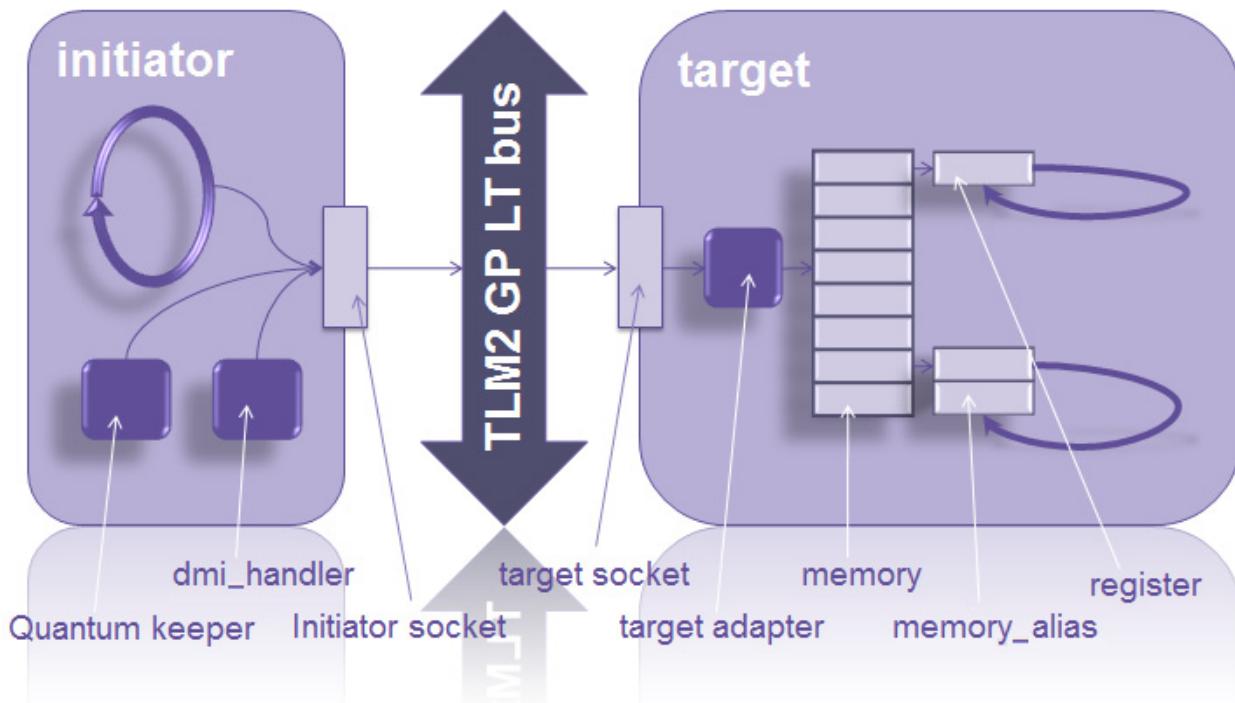
<i>Module</i>	SystemC provides the <code>sc_module</code> base class to enable hierarchical modeling of a system. It represents a component of a system. When a module is defined, it can be instantiated. A module can contain instances of other modules.
<i>Sockets</i>	TLM2.0 and SCML provide socket objects to model the communication between modules. Initiator sockets initiate communication, while target sockets receive communication requests. Sockets carry a protocol, which is a combination of payload and phases. Modules can have multiple sockets. Connections between instances of modules are created by binding sockets.
<i>Interface</i>	Interfaces are the bases for TLM modeling. An interface defines the communication API used by a socket. TLM2.0 defines two basic communication interfaces; sockets need to provide both interfaces: <ul style="list-style-type: none"> <li>• <i>Blocking interface</i>: An initiator calls this interface API and a target implements the API. All communication is expected to happen during this single function call. These interfaces are used in the LT coding style of TLM2.0.</li> <li>• <i>Nonblocking interface</i>: In this case, multiple function calls can be used to model a single data exchange between initiator and target. Both initiator and target can call this interface. Each call to a nonblocking interface has a phase associated with it. <i>Phases</i> are labels for the different time points used to model a transaction. Nonblocking interfaces are used in the AT coding style of TLM2.0.</li> </ul>
<i>Payload</i>	Interfaces carry a payload that contains information about the type of data transaction, as well as the data to be communicated. TLM2.0 provides a generic memory-mapped bus payload. When creating a payload that represents a specific interconnect protocol, the additional information that is not represented in the generic payload of TLM2.0 is added through payload extensions. This is an array of additional data elements for the transaction payload.
<i>Pin</i>	To model simple communication, SystemC provides ports, which are typically used to model pins. Ports are connected through channels that implement the communication behavior. Typically signals are used which represent a vector of data that is exchanged. In the SCML modeling style these are used to model single-bit values, for example, for interrupts or clocks.

<i>Transactor, adaptor</i>	TLM2.0 forbids the connection of sockets that carry different protocols. To make the transition from one protocol to another, transactors are used. These are modules that take care of the protocol conversion. An adaptor has a similar function but it is internal to a module. In SCML, transactors are automatically inserted by protocol conversion logic in the modeling library.
<i>Thread, method</i>	To model concurrent behavior, SystemC provides a multitasking kernel with processes. The execution model is co-operative, which means that a process cannot be interrupted or pre-empted by another process. <ul style="list-style-type: none"> <li>A thread is a process that runs forever and suspends itself and thus hands control back to the multitasking kernel by calling <code>wait()</code>. This allows the kernel to switch to another process. When the kernel switches back to the current thread, it will resume with the statements after the <code>wait()</code> call.</li> <li>A method is a process that executes a function only once. The kernel will execute the method whenever the sensitivity associated with the method is triggered. This sensitivity is defined by event objects. A method cannot be suspended with a <code>wait()</code> call.</li> </ul>
<i>Time</i>	To model time advances in a system, the SystemC kernel provides a global time parameter. Threads can be suspended for certain time; events can be triggered at a certain (future) time point.
<i>Clock</i>	A clock is a SystemC object that contains an event that is triggered at a certain specified rate. It is used to reflect the synchronous behavior of hardware elements in a system. A clock is used to model fine-grain synchronization. SCML provides specialized clock objects with better simulation performance.
<i>Temporal decoupling</i>	The TLM2.0 standard provides a modeling style where different concurrent processes in a system do not synchronize on a clock or on individual data transactions but are running ahead of the SystemC time and keep a local time count. In this case, processes will synchronize when a global <i>quantum</i> value is reached. The quantum represents the system-level synchronization rate for the design. Processes will also need to synchronize when they cannot process a transaction request except if another process is run.
<i>DMI</i>	Direct Memory Interface. This is a simulation speed optimization provided by TLM2.0. A system model can have many modules, threads, methods and functions to model the complete path of a transaction from initiator to target. With the DMI interface, a target can provide a pointer to its internal data storage for the initiator to use, thus avoiding all overhead of the transaction model. DMI is to be used with care but using it can result in a significant speed improvement.
<i>Memory</i>	SCML provides a memory object to model data storage in a component (for example, memory-mapped registers), and to provide interfaces to debug and analysis tools. The object implements the behavior as specified by the TLM2.0 generic protocol and allows overwriting - or adding - behavior through callbacks which will be triggered on a transaction request to this memory object. A memory object defines an array of data elements and can be further refined: <ul style="list-style-type: none"> <li><code>alias</code>: refers to a subrange in a memory object.</li> <li><code>reg</code>: refers to a subrange of size 1.</li> <li><code>bitfield</code>: refers to a range of bits within a single data element.</li> </ul>

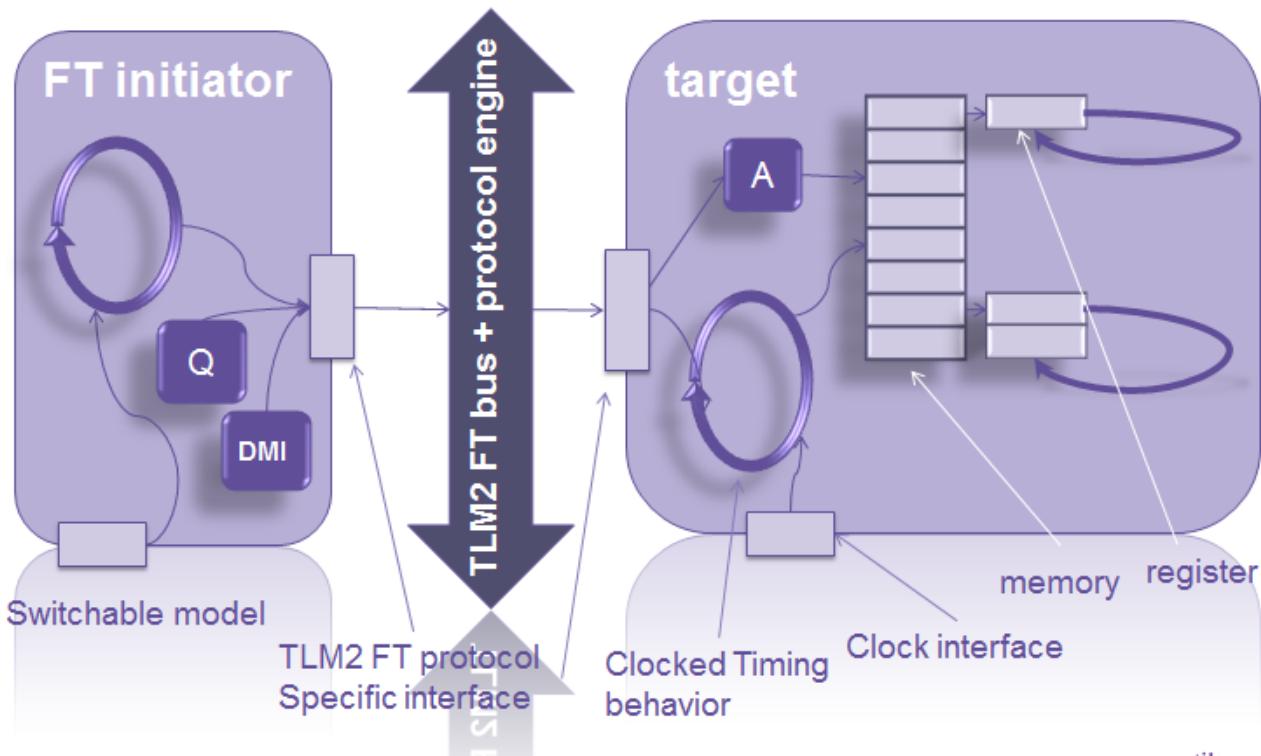
<i>Router</i>	Is an object provided by the SCML library to model the behavior of caches and memory controllers, as well as noncontiguous memory ranges. A router represents a memory-mapped address region through which accesses are done to data further down the transaction path. A router maps incoming addresses to other memory objects or to initiator sockets on a module. This mapping can change during a simulation.
<i>Payload-event-queue (PEQ)</i>	Is a convenience object that allows to model a behavior where multiple transactions are in progress at the same time, the event queue will sort the incoming transaction timing points according to their arrival time (which can include temporal decoupling and timing annotation effects) and provides them for processing to the model at the appropriate moment.
<i>Property, command, status, logging</i>	Are objects provided by SCML that connect a simulation model to the environment in which the simulation will be run. Properties are configurable parameters of a component that can be set through the simulation environment. Commands allow you to interact with the internals of a model from the simulation environment during run time. <i>Status</i> and <i>logging</i> are modeling objects that provide a link to the debugging and analysis environment.

When combining these modeling objects according to the SCML modeling style, the resulting coding style looks as shown in the following figure. Each element in the figure represents a modeling object.

**Figure 8-4 Combining Modeling Objects, LT Case**



**Figure 8-5 Combining Modeling Objects, FT Case**



### 8.3.5 Creating SCML FT Models

There is a considerable effort involved in creating models for all the IP blocks composing a typical hardware platform. Creating good quality models that can be used for the embedded software development use case and thus implement all functions visible by the embedded software and that also run fast enough is not straightforward. Creating models that serve multiple use cases is even harder. The modeling concepts section just barely scratched the surface of all concepts that need to be mastered to be able to develop a model that can be used in a virtual prototype. At the end, the creation of a model is still very much a manually intensive software development process. Therefore, it is very important to follow a process that leads to quality results. Best is to take an approach where prior electronics or embedded systems knowledge is used to understand the requirements of the end user which is then combined with the requirements of modeling efficiently at higher levels of abstraction while avoiding low level details.

The SCML FT modeling style improves the process by providing modeling design patterns and objects so that the overall effort to create the model is reduced. So far the FT Modeling style is proposed as a superset of the LT and AT coding styles as supported by the TLM2.0 standard. The LT and AT coding styles each have their own sweet spot with regards to the use cases that they support. The LT use case is best for the embedded software development use case and the AT use case is more targeting the architecture exploration use case. It is unclear as to which among the two is the best for software optimization and evaluation. As a result, a model created with the TLM2.0 APIs can only fit one use case and a new model needs to be created from scratch to support the other use cases. Models can be integrated into platforms targeting another use case through transactors, but typically these do not provide the necessary additional accuracy so that this approach can be used for components that are critical for the use case. The extensions provided by SCML support a much tighter link between the LT and AT coding styles than is possible in pure TLM2.0. The TLM2.0 standard states that both the LT and AT coding style need to be supported by every socket in a design, but does not offer any additional features to effectively build a model that supports

a wider range of abstraction levels. By using SCML, it is possible to create a component that can be configured so that it can be used for different use cases.

The efficiency improvement coming from the SCML FT modeling style can be further extended through the use of a modeling process that guides the creation of components to maximize the use cases that can be addressed, the quality of the model and reduces the effort to create the model. The key aspects of this process are:

- Use case driven modeling:

Only model what is needed to enable the use cases that will be used. If the model is to be used for embedded software development, then model only what the software needs.

- Iterative modeling:

Model and test in small lock steps. Follow a test driven approach, every time functionality is added an associated test should be developed. This will improve the overall quality of the model.

The ultimate goal of the FT Modeling methodology is to create models that can serve all use cases. This seems an unachievable goal given the performance and accuracy requirements for the use cases that have been discussed. The solution to this is not so much to come up with a single piece of code that miraculously delivers on this requirement, but much more relies on a process to create configurable models that can be tuned to the requirements of the use case as well as to make sure that the additional complexity that is incurred by this configurability is only added when needed.

Not every component of a virtual prototype model plays an equal role to enable the different use cases, only those that are important to multiple use cases and where the additional details are required that can have an important simulation performance impact will need to be configurable:

- Processor cores:

Software execution is a key element of most if not all use cases. For highlevel architecture exploration, it is possible to use an approach based on traffic generation, but as system complexity is increasing actual software execution is becoming more important to have an accurate performance evaluation. Therefore, processor cores need to be configurable and switchable between different levels of accuracy. The development of processor models is a specialized activity that relies on a specific set of technologies and tools and will not be discussed in this document.

- Memory and Interconnect hierarchy IP:

These components are the most critical differentiator for the different use cases, when the focus is on general functionality the interconnect can be simplified to a simple address decoder, however as software performance. Hence, the impact of the memory subsystem on the execution of the software is at the focus of the use case more and more details of the interconnect subsystem need to become available, for example, caching behavior, timing and delay of the communication as well as contention and communication bottlenecks. Therefore, the most flexible and configurable components in the virtual prototypes are the memory and interconnect components. This implies models with switchable accuracy as well as configurable with regards to the functional detail (caches, buffers, and so on) that is enabled in the model.

- Memory mapped components:

Memory mapped components are mostly important for the functional correct execution of the embedded software. Therefore, they can be modeled as register accurate functional models. Timing usually can be added as a latency parameter for the register accesses and possibly a delay model for any interrupts or other system level synchronization they implement. These features can be available in every use case so no configuration is required. As the TLM2.0 standard requires that both the LT and AT interfaces are implemented, these models will typically uplift any AT incoming transaction request into

an LT request, something that comes via the SCML modeling objects, so that there is no additional effort to make these models available for all use cases.

- External interfaces:

It follows the same rules as memory mapped components. Except if the data interface of the component is more complex and is expected to have an impact on the performance or analysis goals of the use case. In such a case, a configurable data interface should be used so that the additional timing details can be enabled when the use case requires it.

- Platform synchronization components:

These components have an impact on the performance of the software execution, but due to their synchronization features they are of key importance to the pure embedded software development use case as well. This means that a single model should be created that provides with an accurate register interface and functional implementation but also with the correct timing accuracy to correctly model the synchronization features of the component. These components do not need to be configurable for different use cases.

As already mentioned, models are best created in a stepwise approach where feature implementation and feature testing are done in lock step. Independent of the model features, the following key steps should always be followed when creating a component model; starting with the creation of what is essentially a loosely timed model and then further extending it with additional timing accuracy.

- Specification study:

For a loosely-timed model, the task involves reviewing the component's specification, identifying the functionality that will be visible to software, or impact other components, and defining the implementation strategy for that functionality including the selection of the SCML and SystemC constructs that will be used. This is the most important step of the model creation. Wrong choices at this point will limit the achievable accuracy and/or simulation performance. The configuration options for the model should be determined at this point. The best starting point for the specification study is usually the programmers' manual of the component since that will explain the features that are visible to the software and give an overview of the programming interface. Extracting timing information may be much harder since these details are not always fully documented.

If they need to be extracted from a detailed description of the hardware component, which will require an additional effort to abstract this to the virtual prototype level in order to avoid modeling all the hardware details of the component which usually results in poor simulation performance. In case the timing information is not documented, the challenge is even bigger since then the information needs to be extracted from an RTL description or by reverse engineering the component through detailed verification tests that help to understand the timing characteristics of the component. Therefore, the best starting point is to first create a loosely timed model so that there is a basis from which to work when extending the component with more timing details later.

- Interface definition:

When starting the implementation of a component model the best thing to start with is to define the interfaces of the model. Usually, this is quite straightforward; from the specification it should be clear how many memory-mapped interfaces are required. Each of these should be modeled with an SCML2 socket. Similarly, the clock interfaces of the component should be easily derived from the specification; these can be modeled using the SCML2 clock interfaces. The other interfaces that are specified should be reviewed whether they are important as synchronizations (interrupt signals) or model to model connections and what their impact will be on the overall software execution. External interfaces can be modeled using real world IO (reusing the IO capabilities of the host) or via virtual IO (for example, using the file system on the host) and thus should not necessarily be available as actual interfaces to the

model. External and other interfaces obviously need to be present when the communication over these interfaces becomes part of the analysis done in the use case.

- Register Interface:

The register interfaces can be added once the black box interface of the model is defined. The register map of a component can be derived from the specification and manually entered as SCML2 code. When the register map is available as an IP-XACT definition or as CSV, the corresponding SCML2 code can be automatically generated using for example, TLM Creator. This tool can be used in general for component modeling and comes with automated IP-XACT import and has a scripting interface that allows importing register descriptions in CSV format. It is even possible to extend this flow using the third-party tools, and generate the SCML2 code from the programmers manual (using PDF or Word to CSV converters) with minimal manual intervention. At this point, it is possible to test the model register interface: a simple unit test can be developed, the model can even be used in combination with a processor to develop the software representation of the register interface.

- Adding behavior:

Next is to add the functional behavior of the component. Behavior can be added in a number of ways, but in order to have good support for the embedded software development use case it is best to use the callback semantics of the SCML2 storage objects and try to associate as much as possible of the functional model behavior with these storage objects. The focus at this step is on the functional behavior of the component, this excludes the timing implications of data exchange and data processing. Behavior that has timing implications through synchronization (interrupts, algorithm sync points) should still be modeled. The decision to add or remove a certain behavioral detail because of the timing implications should be driven by simulation performance considerations and whether it can be added later as a configurable feature, also the performance implications of adding configurability should be considered here.



A configuration also could replace one implementation of the behavior with another one, as such behavior can be modeled using a SCML2 storage object callback for highest performance and be replaced with a clocked callback or SystemC method implementation when the model is configured for more accuracy. As behavior is added a corresponding test should be created to ensure quality results.

Not all behavior might be required to be added at once, it is important to consider how to enable the use of the model as soon as possible, this will help the platform integration effort forward and may enable certain software development tasks already. Furthermore, it will help to determine whether all behavior should be modeled at all, many components are reused in different platforms and may have features that are no longer used, or that are not required for the system under development. Features can always be added to enable a future generation of the platform where they are being used.

- Adding configurations:

Configurability can be added at every step in the model creation. It is possible to make the number of sockets of a model configurable already when defining the model interfaces. Same holds for the register interface and the behavior that is added. The focus of configurability should be to enable wider use of a model. This can be either to make the model more generic and reusable in other platforms or as the bases for components that are very similar but have slightly different specifications. The other case for configurability is to make sure the model can be used for different use cases: here configurability is added to allow performance/accuracy trade-offs that enable different operating points for the component model. Typically, the model will have a loosely timed operating mode where timing information is left at a minimum (none at all, or limited to a delay annotation) versus an approximately

timed operating mode where the AT interfaces get implemented according to the specification to the model and where the data exchange interface behavior is accurately modeled. In many cases, one operating point is sufficient, for example, when the component has a basic data exchange interface that can be modeled with a latency number, else two operating points usually will do. Additional operating points are only useful to enable additional analysis features that have a very negative impact on the simulation performance.

- Adding timing:

The last step is to add timing, if needed. Creating the timing model is mostly important for memory and interconnect IP, in which case the timing model can become quite complex. For the virtual prototype use cases, the timing that needs to be added is related to the way data is being exchanged with other models. The use cases focus on software performance and memory and interconnect architecture optimization or multicore architecture optimization. In all these cases, the critical timing impact is related to the organization of data in memories and how components exchange data and at which rates. Processors are excluded in this discussion, for the internal pipeline behavior of processors is obviously critical to the execution of software as well. Therefore, 'adding timing' effectively corresponds to the implementation of the data exchange architecture of a component. The challenge with this is that this requires the implementation of parallel behaviors and there are no equivalent modeling concepts as there exists for example, for storage modeling. On top of this, the detailed timing behavior may not be documented. Hence rather than specification study, this step is preceded with an RTL implementation study or more often it is driven from reverse engineered specifications extracted from timing verification tests that then are used to validate the virtual prototype model. The SCML FT modeling infrastructure provides with a set of basic modeling objects and paradigms as a starting point for this effort, but still a lot is left to the creativity of the developer.

Following this stepwise modeling approach, it is possible to create virtual prototype component models. Preferably these components are configurable and extensible so that they can be reused in different platforms and use cases or can be modified in order to model slightly different components. Key in all of this is to maximize the reuse of the component model so that the overall effort per component instance in a virtual prototype is minimized.

## 8.4 The SCML Modeling Guidelines for LT

This section describes the rules to create component models for virtual prototypes using SCML. The modeling guidelines and coding style discussed here are focused on the loosely timed coding style. They present the requirements for the FT modeling style model configuration to address the embedded software development use case. These rules are demonstrated through examples in the following sub-sections. All rules that are demonstrated in the examples are listed here. They are split into two lists:

- [Modeling Methodology Guidelines](#)
- [Coding Style Guidelines](#)

### 8.4.1 Modeling Methodology Guidelines

The generic SCML modeling guidelines for virtual prototype components are derived from the requirements to model virtual prototypes described in “[Requirements for a Virtual Prototype Model](#)” on page 233 and the coding style described in “[Fast Timed Modeling \(FTM\) Coding Style](#)” on page 240.

Specification documents for hardware components typically contain a lot of detail that is not important when creating a model that is going to be used in a virtual prototype.

The following table describes guidelines related to modeling only what you need.

**Table 8-1 Rules for Modeling Only What You Need**

Rule ID	Rule Description
1.1	<b>Only that part of the specification that is relevant to the execution of software should be modeled.</b> This is also called the <i>software observable state</i> . It is that part of the specification of the component that is accessible, visible from software, or that can have an impact on the execution flow of software. This means that the section describing the function of the component is important, as well as the part of the specification that describes the programming model with the detailed register layout.
1.2	<b>The detailed I/O interfaces should be abstracted into bus interfaces and modeled using TLM sockets.</b> virtual prototypes are mostly about memory-mapped communication between software and hardware. TLM2 sockets should be used to model these interfaces. Other interfaces should only be modeled when the activity on these interfaces influences the software execution. An example are interrupt interfaces.
1.3	<b>Where possible, timing information should be left out or abstracted to express the software execution rate.</b> There is no need to follow the detailed timing specification of the component. The most important reason to use timing information in a model is to model synchronization with the software and the rest of the system.
1.4	<b>Behavior must be implemented as a simple function call or possibly a state machine.</b>

The simulation performance of a virtual prototype model requires special attention. The goal is to get as close as possible to real-time execution for a model that will execute on a workstation. TLM2.0 defines the standard approach to create models for high-speed simulation. It is based on temporal decoupling and DMI (Direct Memory Interface). SCML enables these features in the SCML modeling objects so that these features are available by default. For more information, see “[Synchronization and Modeling for Speed](#)” on page 259.

The following table describes guidelines related to modeling for speed.

**Table 8-2 Modeling for Speed Rules**

Rule ID	Rule Description
2.1	<b>Temporal decoupling must be used for components that initiate transactions.</b> This typically relates to processor models. Temporal decoupling improves speed since it reduces the number of task switches between threads in a virtual prototype model. When all processor models would run in lock step, there would be a task switch every instruction or clock cycle; with temporal decoupling this will only happen every thousand or more instructions.
2.2	<b>Avoid the use of clock sensitive threads and methods.</b> This rule relates for example to timer components. When modeling a component that is only active irregularly or with large time intervals, it is better to use SCML clock objects or to use the SystemC event scheduler to schedule the exact time point where to execute the model behavior. Since task switches are expensive, it is important to avoid them; this also implies avoiding the use of threads and methods as much as possible.
2.3	<b>Minimize the number of transactions.</b> The number of transactions used to communicate a certain chunk of data determines the number of function calls used in the model (each transaction implies a blocking TLM interface). There is an overhead with each function call and payload that is created. By reducing the number of transactions, speed will be improved. For this reason, the TLM2.0 generic payload models bursts as a single transaction.

Rule ID	Rule Description
2.4	<p><b>Use DMI for all sockets that initiate transactions.</b></p> <p>DMI is intended to reduce the impact of the infrastructure required to do communication between a processor and memory. By avoiding the assembly of a payload and the function call overhead to implement a data transaction, simulation speed is improved.</p>
2.5	<p><b>Optimize for simulation speed when required.</b></p> <p>Follow the strategy discussed in “<a href="#">Modeling Fast Target and Router Peripherals</a>” on page 265.</p> <ol style="list-style-type: none"> <li>Minimize synchronization by selecting the right synchronization level for memory-mapped register callbacks.</li> <li>Optimize the most frequently accessed peripheral first.</li> <li>Use read/write callbacks whenever it is possible to differentiate read behavior from write behavior.</li> <li>Use dynamic registration and removal of callbacks for memory-mapped registers</li> </ol>
2.6	<p><b>Apply C++ coding guidelines for speed.</b></p> <p>All methods that improve the execution speed of software on a workstation apply here as well.</p> <ol style="list-style-type: none"> <li>Reduce the number of data copies required to execute the behavior and communication of the model.</li> <li>Avoid repeated external output (file access, terminal output).</li> <li>Avoid repeated creation/destruction of complex data elements.</li> <li>Avoid dynamic memory allocation</li> </ol>

SCML provides memory modeling objects to model all memory-mapped storage and behavior. SystemC threads should be used for all independent and concurrent behavior in a system.

The following table describes guidelines related to modeling behavior.

**Table 8-3 Modeling Behavior Rules**

Rule ID	Rule Description
3.1	<p><b>Use the SCML memory modeling objects for all memory-mapped storage.</b></p> <p>To separate the communication interface from the behavior, SCML provides memory, register, and bitfield objects. SCML memory modeling objects provide a default storage behavior for all accesses to the objects. These objects can interface with a socket of the module. So in a component model, each memory-mapped socket will have a memory object associated with it. The storage objects can be constructed in a detailed hierarchical model of the memory map of a component. The objects also implement the link to platform debug and analysis tools, and provide DMI access for initiators.</p>
3.2	<p><b>Use callbacks on the SCML memory modeling objects for all memory-mapped behavior.</b></p> <p>The default storage behavior of the SCML memory modeling objects can be overridden through callbacks. To model the impact of software accesses to the registers of a component, there are callback interfaces. Internal values of the component can be updated based on the information that comes with the register access. When a memory access results in communication to another component in the system, the behavior needs to synchronize the overall system state. For this purpose, callbacks can indicate their synchronization requirements when they are registered with the storage objects.</p>

Rule ID	Rule Description
3.3	<p><b>Use SystemC threads for all behavior that is concurrent to the execution of software and that initiates memory-mapped communication.</b></p> <p>In many cases, communication that is a consequence of a memory-mapped access can be modeled within the callback associated with the register access. In this case, all communication and behavior will be instantaneous for the software. Whenever the communication happens in the future or will be spread over multiple time points, a SystemC thread should be used. The thread will allow to model concurrent behavior to the execution of software.</p>
3.4	<p><b>Use SystemC methods to model all behavior that is concurrent to the execution of software and that does not initiate memory-mapped communication.</b></p> <p>SystemC methods are more efficient than SystemC threads for simulations speed. In a SystemC method it is not possible to use a quantum keeper or to initiate communication over a blocking interface like the TLM2.0 LT interface.</p>
3.5	<p><b>Use a regular class method to model the state update for a component.</b></p> <p>A typical behavior for a component is that its I/O and register values depend on the changes of both the I/O pins and the registers. This means that each time the value of one of the input pins or registers changes, all output should be recomputed. In such a case, it is best to create a recomputed method that is used both in the register callbacks as well as in the method sensitive to the input changes. An example of this can be found in the <code>recomputeInterrupts()</code> method of the interrupt controller described in “<a href="#">Modeling an Interrupt Controller</a>” on page 289, or in the <code>recalculateTimeOut()</code> method for the watchdog timer described in “<a href="#">Modeling a Watchdog Peripheral</a>” on page 301.</p>
3.6	<p><b>Use submodules to model recurring behavior.</b></p> <p>When a component is defined as a series of submodules with the same behavior, which together form a unique and well-defined component, it is best to create this as a single component for platform assembly. This implies that it is not necessary to use the usual SystemC or TLM communication mechanisms. Use <code>sc_modules</code> to define the submodules in order to ease debugging, so that naming of signals and internals get an additional SystemC hierarchy level. To communicate from the component level to the submodule, simple variable accesses and class method calls are sufficient. There is no need to introduce ports on the submodules or to use signals to communicate with the submodules.</p>

Modules communicate using TLM interfaces. The goal of TLM modeling is to abstract the details of the pin interfaces between components in order to achieve higher simulation speed and make model creation easier. Virtual prototype modeling is based on the LT coding style of TLM2.0. Communication is done through a single function call from initiator to target, carrying the transaction payload and timing information.

The following table describes guidelines related to communication.

**Table 8-4 Communication Rules**

Rule ID	Rule Description
4.1	<b>Use the TLM2.0 generic transaction payload for all generic memory-mapped communication.</b> In SystemC TLM2.0, a generic payload is defined. It carries the typical information of a memory-mapped bus: address, data, data length, command (read/write), byte enables, response status. The SCML storage objects implement the behavior associated with this payload definition. The callbacks for memories can use the standard payload or also simplified information, in which case the memory objects will handle the semantics that are not forwarded to the callbacks. An adapter is provided that sits between the storage objects and the socket. This adapter implements some of the payload semantics that do not fit with the behavior of the storage objects (for example, separating burst accesses into individual accesses).
4.2	<b>Use a protocol-specific TLM2.0 payload when available.</b> When creating a model for a component that uses an interconnect interface for which a TLM2.0 LT payload or socket is defined, that specialized TLM2.0 payload has to be used. This requires that a dedicated protocol adaptor for the protocol is available in order to connect SCML memory objects to the specialized socket. An interconnect model using the specialized payload must be used to connect the components of the virtual prototype together.
4.3	<b>Use SystemC signals for all sideband communication between components.</b> For all infrequent communication between blocks that is not memory-mapped communication, SystemC signals can be used. It is important to make sure to synchronize before and after reading or writing to a signal from a memory-mapped behavior callback. This guarantees that the signal value that is used is set or seen at the right time in the system, so that the most up-to-date value is seen by all blocks involved.

The final foundation layer of the SCML modeling guidelines is timing. In a virtual prototype, timing information is used to model the synchronization rate of different components in a system. For details, see “[Requirements for a Virtual Prototype Model](#)” on page 233. Adding timing to a model can be done in various ways, but always implies to add more synchronization points into the model, which has an impact on simulation performance. For more information on synchronization, see “[Synchronization and Modeling for Speed](#)” on page 259.

The following table describes guidelines related to timing and synchronization.

**Table 8-5 Timing and Synchronization Rules**

Rule ID	Rule Description
5.1	<b>The timing parameter in the TLM2.0 LT interfaces must only be used for synchronization or to influence the software execution rate.</b> The TLM2.0 communication interfaces carry a timing parameter. This is used to annotate the timing of the data transfer. In a temporally decoupled system, it is the initiator that keeps track of its local time in relation to the quantum. So it will pass its current local time along with the transaction for the target to annotate the delay of the transfer. Whenever a target explicitly synchronizes with SystemC, it can use the value of the timing parameter to advance the global simulation time and set the timing annotation to 0 so that the initiator is aware of the synchronization that happened. A target can also use the value of the timing parameter to implement the delay of a software access to this memory-mapped location. This is not a measure for timing accuracy, but will influence the software execution rate of the software accessing the target versus other software that is not accessing this target.

Rule ID	Rule Description
5.2	<p><b>Increment the local time of an initiator with the execution time of the initiator on each transaction.</b></p> <p>The initiators referred to in this rule are SystemC threads that initiate transactions independently of the software execution. As it is the initiator which controls time in a virtual prototype, the local time parameter of the quantum should be incremented on every access. Before the TLM transaction is started, the local time should be updated to the point before the transaction. After the transaction, the time should be further updated with the timing for the current operation and next the quantum should be checked for synchronization (see <a href="#">5.3</a>). For example, an ISS would make sure local time is updated to the point the current instruction is started, do the TLM transaction next, and after the transaction increment the local time with the delay for the current instruction and then check for synchronization.</p>
5.3	<p><b>There should be a check whether the local time is exceeding the quantum synchronization period after every transaction.</b></p> <p>With every transaction, local time of an initiator can be further incremented by the target components. This will possibly cause an overrun on the quantum value; this should force synchronization with the rest of the system. This is done through the <code>need_sync()</code> and <code>sync()</code> API calls of the TLM2.0 quantum keeper. As explained in <a href="#">5.2</a>, the initiator also should take into account the delay for the current operation. This mechanism will ensure that any signal that is written can propagate to its final destination.</p> <p><b>NOTE:</b> Whenever a signal is written, there may be multiple event-sensitive methods between the original source of the change and its destination. The delay after the transaction makes sure all these event-sensitive methods get triggered before the initiator starts another quantum.</p>
5.4	<p><b>Use the SCML clock objects to schedule regular events.</b></p> <p>The SCML clock objects are a more efficient implementation of clocks than the regular clocks found in the SystemC kernel. Avoid the use of SystemC clocks.</p>
5.5	<p><b>Callback behavior on SCML memory objects must be set to AUTO_SYNCING whenever the correct execution of the behavior requires a synchronized system.</b></p> <p>It is best to use AUTO_SYNCING in all cases and only change when optimizing for speed. Cases when AUTO_SYNCING is required include, but are not limited to:</p> <ol style="list-style-type: none"> <li>1. The behavior queries the global SystemC time using <code>sc_time_stamp()</code> or <code>sc_simulation_time()</code>.</li> <li>2. The behavior reads from an <code>scml_counter</code> object using <code>get_count()</code>, <code>read()</code>, or the <code>()</code> operator.</li> <li>3. The behavior reads from or writes to an <code>sc_signal</code> or any other primitive SystemC channel (either internal to the component or when accessing a pin port).</li> <li>4. The behavior notifies an event.</li> <li>5. The behavior calls <code>wait()</code>.</li> <li>6. The behavior accesses a port or socket that does not support temporal decoupling. This includes using the <code>post()</code>, <code>transport()</code>, <code>post_read()</code>, or <code>post_write()</code> methods of the <code>scml_post_port</code>.</li> <li>7. The behavior models a software synchronization primitive, for example, when the storage is used for software semaphores.</li> </ol>
5.6	<p><b>Use SELF_SYNCING behavior whenever the behavior complies with the following rules:</b></p> <ol style="list-style-type: none"> <li>1. The behavior is not AUTO_SYNCING. This means the behavior requires synchronization, maybe not with each invocation but at least in certain cases.</li> <li>2. The behavior accesses ports or sockets that support temporal decoupling. This means they have a timing parameter according to the definition for TLM2.0 <code>b_transport</code>.</li> </ol>

Rule ID	Rule Description
5.7	<p><b>Use NEVER_SYNCING behavior whenever the behavior will never require synchronization.</b></p> <p>This requires that the behavior complies to both of the following rules:</p> <ol style="list-style-type: none"> <li>1. The behavior is not AUTO_SYNCING.</li> <li>2. The behavior is not SELF_SYNCING.</li> </ol>

## 8.4.2 Coding Style Guidelines

This section describes guidelines related to the coding style.

The following table describes SCML-related guidelines.

**Table 8-6 SCML-Related Rules**

Rule ID	Rule Description
6.1	<b>A component must include the <code>&lt;scml2.h&gt;</code>, <code>&lt;tlm.h&gt;</code>, and <code>&lt;systemc&gt;</code> headers.</b>
6.2	<p><b>The template parameter of a <code>tlm_target_socket</code> and a <code>tlm2_gp_target_adapter</code> must be the same.</b></p> <p>This means that both should have the same bit width defined.</p>
6.3	<p><b>The size of the C++ data type specified as template parameter with an <code>scml::memory</code> object must be the same as the bit width of the adapter and socket it is connected to.</b></p>
6.4	<p><b>For readability, it is advised to name the top-level SCML memory object <code>MemoryMap</code> and to put all internal registers of the module together in a <code>InternalRegisters</code> memory object.</b></p> <p>This will improve the readability of the code, but also ensure that the representation of the model in debugging tools like VP Explorer is easy to use. All internal registers will be grouped next to the memory-mapped registers.</p>
6.5	<p><b>Always make sure there is a visual link between the address values used by software and the index representation of <code>scml::memory</code>.</b></p> <p>Addresses are specified as byte addresses in TLM2.0, while indices in an SCML memory are based on word size (for example, 32 bits, or 4 bytes). To avoid confusion over index versus address values, it is best to put the conversion calculation in the code. This can for example be done by specifying all index values as <code>address &gt;&gt; 2</code>.</p>
6.6	<p><b>Use <code>scml2::initiator_socket</code> to model a socket through which software transactions or transactions concurrent to software execution will be issued according to the TLM2.0 generic protocol.</b></p> <p>In case the transactions are not from a processor executing software, or are not concurrent to software execution, a regular TLM2.0 initiator socket can be used. This avoids an independent quantum for transactions going through this socket.</p>
6.7	<p><b>Use an array of pointers to registers or memories when defining a range of registers or memories.</b></p> <p>To make sure each register can be given a name and to make sure it is possible to easily iterate over a range of registers, you are advised to use a pointer to an array of registers or memories so that each individual register can be constructed and properly named.</p>

Rule ID	Rule Description
6.8	<p><b>Whenever possible, use the same name for pins and sockets in the model as in the specification of the component.</b></p> <p>This will improve readability and testing for the component model. This name should be used for the C++ object as well as the SystemC name passed in the initialization. Exception to this rule is when multiple objects would end up with the same name.</p>
6.9	<p><b>Whenever possible, use the same names for memory-mapped registers and bitfields as specified in the specification of the component.</b></p> <p>This will improve readability and testing for the component model. This name should be used for the C++ object as well as the SystemC name passed in the initialization. Exception to this rule is when multiple objects would end up with the same name. An example is bitfields that are specified with the same name as pins on a component.</p>
6.10	<p><b>Use meaningful names for the callback methods.</b></p> <p>When a method is only used as a callback, it should have a name that indicates when it will be called and for which register. In the examples of this modeling guidelines manual, methods are named according to the following naming template:</p> <pre data-bbox="306 840 1393 872">{on   post} {registerName   bitfieldName} {Read   Write   Transport}</pre> <p>For example, <code>onWdogValueRead</code> or <code>postWdogLoadWrite</code>.</p>
6.11	<p><b>Use meaningful names for the methods that will be made sensitive to events.</b></p> <p>In the examples of this manual, methods are named according to the following naming template:</p> <pre data-bbox="306 1051 486 1083">oneeventName</pre> <p>or for signals:</p> <pre data-bbox="306 1157 600 1189">onsignalNameChange</pre> <p>For example, <code>onTimeOutEvent</code> or <code>onIRQInchange</code>.</p>
6.12	<p><b>Bit numbering in SCML registers.</b></p> <p>When defining bitfields in a register, take into account that in SCML the Least-Significant Bit (LSB) is at position 0. When your specification uses a bit numbering with the Most-Significant Bit (MSB) at location 0, this should be taken into account.</p>
6.13	<p><b>Reusing callbacks for multiple registers.</b></p> <p>Use the tagged version of callback registration when the same behavior (or very similar behavior) is associated with a range of registers. This can be achieved by defining a memory alias for this range and register a tagged callback. The callback itself will have an additional parameter that indicates which specific register index triggered the callback. This allows to identify the register to be used in the behavior, or to differentiate parts of the behavior based on the register that was accessed.</p>
6.14	<p><b>Do not duplicate state.</b></p> <p>When registers, bitfields, or memory represent the state values of a component, it is not necessary to add separate variables in the model to represent the state. This leads to unnecessary copying and raises the risk of nonsynchronized values between the register interface and the internal variables.</p>

Rule ID	Rule Description
6.15	<p><b>When the behavior of a register or bitfield access depends on the old and new value in the register or bitfield, use a regular read or write callback; do not use the post_read or post_write callbacks.</b></p> <p>Since the post_ callbacks store the new value before triggering the behavior, the old value is lost. Adding another variable to keep track of the old value is not necessary since the alternative of using a simple read or write callback avoids this. However, remember that a simple read callback does not store the value automatically, so this should now be part of the behavior.</p>
6.16	<p><b>Use arrays or ranges to model a specification that defines register_0...n.</b></p> <p>It is easier to work with myRegister[0] ... myRegister[n] than with myRegister_0 ... myRegister_n.</p>

The following table describes SystemC-related guidelines.

**Table 8-7 SystemC-Related Rules**

Rule ID	Rule Description
7.1	<p><b>Always specify SC_HAS_PROCESS(<i>className</i>) for each component, either in the class definition or in the constructor.</b></p>
7.2	<p><b>Always make sure all SCML, TLM, or SystemC objects are given a name.</b></p> <p>This name is preferably the same as the C++ object. This will improve readability and debugability of the code.</p>
7.3	<p><b>When using sc_core::sc_time variables, it is best to construct these only once and reuse them in the component behavior.</b></p> <p>Constructing sc_time objects is an expensive operation for speed. It is important to avoid constructing these objects each time you need them.</p>
7.4	<p><b>To finish a simulation always use sc_stop().</b></p> <p>Never use exit(). The exception to this rule is the case that an error condition is triggered in the model that forces an immediate exit to avoid the model to core dump or to misbehave.</p>
7.5	<p><b>Initialization of output ports should be done at end of elaboration.</b></p> <p>Pin ports communicate through signals. These are separate objects that are instantiated separately. To be certain that the signal has been constructed, the initialization of output pins should happen at end of elaboration.</p>
7.6	<p><b>Retrieve the clock period of clocks at or after the end of elaboration.</b></p> <p>Clocks are connected through signals, similar to the input and output pins. This means the information about the actual SystemC clock or SCML clock that is connected to an input clock port is only known at the end of elaboration. It is best to store the clock period in an internal variable since the construction of sc_time objects is expensive; so this value can only be initialized at or after the end of elaboration.</p>
7.7	<p><b>Writing to pins and signals from callbacks.</b></p> <p>It is possible and advised to assign a value to an output pin or a signal from within a callback. There is no need to use a separate SystemC thread to do this. In RTL coding styles, signals and pins should be assigned from a single source to avoid multi-driver conflicts. The RTL simulator gives an error when multiple threads or methods assign the same signal. This ensures that in the actual hardware that is modeled, logic is present to decide what the final value for this signal should be. This rule does not apply in virtual prototype models and the simulator will not issue an error when this happens.</p>

## 8.5 Synchronization and Modeling for Speed

The modeling style presented in this manual provides a standardized approach to create models for virtual prototype simulations. The intent of the guideline is to present an easy-to-use approach that delivers on the speed requirements for virtual prototype models, so that the developer can concentrate on making sure the model is functionally complete, has the right timing accuracy and is register accurate. The FT modeling style that has been presented is created to lead to models that deliver on the simulation performance requirements. However, it should be expected that simulation performance always needs to be a key consideration when creating models. In some cases however there may still be a need. Besides that there is always a reason to look for further speed optimization. The goal of this section is to provide some background and guidance so that a virtual prototype model can be further optimized for speed.

- LT-Centric Simulation Techniques Overview
- Debugging Temporally Decoupled Systems
- Modeling Fast Target and Router Peripherals
- Optimizing Simulation Performance for FT Models

### 8.5.1 LT-Centric Simulation Techniques Overview

The first speed optimization available in TLM2.0 and the Synopsys SystemC simulation kernel is a Direct Memory Interface (DMI) infrastructure. This infrastructure basically provides the initiators with a direct *backdoor access* to all storage elements in the system (memories, registers). As explained in the IEEE Std 1666 TLM-2.0 standard, an initiator can request to get immediate access to the storage within the target for simple storage accesses (read and write). The TLM2.0 infrastructure provides with means for the target to enable and disable this direct access. An interconnect component should forward the DMI requests.

In a virtual prototype model, it should be possible to configure the interconnect model to be as simple as an address decoder. Even in that case, the overhead of TLM function calls for communication is too big for high-speed simulations. So this is definitely an issue when using more refined bus models with more accuracy, for example, when adding more interconnect components to increase the accuracy of the functionality that is modeled or when adding more accurate timing information in the model.

So to have a configuration mode that takes full advantage of the speed optimization, the bypassed models (buses, transactors) should not consume any simulation time in idle mode. This can be achieved by an event-driven modeling style or by applying model gating, that is, disabling the clocks in idle mode, or as is done with the SCML2 clock objects: simply by de-registering the callback methods for the clock or by not triggering them.

At any time, it should be possible to switch between the speed-optimized simulation mode and the full simulation mode. The speed-optimized simulation mode uses the fast backdoor access to the target, while the full simulation mode simulates the complete detail of the bus transaction. It is only useful to switch from optimized mode to full simulation mode to run fast through some initialization sequence and then switch to a more accurate simulation mode. Switching from full mode to optimized mode is not useful since typically it is unsafe because of unfinished transactions stored in the buffers of buses and transactors.

The switch of the simulation mode can be initiated from the SystemC debugging tools VP Explorer and SystemC Shell:

- In VP Explorer

Right-click on the system in the SystemC Design Browser (this is the root element for the design) and from the pop-up menu, select either *Switch to full simulation mode* or *Switch to speed-optimized simulation mode* (depending on the current simulation mode).

- In SystemC Shell:
  - To switch the abstraction level to full simulation mode, call:

```
::scsh::set_full_simulation_mode
```

- To switch the abstraction level to speed-optimized simulation mode, call:

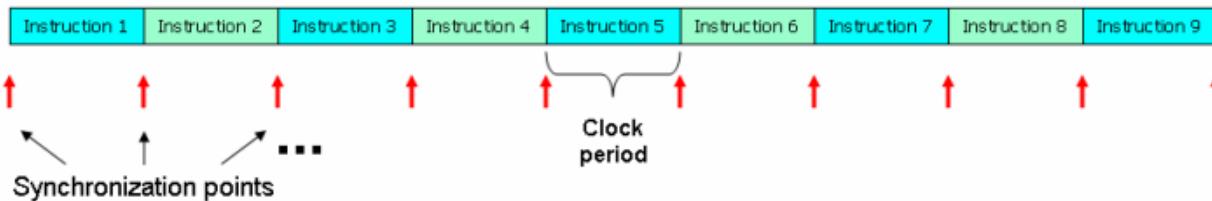
```
::scsh::set_speed_optimized_simulation_mode
```

On its own, the DMI would not have a big impact on the simulation speed. The full potential can only be realized in combination with temporal decoupling. Together, the two techniques reduce the interaction of the platform model with the SystemC kernel to the absolutely required minimum.

The DMI infrastructure described in “[LT-Centric Simulation Techniques Overview](#)” on page 259 provides a high-speed communication mechanism, which bypasses all SystemC interconnect models. This alone does not yield high simulation speed as long as the ISS is activated for every single instruction by the SystemC scheduler. In many cases it is not required for a platform simulation to have this rather accurate temporal granularity. A processor reads and writes to memory most of the time, without any impact on any other hardware block in the system, so there is no need to synchronize at every clock cycle, instruction, or transaction. The frequency with which software running on a processor synchronizes with the software on other processors or with the hardware in the system is much more infrequent than a clock cycle. To take advantage of this, *temporal decoupling* has been defined and standardized in TLM2.0.

A conventional way of building a SystemC simulation is to have a block synchronizing with the rest of the system at the granularity of a clock cycle or individual instructions. The benefit of this is that all internal state and I/O of all hardware blocks is synchronized at each clock cycle or each transaction. In the figure below this is shown by the red arrows.

**Figure 8-6 Clock-Cycle-Based Synchronization as in Full Simulation Mode**



A typical implementation of this is to have an SC\_METHOD or SC\_THREAD process, which is sensitive to an external clock signal. The process is activated by the SystemC scheduler on the raising edge of the clock signal, executes one control step of the ISS, and suspends. In the context of temporal decoupling, such an ISS is called fully synchronized with the SystemC scheduler, because the local time in the ISS is always identical with the global SystemC time. In general, a SystemC process synchronizes with the SystemC kernel by calling `wait()` in case of an SC\_THREAD or `next_trigger()` in case of an SC\_METHOD.

To improve the simulation speed, TLM2.0 standardizes temporal decoupling. This is implemented in the fast ISSes in the Synopsys Virtualizer. As shown in figure 8-7, the concept of a quantum is introduced, that is, the simulation kernel allows the ISS to run a certain amount of time without synchronizing with the SystemC scheduler. This means the SystemC thread containing the ISS executes so many instructions and only at the end of the quantum calls a SystemC wait. During the execution of such a quantum, the ISS basically runs ahead of the global SystemC time. The quantum in the example of figure 8-7 is assumed to be only four clock cycles, but the default duration in a real simulation is in fact 10,000,000 ns.

The *quantum* is a global value for the whole simulation and it is maintained by the quantum keeper. In many cases this could be a fixed value indicating the system-level synchronization for a certain platform and the software running on it. Therefore, it is possible to configure the value for the quantum for a certain simulation run. To improve on this concept, the Synopsys simulation engine introduces a quantum that is controlled by the simulation kernel. When starting a new quantum period, the kernel can make sure that the quantum will be smaller than or equal to the time to the next scheduled event in the kernel. This allows to automatically size the quantum with for example the period of a timer. On top of that, the quantum will terminate before the default duration is elapsed under the following conditions:

- The default duration of the quantum is expired. This is as in the default TLM2.0 definition.
- A SystemC event occurs.

 **Note**

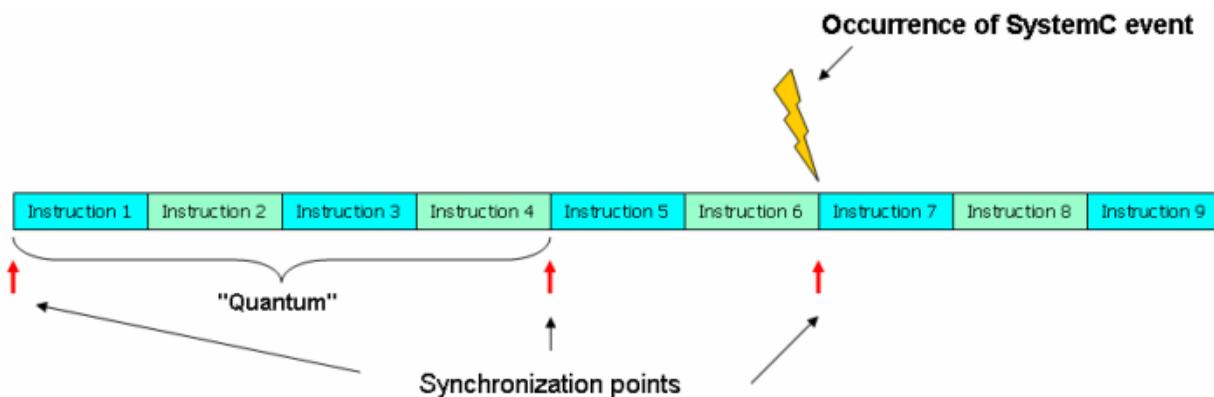
This refers to the point in time, where the SystemC kernel activates the process(es), which is (are) sensitive to the event. It does not refer to the notification of the event, that is, the point in time where a model calls the *notify()* method of the event (see B.70 and B.75 of [IEEE-1666]).

- A peripheral component, which requires synchronization, is accessed. This is as in the default TLM2.0 definition.

Whenever the quantum is "broken" by any of the above conditions, a new quantum will start with the default size or until the next known event. By the support to break a quantum whenever a SystemC event occurs, the Synopsys simulation engine removes the need to determine the size of the quantum before running the simulation. Typically, such a SystemC event is a timer interrupt being signaled or some other system-level event that otherwise should be calculated by the user and passed to the simulation.

The example shown in the following figure assumes that a SystemC event occurs after two cycles into the second quantum. This forces all ISSes in the platform to synchronize with the SystemC scheduler, because it may affect the software running on the ISSes. This may for example be the case if the event occurrence causes the firing of an interrupt. This way the quantum will dynamically resize to align with the system-level synchronization points.

**Figure 8-7 Temporal Decoupling with an Occurrence of a SystemC Event After Instruction 6**



The dynamic quantum is very important to enable a correct simulation. On the other hand, the breaking of the quantum by any event occurrence can have a serious impact on the simulation speed. One careless use of a clocked component forces the quantum's length to be one clock cycle, which completely ruins the simulation speed. You are strongly advised to use notifications of SystemC events with great care and avoid them whenever possible.

As explained with the interrupt controller example in “[Modeling an Interrupt Controller](#)” on page 289, the synchronization property is specified with the callbacks of the registers. By default, a callback should be specified as AUTO\_SYNCING. This means that before and after the callback the quantum will be broken, or in other words, the SystemC time and the local time will be synchronized (so local time is 0; the initiator is no longer running ahead of the SystemC simulation). AUTO\_SYNCING ensures that in a callback it is possible to:

- Query the SystemC time, for example by using calls to `sc_time_stamp()`
- Read and write SystemC signals
- Use the `notify()` methods of SystemC events

The synchronization ahead of the callback ensures that queries of the SystemC time return a correct value, as will a read from a signal give its most up to date value. By synchronizing after the callback, any writes to SystemC signals will be updated for all other initiators to see. The same is true for the event notification.

Alternatively, it is possible to set the synchronization of a callback to SELF\_SYNCING. In this case, no automatic synchronization will be implemented. It is possible to implement the synchronization at any point in the callback by calling `wait()` with the local time as parameter. This enforces synchronization between SystemC time and local time; it is important to reset the local time parameter to 0 after such a call. The local time is the parameter that is passed as an argument to the transport and read/write callbacks.

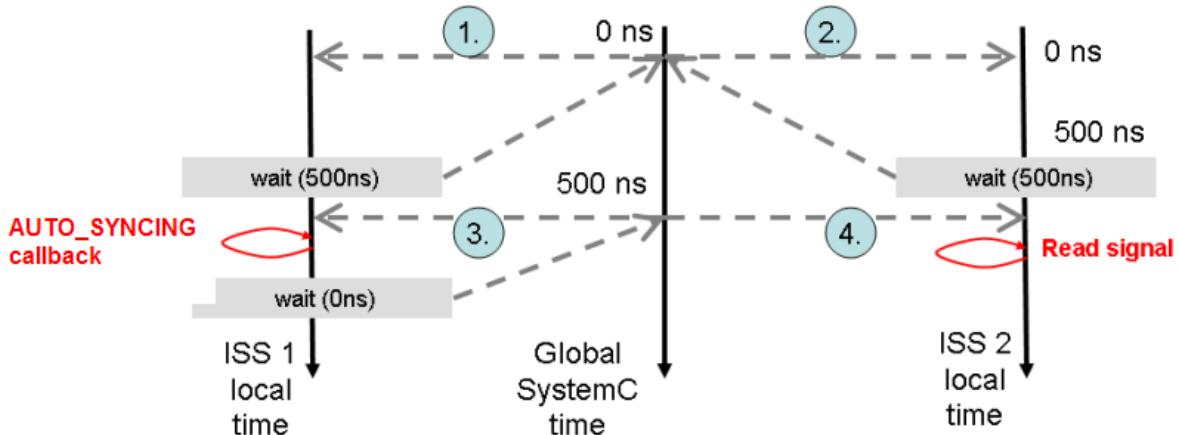
When no synchronization is needed at all, the synchronization option NEVER\_SYNCING can be used to provide better readability of the code.

### 8.5.2 Debugging Temporally Decoupled Systems

This section discusses temporal decoupling by means of example scenarios.

Special care is required when using temporal decoupling in a multiprocessor system. This is illustrated in the example shown in the following figure, assuming a default quantum that is much larger than 500 ns.

**Figure 8-8 Temporal Decoupling in a Multiprocessor System: Scenario 1**



As shown in the above figure:

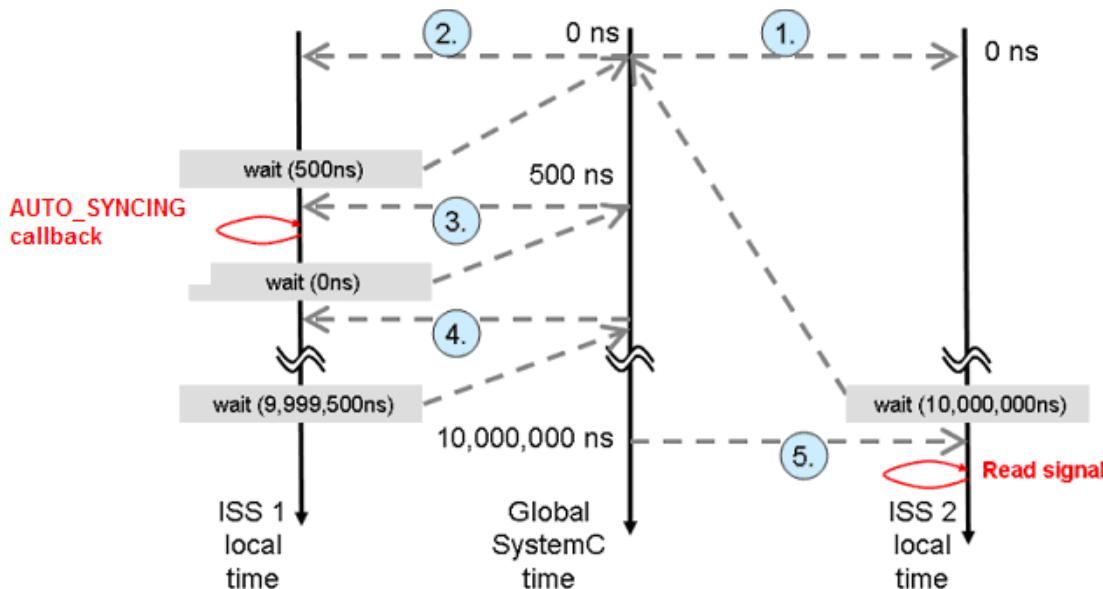
1. ISS1 executes for 500 ns in a temporally decoupled mode and then triggers a hardware block that wants to write an interrupt for ISS2 in an AUTO\_SYNCING callback. As a result, the quantum for ISS1 terminates before the access to the hardware block happens and ISS1 synchronizes by calling `wait(500ns)`.

2. The SystemC kernel immediately activates ISS2 with a reduced quantum of 500 ns (there is an event scheduled at 500 ns; so the quantum size is limited to that point). The ISS runs in temporally decoupled mode and then synchronizes.
3. After 500 ns, the SystemC kernel reactivates ISS1, which performs the access to the peripheral in the AUTO\_SYNCING callback. The behavior inside the callback writes the SystemC signal and after the callback, ISS1 again terminates its quantum by calling inc() and sync() (which amounts to a wait(sc\_zero\_time)).
4. The SystemC kernel activates ISS2, which handles the signal at the correct time and then executes for the duration of the next quantum.

In this case, everything works as expected. However, the sequence of activation between ISS1 and ISS2 is nondeterministic. Therefore, there is a 50% chance that the ISSes are activated in the opposite order.

The following figure illustrates the scenario where ISS2 gets activated first.

**Figure 8-9 Temporal Decoupling in a Multiprocessor System: Scenario 2**



As shown in the above figure:

1. ISS2 executes for the duration of the default quantum and then synchronizes with SystemC time. The default quantum is 10,000,000ns.
2. ISS1 executes for 500 ns and then triggers the AUTO\_SYNCING callback. Like in figure 8-8, ISS1 synchronizes by calling wait(500ns). However, in this case ISS2 has already executed up to 10,000,000 ns, so its next activation will continue from that point forward.
3. After 500 ns, the SystemC kernel activates ISS1, which performs the access to the peripheral with the AUTO\_SYNCING callback. The behavior of the callback writes to the SystemC signal and ISS1 immediately yields to the SystemC kernel.
4. ISS2 remains in waiting mode, so ISS1 is reactivated and executes for the duration of the default quantum.
5. At 10,000,000 ns, ISS2 is reactivated and only now it sees the interrupt signal.

In this second example, ISS2 reacts too late to the interrupt signal from ISS1. As a result, the system may not behave as expected, although the platform model as well as the software running on the platform is correct. In this case, there are a couple of ways to address this issue:

- Reduce the default size of the quantum. This is the brute-force approach. The quantum can be reduced to 0 to make sure the two cores run in lock-step. At this point, the software in the example above would run correctly. Obviously, a trial-and-error approach to find a quantum value that delivers maximal speed can be used as well. Even in the case the quantum is set to zero, it is possible that software that runs correctly on a real system does not run on the virtual prototype. This could be due to the relative speed of software on different cores. On average, one core could be executing more instructions per cycle than another. This could be due to architectural differences between the two cores, but also due to differences in the memory subsystem for each core (caches, interconnect, memory controller).
- When a problem is identified to be caused by quantum effects due to scheduling order, it is possible to use the virtual prototype debugging tools to enforce synchronization only for the points where the interrupt will be set. In other words, breaking the quantum for those points in the simulation where this matters. This can be done by setting a breakpoint on the simulation time where the interrupt will be set. In the Synopsys simulation environment, a breakpoint will cause the simulation to synchronize. This breaks the quantum.

If the exchange would not be through interrupts but in case ISS2 reads a signal through some software access, then the peripheral access in ISS2 is by default AUTO\_SYNCING. This means that it will force the quantum to break at the point the software reads the signal. There would be a synchronization before the callback is called. This would allow ISS1 to run forward to the same time point. ISS1 will set the signal through another AUTO\_SYNCING callback which will make sure the updated signal value is available before ISS2 continued with its own callback.

If these accesses are not for signals but for variables in a memory it does not make sense to make all accesses to that memory AUTO\_SYNCING. In that case it is better to reduce the size of the quantum. However, such a case points to an unsafe approach to synchronization between the two software applications on these processors. In real life it is very hard to predict that a variable access will happen exactly at the time required to ensure write-read order. Typically, such variables should be guarded by a mutex or semaphore, which in turn will typically rely on interrupt signals to ensure safe access. If that cannot be done it is better to try to make all accesses to the memory region that contains the semaphores AUTO\_SYNCING accesses.

Unfortunately, some of the issues introduced by temporal decoupling are hard to pin down.

For example, a callback which is incorrectly registered as NEVER\_SYNCING can be easily overlooked. In fact, the simulation may still behave correctly until a change in the platform or a change in the random order of initiator activations suddenly unearths the problem. In this situation, the problem is typically very hard to detect since the occurrence is completely unrelated to the actual reason. In the same way, the multiprocessor corner case described in “[Debugging Temporally Decoupled Systems](#)” on page 262 can be hard to make out.

Still, there are several strategies to detect simulation issues due to temporal decoupling:

- Check that all behavior registered through callbacks on memory-mapped devices is set to AUTO\_SYNCING. Revert all suspicious NEVER\_SYNCING or SELF\_SYNCING accesses to AUTO\_SYNCING. This will slow down the simulation performance of the platform, but may point out the synchronization problem that exists in the model.
- A simple approach that does not require to inspect every callback in the system is to simply run the simulation with quantum set to zero. If this works correctly, there is a good chance that there is some synchronization problem. The simulation performance impact of this approach is, however, huge.
- Each temporally decoupled core provides a method called `show_optimized_accesses()`. This gives you the complete list of memory regions, which are accessed through DMI.
- Running the simulation with quantum set to zero sometimes takes too long. In this case, the default quantum period can be gradually reduced. This can be done in VP Explorer using the `set_quantum` command. For detailed information, see “[set\\_quantum](#)” in the [VP Explorer Tcl Interface Reference Manual](#).

- Another approach in case running the system with quantum set to zero takes too long is to selectively disable optimized accesses for a suspicious region. This can be done in VP Explorer. To do so, right-click on a memory in the Register view and then select *Backdoor Access* from the pop-up menu to disable it. For detailed information, see "[Controlling the Simulation Mode and Backdoor Accesses](#)" in the *VP Explorer User Guide*.

### 8.5.3 Modeling Fast Target and Router Peripherals

This section discusses the detailed modeling guidelines for target and router peripherals.

As explained in "[LT-Centric Simulation Techniques Overview](#)" on page 259, the type of the callback on an SCML target object (memory, reg, router) determines the simulation speed overhead for accessing this object. Depending on the frequency of accesses, a too conservative callback can seriously limit the overall simulation speed.

The following table associates rough speed level with each type of peripheral access. The three right-most columns give the functional and synchronization features of each type of access that determine the speed impact. As the speed impact increases, the speed-level numbering increases.

The "Synchronization over the Bus" column refers to the type of interconnect model used. By preference, virtual prototypes use an LT memory-mapped model to interconnect the peripherals to the initiator. An LT interconnect is a simple address decoder without any additional synchronization, whereas AT or cycle-accurate interconnect models infer some or many additional synchronization points in the simulation to model for example arbitration.

**Table 8-8 SCML Access Levels**

Speed Level	Type of Access	Synchronization over the Bus	Synchronization Required	Behavior Attached to the Memory
1	DMI pointer access	No	No	No
2	NEVER_SYNCING callback	No	No	Yes
3	AUTO_SYNCING callback	No	Yes	Yes
4	Bus Access over non-LT bus	Yes	Yes	Yes

- Speed level 1 is by far the fastest peripheral access. In this case, the initiator gets access to the internal storage of the peripheral through the DMI APIs and no user-defined behavior is invoked. This corresponds to a target object without callbacks. The speed is not affected by the number of interconnect nodes or the type of interconnect node; also the router objects do not have an impact on this. As long as the DMI request is enabled throughout the whole path from initiator to target object there will be a direct access to the storage of the target object.
- Speed level 2 is the case for a NEVER\_SYNCING callback where no additional synchronization points are set inside the behavior. NEVER\_SYNCING is in essence the same as SELF\_SYNCING but through the name the user shows the intent that no wait() calls will be used in the implementation. In this case, where the interconnect model is still an LT address decoder, no synchronization is implemented for the peripheral access. Since behavior is attached to the target object, it is required to create an TLM2.0 payload and a set of TLM calls need to be made to get to the target peripheral. As a result, the access to a level-2 peripheral is up to 50 times slower than accessing a level-1 storage object.

- Speed level 3 refers to peripherals with a `AUTO_SYNCING` callback attached to the target objects. The access to a level-3 peripheral requires synchronization before and after the call is made. Each synchronization infers a context switch of all ISS initiators in the system, so the simulation speed is another factor 20 slower than at speed level 2.



**Note** This is the default and preferred setting for callbacks on target objects. Therefore, it is important to investigate whether some of these can be reverted to speed level-2 accesses.

- Speed level 4 refers to virtual prototype models where the interconnect model is not a simple LT address decoder but a more complex AT or cycle-accurate bus model. To increase the timing accuracy of these models, additional synchronization points are implemented inside the bus model so that an access from initiator to target will infer a multitude of synchronization points. These typically will have more speed impact than the additional synchronization of an `AUTO_SYNCING` callback. Such a model will be another factor 10-100 slower than the speed level-3 model. Therefore, it is very important for fast virtual prototypes to use an LT address decoder instead of a more accurate bus model.

Obviously, there are many peripherals in a complete virtual prototype model and the impact of an individual peripheral should be weighted with the frequency of accesses to that peripheral. Imagine the case of a platform model with an accurate bus that supports DMI and where only one out of one million accesses are done over the bus (so all other accesses are DMI pointer accesses). Then the speed impact of that single level-4 access will still bring down the total simulation speed significantly but when compared to a platform where all accesses are level-4 accesses it will seem infinitely faster.

As described in “[LT-Centric Simulation Techniques Overview](#)” on page 259, callbacks on target objects are by default `AUTO_SYNCING`. This modeling guidelines manual promotes to use this as a default since it is the safest approach to get the expected behavior. At the same time, “[LT-Centric Simulation Techniques Overview](#)” on page 259 shows that a `NEVER_SYNCING` callback has much better simulation performance. The main benefit of an `AUTO_SYNCING` callback is that it ensures that all initiators in the system have had the chance to catch up to the current local time in the running initiator, and that SystemC time is updated to the current initiator time (that is, local time is now 0). There is still the chance that one or more initiators are further ahead in their local time.

### 8.5.3.1 AUTOSYNCING

In general, it required to stick to `AUTO_SYNCING` callbacks whenever the correct execution of the behavior requires a synchronized system. In the following cases, a `AUTO_SYNCING` callback should be used:

- The behavior queries the global SystemC time using `sc_time_stamp()` or `sc_simulation_time()`. In case `NEVER_SYNCING` would be used, these functions would return the SystemC time at the beginning of the current quantum (that is, they do not take local time into account).
- The behavior reads from an `scml_counter` object using `get_count()`, `read()`, or the `()` operator. Since the return value of these methods is computed based on the global SystemC time, the effect would be the same as in the previous point.
- The behavior reads from or writes to `sc_signal` or any other primitive SystemC channel. Writing to a primitive channel in a `NEVER_SYNCING` callback would only update the projected value. As a result, readers of the primitive channel would only see the new value after the `update()` function of the primitive channel has been called by the SystemC kernel, which is only after the quantum has expired.
- The behavior notifies an event. This operation updates the state of the event queue in the SystemC kernel and hence requires the current thread to yield (that is, call `wait()`). Only then will the SystemC kernel process the notified event correctly. Same as above, an event notification inside a `NEVER_SYNCING` behavior would only be taken into account after the current quantum expires.

- When the behavior has synchronizing semantics, an example is a semaphore register. At first glance, the default behavior of an `scm2::reg` perfectly models a register. However, the semaphore semantics of this register require that every access to this register happens at the correct SystemC time, so that other accesses to the semaphore are correctly handled.
- The behavior uses the `post()`, `post_read()`, or `post_write()` methods of the `scm1_post_port`. This rule is a consequence of the rule on notifying an event since the implementation of these methods notifies the `end_event` of the `scm1_transaction` object.
- The behavior calls the `transport()` method of `scm1_post_port`. Unlike the `post_` methods listed above, the `transport()` method of `scm1_post_port` does not notify any events and does not have any other side-effects. However, any component called through this `transport()` method could have synchronizing behavior and since the `transport()` call does not have a timing parameter, it is required to synchronize with the rest of the system before calling `transport`.

#### 8.5.3.2 SELF\_SYNCING

It is safe to use `SELF_SYNCING` callbacks in the following cases:

- The behavior calls `wait()`. By calling `wait()`, the behavior effectively has become a synchronizing behavior itself so there is no need to use `AUTO_SYNCING`. Whenever `wait()` is called, the local time should be passed as an argument to the `wait()` call to make sure local time and SystemC time are correctly aligned. The timing argument of the `wait` call may further be incremented with any additional delay that needs to be modeled. After the `wait()` call, the local time parameter should be reset to 0.
- When the behavior calls the `b_transport()` call on an initiator socket. As the timing parameter will be forwarded, any behavior further in the call chain can correctly synchronize.
- With any APIs that take a timing parameter to support temporal decoupling. These could be available from convenience sockets that come with extended TLM2.0 protocol definitions.

As mentioned before, in case of doubt, a callback should be registered as `AUTO_SYNCING` to ensure the correct execution of the functionality.

#### 8.5.3.3 NEVER\_SYNCING

Use `NEVER_SYNCING` behavior whenever the behavior will never require synchronization. This requires that the behavior complies to both of the following rules:

- The behavior is not `AUTO_SYNCING`.
- The behavior is not `SELF_SYNCING`.

For example:

- The behavior does not use any SystemC constructs and does not have synchronizing semantics. This applies to all purely arithmetic blocks like hardware accelerators.
- The behavior only accesses `scm2::memory` or `scm2::reg` objects.

This section discusses a number of additional strategies to optimize for simulation speed.

- Optimize the most frequently accessed peripheral first.

This means that the memory that is accessed most frequently should have the lowest speed level. The practical application of this simple rule requires some knowledge about the system and should be generalized to all accesses to a peripheral.

A simple example illustrates this idea: Imagine a peripheral that has three registers: A, B, and C. The behavior of the peripheral is that register C always contains the sum of A and B. There are two options to model this:

- This can be modeled by adding a NEVER\_SYNCING write callback to registers A and B so that on every write to these registers the value of C is updated. Register C does not need a callback.
- An alternative approach is to have a NEVER\_SYNCING read callback on C where every time C is read, A and B are added and stored in C.

Depending on the number of times A, B, and C are read or written, one alternative is better than the other. This decision may depend on the system context.

- Use read/write callbacks whenever it is possible to differentiate read behavior from write behavior.

The SCML memory object offers the following kinds of callbacks:

- The transport callback gives access to all attributes in the TLM2.0 generic payload.
- The read and write callbacks are essentially a convenience shortcut which give access to a limited set of attributes.

There is basically no difference in simulation speed if you use either a transport callback or both read and write callbacks of the same speed level. A speed advantage can be achieved if the speed level for read and write can be different. For example, if a component behaves as plain storage for read accesses but for write accesses there is some behavior implied then it is better to implement a write callback if possible. Using a transport callback to model such a component would be unnecessarily slow. Only having a write callback ensures that DMI is still enabled for read accesses.

All default behaviors (as `write_only`, `ignore_access`, and so on) are also implemented using callbacks with NEVER\_SYNCING synchronization.

- Use dynamic registration and removal of callbacks.

One more strategy to optimize simulation speed is to adjust the speed level of the callback during simulation time. An example is shown in the watchdog timer (see “[Modeling a Watchdog Peripheral](#) on page 301”), where the behavior of the module changes so that certain registers get simple storage behavior in certain modes. In the example, callbacks are removed and reregistered when needed, providing a simulation-speed improvement for the cases when callbacks are removed since DMI pointer accesses will be allowed in that case. In a similar way, callbacks can be registered to change from NEVER\_SYNCING to AUTO\_SYNCING, for example, when the register models a serial buffer that needs to synchronize with the rest of the system every 512 accesses. In this case, the callback will keep track of the number of accesses and switch itself to AUTO\_SYNCING when the update will need to happen.



Registration and removal of callbacks are in itself expensive operations so a careful trade-off has to be made.

#### 8.5.4 Optimizing Simulation Performance for FT Models

This section provides an overview of the simulation infrastructure.

The SCML FT Modeling methodology is built on top of SystemC. SystemC is a C++ based modeling library with a cooperative event-driven process scheduler at its core. This means that at the core SystemC provides with deterministic mechanisms to model the details of the concurrent hardware. The drawback of these mechanisms is that they rely on a scheduler and process context switches. Hence, have a very negative impact on simulation performance. Typically, the simulation speed of a fast stand-alone instruction-accurate ISS drops by more than one order of magnitude when it is integrated into a SystemC platform.

Over time, several extensions have been added to SystemC, primarily to circumvent the process and event based modeling basics of SystemC. This is done to improve the simulation performance so that SystemC-based solutions can keep up with the ever increasing complexity of hardware systems. In the first step, SystemC has been extended with *Interface Method* calls for communication, which is better known and

further standardized as Transaction Level Modeling (TLM). This replaces signal and event based communication between models with interface method calls. This increases the risk for non-determinism and the ugly debugging problems that come along with it, but this risk is low compared to the simulation speed benefit.

In the next step the synchronization requirements to model communication between components is further reduced. In the TLM2.0 standard, the concepts of temporal decoupling and direct memory interface have been introduced.

Using a TLM modeling style with function calls for communication with the platform context of the ISS is not sufficient. The overhead incurred by requiring TLM communication for every data exchange by an ISS limits the reachable simulation speed. To enable the use of SystemC-based platforms for software development use cases, TLM2.0 has standardized two major optimization techniques, which together enable simulation speed of 10-500 MIPS and beyond for an entire SOC platform. The two optimizations are:

- DMI infrastructure

Direct Memory Interface (DMI) forwards memory requests directly from the initiator to the target. This technique - also known as *backdoor access* - bypasses the complete interconnect model, which typically comprises components like buses, bridges, and transactors.

- Temporal decoupling

*Temporal decoupling* reduces the interaction between the platform simulation and the SystemC scheduler. This is achieved by drastically extending the synchronization interval between the model and the simulation kernel. A traditional model is activated by the SystemC kernel at the level of cycles or transactions. A temporally decoupled model synchronizes with the SystemC kernel at the level of *quanta*, which can last as long as 100,000 cycles.

Taken together, the two techniques basically turn any SystemC platform model into an LT platform model. These techniques can be enabled or disabled from the SCML modeling objects which in turn can be used to configure the model in what is referred to as full simulation mode and speed optimized mode for an loosely-timed platform. Basically, these indicate whether an initiator will initiate a TLM communication call for each data exchange or whether it will use the DMI interface and temporal decoupling to reach an optimal simulation speed.

TLM communication interfaces address the performance overhead of exchanging data between components and temporal decoupling that is usually associated with loosely-timed modeling. The FT modeling style also addresses the AT modeling style requirements where there is a need for performance improvements due to the much finer grain synchronization. To enable the use of SystemC-based platforms for the performance optimization and architecture exploration, use cases the SCML FT modeling style extends two more optimization techniques available in SystemC and TLM2.0. These are:

- Clock interfaces:

Traditional SystemC models will use a `sc_clock` to model clocked behavior. This is a standard SystemC object built on top of the event and process scheduler in SystemC. These clock objects deliver extremely poor simulation performance. The Synopsys SystemC simulation engine provides an improved implementation for those. However, this can be improved further through the introduction of a clock interface that provides more information about the clock and that allows to schedule behavior in the future. The benefit of this is that it reduces the number of processes and events in the simulation by providing a central clock scheduler for all components that rely on the same clock rather than reusing the basic SystemC kernel for that.

- Timing annotation:

Basically, this technique is the same as temporal decoupling. In the TLM2.0 standard, temporal decoupling is typically linked to the loosely-timed coding style, in the FT modeling style, timing annotation is used in combination with a quantumkeeper, as provided for temporal decoupling. While the granularity of temporal decoupling is much smaller there is still a performance gain to be had.

Based on the basic simulation technology described in [“Optimizing Simulation Performance for FT Models” on page 269](#), the following rules should be considered while creating SCML FT component models moving from how to use the SystemC kernel up to the optimization process:

- Kernel activity:

Minimize the interaction with the process and event scheduler; this should be the first concern of any model architecture since this is the key cause of simulation performance degradation. As already explained, there have been several enhancements made to SystemC specifically with this goal in mind. However, this does not imply that minimizing kernel overhead should be the number one concern with regards to the simulation performance. Consider the following points:

- It is not necessary to mimic the internal parallelism and concurrency of a component. It is sufficient to make sure that the interaction with the external world happens at the correct time points. Parallelism and concurrency implies events and processes in SystemC, these are only useful when there is contention; that is, several activities fighting over a resource. There should be a synchronization point modeled only when there is a possible contention (that is, the resource is free and there is activity) when the resource assignment schedule can be determined for a series of activities, based on current inputs. Then, this can be used to avoid synchronization in the SystemC kernel.
- Use TLM also for component interfaces that are not memory mapped. Use TLM1.0 API with a dedicated payload or use the TLM2.0 base protocol restricted to the data pointer. This will avoid signal events and method sensitivity.
- When reusing code that intensively uses signal interfaces and `sc_clocks`, it may be worth wrapping the code with a TLM interface and a clock gate, so that the component is only active when there is data exchange.
- Use SystemC methods which are cheaper than threads; immediate notification is cheaper than event notification or timed notification.

- Clock scheduling:

A special case of kernel activity is incurred through clock scheduling. Traditional SystemC clocks will cause event and process overhead for every tick of the clock. In many cases, users overcome this by not modeling the clock connectivity and revert to use a model parameter indicating the clock period for that model and use that to schedule events which then mimic the clocked activity in the model. This approach causes a mess when trying to mimic the clock tree of a design through parameter settings. It makes it a nightmare when clocks are becoming programmable and even more when optimizations for power (and hence clock frequency scaling) become a key use case for virtual prototypes. Therefore, the SCML2 modeling objects library provides specialized clock interfaces to propagate the clock period to every component that needs it. Further optimizations are possible when each component registers their clocked behavior with a central clock scheduler, thus avoiding event and process overhead by the SystemC kernel for each of the components in the design. Furthermore, this modeling approach also allows the central clock scheduler to keep track of any period changes for the clock during temporal decoupling and to reschedule the callback requests accordingly. Obviously, this last feature has a performance overhead since additional checks and calculations are required for each callback that is scheduled. The key guideline for clocks is to use the SCML2 modeling objects and then to apply the previous rule to minimize the clock based activity.

- TLM calls:

TLM calls model the communication between components. Each TLM call needs to be forwarded over the memory and interconnect infrastructure even if these components do not modify or use any of the information provided with the call. This is one of the reasons to come up with the TLM2.0 Direct Memory Interface. While using TLM calls it is important to minimize the number of transport calls to implement the communication. The following are the different ways to achieve this:

- The most efficient TLM interface is the blocking transport interface since it has only two timing points for a data transfer of arbitrary length and it is implemented with a single interface method call. Therefore, it is the preferred API for the loosely-timed modeling style where speed is most important. However, it can also be used for the approximately-timed coding style. The possible drawback is that it must be called from a `sc_thread`, which is more expensive than methods. And more importantly, when a transaction needs to be converted from a non-blocking transport interface to the blocking transport interface, a `sc_thread` needs to be added in the conversion which adds additional synchronization overhead. Conversion from the blocking interface to the non-blocking interface is cheaper since there is no additional thread and synchronization required than would typically be incurred when using a non-blocking interface to start with. A component can limit itself to only support the blocking interface in the following cases:
  - An initiator that only needs the two timing points supported by the blocking API, or it can use the API for the transactions for which this timing information is sufficient. Typically, the speed improvement will be very limited since most likely the blocking interface will need to be converted to a non-blocking interface somewhere in the interconnect path.
  - A target for which it is sufficient to model the latency behavior. Also, here the speed improvement is limited due to the conversion overhead that is likely to be there.

In both cases however, the simplified coding style is a reason to expect improved performance (less code, so simpler to maintain and optimize).

- When using the non-blocking interface, there are typically a number of calls required to complete a transaction. Usually, there is a complete set of calls defined that allow full timing accuracy for all possible protocol state transitions and delays of a certain protocol, for example, they enable to exchange data on a beat per beat for burst transactions. The key speed improvement here is to do as few as possible TLM interface calls. The definitions of the protocol state machines is such that they provide with shortcuts to skip states if they happen to be on the same timing point or when they do not provide any meaningful accuracy improvement for initiator or target. These shortcuts should be used whenever possible.
- One additional way of reducing the number of protocol states required to execute a transaction is to use the TLM2.0 base protocol. This protocol has four timing points for the non-blocking interface. When this provides all features to model the timing of a component, then the TLM2.0 base protocol should be used. When used in combination with other components that use a more detailed protocol engine, protocol conversion will be inserted. This reduces the potential for speed improvement. The protocol state conversion adds the code that would be needed in case all components used the detailed protocol; but there will also be an additional conversion logic to convert the attributes. When used in combination with the SCML storage objects that conversion is anyway needed.

- Use DMI:

DMI is a key TLM2.0 speed improvement feature. It should be used as much as possible and should be made switchable so that less and more accurate modes are enabled. A DMI handler can be protocol specific or even component specific to make sure that there is a maximum number of DMI calls done. DMI is typically limited to the loosely-timed coding style but should be considered for the AT case as

well. Whenever a blocking call makes sense it is also useful to consider whether the full interconnect path can be skipped via a DMI access. DMI allows for a latency parameter which can be used to identify start and end of the transaction. Still there might be an accuracy impact when doing this which should be considered in the evaluation: the impact of resource contention in the interconnect is not taken into account.

- Use temporal decoupling:

This is a specific variant of minimizing kernel activity by using timing annotation in the TLM interface calls to model timing rather than to use process and event synchronization for every time advance. Temporal decoupling is the key to the performance of loosely-timed simulations. In the FT modeling style, timing annotation or temporal decoupling should be used whenever possible.

- Configurability:

Have an LT/AT abstraction switch. Models should be configurable to run at different abstraction levels. The timing accuracy of a simulation relates to the internal accuracy of the model as well as the timing accuracy of the communication interface it is using. For ease of use and overall consistency, each model should be aware of its own accuracy. This means that it should determine what level of timing accuracy it will enable and in which way it wants to initiate transactions or wants to respond to them. The alternative where a model uses the abstraction of the incoming transaction request to determine the timing accuracy it will provide is acceptable, but tends to be lean to less accuracy than you would expect. Transactions that are handled with less accuracy do not only impact their own timing accuracy but also the accuracy of the following transactions.

More configurability can be added for complex models, so that the simulation performance can be optimized for the specific use case, although typically an on/off switch for accuracy should be sufficient. A model should only provide support to switch on accuracy once, the complexity to support a switch from accurate to less accurate is too high compared to the very limited use cases where it can be used. This complexity is caused by the handling of timed transactions that are in flight at the moment the accuracy is turned off, additional modeling is required to make sure an inaccurate transaction overtakes a more accurate one.

- Use profiling tools:

Despite the advises discussed above, the best approach to optimize performance is to profile the models, preferably exercised in a realistic context and focus on the specific performance bottlenecks of the model itself.

- Use the SystemC debugging tools to the following:

- Check the number of activations of threads or the number of event notifications.
    - Validate whether the DMI configuration is done correctly.
    - Check the number of TLM interface calls that are done.

- Use generic C++ profiling tools to check the computational overhead for the model.

Use this to validate the results of the SystemC debugging tools in context of the overall performance. One of the factors that affect the performance of an SCML2 based system is the overhead of the frequent calls to the `nb_transport_fw()` and `nb_transport_bw()` functions of the FT (TLM2.0) interface of the models.

For example, consider an SCML2 FT AXI based system. In the FT AXI protocol, a sixteen beat read transaction requires a minimum of seventeen calls to the TLM2.0 `nb_transport` (fw/bw) functions, when all the timing points of the transaction are to be modeled.

This can cause a substantial speed degradation, especially when the communicating components have multiple intermediate blocks and each call is routed through all these blocks.

To mitigate this, the FT methodology allows skipping protocol states in order to gain on simulation performance, for example, for the read transaction above, a target can send all the data using a single `nb_transport_bw()` call with the protocol state set to `RVALID_LAST`. While using a single call results in a significant improvement in simulation performance, it causes a loss of timing information for the individual beats of the transaction.

Data beat timing arrays allow model developers to reap the benefits of state skipping while retaining the timing information on their FT interface. Using data beat timing arrays the sender of the data can indicate the timings of the individual beats of data with a single call to the `nb_transport_(fw/bw)` function on its FT interface.

#### 8.5.4.1 SCML2 Payload Extensions for Using Data Beat Timing Arrays

This section describes the data beat timing array specific extensions as well as convenience macros that have been provided for their use.

Two array type extensions have been defined to allow components to communicate the start and end times of the individual data beats of a transaction using a single `nb_transport_fw()` or `nb_transport_bw()` call. These extensions contain an array of unsigned integers, where each entry of the array is the offset in number of clock cycles from `sc_time_stamp() + time` (annotated time of the `nb_transport_fw/bw()` call).

- `data_beat_avail_extension`

```
DECLARE_ARRAY_EXTENSION(data_beat_avail_extension, unsigned int);
```

- This extension is used by the sender of the data (initiator for writes and the target for reads) to indicate the best effort availability time for each beat of data that it has to send for a transaction.
- The individual entries in the array should be interpreted by the receiver as the offset in clock cycles from the `sc_time_stamp () + t` at which the `nb_transport_fw/bw()` call is made.

- `data_beat_used_extension`

```
DECLARE_ARRAY_EXTENSION(data_beat_used_extension, unsigned int);
```

- This extension is used by the receiver of the data to indicate the time at which each beat of data is consumed.
- The individual entries in the array should be interpreted by the sender of the data as the offset in clock cycles from the `sc_time_stamp () + t` at which the `nb_transport_fw/bw()` call is made.

The semantics for the use of data beat arrays (see “[General Rules for the Use of Data Beat Arrays](#)” on [page 276](#)) mandate that each transaction must go through a handshake process in the address phase to determine if the receiver of the data will be able to honor data beat timing arrays. This per transaction handshake ensures that data beat timing arrays are used only in the cases where the receiver of the data agrees that it will be able to process the same.

The following extension is used for the above-mentioned handshake procedure.

- `can_accept_data_beat_array_extension`

```
DECLARE_EXTENSION (can_accept_data_beat_array_extension, bool, false);
```

- This is an extension of type `bool`, and it must be set to true by the receiver of the data to indicate that it can accept data beat timing array for the particular transaction.

- The default value for this extension is `false` and it is imperative that it is set prior to the data phase (if any) of the transaction.
- The sender of data must not use data beat timing array for the transaction in case the extension is set to `false` by the receiver.



**Note** The `can_accept_data_beat_array_extension` is just a hint from the receiver of the data to the sender, to indicate that it can process data beat timing array. The initiator is free to not use data beat arrays even if the extension is set to `true`.

#### 8.5.4.2 Convenience Macros for Array Type Extensions

As explained in “[Optimizing Simulation Performance for FT Models](#)” on page 269, the use of data beat timing requires adding and modifying payload extensions that contain an array of unsigned integers.

The following macros have been provided to define, set and retrieve the extensions that contain arrays.

An extension containing an array can be defined using:

```
DECLARE_ARRAY_EXTENSION(ext_name, attr_type)
```

The parameters for this macro are:

- The name of the extension.
- The type of the array.

For example: `DECLARE_ARRAY_EXTENSION(my_array_extension, int);` defines an extension that consists of an array of integers.

The following convenience macro can be used to add an array type extension to a payload, or to change the valid count of an array type extension that is already present with the payload.

```
SET_ARRAY_EXT_ATTR(payload_ptr, ext_name, attr_type, array_ptr, valid_count);
```

This macro takes the following parameters:

<code>payload_ptr</code>	Specifies pointer to the payload for which the extension needs to be added or modified.
<code>ext_name</code>	Specifies the name of the array extension.
<code>attr_type</code>	Specifies the type of the array.
<code>array_ptr</code>	This returns the pointer to the start of the array.
<code>valid_count</code>	Specifies the number of valid entries in the array.

Usage:

The following demonstrates how this macro can be used to set the array size of `data_beat_avail_extension`.

```
ft_generic_payload* trans;
SET_ARRAY_EXT_ATTR(trans, scml2::data_beat_avail_extension, unsigned int, beat_array_ptr, 8);
```

- The macro will add the `data_beat_avail_extension` extension to the payload, if it does not exist already.

- This timing array gets automatically resized in case, the extension existed already and the original valid count was less than the count specified.



The memory management for the array is done in the extension class itself.

The following convenience macro can be used to retrieve the array and the number of valid entries in the array type extension.

```
GET_ARRAY_EXT_ATTR(payload_ptr, ext_name, attr_type, array_ptr, valid_count);
```

This macro takes the following parameters.

payload_ptr	Specifies the pointer to the payload for which the extension is to be retrieved.
ext_name	Specifies the name of the extension that needs to be retrieved.
attr_type	Specifies the type of the array.
array_ptr	This returns the pointer to the start of the array.
valid_count	This returns the number of valid entries in the extension.

Usage:

The following demonstrates how the above macro can be used to get start of the array as well as the number of valid entries in the array.

```
ft_generic_payload* trans;
GET_ARRAY_EXT(trans, scml2::data_beat_avail_extension, unsigned int, beat_array_ptr, num_entries);
```

- The number of valid entries gets stored in the num\_entries variable.
- The start of the array is stored in an beat\_array\_ptr variable which is of type unsigned int\*.



In case, the extension does not exist, the number of valid entries is set to 0. This implies that the data beat semantics are not being used by the sender and the data beat array should not be accessed.

#### 8.5.4.3 General Rules for the Use of Data Beat Arrays

This section enumerates some rules which must be followed in case data beat arrays are being used by FT based models.

- It is mandatory for the receiver of data to indicate that it can accept data beat arrays for the data phase using the can\_accept\_data\_beat\_array\_extension.
- The can\_accept\_data\_beat\_array\_extension should be set to true only when it is found in the transaction payload. The receiver should use the standard TLM2.0 get\_extension() API, for this extension to determine its presence in the payload.
- Prior to starting the data phase, the sender must check the can\_accept\_data\_beat\_array\_extension and should use data beat array semantics only if the extension is set to true.

- Using data beat arrays to transfer a part of the total data is not allowed. For instance, for a sixteen beat burst, the sender cannot send two data transfers for the data phase each using a data beat array of size eight.
- The size of the `data_beat_avail_extension` must always be set to the total number of beats in the transaction.
- The first entry in the `data_beat_avail_extension` must always be 0. This helps to avoid unnecessary manipulation of the data beat timing array in case the data phase is stalled for some reason.
- A receiver capable of handling data beats should use the valid count of the `data_beat_avail_extension` to determine whether data beat semantics are being used or not. A value of 0 for this attribute indicates that the data beat arrays are not being used. This can happen in case, the sender decides not to use data beat timing arrays even when the receiver hints that it can accept the data beat arrays.
- The valid count of the `data_beat_used_extension` should always be set to the count retrieved from the `data_beat_avail_extension`.
- The receiver is not allowed to accept a subset of total beats using the `data_beat_used_extension`.
- The values in the `data_beat_avail_extension` indicate best effort availability of data. While populating the `data_beat_used_extension`, the receiver should also account for the back-pressure that may get created due to the data acceptance delays. The following may be used by the receiver in case the data transfer is to be completed without synchronization.

```

for (unsigned int i = 0; i < valid_count; i++)
{
    if ( i == 0 )
    {
        data_used [i] = data_avail [i] + data_accept_delay;
    } else {
        data_used [i] = ( data_avail [i] > data_used [i-1] ) ? data_avail [i] + data_accept_delay
                                                : data_used [i - 1] + 1+ data_accept_delay;
    }
}

```

- In case the receiver wishes to synchronize and change context before sending the data acknowledgement, it must set the first entry in the `data_used_time_exetension` to 0.

This section covers the FT GFT specific semantics for the use of data beat arrays.

#### 8.5.4.4 Data Beat Array Handshake

The first step of the data beat array approach is the handshake between the initiator and the target regarding the use of data beat arrays using the `can_accept_data_beat_array_extension`. For FT GFT transaction, this handshake must happen during the CMD phases namely, the RD\_CMD and the WR\_CMD phases.

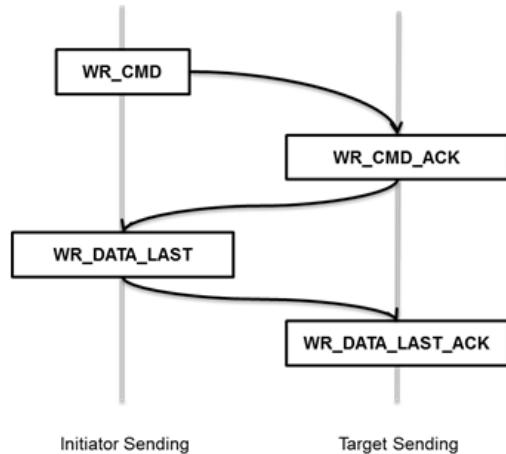
Note the following points in this context with regards to read and write transactions respectively.

- For a write transaction, the initiator must explicitly set the value of the extension to `false`, before sending the WR\_CMD. The target on receiving the WR\_CMD checks for the presence of this extension and if it is present, sets it to `true` or `false` depending on its behavior.
- For a read transaction, the initiator must initialize the `can_accept_data_beat_array_extension` appropriately by setting it to either `true` or `false` explicitly, before sending the RD\_CMD.

#### 8.5.4.5 Transaction State Machine for Write Transactions

The following figure depicts the protocol state machine for a write transaction when the data beat arrays are used.

**Figure 8-10 GFT State Machine for Write Accesses Using Data Beat Arrays**



The state machine is a subset of the one shown in figure 3-16 where the initiator is also allowed to send the WR\_DATA states.

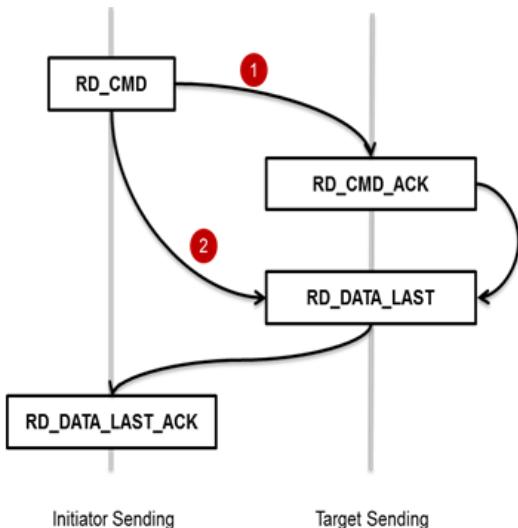
Observations on the state machine:

- The initiator on receiving the WR\_CMD\_ACK must set the protocol state to WR\_DATA\_LAST. The data setup delay should be accounted for in the annotated time and the first entry in the data\_beat\_avail\_extension should be set to 0.
- Target on receiving the WR\_DATA\_LAST state should look for the data\_beat\_avail\_extension and if present should populate the data\_beat\_used\_extension keeping in mind its data-accept delays.

#### 8.5.4.6 Transaction State Machine Read Transactions

The following figure depicts the protocol state machine for a read transaction when the data beat arrays are used.

**Figure 8-11 GFT State Machine for Read Accesses Using Data Beat Arrays**



This state machine is a subset of the one shown in figure 3-18, which also allows the target to send RD\_DATA protocol states.

Observations on the state machine:

- The target is allowed to skip the RD\_CMD\_ACK phase and move directly to the RD\_DATA\_LAST phase.
- In case the target wishes to exercise path ② in the state machine, the RD\_CMD\_ACK is assumed to be implicit and the time at which it is received should be taken as the `sc_time_stamp() + t`.
- Moreover for path ②, the address-accept delay if any should be accounted for using the annotation. The rule that the first entry in the `data_beat_avail_extension` should be 0 must be followed.
- The target can also choose path ①, where it wants to send an explicit RD\_CMD\_ACK phase. In that case, it must start the RD\_DATA\_LAST from a different SystemC context.
- On receiving the RD\_DATA\_LAST\_ACK, the target must look for the data beat array specific extensions in the payload so that it is able to mark the free time of the GFT read channel correctly.

This section covers the FT\_AXI specific semantics for the use of data beat arrays.

#### 8.5.4.7 Data Beat Array Handshake

For the FT\_AXI protocol, the handshake between the initiator and the target regarding the use of data beat arrays must happen during the address phases of a transaction, namely, the ARVALID and the AWVALID phases.

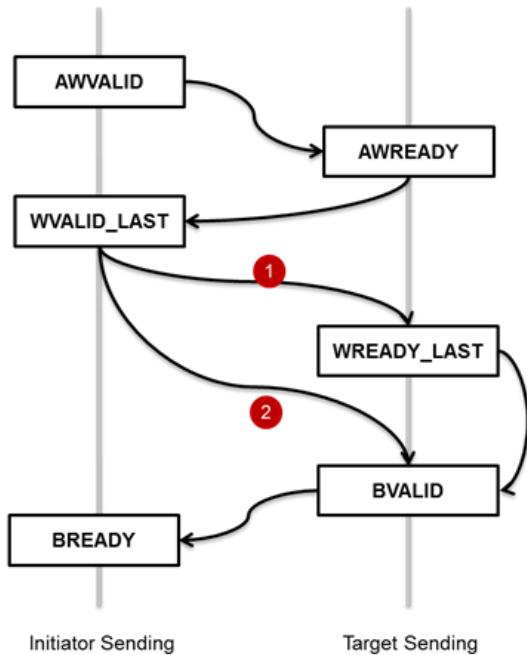
Note the following points in this context with regards to read and write transactions respectively.

- At the start of a read transaction (that is, the ARVALID phase) the initiator must initialize this extension appropriately by setting it to either `true` or `false` explicitly.
- At the start of the write transaction (that is, AWVALID phase) the initiator must set the value of the extension to `false`. The target on receiving the AWVALID checks for the presence of this extension and if it is present sets it to `true` or `false` depending on its behavior.

#### 8.5.4.8 Transaction State Machine for Write Transactions

The following figure depicts the protocol state machine for a write transaction when the data beat arrays are used.

**Figure 8-12 FT AXI State Machine for Write Accesses Using Data Beat Arrays**



This state machine is a subset of the one shown in figure 3-24, which also allows the initiators to send WVALID protocol states.

Observations on the state machine:

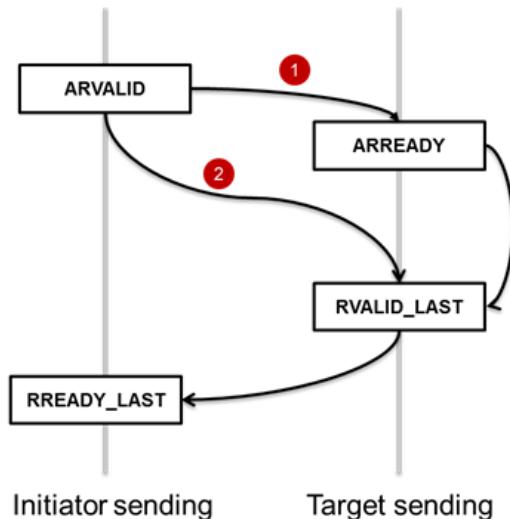
- The initiator on receiving the AWREADY must set the protocol state to WVALID\_LAST. The data setup delay should be accounted for in the annotated time and the first entry in the data\_beat\_avail\_extension should be set to 0.
- Target on receiving the WVALID\_LAST state should look for the data\_beat\_avail\_extension and if present should populate the data\_beat\_used\_extension keeping in mind its data-accept delays.
- On receiving the WVALID\_LAST state, the target can choose path ① to advance the state machine of the transaction and send an explicit WREADY\_LAST state.
- In case the target sends an explicit WREADY\_LAST, the target would need to send a BVALID using a new SystemC context.
- Alternatively, the target can also choose to skip WREADY\_LAST state altogether and send directly move on to the BVALID state using path ②.
- The target must populate the data\_beat\_used\_extension even when it is skipping the WREADY\_LAST state.
- The time at which BVALID is sent should be calculated as `sc_time_stamp () + t + used_array [last_beat-1]`.
  - In such a scenario, it is the responsibility of the initiator to extract the timing information from the data beat array and transfer it to the time annotation while sending the BREADY.

- In other words, it is incumbent on the initiator to transfer the timing information from the data beat array into the timing annotation once the data phase of the transaction is handled.

#### 8.5.4.9 Transaction State Machine Read Transactions

The following figure depicts the protocol state machine for a read transaction when the data beat arrays are used.

**Figure 8-13 FT AXI State Machine for Read Accesses Using Data Beat Arrays**



This state machine is a subset of the one shown in figure 3-26, which also allows the target to send RVALID protocol states.

Observations on the state machine:

- The target is allowed to skip the ARREADY phase and move directly to the RVALID\_LAST phase.
- In case the target wishes to exercise path ② in the state machine, the ARREADY is assumed to be implicit and the time at which it is received should be taken as the `sc_time_stamp() + t`.
- Moreover for path ② the address-accept delay if any should be accounted for using the annotation. The rule that the first entry in the `data_beat_avail_extension` should be 0 must be followed.
- The target can also choose path ① where it wants to send an explicit ARREADY phase. In that case, it must start the RVALID\_LAST from a different SystemC context.

On receiving the RREADY\_LAST, the target must look for the data beat array specific extensions in the payload so that it is able to mark the free time of the FT AXI read channel correctly.

## 8.6 Getting Started

With the Virtualizer and Platform Architect releases, there are a number of examples provided for the FT modeling coding style. There are examples for models that use a pure LT TLM2 base protocol interface, which focuses on the usage of the storage objects, there are also examples that introduce the GFT and AXI protocol definitions for AT modeling. These examples can be found at

`$SNPS_VP_HOME/.../any/examples/TLM2ModelingExamples/` location and also as the `TLM2ModelingExamples` component library in Platform Creator.

To introduce the coding style and modeling objects, this section explains some examples. The focus is on the modeling guidelines for LT modeling with the TLM2 base protocol. There is also an introduction to how components can be tested. Each construct is explained in detail when first introduced; later examples focus on the additional constructs that are used in these examples.

- Modeling a Memory
- Modeling an Interrupt Controller
- Modeling a Watchdog Peripheral
- Modeling a DMA
- Modeling a Cache
- Example Timer Specification

### 8.6.1 Modeling a Memory

In this first example, the focus is on the use of the core language primitives of SCML, TLM2.0, and SystemC to create a model. These are introduced using a memory model and a test component for the memory.

The simplest possible component to model using SCML is a plain memory. Since the SCML memory objects implement all the TLM2.0 interface features and the storage behavior, their code is limited to instantiating and connecting the SCML memory object. The code looks as follows:

```
1 #include <scml2.h>
2 #include <tlm.h>
3 #include <systemc>
4
5 class ExampleMemory : public sc_core::sc_module
6 {
7     private:
8         static const unsigned int MEMORY_SIZE = 0x10000;
9
10        public:
11            tlm::tlm_target_socket<32> socket;
12
13        public:
14            SC_HAS_PROCESS(ExampleMemory);
15
16            explicit ExampleMemory(sc_core::sc_module_name name) :
17                sc_core::sc_module(name),
18                socket("socket"),
19                mAdapter("mAdapter", socket),
20                mMemory("mMemory", MEMORY_SIZE >> 2)
21            {
22                mAdapter(mMemory);
23            }
24
25        private:
26            scml2::tlm2_gp_target_adapter<32> mAdapter;
27            scml2::memory<unsigned int> mMemory;
28        };
29
```

The labeled code fragments are explained as follows:

## 1 Header files

Any component should include the SCML, TLM, and SystemC header files. With these three header files, all modeling constructs, interfaces, and objects from SCML, SystemC, and TLM2.0 are available.

**NOTE:** The SystemC, TLM2.0, and SCML2 libraries use namespaces and prefixes:

- All SystemC objects have an `sc_` prefix and are defined in the `sc_core` namespace.
- All TLM2.0 objects have a `tlm_` prefix and are defined in the `tlm` namespace.
- SCML objects do not have a prefix and are defined in the `scml2` namespace.

## 2 Class definition

In SystemC, components are modeled as C++ classes that derive from the generic component model `sc_module`. In this case, the module will define an `ExampleMemory`.

## 3 Sockets

The memory should be able to receive communication requests from other components in a system. Therefore, a member is added to the module: `tlm::tlm_target_socket<32> socket`. The integer template parameter indicates the width of individual data elements that pass through the socket; in this case 32 bits wide.

## 4 Memory and adapter

1. In the `private` section at the bottom, two more members are defined for the memory component.
2. There is an adapter object to interface the generic payload carried through the socket to the internal storage of the memory (`scml2::tlm2_gp_target_adapter<32> mAdapter`).
3. There is the storage itself (`scml2::memory<unsigned int> mMemory`).
4. The template parameter for the adapter is 32, and should be the same as for the socket. The template parameter for the memory indicates the behavioral representation for the data elements of the memory. This indicates how the data will be used in expressions. The size of the C++ data type used should be equal to the bit width of the socket and adapter. For the list of supported C++ data types, see “[Memory Objects](#)” on page 25.

## 5 Module constructor

1. The constructor for the module initializes the different members.
2. The constructor takes an argument `sc_core::sc_module_name name`, so that each instantiation of this module can be given a name. The name is passed to the constructor of the `sc_module` base class.
3. The socket is given a name, so that sockets can be identified in debugging tools.
4. Next, the adapter is initialized with a name and it is connected to the socket of the module.
5. Finally, the memory is initialized with a name and a size. The size of an SCML memory is specified in units of the data type specified. Here `unsigned int` is used while the size of a memory is typically defined in address space, which is usually defined in byte. This is why the size of the memory is shifted down by 2 to convert byte addresses into word index values.

## 6 Constructor body

In the body of the constructor, the memory is connected to the adapter. Through this initialization, all communication received by the socket will be interpreted by the adapter to fit with the memory behavior and the memory will store or retrieve the requested data.

## 7 SC\_HAS\_PROCESS:

The `SC_HAS_PROCESS` (`SCML2Memory`) macro is required so that all initialization code can know which module is currently being defined.

**NOTE:** This macro was originally used in SystemC by processes to know to which module they belong to; hence the name for the macro.

## 8 Constant parameters

Whenever a constant parameter for a design is used, it is best to declare that as a constant member of the class. In this case, the size of the memory is a constant. It is only used in one place but usually there will be several points in the code where this parameter will be needed. Obviously, these parameters are prime candidates for parameterization through `scml_property` objects, so that these values can be set through configuration tools like Platform Creator.

To test the memory, we need a module that initiates memory-mapped transactions. An example of such an initiator module is defined below.

```
3.2 #include <scml2.h>
      #include <tlm_utils/tlm_quantumkeeper.h>
      #include <tlm.h>
      #include <systemc>

1   class MemoryCheck : public sc_core::sc_module
2   {
3       public:
4           scml2::initiator_socket<32> socket;

5       public:
6           SC_HAS_PROCESS(MemoryCheck);

7           explicit MemoryCheck(sc_core::sc_module_name name) :
8               sc_core::sc_module(name),
9               socket("socket")
10          {
11              tlm_utils::tlm_quantumkeeper::set_global_quantum(
12                  sc_core::sc_time(1, sc_core::SC_MS));
13              mQuantumKeeper.reset();
14
15              socket.set_quantumkeeper(mQuantumKeeper);
16
17              SC_THREAD(run);
18          }
19      }

20  protected:
21      tlm_utils::tlm_quantumkeeper mQuantumKeeper;
22  };
```

```

5    void run()
6    {
7        unsigned int address = 0;
8        unsigned int data = 0;
9.4     sc_core::sc_time period(100, sc_core::SC_NS);

10       std::cout << "TEST STARTED" << std::endl;

11       for (; address < 0x4000; address += 4) {
12           data = address * 2;
13           if (!socket.write<unsigned int>(address, data)) {
14               std::cerr << "ERROR: Write failed at address 0x"
15                           << std::hex << address << std::endl;
16           }
17           timeAdvance(period);
18       }

19       do {
20           address -= 4;
21           data = 0;
22           if (!socket.read<unsigned int>(address, data)) {
23               std::cerr << "ERROR: Read failed at address 0x"
24                           << std::hex << address << std::endl;
25           }
26           if (data != (address * 2)) {
27               std::cerr << "ERROR: TEST FAILED, got 0x" << std::hex << data
28                           << " at address 0x" << std::hex << address << std::endl;
29           }
30           timeAdvance(period);
31       } while (address != 0);

32       std::cout << "TEST DONE" << std::endl;
33.5    sc_core::sc_stop();
34   }

35.3   void timeAdvance(sc_core::sc_time &period)
36   {
37       mQuantumKeeper.inc(period);
38       if (mQuantumKeeper.need_sync()) {
39           mQuantumKeeper.sync();
40       }

```

The labeled code fragments are explained as follows:

- 1 The code defines a `MemoryCheck` module.

## 2 Initiator socket

The code contains an initiator socket `scml2::initiator_socket<32> socket`; this socket is part of the SCML library. SCML has an initiator socket defined that has a simplified communication interface and also implements a handler for the TLM2.0 DMI. This means that the socket keeps track of the memory regions for which DMI is allowed and on every access a check is performed to see whether it is possible to retrieve the data via DMI.

## 3 Quantum keeper

All *free-running initiators*, which are components that model behavior that executes software or behavior that is concurrent to software execution, must use a quantum keeper.

**3.1** : There is a quantum keeper to support temporal decoupling: `tlm_utils::tlm_quantumkeeper mQuantumKeeper`. The quantum keeper will keep track of the local time in the initiator versus the global SystemC time and will force synchronization at the rate of the global quantum.

**3.2** : The quantum keeper is part of the `tlm_utils` namespace; so an additional header file is required.

## 4 Constructor

1. In the constructor of the module, the global quantum is set to 1 millisecond. SystemC defines an object `sc_time` which is used to identify time periods. Several units are defined for time: `SC_MS` for milliseconds, `SC_NS` for nanoseconds (others are possible as well).
2. The quantum keeper of this module is also passed to the socket. The socket will use it to pass the local time information along with the communication requests as is defined in TLM2.0 and synchronize if the local time becomes larger than the global quantum.
3. The `SC_THREAD(run)` macro registers the `run()` method of the `MemoryCheck` module to be used as a thread in the multitasking kernel. Since the LT coding style is used in this initiator, the code that initiates transactions must be called from a SystemC thread. In a SystemC thread it is allowed to return control to the SystemC kernel from any point in the code; this is not possible for a SystemC method. Using a thread, it is possible to synchronize with the SystemC kernel while processing a transaction request.

## 5 The run() class method

The `run()` method should implement the concurrent behavior of this module. Through the registration in the constructor (see 4.3 above), the `run()` method will become an `SC_THREAD`. Typically, an `SC_THREAD` models a concurrent process that performs a task repeatedly. For that purpose, it usually contains an infinite loop. In this case, the test should be executed only once; therefore the infinite loop is not there.

To initiate a transaction, the read and write methods of the initiator socket are used. Both have an address and data argument.

**5.1** : First, each of the `0x1000` elements in the memory is written to. The value stored in each location is equal to the address multiplied by 2. This value will be read back to test the memory. If the write fails, an error is printed. Since the TLM2.0 address is a byte address and each access will store 32 bits (or 4 bytes) the address needs to be incremented by 4 after each write. This also explains why the upper boundary for the address value (`0x4000`) is four times larger than the size of the memory (`0x1000`).

**5.2** : Next, each word that was written is read back and checked. If the read fails, an error is printed; Also when the value returned is not the same as the one that was written, an error is printed.

**5.3** : When modeling components for a virtual prototype, it is the initiator that controls time. So after every access, the local time is incremented with the initiator execution time. The timing annotation in target models is intended to model any additional delay on top of the execution time that should be expected from the initiators. At every time increment, the quantum should be checked and a system-level synchronization should be performed if needed. For this purpose, the `timeAdvance()` method is added.

**5.4** : To model the initiator execution time, a `period` parameter is added. This `sc_time` variable is constructed once and reused in order to avoid simulation speed overhead when creating a new `sc_time` object.

It is not good to write `timeAdvance(sc_core::sc_time(100, sc_core::SC_NS))` since a new `sc_time` object will be constructed on each invocation of `timeAdvance()`.

**5.5** : The `sc_stop()` call terminates the simulation.

To run the test, the `ExampleMemory` and `MemoryCheck` modules are imported into Platform Creator, connected together into a test system, compiled, and run. This section describes the procedure. For detailed information, see the *Platform Creator Reference Manual*, which is included in the Platform Architect documentation.

### To run the memory test:

- 1 Launch Platform Creator:

```
pct &
```

- 2 Load the `ExampleMemory` and `MemoryCheck` modules. To do so:

1. From the menu bar, select *Project > Import SystemC Modules*.



Make sure that the `TLM2_PL` library is opened before you take this step. To do so, on the *Definitions* tab page on the left, scroll down to the `TLM2_PL` item, right-click on it and from the pop-up menu, select *Open*.

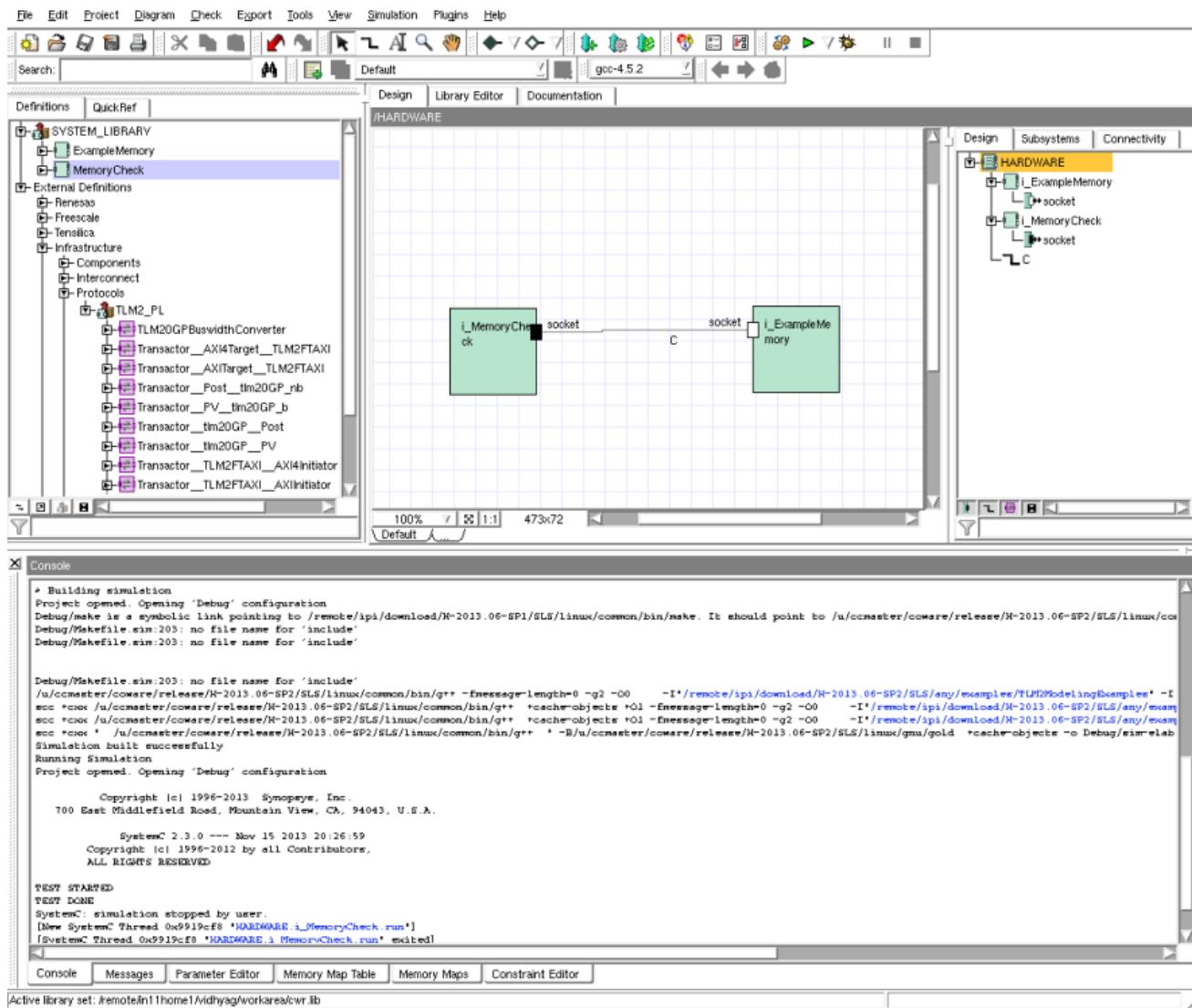
The Import SystemC Modules dialog box appears.

2. Click *Add* to add the `ExampleMemory.h` and `MemoryCheck.h` files.
  3. Click *OK*. The modules are loaded into `SYSTEM_LIBRARY` (see the *Definitions* tab page).
- 3 Instantiate the two modules by dragging them from the *Definitions* tab page into the System Diagram.
- 4 On the toolbar, click to connect the two ports.

- 5 On the toolbar, click to build and run the test.

The following figure shows the result.

**Figure 8-14 Result of the Memory Test**

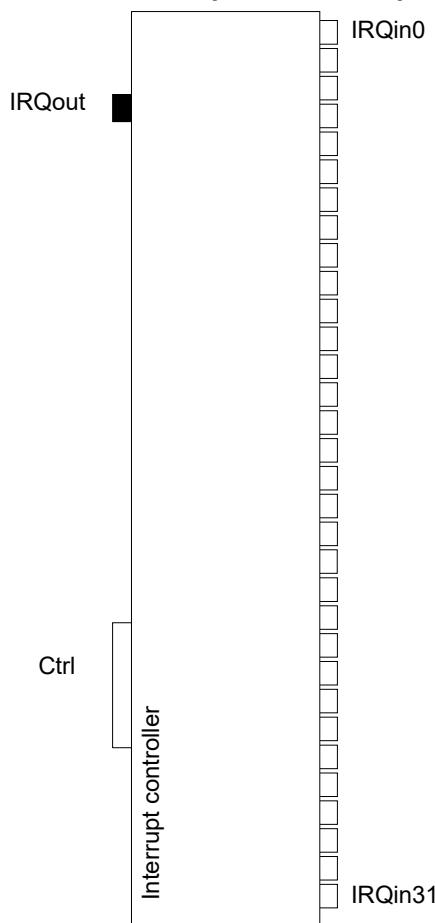


## 8.6.2 Modeling an Interrupt Controller

The purpose of this example is to introduce modeling of a register map, memory-mapped behavior, and modeling of non-memory-mapped interfaces. In this example, the flow from text specification to model testing is introduced.

The starting point for any component modeling activity is the specification for the component. In this case, the specification looks as shown in the following figure.

**Figure 8-15 Interrupt Controller Specification**



The interrupt controller is specified as follows:

- The interrupt controller has 32 interrupt inputs and one interrupt output. The output interrupt is active high. The active level of the input interrupts is programmable.
- The following tables outlines the registers by which the programming interface of the interrupt controller is defined.

**Table 8-9 Registers of the Interrupt Controller**

Register	Description
Enable	Each interrupt input can be enabled or disabled separately through this register. When a bit is set high, the corresponding interrupt inputs will be forwarded to the output.
Clear	When a bit is set high, the corresponding bit in the RawIrq register is cleared; this will clear that interrupt from the output.
ActiveHighNotLow	When a bit in this register is set high, the corresponding input is active high; else it is active low.
Status	Each bit in this register indicates whether the corresponding input is active and enabled.

Each bit in the Enable, Clear, ActiveHighNotLow, and Status registers corresponds to the interrupt with the same number.

- The following table shows how the memory map is defined.

**Table 8-10 Memory-Map Specification of the Interrupt Controller**

Address	Type	Width	Name
0x000	Read-Write	32	Enable
0x004	Read-Write	32	Clear
0x008	Read-Write	32	ActiveHighNotLow
0x00C	Read-only	32	Status

- The interrupt controller also has two internal registers:
  - RawIrq, which indicates the interrupt inputs that are currently active.
  - LatchedIrq, which indicates the active interrupts that are not cleared.These registers are not visible in the memory map.
- The output interrupt is active high and will be set whenever one of the input interrupts is active and enabled.

Before writing any code, it is important to define the approach that will be taken to create a model for a component. The modeling approach determines how the behavior of the component will be mapped to the SCML and SystemC modeling objects.

The modeling approach for the interrupt controller is as follows:

- There is a need for a memory-mapped interface and also an interface for the interrupt pins.
- The register set is modeled with SCML objects.
- The behavior of the component can be modeled through the callbacks that can be associated with the SCML registers.
- A SystemC process is required to model the independent or concurrent behavior of the interrupt controller. The callbacks on the registers are not sufficient since changes on the interrupt interfaces will also trigger changes to the registers and the output interrupt.
- The changes to registers and outputs can be calculated without accessing a memory-mapped socket; so it suffices to use a SystemC method, sensitive to the interrupt inputs. A SystemC method is more efficient than a SystemC thread for simulation speed.

Based on the above information, the definition of the interrupt controller can be created:

```
#include <scml2.h>
#include <tlm.h>
#include <systemc>

class ExampleInterruptController : public sc_core::sc_module
{
private:
    static const unsigned int NB_IRQ = 32;

public:
    sc_core::sc_in<bool>* IRQin[NB_IRQ];
    sc_core::sc_out<bool> IRQout;
1   tlm::tlm_target_socket<32> Ctrl;
2
```

The labeled code fragments are explained as follows:

- 1 The module contains a socket for the interface to the memory-mapped bus.
- 2 To model the interrupts, SystemC ports of type `sc_in` and `sc_out` are used. The template parameter for the `sc_in` and `sc_out` objects indicates the data type of the values carried over these ports, here `bool` to represent a one-bit value. For the 32 interrupt inputs, an array of pointers to input ports is defined. The pointers are used to avoid the default constructor for the pins, which does not allow changing the name for the pins. By using a pointer, it is possible to construct and initialize each pin with a meaningful name. The size of the array is set by a constant parameter of the module class.



**Note** The SystemC modeling object `sc_signal` should be used in order to connect `sc_in` and `sc_out` ports. Signals are a SystemC construct to represent hardware at a low level. The value of a signal is not immediately updated; it is updated only when all processes that are scheduled to run at the current time point have finished. The execution of a SystemC process can be made sensitive to a value change of a signal. This also means that new processes can be scheduled to run at a certain time point as a consequence of signal value changes that happened at the same time point in another process. Only when there are no new processes, scheduled time will advance. In the SCML methodology, signals are used to represent low-level hardware signals like interrupts and clocks.

```

private:
3 scml2::tlm2_gp_target_adapter<32> mAdapter;

4 scml2::memory<unsigned int> MemoryMap;
scml2::reg<unsigned int> Enable;
scml2::reg<unsigned int> Clear;
scml2::reg<unsigned int> ActiveHighNotLow;
scml2::reg<unsigned int> Status;

5 scml2::memory<unsigned int> InternalRegisters;
scml2::reg<unsigned int> RawIrq;
scml2::reg<unsigned int> LatchedIrq;

};


```

The labeled code fragments are explained as follows:

- 3 To model the register interface, the memory object `MemoryMap` is defined. This is the parent object for all memory-mapped registers in the component.
- 4 For the individual register, `scml2::reg` objects are used. The use of the register objects is to provide easy access and to define different behavior for the different registers. The actual storage is maintained by the `MemoryMap` object, as well as the interaction with the interface of the module.
- 5 For the internal registers, a separate memory called `InternalRegisters` is defined. Here it contains two additional registers `RawIrq` and `LatchedIrq`. `LatchedIrq` is a consequence of the implementation for the behavior.

To model the behavior of the interrupt controller, the memory-mapped behavior should be specified, as well as the concurrent behavior that is triggered on a change on one of the input interrupts. The setup for this is done in the constructor of the model class.

The public section looks as follows:

```
public :  
    SC_HAS_PROCESS(ExampleInterruptController);  
  
    explicit ExampleInterruptController(sc_core::sc_module_name name) :  
        1      sc_core::sc_module(name),  
              IRQout("IRQout"),  
              Ctrl("Ctrl"),  
              mAdapter("mAdapter", Ctrl),  
              MemoryMap("MemoryMap", 4),  
              Enable("Enable", MemoryMap, 0x0 >> 2),  
              Clear("Clear", MemoryMap, 0x4 >> 2),  
              ActiveHighNotLow("ActiveHighNotLow", MemoryMap, 0x8 >> 2),  
              Status("Status", MemoryMap, 0xc >> 2),  
              InternalRegisters("InternalRegisters", 2),  
              RawIrq("RawIrq", InternalRegisters, 0),  
              LatchedIrq("LatchedIrq", InternalRegisters, 1)  
    {  
        2      for (std::size_t i = 0; i < NB_IRQ; ++i) {  
              IRQin[i] = new  
                  sc_core::sc_in<bool>(sc_core::sc_gen_unique_name("IRQin"));  
        }  
  
        5      SC_METHOD(onIRQInChange);  
        for (std::size_t i = 0; i < NB_IRQ; ++i) {  
            sensitive << *IRQin[i];  
        }  
  
        mAdapter(MemoryMap);  
  
        6      MemoryMap.initialize(0);  
        InternalRegisters.initialize(0);  
        scml2::set_post_write_callback(Enable,  
                                      SCML2_CALLBACK(recomputeInterrupts));  
        4      scml2::set_post_write_callback(Clear,  
                                      SCML2_CALLBACK(postClearWrite));  
        scml2::set_read_only(Status);  
    }  
}
```

The labeled code fragments are explained as follows:

- 1 In the initialization, all objects get a name and the relation between the registers and the memory is defined. This includes the definition of the local address map for this component.  
This is done in the initialization of the register objects. The third parameter in the initialization is their index in the array of memory elements. Since the memory map is defined in byte addresses, we need to shift the address by 2 (or divide by 4).
- 2 The input interrupt pins need to be constructed and initialized in the constructor of the module since an array of pointers is used for the input interrupts.

- 3** By default, simple storage behavior is implemented by the memory and registers. To overwrite this behavior, callbacks functions can be set for these objects.
- It is possible to define different behavior for read and write accesses. Also a number of common behaviors are available. These can be set through global functions in the `scml2` namespace.
- An example is `scml2::set_read_only()`, which disables write accesses by returning an error response on the write transaction request.
- A consequence of overwriting the default behavior of a memory or register is that accesses via DMI will no longer be possible.
- 4** As mentioned in the specification (see “[Interrupt Controller Specification](#)” on page 290), the behavior of the registers is specified as follows: All registers are 32-bit registers, where each bit represents the corresponding interrupt input.
- **Enable:** When a bit is set high, the corresponding interrupt inputs will be forwarded to the output.
  - **Clear:** When a bit is set high, the corresponding bit in the `RawIrq` register is cleared.
  - **ActiveHighNotLow:** When a bit is set high, the corresponding input is active high.
  - **Status:** Each bit indicates whether the corresponding input is active and enabled.
- For the `Enable` and `Clear` register, it is required to add additional behavior through a callback. Different types of callbacks are possible. In this example, the `scml2::set_post_write_callback()` function is used to define the callback function. The arguments of this function are the memory object and a method of the current module which will be used as callback. There is also a third argument indicating synchronization. By default, the latter is set to `AUTO_SYNCING`. In this example, all callbacks are `AUTO_SYNCING`. Other functions exist for other types of behavior and for the different memory objects.
- The synchronization argument - when defining a callback for a register - is by default set to `AUTO_SYNCING`. This means that the system will synchronize before and after calling the callback function. As a consequence, the local time for the current initiator will be zero when the callback function is called. This allows to schedule and trigger events as well as to write to signals (for example, an output port). An alternative value for this argument is `SELF_SYNCING`. This should be used for simulation speed optimization. `SELF_SYNCING` can be used when the callback contains behavior that only in certain cases (or never) requires the system to synchronize. When using `SELF_SYNCING`, the local time for the initiator starting the transaction will be passed as an argument to the callback, which can use this information to synchronize the system. For callbacks that will never need to synchronize with the rest of the system, there is also a `NEVER_SYNCING` value for this argument (which is essentially the same as `SELF_SYNCING` but is provided for readability).
- 5** In the constructor, the method for the interrupt input changes is defined via the `SC_METHOD` macro and it is made sensitive to all input interrupts. This is done via the `sensitive <<` notation. A method is made sensitive to all events and signals that are specified following the macro invocation.
- 6** Finally, the initial value for the registers is set.

The private section looks as follows:

```
private:  
7     void end_of_elaboration()  
    {  
        IRQout = false;  
    }  
  
9     void postClearWrite()  
    {  
        LatchedIrq &= ~Clear;  
        recomputeInterrupts();  
    }  
  
10    void onIRQInChange()  
    {  
        RawIrq = 0;  
        int mask = 1;  
        for (int i = 0; i < NB_IRQ; ++i, mask <<= 1) {  
            if (IRQin[i]->read() ==  
                scml2::extract_bits(ActiveHighNotLow.get(), 1, i)) {  
                    RawIrq |= mask;  
                    LatchedIrq |= mask;  
                }  
        }  
  
        recomputeInterrupts();  
    }  
  
8     void recomputeInterrupts()  
    {  
        Status = Enable & LatchedIrq;  
        IRQout = (Status != 0); // Active high  
    }  

```

The labeled code fragments are explained as follows:

- 7 The output interrupt should not be initialized in the constructor as it may be writing to a signal that may not be constructed yet. Output signals should be initialized in the `end_of_elaboration()` method. This method is part of the `sc_module` base class and is called by the simulation kernel after all objects have been constructed. The output interrupt is active high.
- 8 The core behavior of the interrupt controller is to compute whether the output interrupt signal should be active. This behavior is captured in the `recomputeInterrupts()` method. This method is also used as the callback function for the `Enable` register, since this value is directly related to the interrupt output. For the computation, the `LatchedIrq` register is used. The naming for this method does not follow the naming rules because it is used both as a callback as well as a regular method.
- 9 For the `Clear` register, `LatchedIrq` is first updated with the new `Clear` value and then the output is recomputed.

- 10** Whenever the value of an input interrupt changes, the `onIRQInChange()` method is called. In this method, `LatchedIRQ` is updated based on the value of the input pins and the active level for that pin, set in the `ActiveHighNotLow` register. This value is also stored in the `RawIRQ` register.

Each model requires a test setup to ensure that all features are correctly modeled. The test setup needs to make sure that the following features are validated:

- The memory map is correctly specified, including read/write permissions.
- Reading and writing to the individual registers delivers the expected results.
- All behavior scenarios that can be derived from the specification are correctly modeled.

To test an individual component, the SCML2 example library contains `TclInitiator`. This is an example of an initiator targeted to test individual models. It has a TLM2.0 generic payload socket and a flexible number of input and output pins. Most important is that it has a Tcl scripting interface targeted to create test scripts for a model.

The Tcl interpreter is linked to the module and adds some additional Tcl commands that allow to:

- Read and write to the initiator socket
- Get values from input pins and set values on the output pins
- Advance local time and manage the quantum
- Wait for input signal changes or advance global time

The module reads commands from a script and terminates the simulation when the script is finished or when there are no more events in the SystemC simulation. With this test infrastructure it is possible to develop unit tests for a component using the Tcl scripting language.

For the interrupt controller, a unit test can be created, as described in the following procedure.

#### To run the interrupt controller test:

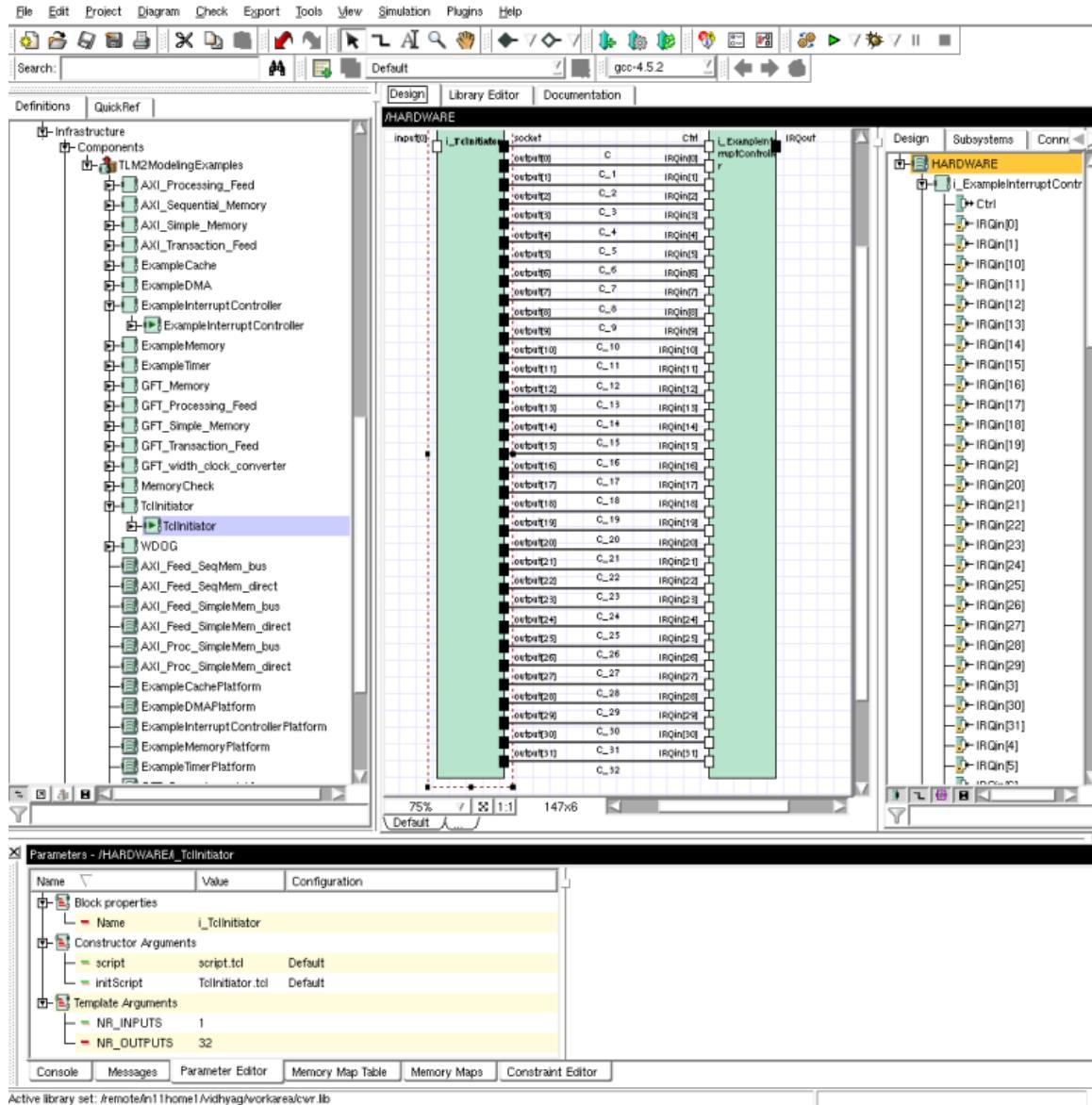
- 1 Launch Platform Creator:

```
pct &
```

- 2 Open the `TLM2ModelingExamples` library. To do so, right-click on `TLM2ModelingExamples` on the *Definitions* tab page and from the pop-up menu, select *Open*.
- 3 Instantiate the `ExampleInterruptController` module. To do so, drag `ExampleInterruptController` from the *Definitions* tab page into the System Diagram.
- 4 Instantiate the `TclInitiator` module. To do so, drag `TclInitiator` from the *Definitions* tab page into the System Diagram.
- 5 Configure `TclInitiator`. To do so:
  1. In the System Diagram, select the `TclInitiator` instance.
  2. Click the *Parameters* tab at the bottom of the Platform Creator window.
  3. Set the number of output ports to 32 (to drive all the interrupt inputs of the interrupt controller).
  4. Update the instance view by right-clicking on the instance in the System Diagram and selecting *Update Dynamic Port Arrays* from the pop-up menu.
  5. If desired, you can change the names of the Tcl scripts that will be run (`initscript` and `script`).
- 6 Connect the initiator socket with the `Ctrl` socket of the interrupt controller, connect the output pins to the `IRQin` pins (make sure the indices correspond), and connect the input pin to the `IRQout` pin.
- 7 On the toolbar, click  to build the test.

The following figure shows the result.

**Figure 8-16 Result of Interrupt Controller Example**



To run the test, a Tcl script is required. According to the parameter settings for the constructor arguments it should be called `script.tcl`. In this script, a number of procedures are defined to access the individual registers of the memory map. These should be added to improve readability. For each register, a set and

get function should be added. These functions use the `readX` and `writeX` commands of `TclInitiator`, where  $X$  indicates the number of bytes that will be read or written. The following code shows an example.

```
proc setEnable { value } {
    write4 0x0 $value
}

proc getEnable { } {
    return [read4 0x0]
}
```

To access the input and output pins, `TclInitiator` provides the `setOutput index value` and `getInput index` commands, where  $index$  gives the index of the port to be used for the access.

The test itself goes through the following scenarios:

1. The test tries an access on all registers (read and write). This should give an error when attempting to write to the Status register.
2. Next, the test initializes the ActiveHighNotLow register so that the first 16 interrupt inputs are active high and the rest are active low.
3. Then the test tries every interrupt by enabling it, setting it, waiting for the interrupt to come in, and clearing it again through the Clear register.

If the test fails, this usually means that it is waiting for an event that will never come. This will finish the simulation before the whole test is done.

The remaining code for the test script looks as follows:

```
puts stderr "TEST STARTED"
puts stderr "-----"

# register access test
puts stderr "reading all registers"
puts stderr [format "Enable:          0x%X" [getEnable] ]
puts stderr [format "Clear:           0x%X" [getClear] ]
puts stderr [format "ActiveHighNotLow:0x%X" [getActiveHighNotLow] ]
puts stderr [format "Status:          0x%X" [getStatus] ]
puts stderr "done"
puts stderr "writing all registers"
setEnable 1
setClear 1
setActiveHighNotLow 1
setStatus 1
puts stderr "done"

# initialize output pins ( first 16 active high, rest active low)
setActiveHighNotLow 0xFFFF
puts stderr [format "ActiveHighNotLow set to 0x%x" [getActiveHighNotLow] ]
for { set i 0 } { $i < 32 } { incr i 1 } {
    if { $i < 16 } { setOutput $i 0 } else { setOutput $i 1 }
}

# test all interrupts
for { set i 0 } { $i < 32 } { incr i 1 } {
    setEnable [ expr ( 1<<$i ) & 0xFFFFFFFF ]
    if { $i < 16 } { setOutput $i 1 } else { setOutput $i 0 }
    waitForInput 0 0
    puts stderr [format "received interrupt 0x%X" [getStatus] ]
    setClear [ expr ( 1<<$i ) & 0xFFFFFFFF ]
    waitForInput 0 1
    if { $i < 16 } { setOutput $i 0 } else { setOutput $i 1 }
    setEnable 0
}

puts stderr ""
puts stderr "TEST DONE"
puts stderr "-----"
```

When running the test, the following output is generated:

```
Copyright (c) 1996-2009 CoWare, Inc.  
1731 Technology Drive, San Jose, CA, 95110, U.S.A.  
  
Copyright (c) 2004 Cadence, Inc.  
2655 Seely Avenue, San Jose, CA, 95134, U.S.A.  
  
SystemC 2.2.0 --- Jul 23 2009 11:32:27  
Copyright (c) 1996-2006 by all Contributors  
ALL RIGHTS RESERVED  
  
TEST STARTED  
-----  
reading all registers  
Enable: 0x0  
Clear: 0x0  
ActiveHighNotLow:0x0  
Status: 0x0  
done  
writing all registers  
write of address 0xc failed  
done  
ActiveHighNotLow set to 0xfffff  
received interrupt 0x1  
received interrupt 0x2  
received interrupt 0x4  
received interrupt 0x8  
received interrupt 0x10  
received interrupt 0x20  
received interrupt 0x40  
received interrupt 0x80  
received interrupt 0x100  
received interrupt 0x200  
received interrupt 0x400  
received interrupt 0x800  
received interrupt 0x1000  
received interrupt 0x2000  
received interrupt 0x4000  
received interrupt 0x8000  
received interrupt 0x10000  
received interrupt 0x20000  
received interrupt 0x40000  
received interrupt 0x80000  
received interrupt 0x100000  
received interrupt 0x200000  
received interrupt 0x400000  
received interrupt 0x800000  
received interrupt 0x1000000  
received interrupt 0x2000000  
received interrupt 0x4000000  
received interrupt 0x8000000  
received interrupt 0x10000000  
received interrupt 0x20000000  
received interrupt 0x40000000  
received interrupt 0x80000000  
  
TEST DONE
```

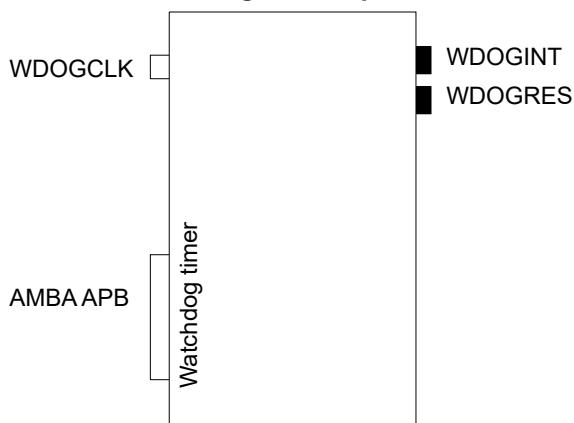
### 8.6.3 Modeling a Watchdog Peripheral

In this example, the specification is derived from an actual peripheral. The key features introduced in this example are system-level synchronization and timing modeling. These are introduced via an example watchdog peripheral. A *watchdog timer* is a peripheral used to reset a system if the application or OS hangs or has run into a failure. A watchdog timer is a counter with a programmable time-out interface which will generate an interrupt whenever the time-out is reached.

As part of the AMBA peripheral IP delivered by Arm, a watchdog is specified (reference number: SP805). A model of this component is delivered as part of the Synopsys Primecell Model Library. The specification can

be found on the website of Arm ([www.arm.com](http://www.arm.com)). The watchdog described in this section is a simplified version of that component. The specification looks as shown in the following figure.

**Figure 8-17 Watchdog Timer Specification**



The functionality of the watchdog timer is specified as follows:

- The watchdog will count down at the rate of a clock input, starting from a value stored in the load register. When the counter reaches zero, an interrupt is sent and the counter restarts. When the counter reaches zero again, a reset signal is set. The counter can be cleared through the interrupt clear register.
- The following table outlines the interface of this component.

**Table 8-11 Interface of the Watchdog Timer**

Name	Description
AMBA APB	Memory-mapped interface (APB in this case).
WDOGCLK	Clock input.
WDOGINT	Interrupt output. It is active high.
WDOGRES	Interrupt-reset output. It is active low.

- The following tables outlines the registers and bitfields by which the programming interface of the watchdog timer is defined.

**Table 8-12 Registers and Bitfields of the Watchdog Timer**

Register	Description
WdogLoad	The value written to this register sets the value for the time-out. The default value is 0xFFFFFFFF
WdogValue	Whenever this register is read, the current counter value should be returned. The default value is 0xFFFFFFFF.
WdogControl	Contains the bitfields RESEN and INTEN to enable the functionality of the watchdog timer.
WdogIntClr	A write access to this register causes the counter to be stopped.

Register	Description
WdogRIS	Is a register that contains the raw interrupt status.
WdogMIS	Is a register that contains the masked interrupt status.
WdogLock	When the value 0x1ACCE551 is written to this register, all registers are accessible. Any other value will lock the watchdog registers. Reading the register should return 1 on lock; 0 when not locked. At reset, the registers are unlocked.
WdogITCR	Contains the ITEN bitfield to control the integration test mode.
WdogITOP	Contains the WDOGINTBIT and WDOGRESBIT to be used in integration test mode.
Bitfield	Description
RESEN	Is used to enable the watchdog reset outputs. It needs to be set high to enable the reset. By default, the reset is disabled.
INTEN	Should be set high to enable the counter and the interrupt. When re-enabled, the counter value is reloaded. By default, the interrupt is disabled.
ITEN	Is the integration test enable bit. When set high, the watchdog is set into integration test mode.
WDOGINTBit	Is the value output on WDOGINT when in integration test mode.
WDOGRESBit	Is the value output on WDOGRES when in integration test mode.

- The watchdog timer has a memory range defined of size 0x1000. The following table outlines its register definitions.

**Table 8-13 Register Definitions of the Watchdog Timer**

Address	Type	Width	Name
0x000	Read-Write	32	WdogLoad
0x004	Read-only	32	WdogValue
0x008	Read-Write	2	WdogControl
0x00C	Write-only		WdogIntClr
0x010	Read-only	1	WdogRIS
0x014	Read-only	1	WdogMIS
0x018-0xBFC	Reserved		
0xC00	Read-Write	32	WdogLock
0xC04-0xEFC	Reserved		
0xF00	Read-Write	1	WdogITCR
0xF04	Write-only	2	WdogITOP
0xF08-0xFFC	Reserved		

- Bit specification

In the WdogControl register, different bits have a special meaning, as shown in the following table.

**Table 8-14 WdogControl Bit Specification**

Bits	Type	Name
[31:2]	Reserved	
[1]	Read-Write	RESEN
[0]	Read-Write	INTEN

The WdogITCR register contains one bitfield ITEN, as shown in the following table.

**Table 8-15 WdogITCR Bit Specification**

Bits	Type	Name
[31:1]	Reserved	
[0]	Read-Write	ITEN

The WdogITOP register contains two bitfields, as shown in the following table.

**Table 8-16 WdogITOP Bit Specification**

Bits	Type	Name
[31:2]	Reserved	
[1]	Write-only	WDOGINT
[0]	Write-only	WDOGRES

The two bitfields specified for WdogITOP have the same name as the output pins they drive. As it is not possible to have several objects with the same name in C++, the name for the bitfields will be extended with the postfix Bit.

Before writing any code it is important to define the approach that will be taken to create a model for a component. The modeling approach determines how the behavior of the component will be mapped to the SCML and SystemC modeling objects.

The modeling approach for the watchdog timer is as follows:

- The memory-mapped registers and bitfields are modeled using the SCML2 memory objects.
- The memory-mapped interface is modeled using a TLM2.0 socket.
- The interrupt and reset output is modeled using SystemC output pins.
- The clock input can be modeled using an input pin. However, in general, clocks are not required for a peripheral model for virtual prototypes as the timing is driven from the initiators and timing is mainly used as an easy way to model system synchronization. Since the behavior depends on the timing set by the clock, the clock input pin is required.

- To model the behavior, SCML provides callbacks that can be associated with the register accesses.
- The interrupt and reset output behavior cannot be modeled using callbacks on register accesses as the interrupt and reset signals will be set independent of a current register access. Their value needs to be set at a future simulation point.
- The obvious solution is to have a method sensitive to the clock and count down on each clock tick. This would be very inefficient for simulation speed as this implies a process switch on every clock cycle in the simulation model. This should be avoided at all cost.

Another approach is used in this example, which limits the number of process switches in the SystemC simulation kernel: An SCML2 `clocked_callback` is used. The callback is scheduled to trigger at the correct time point.

This point can be calculated whenever the counter value is written. The output signals are then set from the callback. This approach transparently handles all parameter changes of the clock, like changes of the clock frequency or disabling and re-enabling gated clocks.

- The behavior of the lock register and the integration test registers is that they change the access rights and behavior of other registers. To model that, each register could get a callback registered to check whether the component is in lock mode or in integration test mode. The drawback of this is additional complexity to manage the state machine implied by this; furthermore, it disables DMI accesses to all registers. An alternative approach is to use dynamic registration and removal of callbacks, which removes the need to check in every register access which state the component is in, and allows to revert to default behavior whenever possible.

Based on the information so far, an initial skeleton for the model can be created, as shown in the following code.

```
#include <systemc>
#include <tlm.h>
#include <scml2.h>

class WDOG : public sc_core::sc_module
{
public:
    tlm::tlm_target_socket<32> APB;
    sc_core::sc_out < bool > WDOGINT;
    sc_core::sc_out < bool > WDOGRES;
    sc_core::sc_in < bool > WDOGCLK;

1   private:
    scml2::tlm2_gp_target_adapter<32> mAdapter;

    scml2::memory<unsigned int> MemoryMap;

2   scml2::reg<unsigned int> WdogLoad;
    scml2::reg<unsigned int> WdogValue;
    scml2::reg<unsigned int> WdogControl;
    scml2::reg<unsigned int> WdogIntClr;
    scml2::reg<unsigned int> WdogRIS;
    scml2::reg<unsigned int> WdogMIS;
    scml2::reg<unsigned int> WdogLock;
    scml2::reg<unsigned int> WdogITCR;
    scml2::reg<unsigned int> WdogITOP;

    scml2::bitfield<unsigned int> RESEN;
    scml2::bitfield<unsigned int> INTEN;
    scml2::bitfield<unsigned int> ITEN;
    scml2::bitfield<unsigned int> WDOGINTBit;
    scml2::bitfield<unsigned int> WDOGRESBit;

3   scml2::clocked_callback mTimeout;

4   bool mWdogTimeoutForReset;
```

The labeled code fragments are explained as follows:

- 1 The module contains a socket for the interface to the memory-mapped bus. To model the interrupt and clock signals, SystemC ports of type `sc_in` and `sc_out` of type `bool` are used. The reason to have a clock port is to allow this component to be used in virtual prototypes running at different frequencies. The clock is used to schedule a callback for the clock tick, at which the watchdog times out. This approach will only work if the complete clock tree in the design is using `scml_clocks` (which is anyway advised; `scml_clocks` are optimized for speed). Clock objects are defined in SCML1, so they are not part of the `scml2` namespace.

- 2 The `MemoryMap` object represents the whole address range of the component. The individual registers are defined using the SCML object `scml2::reg`. A register object is a convenience object to refer to a single element within a memory object. Same holds for a bitfield object: It allows access to individual bit ranges in a register and simplifies the code to manage accesses to these bits. Bitfields are defined for the bits in the `WdogControl`, `WdogITCR`, and `WdogITOP` registers, since these bitfields can be written. In the `WdogRIS` and `WdogMIS`, there is a single read-only bit. No extra objects are added for those as these values can be easily managed from within the behavior.
- 3 To model the time-out of the watchdog, an SCML2 `clocked_callback` is defined.
- 4 Finally, there is a `bool` variable that will be used to check whether the signal that will be set when the counter reaches zero is the interrupt output or the reset output.

The setup for the modeling approach is done in the class constructor of the model. The constructor defines the memory map for the component. It is also the place to define the callbacks for the registers and the threads and methods for concurrent behavior. The code for the constructor is shown below.

```
public:  
    SC_HAS_PROCESS(WDOG);  
  
WDOG(sc_core::sc_module_name name) :  
    sc_core::sc_module(name)  
  
    , APB("APB")  
    , WDOGINT("WDOGINT")  
    , WDOGRES("WDOGRES")  
    , WDOGCLK("WDOGCLK")  
    , mAdapter("mAdapter", APB)  
    , MemoryMap("MemoryMap", 0x1000 >> 2)  
    , WdogLoad("WdogLoad", MemoryMap, 0x000 >> 2)  
    , WdogValue("WdogValue", MemoryMap, 0x004 >> 2)  
    , WdogControl("WdogControl", MemoryMap, 0x008 >> 2)  
    , WdogIntClr("WdogIntClr", MemoryMap, 0x00C >> 2)  
    , WdogRIS("WdogRIS", MemoryMap, 0x010 >> 2)  
    , WdogMIS("WdogMIS", MemoryMap, 0x014 >> 2)  
    , WdogLock("WdogLock", MemoryMap, 0xC00 >> 2)  
    , WdogITCR("WdogITCR", MemoryMap, 0xF00 >> 2)  
    , WdogITOP("WdogITOP", MemoryMap, 0xF04 >> 2)  
    , RESEN("RESEN", WdogControl, 1, 1)  
    , INTEN("INTEN", WdogControl, 0, 1)  
    , ITEN("ITEN", WdogITCR, 0, 1)  
    , WDOGINTBit("WDOGINTBit", WdogITOP, 1, 1)  
    , WDOGRESBit("WDOGRESBit", WdogITOP, 0, 1)  
    ("timeout")  
    , mWdogTimeoutForReset(false)  
  
{  
    mAdapter(MemoryMap);  
2    scml2::set_read_only(WdogValue);  
    scml2::set_write_only(WdogIntClr);  
    scml2::set_read_only(WdogRIS);  
    scml2::set_read_only(WdogMIS);  
3    scml2::set_write_only(WdogITOP);  
  
4    setAllCallbacks();  
  
5    mTimeout.set_callback(this, &WDOG::onTimeoutEvent);  
    MemoryMap.initialize(0);  
    WdogLoad = 0xfffffffffu;  
    WdogValue = 0xfffffffffu;  
}
```

The labeled code fragments are explained as follows:

- 1 All objects get a name and the register file is connected to the socket via the adapter. For each register and bitfield, the location in the register file is specified. Finally, the internal variables are initialized.
- 2 The access restrictions for the different registers are specified here as well.
- 3 The callback to register the behavior for the memory-mapped registers is set up through the `setAllCallbacks()` method, which is discussed [below](#).
- 4 Next, the callback which will be used to implement the behavior for signaling the time-out is defined and hooked into the `clocked_callback` object.
- 5 The final piece of code in the constructor initializes the memory-mapped storage to 0. The registers for which a default value is specified (see “[Modeling a Watchdog Peripheral](#)” on page 301) also are initialized.

The `setAllCallbacks()` method takes care of registering the callbacks with the memory-mapped behavior. For the registers where an access has an impact on the further behavior of the component, a callback is registered with the register to model this behavior. The behavior of the registers is defined as follows (see also “[Modeling a Watchdog Peripheral](#)” on page 301).

WdogLoad	The value written to this register sets the value for the time-out. So on an access to this register, a future callback needs to be scheduled to trigger the interrupt.
WdogValue	Whenever this register is read, the current counter value should be returned. Since the counter will not be updated on every clock cycle, this value needs to be calculated on each access.
WdogIntClr	A write access to this register causes the counter to be stopped.
WdogLock	When the value 0x1ACCE551 is written to this register, all registers are accessible. Any other value will lock the watchdog registers. Reading the register should return 1 on lock, 0 when not locked.

Also for the bitfields, behavior can be added through callbacks. The specification of the bitfields is as follows:

RESEN	Is used to enable the watchdog reset outputs. It needs to be set high to enable the reset.
INTEN	Is used to enable the watchdog reset outputs. It needs to be set high to enable the reset.
ITEN	Is the integration test enable bit. When set high, the watchdog is set into integration test mode.
WDOGINTBit	Is the value output on WDOGINT when in integration test mode.
WDOGRESBit	Is the value output on SDOGRES when in integration test mode.

In most cases, the read behavior is to return the current value or a predefined value for the register. As this is the default behavior, there is no need to overwrite this through a function call. This will also make sure that DMAs to those registers are still enabled, which guarantees that for read accesses there is no speed impact.

Since it is required to register and remove callback behavior from the registers, the code for registering the callbacks is moved to a couple of methods that will be called from the constructor and whenever behavior needs to be reinstated. The code for these functions is shown below.

```
void setAllCallbacks()
{
    setNormalCallbacks();
    setTestCallbacks();
}

void setNormalCallbacks()
{
    scml2::set_post_write_callback(WdogLoad,
                                    SCML2_CALLBACK(postWdogLoadWrite));
    scml2::set_word_read_callback(WdogValue,
                                  SCML2_CALLBACK(onWdogValueRead));
    scml2::set_post_write_callback(WdogIntClr,
                                  SCML2_CALLBACK(postWdogIntClrWrite));
    scml2::set_write_callback(WdogLock, SCML2_CALLBACK(onWdogLockWrite),
                             scml2::NEVER_SYNCING);
    scml2::set_write_callback(INTEN, SCML2_CALLBACK(onINTENWrite));
}

void setTestCallbacks()
{
    scml2::set_word_write_callback(ITEN, SCML2_CALLBACK(onITENWrite));
    scml2::set_post_write_callback(WDOGINTBit,
                                  SCML2_CALLBACK(postWDOGINTBitWrite));
    scml2::set_post_write_callback(WDOGRESBit,
                                  SCML2_CALLBACK(postWDOGRESBitWrite));
}
```

For most of the write accesses, there is no need to change the storage behavior of the register. However, some additional behavior should be modeled whenever a value is written to the register. In this case, callback behavior can be added using the `scml2::set_post_write_callback()` function. For the watchdog module, most of the callbacks are AUTO\_SYNCING. Most behaviors will need to schedule and/or cancel the time-out event, or set an output port for the module. Since AUTO\_SYNCING is the default setting, it is not mentioned in the calls.

The labeled code fragments are explained as follows:

- 1 For the lock register, the value written is not stored. So the callback behavior is added to the register using the `scml2::set_write_callback()` function. The synchronization for the lock register can be set to `NEVER_SYNCING` since this register access has no side-effects.
- 2 A special case is the callback on a read access for the `WdogValue` register. Since the model will not implement an actual counter for the time-out, there is no register holding the current counter value; this will need to be calculated on each access to the `WdogValue` register. Therefore, the callback used is set via the `scml2_set_read_callback()` function, which means that the data to be stored in the register will be calculated in the behavior (as opposed to the `post_callbacks`). To be able to calculate the current value of the counter, the system should be synchronized to the SystemC time; therefore this callback is `AUTO_SYNCING`.
- 3 For the `ITEN` bitfield, the behavior is to check if the model changes from a test mode into normal mode. This should be checked before the value is stored in the bitfield. So also here it is not possible to use a `post_write_callback`; therefore, a regular `write_callback` is used.

The output pins of the component are initialized in the `end_of_elaboration()` method. Here also, the clock is assigned to the `clocked_callback`. This can only happen now, since it requires that the clock port is already connected to the clock generator. Now where the clock and its period parameter are defined, it is possible to request the callback for the default time-out through the `reCalculateTimeout()` method, which is shown below.

```
void end_of_elaboration()
{
    mTimeout.set_clock(WDOGCLK);
    WDOGRES = 1;
    WDOGINIT = 0;
    reCalculateTimeout();
}
```

The core of the behavior of the watchdog module is in the calculation of the time-out callback and the method signaling the interrupt and reset outputs, as shown below.

```
1 void reCalculateTimeout()
{
    mTimeout.cancel_trigger();
    if (INTEN) {
        mTimeout.request_trigger(WdogLoad);
    }
}

2 void onTimeoutEvent()
{
    if (mWdogTimeoutForReset) {
        WDOGRES = RESEN ? 0 : 1;

    } else {
        WDOGINT = 1;
        WdogRIS = 1;
        WdogMIS = INTEN & WdogRIS;
        mWdogTimeoutForReset = true;
        reCalculateTimeout();
    }
}

3 unsigned int getCounterValue()
{
    if (!INTEN) {
        return WdogValue;
    }

    if (!mTimeout.is_clock_trigger_requested()) {
        return 0;
    }

    return (unsigned int)(mTimeout.get_scheduled_clock_tick() -
                         mTimeout.get_clock()->get_tick_count()-1);
}
```

The labeled code fragments are explained as follows:

- 1 The calculation of the time-out is captured in the `reCalculateTimeout()` method, which is called whenever the time-out needs to be recalculated. The time-out is issued by a clock tick callback that happens at the specified tick of the clock. At first, a potential former request for a clock tick callback is canceled. Next, a new callback is requested for the case that the watchdog counter is enabled. The future clock tick for the callback is determined by the value of the `WdogLoad` register. The argument to `request_trigger()` specifies how many clock ticks to skip before receiving the callback. Therefore a value of 0 will request the callback for the beginning of the next clock period.
- 2 The implementation of the method that is triggered on the time-out signals the interrupt or reset output signal and sets the corresponding bitfields. The `mWdogTimeoutForReset` member variable keeps track of whether the time-out is scheduled for an interrupt output or a reset output. In case the time-out is for interrupt, the time-out for the reset needs to be scheduled via the `reCalculateTimeout()` method.
- 3 As mentioned in “[Modeling a Watchdog Peripheral](#) on page 301”, the value for the `WdogValue` register will be calculated each time it is read. The calculation considers the difference between the tick count for which the time-out callback is scheduled and the current tick count of the clock. Two cases must be handled in a special way: If the timer is not enabled, then the last value of the watchdog counter is returned, as it was saved when the timer was disabled. Next, if the register is read from a time-out callback, and the next callback was not yet requested, then a zero value is returned.

The callbacks on the `WdogLoad`, `WdogValue`, and `WdogIntClr` registers are used to control the time-out, as shown below.

```
1 void postWdogLoadWrite()
{
    if (INTEN) {
        reCalculateTimeout();
    }
}

2 bool onWdogValueRead(unsigned int& data)
{
    data = getCounterValue();
    return true;
}

3 void postWdogIntClrWrite()
{
    if (INTEN) {
        reCalculateTimeout();
        mWdogTimeoutForReset = false;
        WDOGINT = WdogMIS = 0;
    }
    WdogRIS = 0;
}
```

The labeled code fragments are explained as follows:

- 1 In the callback for the load register, the time-out is recalculated if the watchdog is enabled.

- 2 For the read callback on the value register, the counter value is calculated on demand, as explained [above](#).
- 3 On interrupt clear, the interrupt output is cleared and the callback is rescheduled. Also here the state is set so that the next time-out will lead to an interrupt output.

### Note

Callbacks registered via the `scm12::set_post_write_callback` method do not take any argument and also do not have a return value. The callbacks registered via the `scm12::set_word_write_callback` method have a `bool` return and take a data argument of the same type as the register has. The `bool` return value for the callback indicates an error for the word access. These callbacks imply that only word accesses are allowed for this register; subword or byte accesses will return an error. Other callback registration functions allow registering callbacks that can handle subword accesses as well.

When the `INTEN` bitfield is written, the watchdog can be disabled. In that case, the time-out must be canceled. When the watchdog is re-enabled after disabling it, the time-out restarts. The code looks as follows:

```
bool onINTENWrite(const unsigned int &data)
{
    if (data == INTEN) {
        return true;
    }

    if (data == 0) {
        WdogValue = getCounterValue();
        WDOGINT = WdogMIS = 0;
    } else {
        mWdogTimeoutForReset = false;
        WDOGINT = WdogMIS = WdogRIS;
    }

    INTEN = data;
    reCalculateTimeout();
    return true;
}
```

The behavior of the lock register requires that no register can be accessed except for the lock register itself. To model this, all callbacks can be removed and for the complete memory range, all accesses are set to be ignored. This means no updates will be allowed. When accesses are unlocked by writing the correct value to

the lock register, callbacks are reset and accesses to the memory range are allowed again. The following code illustrates this.

```
bool onWdogLockWrite(const unsigned int &data)
{
    if (data == 0x1acce551u) {
        if (WdogLock) {
            MemoryMap.remove_callback();
            if (!ITEN) {
                setNormalCallbacks();
            }
            setTestCallbacks();
            WdogLock = 0;
        }
    } else {
        removeAllCallbacks();
        scml2::set_ignore_access(MemoryMap);
        WdogLock = 1;
    }
    return true;
}
```



When removing callbacks, also the read-only or write-only settings are removed (since these are implemented through default callbacks). So only the behavioral callbacks should be removed. The following code illustrates this.

```
void removeNormalCallbacks()
{
    WdogLoad.remove_callback();
    WdogValue.remove_read_callback();
    WdogIntClr.remove_write_callback();
    INTEN.remove_callback();
    RESEN.remove_callback();
}

void removeTestCallbacks()
{
    ITEN.remove_callback();
    WDOGINTBit.remove_callback();
    WDOGRESBit.remove_callback();
}
```

Finally, there is the code for the integration test mode, as shown below. This is similar to the code for the ITEN enable register and the lock register. When the watchdog is in integration test mode, the callbacks for the registers that should not be used for integration test are removed and they are restored when the

watchdog exists from integration test mode. In the callbacks for the WDOGINTBit and WDOGRESBit bitfields, the corresponding output signals are set.

```
bool onITENWrite(const unsigned int &data)
{
    if (data) {
        if (!ITEN) {
            mTimeout.cancel();
            removeNormalCallbacks();
            ITEN = 1;
            WDOGINT = WDOGINTBit;
            WDOGRES = WDOGRESBit;
        }
    } else {
        if (ITEN) {
            reCalculateTimeout();
            setNormalCallbacks();
            ITEN = 0;
            WDOGINT = WdogMIS;
            WDOGRES = 0;
        }
    }
    return true;
}

void postRESENWrite()
{
    if (!RESEN) {
        WDOGRES = 1;
    }
}

void postWDOGINTBitWrite()
{
    if (ITEN) {
        WDOGINT = WDOGINTBit;
    }
}

void postWDOGRESBitWrite()
{
    if (ITEN) {
        WDOGRES = WDOGRESBit;
    }
}
```

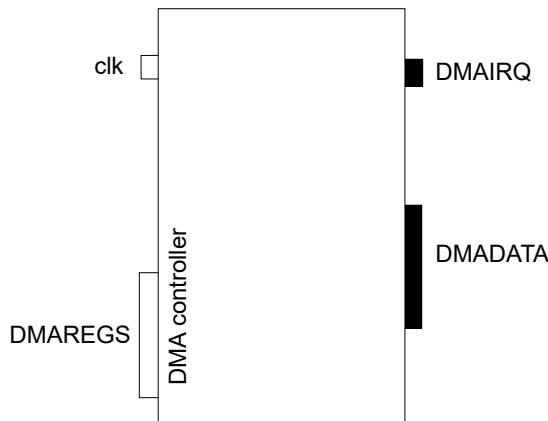
#### 8.6.4 Modeling a DMA

The example in this section is not derived from an actual peripheral, but is intended to show the modeling approach to create models for components that initiate transactions in a system. The example also

introduces the use of parameters to create configurable models. The example component is a Memory Access (DMA), as this is a typical component for a system while still different from a pure initiator like a processor. A DMA is a device that allows moving data from one memory to another while allowing the software on a processor to continue to operate. A DMA is programmable through some registers and it will behave as an initiator in the system to move the data around. To return control to the software, it will emit an interrupt.

The specification looks as shown in the following figure.

**Figure 8-18 DMA Specification**



The DMA is specified as follows:

- The DMA example has three different interfaces: One interface to allow software to program the data transfers, a second interface to initiate the data transfers, and an interrupt output that will be used to signal back to the processor executing software. Further, the DMA example has a clock pin. The following table outlines the specification for the pins of this example.

**Table 8-17 Interface of the DMA**

Name	Description
DMADATA	Interface for data transfers
DMAREGS	interface to access the memory-mapped registers of the DMA
DMAIRQ	interrupt output
Clk	Clock input

- The programming interface for the DMA is specified through the DMAREGS interface, which provides access to the configuration registers of the DMA. The memory range of this simple DMA is 0x18 and the internal registers are defined as shown in the following table.

**Table 8-18 DMA Specification**

Address	Type	Width	Name
0x000	Read-Write	1	DmaEnable
0x004	Read-only	1	DmaStatus

Address	Type	Width	Name
0x008	Write-only	n/a	DmaIrqClr
0x00C	Read-Write	32	DmaSize
0x010	Read-Write	32	DmaSrcAddr
0x014	Read-Write	32	DmaDstAddr

- As this is a simple DMA, only one DMA access can be programmed at any time. The following table shows the use for the different registers.

**Table 8-19 Registers of the DMA**

Register	Description
DmaEnable	Indicates a request to start a DMA transfer from the software. This register is only 1 bit wide, but can be accessed through a regular 32-bit access. This means only the first bit contains an actual value.
DmaStatus	Is used by the DMA to reflect the output interrupt status. This register is only 1 bit wide.
DmaIrqClr	Is used to clear the output interrupt. The value of this register has no meaning.
DmaSize	The register should be programmed to contain the size of the data transfer (in bytes).
DmaSrcAddr	Should be programmed to contain the source address of the data to be copied (byte address).
DmaDstAddr	Should be programmed to contain the destination address for the data to be copied to (in byte addresses).

In its simplest form, a DMA has a target socket to access the configuration registers, an initiator socket for the data transfers, and an interrupt port to signal back to the software. A thread is required to model the data transfers as these should run independently of any register access.

- The behavior of the DMA module is driven from the DmaEnable and DmaIrqClr registers. When the DmaEnable register is written, a DMA transfer is started unless one is already underway, in which case the request is ignored.
- When a DMA transfer is finished, the interrupt output is set. To reset the interrupt value, the DmaIrqClr register should be written to.
- The thread that will execute the DMA transfer runs independently of the register accesses. To synchronize between the DmaEnable access and the thread, an event is used. The thread is suspended until the event is notified and after finishing processing the DMA transfer the thread goes back to sleep.
- When doing a DMA transfer, the DMA model should store the data that is read into some local buffer and then write it to the destination address. The size of that buffer can be fixed; in this model the size of the buffer is made configurable. For this purpose, an scml\_property is added that will represent the size of the buffer.

- Since this component will initiate transactions independent of the software execution, there is a need to determine the relative synchronization of the DMA transfer to the software execution. Timing is used to implement this synchronization. Therefore, the clock input is present in this model; it allows to get the clock period for this component and a simple delay model for initiating the individual DMA transfers can be computed.



- It is possible to define a static sensitivity list for a SystemC thread (in “[Modeling Concepts](#)” on [page 243](#), sensitivity is only explained for methods). Whenever a static sensitivity is defined for a SystemC thread, it is possible to use a `wait()` call to suspend execution until the next event in the sensitivity list is triggered.  
It is not possible to use a method with static sensitivity to model the DMA transfers since we will be using the blocking interface together with a quantum keeper, which may cause additional wait calls in the implementation (for example, when reaching a quantum boundary); this is not allowed in a method. Through the combination of waiting for time and for the event it is possible to miss events. When the thread is waiting for time to advance, it will miss all events that were triggered during that time-span.
- When using `scml_property` objects in models used in the Platform Architect and virtual prototype tools, the value for these parameters is read from a configuration file at the start of the simulation. When loading the simple DMA model into Platform Creator, a static parameter as defined here will be recognized so that when building a system using this DMA model the size of the buffer can be defined from within Platform Creator.

The class definition for the DMA looks as follows:

```
class ExampleDMA : public sc_core::sc_module
{
public:
    sc_core::sc_out<bool>          DMAIRQ;
    scml2::initiator_socket<32> DMADATA;
    tlm::tlm_target_socket<32> DMAREGS;
```

This piece of code contains the pin and port definitions. Additionally, there is the following code:

```
private:  
    tlm_utils::tlm_quantumkeeper mQuantumKeeper;  
  
1     scml_property<unsigned int> mMaxDmaByteSize;  
2     sc_core::sc_time mClockPeriod;  
  
    scml2::tlm2_gp_target_adapter<32> mAdapter;  
  
2     scml2::memory<unsigned int> MemoryMap;  
    scml2::reg<unsigned int> DmaEnable; // kicks of DMA transfer  
    scml2::reg<unsigned int> DmaStatus; // 1 in case of active IRQ  
    scml2::reg<unsigned int> DmaIrqClr; // clears the IRQ  
    scml2::reg<unsigned int> DmaSize; // in bytes  
    scml2::reg<unsigned int> DmaSrcAddr; // byte addressing  
    scml2::reg<unsigned int> DmaDstAddr; // byte addressing  
  
    unsigned char* mData;  
3     bool mBusy;  
4     sc_core::sc_event mStartDmaEvent;  
};
```

This section contains the register definitions, quantum keeper, adapter for the target socket, and the local variables that will be used in the behavior.

The labeled code fragments are explained as follows:

- 1 The SCML property `mMaxDmaByteSize` is defined as an `unsigned int` and is used to define the size of the `mData` array for the local data.
- 2 `mClockPeriod` is used to keep track of the clock period for this block, so that a correct delay can be calculated for each transaction.
- 3 `mBusy` is a flag that indicates whether an DMA transfer is currently busy.
- 4 The `mStartDmaEvent` event is used to start the DMA transfer from the register callback.

The constructor and destructor for the module look as follows:

```
public:  
    SC_HAS_PROCESS(ExampleDMA);  
  
1 explicit ExampleDMA(sc_core::sc_module_name module_name) :  
  
    sc_core::sc_module(module_name),  
    DMAIRQ("DMAIRQ"),  
    DMADATA("DMADATA"),  
    DMAREGS("DMAREGS"),  
    clk("clk"),  
    mMaxDmaByteSize("MaxDmaByteSize", 0x100),  
    mAdapter("mAdapter", DMAREGS),  
    MemoryMap("MemoryMap", 0x18 >> 2),  
    DmaEnable("DmaEnable", MemoryMap, 0x0 >> 2),  
    DmaStatus("DmaStatus", MemoryMap, 0x4 >> 2),  
    DmaIrqClr("DmaIrqClr", MemoryMap, 0x8 >> 2),  
    DmaSize("DmaSize", MemoryMap, 0xc >> 2),  
    DmaSrcAddr("DmaSrcAddr", MemoryMap, 0x10 >> 2),  
    DmaDstAddr("DmaDstAddr", MemoryMap, 0x14 >> 2)  
{  
    // configure quantum keeper  
    DMADATA.set_quantumkeeper(mQuantumKeeper);  
  
    // bind register file to target adapter  
    mAdapter(MemoryMap);  
  
2    //setup register handling  
    scml2::set_ignore_write_access(DmaStatus);  
    scml2::set_word_write_callback(DmaEnable,  
                                    SCML2_CALLBACK(onDmaEnableWrite));  
    scml2::set_word_write_callback(DmaIrqClr,  
                                    SCML2_CALLBACK(onDmaIrqClrWrite));  
  
    MemoryMap.initialize(0);  
    mBusy = false;  
  
3    //setup interrupt handling  
    SC_THREAD(f_dma_transfer_thread);  
    sensitive << mStartDmaEvent;  
  
4    mData = new unsigned char[mMaxDmaByteSize];  
}  
  
~ExampleDMA() {  
    delete mData;  
}
```

The labeled code fragments are explained as follows:

- 1 In the constructor, the memory map is defined and all ports and sockets get a name. The module has a quantum keeper for the initiator socket and an adapter for the target socket.
- 2 The `DmaSize`, `DmaSrcAddr`, `DmaDstAddr` registers can be modeled using the default SCML storage behavior, so no callbacks are registered. Potentially, it would be possible to restrict accesses to word-only accesses, but this is ignored in this example. The `DmaStatus` register has default read behavior but write accesses should be disabled. The `DmaEnable` and `DmaIrqClr` each need a callback to enable the DMA transfer and to clear the interrupt output. The callbacks are autosyncing; one will trigger the sensitivity of a thread and the other will write to an output signal.
- 3 For the actual data transfers, an `SC_THREAD` is registered with the kernel.
- 4 The destructor is needed to clean up the local storage for the DMA transfers which is created in the constructor based on the `mMaxDmaByteSize` property.

The additional behavior of the DMA that needs to be specified is fully captured in the callbacks for the register accesses and the method that is registered as thread:

```
1  bool onDmaEnableWrite(const unsigned int& data)
   {
      if(!mBusy) {
         DmaEnable = 1;
         mStartDmaEvent.notify();

      } else {
         // already busy: ignore access
      }
      return true;
   }

2  bool onDmaIrqClrWrite(const unsigned int& data)
   {
      DmaStatus = 0;
      DMAIRQ = false;
      return true;
   }
```

The labeled code fragments are explained as follows:

- 1 When the `DmaEnable` register is written, the DMA transfer is started by notifying the `mStartDmaEvent` event. Whatever value is written, the value `1` is stored in the register. Before doing this, the status of the DMA is checked via the `mBusy` flag. If a transfer is busy, the event will not be notified and no new DMA transfer is enabled. When using this DMA, it is the programmer's responsibility to first check the DMA enable value before starting a new DMA transfer.
- 2 When the `DmaIrqClr` register is written, the `DMAIRQ` output is cleared, whatever its value was. Also the `DmaStatus` register is cleared (value zero is written). This means that here it is the programmer's responsibility to check the `DmaStatus` register before clearing the IRQ output of the DMA.

The code for the thread is as follows:

```
3 void f_dma_transfer_thread()
{
    while(1) {
        // wait till next DMA transfer
        sc_core::wait();
        // reset quantumkeeper (calculates next sync point)
4     mQuantumKeeper.reset();

5     mBusy = true;
    assert (mMaxDmaByteSize >= DmaSize);
    // initial set-up delay
    inc_and_sync(sc_core::sc_time(5, sc_core::SC_NS));

    // read data from source address
    {
        bool result = DMADATA.read<unsigned char>(DmaSrcAddr, mData,
                                                     DmaSize);
        assert(result); (void)result;
        inc_and_sync(mClockPeriod * (DmaSize + 3) / 4);
    }

    // write data to destination address
    {
        bool result = DMADATA.write<unsigned char>(DmaDstAddr, mData,
                                                      DmaSize);
        assert(result); (void)result;
        inc_and_sync(mClockPeriod * (DmaSize + 3) / 4);
    }

6    // let the SystemC kernel catch up with my local time
    mQuantumKeeper.sync();

7    DmaEnable = 0;
    mBusy = false;

    // generate irq
    DmaStatus = 1;
    DMAIRQ = true;
}
}
```

The labeled code fragments are explained as follows:

- 3 The behavior of the thread is concurrent to the execution of software and an unknown number of DMA transfers has to be executed; so the thread contains a `while` loop that can run an infinite number of iterations. Each iteration starts by waiting for the static sensitivity of the thread. In this case, this means the wait will finish when the `mStartDmaEvent` event is notified.

- 4 On each invocation, the quantum keeper of this module is reset. This will force to recalculate the next synchronization point. Since it is unknown when the `mStartDmaEvent` event will be triggered, any previous quantum keeper synchronization event may be long passed.
- 5 The actual DMA transfer is implemented by using the read and write methods of the `DMADATA` port. In the read access, the data is stored in the local storage (`mData`), which is used again when writing. After each access, the local time is incremented and there is a check for synchronization through the `inc_and_sync()` method, which is local to this class. At every step, there is an assert to check whether all of the boundary conditions are met (`burstsize` is not bigger than array size, each access is successful). Obviously, these can be replaced by more elaborate error checking and reporting or even an upgraded specification that would allow the software to catch these errors.
- 6 Once the complete DMA transfer is done, there is another synchronization with the rest of the system through the quantum keeper. Since the `inc_and_sync()` call will only synchronize when the quantum boundary is reached, no synchronization may have been done in the last `inc_and_sync`. The additional synchronization is required to make sure the complete delay of the DMA transfer is taken into account and no new DMA transfer is started before the previous was completed. Removing the synchronization would allow the next `mStartDmaEvent` to come before the delay was completed.
- 7 At the end of the DMA transfer, the `DmaEnable` register is reset, the `mBusy` flag is set to `false`, and the interrupt is generated plus the status register is set.

At end of elaboration, the interrupt output is initialized and the period of the clock is stored in `mClockPeriod`. The code looks as follows:

```
void end_of_elaboration()
{
    DMAIRQ.initialize(false);
    mClockPeriod = scml2::get_period(clk);
}
```

The `inc_and_sync()` method will increment the quantum, check if we reached the quantum boundary, and synchronize if necessary. The code looks as follows:

```
void inc_and_sync(const sc_core::sc_time& time)
{
    mQuantumKeeper.inc(time);
    if (mQuantumKeeper.need_sync()) {
        mQuantumKeeper.sync();
    }
}
```

## 8.6.5 Modeling a Cache

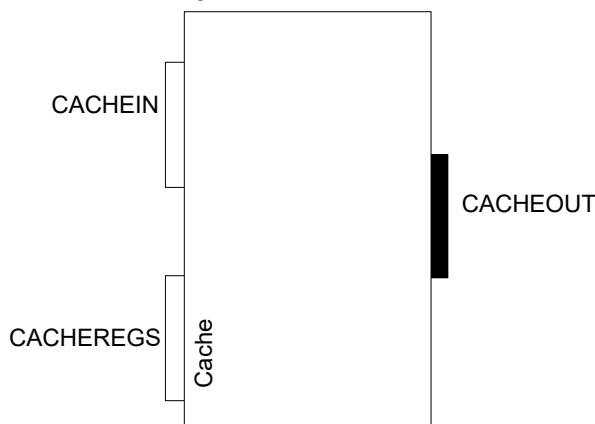
This section describes a cache model. The goal of the example is to introduce the modeling approach for a component that deals with memory accesses. The key SCML2 feature introduced in this example is the router object. This example also introduces the TLM2.0 style transport callbacks for memories.

A *cache* is a component that provides a local copy for part of the data that is stored further in the memory path. As such, a cache is similar to the DMA of “[Modeling a DMA](#)” on page 316, that is, both receive and initiate transactions. The key difference is that a cache will forward a transaction in case no local copy is

available for the data. So the software will need to wait for the forwarded transaction to finish before it can continue. A DMA is programmed through its memory-mapped registers to initiate transactions that copy data from one memory to another independently from the software that is running on the core.

The specification looks as shown in the following figure.

**Figure 8-19 Cache Specification**



The cache is specified as follows:

- The cache example has three memory-mapped interfaces: One interface to allow the software to program the cache, a second through which the incoming transactions are routed, and an output interface that is used to fetch data copies to be stored in the cache. The following table outlines the specification for the pins of this example.

**Table 8-20 Interface of the Cache**

Name	Description
CACHEIN	Interface for data transfers.
CACHEREGS	Interface to access the memory-mapped registers of the cache.
CACHEOUT	Output interface of the cache.

- The programming interface for the cache is specified through the CACHEREGS interface which provides access to the configuration registers of the cache. This simple cache has a single internal register, Control at local address 0x0, which has two bitfields which are defined as shown in the following table.

**Table 8-21 Bit Specification of Cache**

Bits	Type	Name
0x0	Read-Write	Enable
0x1	Read-Write	WritePolicy

The Enable bitfield is used to enable or disable the behavior of the cache. In case the cache is disabled, all transactions are forwarded to the CACHEOUT port.

The `WritePolicy` bitfield is defined as read-write. However, this is only true as long as the cache is disabled. Once the cache is enabled, `WritePolicy` cannot be changed.

- The cache has an internal storage where it will store a local copy of the result of every read transaction. Whenever the same address is read again, it will return the local copy of the data. The cache will overwrite already existing local copies when a transaction comes in for an address for which the data is not stored locally and the local storage is fully used (a cache miss). The cache keeps track of the addresses for which a local copy is stored.
- The behavior of the cache in case of write transactions is configurable:
  - When `WritePolicy` is set to `WriteThrough`, all write transactions are forwarded to the `CACHEOUT` port.
  - In case the policy is to set `WriteBackAll`, writes are stored locally in the cache and whenever there is a cache miss, the data is written back to its original location.

In this simple example, the focus is on the modeling approach and the SCML2 features, so the storage is done per byte and the cache does not support burst accesses.

The key SCML2 feature that enables a simple cache model is the router object. The main goal for the router is to manage accesses for components that reroute transactions. In the case of a cache, transactions either will end up in the internal storage or will be forwarded to the `CACHEOUT` port. All accesses should be routed to one of these two. The management of this decision is implemented in the router object. A router object has an API to define the mapping between incoming addresses and memory objects or output ports. This mapping can be changed at any time. The mapping is used to route incoming transactions. Callbacks can be associated to the router to implement transaction handling for any address range that is not mapped. The router object does not implement any storage behavior, so failing to forward transactions will lead to transactions that are not handled at all.

The model of the cache contains a router object that implements the transaction routing for the cache and a memory object that implements the default storage behavior for the cache storage. Next to that, there should be a register to model the control register. The behavior of the cache in case of a cache miss is implemented in the callback of the router.

Although the cache has an initiator port, it will not contain a quantum keeper since its behavior is not concurrent with the execution of software.

The class definition for the cache looks as follows:

```
class ExampleCache : public sc_core::sc_module
{
public:
    tlm::tlm_target_socket<32> CACHEIN;
    tlm::tlm_target_socket<32> CACHEREGS;
    scml2::initiator_socket<32> CACHEOUT;

    scml_property<unsigned int> mCacheLines;
    scml_property<unsigned int> mMemorySize;
```

This part of the definition contains the socket definition and the parameters for the cache.

- The `mCacheLines` parameter defines the size of the local storage of the cache.
- The `mMemorySize` parameter defines the range of addresses the cache will cover.

The definition of the other internals of the model looks as follows:

```
private:  
1 scml2::tlm2_gp_target_adapter<32> mRouterAdapter;  
scml2::router<unsigned int> mRouter;  
  
2 scml2::tlm2_gp_target_adapter<32> mMemoryMapAdapter;  
scml2::memory<unsigned int> MemoryMap;  
scml2::reg<unsigned int> Control;  
scml2::bitfield<unsigned int> Enable;  
scml2::bitfield<unsigned int> WritePolicy;  
  
scml2::memory<unsigned int> cacheMemory;  
  
enum WritePolicy { WriteThrough = 0x0, WriteBackAll = 0x1 };  
  
// Maps cache address to real address  
3 typedef std::map<unsigned long long, unsigned long long> CacheTags;  
CacheTags mCacheTags;  
};
```

The labeled code fragments are explained as follows:

- 1 The router object is defined together with the target adapter to connect it to its socket.
- 2 The adapter and storage for the memory-mapped register is defined, as well as the local storage for the cache. Obviously, no adapter is defined for the local storage since accesses will come through the router object.
- 3 `std::map` is defined, which will be used to keep track of the address regions that are cached in the local storage.

The constructor of the cache looks is used to initialize all objects of the model and to set up the memory map for the cache model. What is different in this model is the setup of the callbacks for the router object.

```
public:  
    SC_HAS_PROCESS(ExampleCache);  
  
    ExampleCache(sc_core::sc_module_name name) :  
        sc_core::sc_module(name),  
        CACHEIN("CACHEIN"),  
        CACHEREGS("CACHEREGS"),  
        CACHEOUT("CACHEOUT"),  
        mCacheLines(32),  
        mMemorySize(0x10000),  
        mRouterAdapter("mRouterAdapter", CACHEIN),  
        mRouter("mRouter", mMemorySize >> 2),  
        mMemoryMapAdapter("mMemoryMapAdapter", CACHEREGS),  
        MemoryMap("MemoryMap", 0x4 >> 2),  
        Control("Control", MemoryMap, 0x0 >> 2),  
        Enable("Enable", Control, 0, 1),  
        WritePolicy("WritePolicy", Control, 1, 1),  
        cacheMemory("cacheMemory", mCacheLines)  
    {  
        mRouterAdapter(mRouter);  
        mMemoryMapAdapter(MemoryMap);  
  
        scml2::set_callback(mRouter,  
                            SCML2_CALLBACK(handleUncachedMemoryAccess),  
                            scml2::SELF_SYNCING);  
        scml2::set_debug_callback(mRouter,  
                                SCML2_CALLBACK(handleDebugAccess));  
  
        // Must be AUTO_SYNCING since doWriteBack (write) may call wait  
        scml2::set_write_callback(Enable, SCML2_CALLBACK(onEnableWrite));  
  
        WritePolicy = WriteThrough;  
        Enable = false;  
  
        doDisable();  
    }  
}
```

The labeled code fragments are explained as follows:

- 1 The callback type used for the router is different from the ones in the other examples. A router object only supports TLM2.0-style transport callbacks. Since a router needs to make a decision on the further trajectory of an incoming transport call, the callback should have access to the full TLM2.0 transaction payload. The callback is registered as SELF\_SYNCING so that the syncing strategy can be further decided by the components to which the transaction is forwarded. In case of an internal memory, no syncing may be required, while an access over the DATAOUT socket may lead to further synchronization.
- 2 A debug callback is also registered for the router object. This callback will be used to handle the debug accesses that come through the TLM2.0 interface.

In the constructor, the cache is initialized to be disabled and with WritePolicy set to WriteThrough. The behavior of the cache is controlled by the Enable bitfield. The callback registered for the Enable bitfield is used to check whether there is a change in mode for the cache. When the cache moved from enabled to disabled state or vice versa the router mapping needs to be updated. The code for managing the enable/disable state switch looks as follows:

```

bool onEnableWrite(const unsigned int& data)
{
    if (data) {

        if (!Enable) {
            doEnable();
        }
    } else {
        if (Enable) {
            if (WritePolicy == WriteBackAll) {
                doWriteBackAll();
            }
            mCacheTags.clear();
            doDisable();
        }
    }

    Enable = data;
    return true;
}

void doEnable()
{
    mRouter.unmap_all();
}

void doDisable()
{
    mRouter.unmap_all();
    bool result = mRouter.map(0x0, mRouter.get_size() * 4, CACHEOUT, 0x0);
    assert(result);
}

```

The labeled code fragments are explained as follows:

- 1** When disabling after the cache has been enabled, the content of the cache should be written back to the destination memory in case the write policy was set to WriteBackAll. Also the mapping of internal to external addresses should be cleared. Further, disable functionality is split into a separate function since that code is reused for the initialization of the cache.
- 2** When enabling the cache, all current mappings of the router should be removed.

- 3** When disabling the cache, all current mappings of the router should be removed; typically these are the mappings that occurred during the operation of the cache. A new mapping is defined which will forward all requests to the output port. This will ensure that the system continues to work correctly. This mapping needs to be removed when enabling the cache.

Next is the code to maintain and update the cache contents when it is enabled. This is done in the callback for the router object. This callback will only be called for memory accesses that are currently not mapped (cache misses). All mapped accesses will be automatically forwarded. In case the cache is disabled, the whole memory region of the cache is mapped to the output port; so the callback on the router will never be called. In case the cache is enabled, all mappings are removed, so the first accesses will be directed to the callback. In the callback, the decision is taken whether a copy for the requested access will be stored in the

local memory. Once the cache is full, the callback will also need to handle the replacement strategy for the cache. In this simple example, only the current access is stored. The code for the callback looks as follows:

```
void handleUncachedMemoryAccess(tlm::tlm_generic_payload& trans,
                                sc_core::sc_time& time)
{
    1   const unsigned long long unalignedAddress = trans.get_address();
    2   const unsigned long long alignedAddress = unalignedAddress & ~0x3ull;
    3   const unsigned long long unalignedCacheAddress = unalignedAddress %
                                                (mCacheLines *4);
    4   const unsigned long long alignedCacheAddress = alignedAddress %
                                                (mCacheLines *4);

    if (!checkTransaction(trans)) {
        return;
    }

    2   if (trans.is_read()) {
        unsigned int data;
        CACHEOUT.read<unsigned int>(alignedAddress, data, time);

        // Record the entry in the cache
        std::pair<CacheTags::iterator, bool> tag = mCacheTags.insert(
            std::make_pair(alignedCacheAddress, alignedAddress));
        if (!tag.second) {
            // Evict the existing entry
            5   if (WritePolicy == WriteBackAll) {
                doWriteBack(tag.first->first, tag.first->second, time);
            }
        }
        4   bool result = mRouter.unmap(tag.first->second);
        assert(result);
        tag.first->second = alignedAddress;
    }

    6   cacheMemory.put(alignedCacheAddress,
                      reinterpret_cast<const unsigned char*>(&data), 4);
    if (WritePolicy == WriteThrough) {
        5   bool result = mRouter.map_read(alignedAddress, 4, cacheMemory,
                                         alignedCacheAddress);
        assert(result);

        6   else {
            bool result = mRouter.map(alignedAddress, 4, cacheMemory,
                                      alignedCacheAddress);
            assert(result);
        }
    }

    7   cacheMemory.get(unalignedCacheAddress, trans.get_data_ptr(),
                      trans.get_data_length());
    trans.set_response_status(tlm::TLM_OK_RESPONSE);

    8   } else if (trans.is_write()) {
        if (WritePolicy == WriteThrough) {
            CacheTags::const_iterator i =
                mCacheTags.find(alignedCacheAddress);
            if (i != mCacheTags.end() && i->second == alignedAddress) {
                cacheMemory.put(unalignedCacheAddress, trans.get_data_ptr(),
                               trans.get_data_length());
            }
        }
        CACHEOUT.b_transport(trans, time);
    }
}
```

The labeled code fragments are explained as follows:

- 1 First task in the callback is to calculate the aligned address for the transaction and determine the local address that will be used in the cache. The local address is determined by the lower bits in the address (here calculated as the remainder of CacheLines expressed in byte addresses).

- 2 In case of a read transaction, we first fetch the data via the output port.
- 3 The pair of local address and external address is stored in CacheTags. This is done via the `insert()` method of `std::map`. For code simplicity, the version of the `insert()` method is used that returns a pair containing the iterator in the map where the insertion was done and a `bool` value that indicates whether an insert happened. If the `bool` is `false`, there was already a value stored with the same key. In this case, the key is the local address in the cache.
- 4 If there is already a value stored in the cache for this local address, the current stored address should be evicted from the cache and replaced with the new request. This is done by unmapping the old address in the router. The pairing of local address with external address is updated in the map.
- 5 When the write policy is set to write back, the currently stored value in the cache should be written back to memory.
- 6 When all cache management is finished, the value can be stored in the cache and the router mapping is updated. In case the write policy is set to write through, the router should only forward reads to the local memory. This means that writes to the external address will still be forwarded to the output port. In case the write policy is set to write back, all accesses to the external address should be forwarded to the local storage.
- 7 Finally, the read transaction is finished by retrieving the data from the local memory and setting the response for the transaction.
- 8 In case of a write transaction, the callback will only be triggered for an address that is not cached or in case the write policy is set to write through. For noncached write accesses, the transaction should be forwarded to the output port. A copy of the data is stored in the local memory in case the write policy is set to write through and the external address is cached.

The callbacks on the `Enable` bitfield as well as the callback on the router object use the methods that implement the write back.

- The `doWriteBack` function implements the write back for a single value by getting the data from the local memory and writing it back to the real address.
- The `doWriteBackAll` function goes through the complete map of the cache and writes back every value in the cache.

The code for these functions looks as follows:

```
void doWriteBack(unsigned long long cacheAddress,
    unsigned long long realAddress, sc_core::sc_time& t)
{
    unsigned int cachedData;
    cacheMemory.get(cacheAddress,
        reinterpret_cast<unsigned char*>(&cachedData), 4);
    CACHEOUT.write<unsigned int>(realAddress, cachedData, t);
}

void doWriteBackAll()
{
    CacheTags::const_iterator it = mCacheTags.begin();
    const CacheTags::const_iterator end = mCacheTags.end();
    sc_core::sc_time t = sc_core::SC_ZERO_TIME;
    for (; it != end; ++it) {
        doWriteBack(it->first, it->second, t);
    }
    if (t != sc_core::SC_ZERO_TIME) {
        wait(t);
    }
}
```

Finally, there is the implementation for the debug callback. In this example, the debug accesses are simply forwarded to the output port. The assumption is that for all cached data, DMI is supported and that debug accesses will be using DMI. In that case, only uncached accesses for which the destination memory does not support DMI will be using this callback and need to be forwarded through the output port. The code looks as follows:

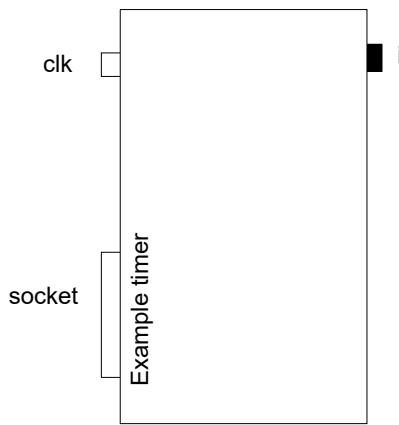
```
unsigned int handleDebugAccess(tlm::tlm_generic_payload& trans)
{
    return CACHEOUT.transport_dbg(trans);
}
```

Finally, one more call checks whether the cache supports a certain type of transaction and returns an error response in case the transaction is not supported.

### 8.6.6 Example Timer Specification

The Example timer specification looks as shown in the following figure.

**Figure 8-20 Example Timer Specification**



The timer is specified as follows:

- The timer will count down at the rate of a clock input, starting from a value stored in the reload register. When the count reaches zero, and the interrupt is enabled, an interrupt is sent. If the timer is configured as a periodic timer, it will start counting down again from the reload value after it reached zero. The interrupt can be cleared by writing zero to the interrupt status register.
- The following table outlines the interface of this component:

**Table 8-22 Interface of the Example Timer**

Name	Description
clk	Clock input
irq	Interrupt output. It is active high.
socket	Memory-mapped interface

- The following table outlines the registers and bitfields by which the programming interface of the timer is defined.

**Table 8-23 Registers and Bitfields of the Example Timer**

Registers	Description
ControlStatus	Contains the bitfields <code>TimerEnable</code> , <code>OneShot</code> and <code>InterruptEnable</code> to control the functionality of the timer, and the bitfield <code>InterruptStatus</code> to read out the status of the timer.
ReloadValue	The value written to this register will be the initial value for the timer when it is reloaded. The default value is 1000.
Counter	Whenever this register is read, the current counter value is returned.
Bitfields	Description
TimerEnable	Should be set high to enable the timer. When re-enabled, the counter value is reloaded. By default, the interrupt is disabled.

Registers	Description
OneShot	Should be set <code>high</code> to configure the timer as an one-shot timer, if set to <code>low</code> , the timer is configured as a periodic timer. By default, the timer is configured to be a periodic timer.
InterruptEnable	Should be set <code>high</code> to enable the interrupt. By default, the interrupt is disabled.
InterruptStatus	Is the value output in <code>irq</code> .

- The timer has a memory range defined of size `0xC`. The following table outlines its register definition

**Table 8-24 Register Definitions of the Example Timer**

Address	Type	Width	Name
<code>0x0</code>	RW	6	ControlStatus
<code>0x4</code>	RW	32	ReloadValue
<code>0x8</code>	R	32	Counter

- Bit specification

The `ControlStatus` register contains the bitfields, as shown in the following table.

**Table 8-25 ControlStatus Bit Specification**

Bits	Type	Name
[31:6]	Reserved	
[5]	RW	InterruptStatus
[4]	RW	InterruptEnable
[3:2]	Reserved	
[1]	RW	OneShot
[0]	RW	TimerEnable





# Index

## A

adaptors, definition [244](#)

Address Type. *See* AT

API, definition [12](#)

Application Programmer's Interface. *See* API

Approximately Timed. *See* AT

Architect's View. *See* AV

architecture analysis use case [240](#)

ASI TLM WG, definition [12](#)

AT

    description [240](#)

AT, definition [12](#)

AUTO\_SYNCING callback [267](#)

AV, definition [12](#)

## B

backdoor access, definition [270](#)

bind() function [162](#), [167](#)

bitfield

    get\_description() method [55](#)

    get\_name() method [55](#)

    get\_offset() method [55](#)

    get\_register() method [55](#)

    get\_size() method [55](#)

    has\_default\_debug\_read\_behavior() method [55](#)

    has\_default\_debug\_write\_behavior() method [55](#)

    has\_default\_read\_behavior() method [55](#)

    has\_default\_read\_restriction() method [55](#)

    has\_default\_write\_behavior() method [55](#)

    has\_default\_write\_restriction() method [55](#)

    has\_never\_syncing\_read\_behavior() method [55](#)

    has\_never\_syncing\_write\_behavior() method [55](#)

    is\_dmi\_read\_allowed() method [55](#)

    is\_dmi\_write\_allowed() method [55](#)

    set\_debug\_callback() method [58](#)

    set\_debug\_read\_callback() method [58](#)

    set\_debug\_write\_callback() method [58](#)

set\_description method [55](#)

set\_post\_write\_callback method [57](#)

set\_read\_callback() method [57](#)

set\_read\_no\_store\_callback() method [57](#)

set\_write\_callback() method [57](#)

bitfield\_debug\_read\_callback\_base [68](#)

bitfield\_debug\_write\_callback\_base [68](#)

bitfield\_read\_callback\_base [67](#)

bitfield\_write\_callback\_base [67](#)

## C

cache, modeling [324](#)

callback base classes [66](#)

callbacks

    AUTO\_SYNCING [267](#)

    NEVER\_SYNCING [268](#)

    SELF\_SYNCING [268](#)

    types [265](#)

clock objects [157](#)

    example [183](#)

clocks, definition [244](#)

coding style guidelines [256](#)

coding styles

    AT [240](#)

    LT [240](#)

commands, definition [245](#)

communication components, in virtual prototype [235](#)

Compute Express Link. *See* CXL

CXL, definition [12](#)

## D

Data processing, in virtual prototype [236](#)

Data Type. *See* DT

data\_type() function [167](#)

debugging quantum effects [265](#)

Direct Memory Access. *See* DMA

disable() function [160](#)

disabled() function [160](#)

modeling 317

DMA, definition 12

DMI

definition 244, 270

infrastructure 259

dmi\_handler 191

disable\_dmi() method 192

enable\_dmi() method 192

invalidate\_direct\_mem\_ptr() method 192

is\_dmi\_enabled() method 191

read\_debug() method 192

read() method 192

set\_interface() method 191

transport\_debug() method 192

transport() method 192

write() method 192

DT, definition 12

## E

embedded software development use case 238

enable() function 160

event() function 161

examples

modeling a cache 324

modeling a DMA 316

modeling a memory 282

modeling a watchdog peripheral 301

modeling an interrupt controller 289

External interfaces, in virtual prototype 235

## F

FIFO, definition 12

First In First Out. *See* FIFO

functional specification use case 239

## G

get\_count() function 167

get\_divider() function 162

get\_duty\_cycle() function 159

get\_period\_multiplier() function 160

get\_period() function 159

get\_posedge\_first() function 159

get\_start\_time() function 159

getBoolProperty() function 203, 204, 206

getDoubleProperty() function 203, 204, 206

getIntProperty() function 203, 204, 206

getName() function 202

getStringProperty() function 203, 204, 206

getting started 281

getType() function 202

getUIntProperty() function 204, 206

## H

hardware verification use case 240

## I

initiator\_socket 192

b\_transport() method 195

disable\_dmi() method 193

enable\_dmi() method 193

get\_local\_time() method 193

inc() method 193

is\_dmi\_enabled() method 193

need\_sync() method 193

read\_debug() method 194

read() method 193

set\_endianness() method 193

set\_quantumkeeper() method 193

set() method 193

sync() method 193

transport\_dbg() method 195

write\_debug() method 194

write() method 193

Intellectual Property. *See* IP

interconnect, in virtual prototype 234

interfaces

blocking 243

definition 243

nonblocking 243

interrupt controller, modeling 289

IP, definition 12

## L

load() function 206

logging, definition 245

Loosely Timed. *See* LT

LT

description 240

LT, definition 12

## M

mappable\_if 65  
  b\_transport() method 66  
  get\_direct\_mem\_ptr() method 66  
  get\_mapped\_name() method 65  
  register\_bw\_direct\_mem\_if() method 66  
  transport\_dbg() method 66  
  unregister\_bw\_direct\_mem\_if() method 66

memory  
  definition 244  
  in virtual prototype 235

memory\_debug\_callback\_base 66

memory\_index\_reference 64  
  put\_debug() method 65  
  put() method 65

memory, modeling 282

memory-mapped components, in virtual prototype 235

methods, definition 244

modeling a cache, example 324

modeling a DMA, example 316

modeling a memory, example 282

modeling a watchdog peripheral, example 301

modeling an interrupt controller, example 289

modeling guidelines  
  communication 253  
  model only what you need 250  
  modeling behavior 252  
  modeling for speed 251  
  SCML-related coding guidelines 256  
  SystemC-related coding guidelines 258  
  timing and synchronization 254

modeling methodology guidelines 250

modeling objects  
  combining 245  
  concepts 243

modeling utilities 185, 191

modules, definition 243

N

negedge\_event() function 161  
negedge() function 161  
NEVER\_SYNCING callback 268

## O

operator() function 162, 167

optimizing simulation speed, strategies 268

original clock period, definition 162

overview 19, 22

P

payload, definition 243

Payload-event-queue, definition 245

PCI Express, definition 12

Peripheral Component Interconnect Express. *See PCI Express*

phases, definition 243

pins, definition 243

Plain Old Data Type. *See PODT*

Platform synchronization, in virtual prototype 235

PODT, definition 12

posedge\_event() function 161

posedge() function 161

processor model, in virtual prototype 234

Programmer's View. *See PV*

properties, definition 245

Protocol Checker 150

Pulse Width Modulation. *See PWM*

PV  
  definition 12

PWM  
  definition 12

Q

quantum  
  debugging quantum effects 265  
  definition 261  
  dynamic adjustment 261

R

read() function 167

register\_observer() function 160

router  
  b\_transport method 62  
  disable\_dmi() method 60  
  enable\_dmi() method 60  
  get\_description() method 60  
  get\_direct\_mem\_ptr method 62  
  get\_name() method 60



get\_offset() method 60  
get\_size() method 60  
get\_width() method 60  
is\_dmi\_enabled() method 60  
map\_read() method 61  
map\_write() method 61  
map() method 61  
set\_callback() function 64  
set\_debug\_callback function 64  
set\_description method 60  
transport\_dbg method 62  
unmap\_all() method 62  
unmap\_read() method 61  
unmap\_write() method 62  
unmap() method 61  
router peripherals, modeling 265  
router\_callback\_base 67  
router\_debug\_callback\_base 67  
router, definition 245  
running() function 160

## S

SCML  
modeling guidelines 250  
purpose 236, 237  
SCML callback levels 265  
SCML modeling, introducing 236, 237  
scml\_clock\_counter 166  
scml\_command\_processor 197  
scml\_loader 199  
scml\_property 200  
scml\_property\_registry 202  
scml\_property\_server\_if 204  
SCML1, definition 12  
SCML2, definition 12  
SELF\_SYNCING callback 268  
set\_duty\_cycle() function 159  
set\_period\_multiplier() function 160  
set\_period() function 159  
set\_posedge\_first() function 159  
set\_start\_time() function 159  
setCustomPropertyServer() function 203  
simulation modes 269  
simulation speed, strategies for optimizing 268

Socket Transaction Language. *See* STL  
sockets, definition 243  
status 207  
status, definition 245  
STL, definition 12  
Subsystems, in virtual prototype 236  
synchronization

and temporal decoupling 260  
synchronizing behavior 262

SystemC Modeling Library 1. *See* SCML1  
SystemC Modeling Library 2. *See* SCML2

## T

target peripherals, modeling 265  
temporal decoupling  
and synchronization 260  
definition 244, 260, 270  
examples 262  
multiprocessor example 262

threads, definition 244

time, definition 244

TLM  
description 237

TLM, definition 12  
tlm2\_gp\_initiator\_adapter 73  
tlm2\_gp\_target\_adapter 71

operator() method 72

transaction payload 254

Transaction-Level Modeling. *See* TLM  
transactor, definition 244

## U

unregister\_observer() function 160  
use cases 238

## V

value\_changed\_event() function 161  
Virtual Processing Unit. *See* VPU  
virtual prototypes

components 234  
requirements 233

VPU, definition 12

## W

watchdog peripheral, modeling 301  
write() function 167