Synopsys Virtualizer Product Family:

SystemC Modeling Library Manual

Version F-2011.06-SP2
December 2011

Synopsys, Inc.
700 E. Middlefield Road
Mountain View, CA 94043

www.synopsys.com

Contents

Preface

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

- About This Manual
- Documentation Conventions
- Terminology

About This Manual

This manual describes SystemC Modeling Library 1 (SCML1) modeling objects.

Note This release supports SCML1 as well as SCML2. The latter is described in the *SystemC Modeling Library 2 Manual*.

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.
- Target-Side Objects describes target-side objects.
- Clock Objects describes clock objects.
- Initiator-Side Objects describes initiator-side objects.
- TLM2 Adapters describes TLM2 adaptors, which convert TLM2 transactions into PV requests and vice versa.
- Modeling Utilities describes modeling utilities.
- Using the PV Abstraction Level describes the Programmer's View (PV) abstraction level and the PV API functions.

Documentation Conventions

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

Table 1-1 Documentation Conventions

<table>
<tr><th>Convention</th><th>Description and Examples</th></tr>
<tr><td>italic</td><td>Is used in running text for:
GUI elements. For example:
The Enumeration field contains a space-separated list of values.
New terms. For example:
A protocol library is a collection of protocol definitions.
Web sites. For example:
For more information, see www.eclipse.org/rcp.
E-mail addresses. For example:
Please contact customer support via e-mail at vp_support@Synopsys.com.
Manual names. For example:
The preface of the Analysis Manual describes:</td></tr>
<tr><td><code>courier</code></td><td>Is used for:
Code text. For example:
<code>list_library_configurations</code>
<code>myConfig</code>

In this example, <code>myConfig</code> is used.
System messages. For example:
<code>JVM not found.</code>
Text you must type literally. For example:
At the prompt, type <code>go</code>.
Names (of environment variables, commands, utilities, prompts, paths, macros, and so on). For example:
The <code>build-options</code> command sets build parameters.</td></tr>
<tr><td><code>courier italic</code></td><td>Indicates variables. For example:
<code>scope</code> specifies a module, a channel, or a refined port.</td></tr>
<tr><td>bold</td><td>Serves to draw your attention to the text in question. For example:
<code>coreId = cwrSAGetCoreId("ARM7");</code></td></tr>
</table>

Table 1-1 Documentation Conventions

<table>
<tr><th>Convention</th><th>Description and Examples</th></tr>
<tr><td>[ ]</td><td>Square brackets enclose optional items. For example:

clean [-pch]

If you must type a square bracket as part of the syntax, it is enclosed in single quotes. For example:

'['--use-vector']'</td></tr>
<tr><td>{ }</td><td>Braces enclose a list from which you must choose one or more items. For example:

add {signalPattern | portPattern} ID

If you must type a brace as part of the syntax, it is enclosed in single quotes. For example:

DECLARE '{'
Item1
Item1
'}'</td></tr>
<tr><td>|</td><td>A vertical bar separates items in a list of choices. For example:

autoflush {on | off}</td></tr>
<tr><td>></td><td>A right angle bracket separates menu commands. For example:

The Library > Update System Library menu command is available.</td></tr>
<tr><td>...</td><td>A horizontal ellipsis in syntax indicates that the preceding expression may have zero, one, or more occurrences. For example:

build-options -option optionArgs ...

A horizontal ellipsis in examples and system messages indicates material that has been omitted. For example:

::scsh> dtrace add top1.signal_* $t1
::scsh> dtrace add top1.clk_* $t1
...
::scsh> dtrace flush *</td></tr>
</table>

Terminology

AT	In the context of PV, AT stands for Address Type. In the context of TLM2, AT stands for Approximately Timed.
API	Application Programmer's Interface
AV	Architect's View
DMA	Direct Memory Access

DT	Data Type
IP	Intellectual Property
LT	Loosely Timed
OSCI TLM WG	Open SystemC Initiative Transaction-Level Modeling Work Group
PODT	Plain Old Data Type
PV	Programmer's View
SOC	Stands for System-On-a-Chip.
SCML1	SystemC Modeling Library 1. SCML1 is described in this manual.
SCML2	SystemC Modeling Library 2. SCML2 is described in the *SystemC Modeling Library 2 Manual*.
STL	Socket Transaction Language
TLM	Transaction-Level Modeling
VPU	Virtual Processing Unit

Chapter 1
Introduction

This chapter describes:

- Introduction
- Overview of Modeling Objects
- Including Header Files

1.1 Introduction

In *TLM Peripheral Modeling for Platform-driven ESL Design*, a whitepaper included with this kit, CoWare presents a reuse-driven methodology for the transaction-level modeling of SOC peripherals. Under the assumption that ISSes and bus models at different levels of abstractions are available from IP and ESL tool vendors such as CoWare, peripheral modeling is an essential ingredient to ease the adoption of ESL design.

The investment in peripheral modeling can be justified much more easily when multiple platform models for different purposes can be built from the same model base. In other words, the individual ESL design tasks like architecture exploration, software development, and verification should be combined into a seamless ESL design flow.

Furthermore, the SystemC Modeling Library 1.0 described in this reference manual is available in source code for use in other simulation environments. It provides the same open API interfaces as the object code version shipped with CoWare tools, preserving source-level compatibility for user-defined TLM peripheral models. CoWare is committed to maintaining this compatibility to help protect your modeling investment in SCML.

1.2 Overview of Modeling Objects

The following categories of modeling objects can be distinguished:

- *Target-side objects* are used to model storage objects and memory-mapped registers.

  Target-side objects are described in detail in "Target-Side Objects" on page 13.
- *Clock objects* are used to model clock hierarchies.

  Clock objects are described in detail in "Clock Objects" on page 39.
- *Initiator-side objects* are used to model the communication of initiator peripherals in a bus-protocol-agnostic manner to support reuse.

  Initiator-side objects are described in detail in "Initiator-Side Objects" on page 47.
- *TLM2 adapters* are used to convert TLM2 transactions into PV requests and vice versa.

  TLM2 adapters are described in detail in "TLM2 Adapters" on page 59.

1.3 Including Header Files

The memory modeling objects are available after the `scml_memory.h` file has been included.

```
#include "scml_memory.h"
```

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

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

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

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

Chapter 2
Target-Side Objects

This chapter describes the target-side objects.

- Overview
- scml_bitfield
- scml_memory
- scml_memory_user
- scml_memsize
- scml_pv_decoder
- scml_pv_decoder_port
- scml_router
- Code Example

2.1 Overview

The following table summarizes the target-side objects.

Table 2-1 Target-Side Objects

Modeling Object	Summary
scml_bitfield	Bitfields can be attached to memory objects to alias a few bits in the original word.
scml_memory	Models memories and register files.
scml_memory_user	Is a helper class to provide access to information that would otherwise not be available in call-back functions.
scml_memsize	Is a proxy class to identify a constructor argument in scml_memory as a size parameter rather than an initial value.
scml_pv_decoder	Is an address decoder for PV requests.
scml_pv_decoder_port	Is the port that provides the functionality offered by the scml_pv_decoder module.
scml_router	Maps a memory region to a region in an scml_memory, scml_post_port, or scml_tlm2_initiator_adaptor.

2.2 scml_bitfield

Bitfields can be attached to memory objects to alias a few bits in the original word. Additional behavior, besides storing on write and retrieving on read, can be attached by means of call-back functions.

The signature of the call-back function is:

```
data_type readLikeCB();
void writeLikeCB(data_type);
```

Two callbacks can be attached to a bitfield: one for read and one for write. Multiple `scml_bitfield` objects can be attached to an `scml_memory` object.

2.2.1 Types

The following type definitions are available:

```
typedef unsigned int size_type
typedef unsigned int value_type;
```

2.2.2 Constructors

The following constructor is available:

```
scml_bitfield(const char * name,
              scml_bitfield_container_if & mem,
              size_type offset,
              size_type size);
```

where:

name	Specifies a name for the bitfield.
mem	Specifies the memory object being aliased.
offset	Specifies the first bit to be aliased in the referenced memory object.
size	Specifies the number of bits to be aliased.

The bitfield container interface (`scml_bitfield_container_if`) functionality is implemented by `scml_memory` objects, so bitfields can be attached to memory objects.

2.2.3 Operators

The following assignment operator is available:

```
this_reference_type operator=(value_type);
```

Assignments to `scml_memory` objects behave as the underlying value type.

The following arithmetic assignment operators are available. Assignments to `scml_bitfield` objects behave as the value type, restricted to the defined bit width.

```
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);
```

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

```
this_reference_type operator -- ();
value_type operator -- (int);
this_reference_type operator ++ ();
value_type operator ++ (int);
```

A memory object can be converted to the value type:

```
operator value_type() const;
```

2.2.4 Access Functions

The following functions are available to access the value in the bitfield:

```
value_type get() const;
void put(value_type);

value_type read() const;
void write(value_type);
```

If a read callback is attached to the bitfield, `read()` triggers the call-back function and returns the resulting value. If no read callback is attached to the bitfield, `read()` calls `get()` and returns the resulting value.

If a write callback is attached to the bitfield, `write()` triggers the call-back function. If no write callback is attached to the bitfield, `write()` calls `put()`.

> **Note** The difference between `read()` and `write()` on the one hand and `get()` and `put()` on the other hand, is that the former trigger the registered call-back functions, while the latter do not.

2.2.5 Behavior

The `scml_bitfield` object has value type variable behavior, so that its value can be accessed in the call-back functions. The following operators are available to achieve this:

```
value_type operator=(value_type var);
operator value_type() const ;
```

Call-back functions can be attached to bitfield objects to implement behavior. The following macros are available to achieve this:

```
BITFIELD_REGISTER_READ(mem, func)
BITFIELD_REGISTER_READ_USER(mem, func, ID)
BITFIELD_REGISTER_NB_READ(mem, func)
BITFIELD_REGISTER_NB_READ_USER(mem, func, ID)
BITFIELD_REGISTER_WRITE(mem, func)
BITFIELD_REGISTER_WRITE_USER(mem, func, ID)
BITFIELD_REGISTER_NB_WRITE(mem, func)
BITFIELD_REGISTER_NB_WRITE_USER(mem, func, ID)
```

where:

mem	Is an `scml_bitfield` object.

func	Is a pointer to a member function, having the following signature in case of a read or a write callback:
	`value_type readCb();`
	`void writeCb(value_type);`

ID	Specifies user-provided data. A type definition is provided in the `scml_memory_user` class for the user-provided ID. `typedef unsigned int user_id_data_type;` This user-provided ID data is set in the instance of the `scml_memory_user` class when the callback function is triggered for the referenced memory object. The arguments to the callbacks are taken from the PV request structure.

By default, the implementations of these callbacks are allowed to call the System C `wait()` function. In case the implementations do not call the System C `wait()` function and do not write to signals or notify events, you are advised to register them as nonblocking callbacks. This is done by using the `*_NB_*` variants of the macros.

2.2.5.1 Maintaining Control over Allowed Operations

Bitfield objects check whether accesses made over the bus are allowed operations.

Allowed operations are read and write. An illegal read operation on a bitfield returns zero for this bitfield. An illegal write operation on a bitfield is ignored.

Illegal operations do not generate an error response.

```
unsigned int get_allowed_operations() const;
void set_allowed_operations(unsigned int);
```

Gets and sets allowed operations controls.

```
void set_read_only();
void set_write_only();
```

Convenience functions to declare a bitfield read only or write only.

Note Debug transactions, that is, transactions having `pvDebug` as mode in the PV request structure, are allowed. Hence debug read and debug write operations will be performed.

```
bool is_reading_allowed() const;
bool is_writing_allowed() const;
```

Convenience boolean test functions.

```
static const unsigned int scml_memory_base::OPERATIONS_ALLOW_READ = 1;
static const unsigned int scml_memory_base::OPERATIONS_ALLOW_WRITE = 2;
```

Symbolic constants.

2.3 scml_memory

Objects of type `scml_memory` can be used to model memories and register files. Additional behavior, besides storing on write and retrieving on read, can be attached by means of call-back functions.

2.3.1 Types

The `scml_memory` class is templated with the underlying value type.

The following type definitions are available to support generic programming:

```
template<typename DT> class scml_memory;
typedef DT value_type;
typedef value_type* pointer;
typedef const value_type* const_pointer;
typedef value_type* iterator;
typedef const value_type* const_iterator;
typedef value_type& reference;
typedef const value_type& const_reference;
typedef unsigned long long size_type;
typedef ptrdiff_t difference_type;
typedef scml_memory<value_type> this_type;
typedef this_type & this_reference_type;
typedef scml_memory_pv32_if::request_type request32_type;
typedef scml_memory_pv32_if::response_type response32_type;
```

2.3.2 Enumeration Types

The following enumeration type is available:

```
enum scml_endian { scml_little_endian, scml_big_endian };
```

The `scml_endian` enumeration type is used to describe the endianness.

2.3.3 Constructors

Two types of `scml_memory` can be distinguished: those that represent storage and those that do not. The latter are known as *aliases*, since they alias - or refer to - storage in another `scml_memory` object.

Objects of type `scml_memory`, representing storage, can be constructed using one of the following constructors:

```
scml_memory();
scml_memory(value_type var);
scml_memory(const ::std::string & name);
scml_memory(const char * name);
scml_memory(scml_memsize memSize);
scml_memory(const ::std::string & name,
            scml_memsize memSize);
scml_memory(const char * name,
            scml_memsize memSize);
```

Aliases need to be constructed with a reference to an already existing memory object:

```
scml_memory(scml_memory & mem,
            size_type offset,
            size_type size);
scml_memory(const char * name,
```

```
            scml_memory & mem,
            size_type offset,
            size_type size);
scml_memory(const ::std::string & name,
            scml_memory & mem,
            size_type offset,
            size_type size);
```

where:

name	Specifies a name for the memory. The name must be the same as the name of the target location that corresponds with this object. For more information, see "Specifying Target Locations" in the *Platform Creator User's Guide*.
var	Specifies an initial value.
memSize	Specifies the memory size in terms of items of `value` type. The default size of a memory is `1`.
mem	Specifies the memory object being aliased.
offset	Specifies the first index to be aliased in the referenced memory object.
size	Specifies the number of indices to be aliased.

To avoid memory objects being constructed unintentionally, all single-argument constructors are explicit. The copy constructor is disabled.

2.3.4 Initialization

The `initialize()` method can be used to put the specified initial value (*a* in the following function prototype) in the whole memory array. In case no argument is given, the value returned by the default constructor for the underlying data type is used.

```
void initialize (value_type a = DT());
```

2.3.5 Operators

The following assignment operators are available:

```
this_reference_type operator=(const scml_memory & mem);
this_reference_type operator=(value_type var);
```

Assignments to `scml_memory` objects behave as the underlying value type.

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

```
this_reference_type operator += (value_type var);
this_reference_type operator -= (value_type var);
this_reference_type operator /= (value_type var);
this_reference_type operator *= (value_type var);
this_reference_type operator %= (value_type var);
this_reference_type operator ^= (value_type var);
this_reference_type operator &= (value_type var);
this_reference_type operator |= (value_type var);
this_reference_type operator <<= (value_type var);
```

```
this_reference_type operator >>= (value_type var);
```

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

```
this_reference_type operator -- ();
value_type operator -- (int);
this_reference_type operator ++ ();
value_type operator ++ (int);
```

A memory object can be converted to the underlying value type:

```
operator DT() const;
```

2.3.6 Access Functions

The following access functions are available to access the memory:

```
scml_memory_index_reference<value_type> operator[] (size_type index);
value_type operator[] (size_type index) const ;
```

Objects of the `scml_memory` can be used as arrays. The operator `[]` is available for indexing. The `lvalue` version returns an index object, which will forward the operations appropriately. The `const` version returns the stored value.

Taking the address of an index object returns a random access iterator that can be used with algorithms like `std::copy`. For more information, see "Algorithms" on page 21.

Note This is C++ behavior, not Verilog behavior. For example, the following code:

```
scml_memory<unsigned int> reg_FIVECSR;
reg_FIVECSR.initialize(0xffffffff);
```

calling:

```
reg_FIVECSR[0]
```

produces:

```
0xffffffff
```

which is the full 32-bit word, not just one selected bit (as would be the case in Verilog).

```
value_type get(size_type index = 0) const ;
value_type get(size_type index,
               unsigned int acSiz,
               unsigned int offSet) const ;
void put(value_type var,
         size_type index = 0);
void put(value_type var,
         size_type index,
         unsigned int acSiz,
         unsigned int offSet);

value_type read(size_type index = 0);
value_type read(size_type index,
```

```
                unsigned int acSiz,
                unsigned int offSet);
void write(value_type var,
           size_type index = 0);
void write(value_type var,
           size_type index,
           unsigned int acSiz,
           unsigned int offSet);
```

In case the access size (in bits) is not equal to the default access size (that is, eight times the size (in bytes) of the underlying `value_type`), the access size can be specified with the *`acSiz`* variable. The access size (*`acSize`*) has to be a multiple of `8`.
`offset` specifies the lowest bit to be accessed. This is to be interpreted as the second argument to an arithmetic shift operation.

Note The difference between `read()` and `write()` on the one hand and `get()` and `put()` on the other hand, is that the former trigger the registered call-back functions, while the latter do not. The type conversion operator and the assignment operator behave as `get()` and `put()` and hence do not trigger the call-back functions either.

```
value_type readDebug(size_type index = 0);
value_type readDebug(size_type index,
                     unsigned int acSiz,
                    unsigned int offSet);
void writeDebug(value_type var,
               size_type index = 0);
void writeDebug(value_type var,
                size_type index,
                unsigned int acSiz,
               unsigned int offSet);
```

These functions do a debug access to the `scml_memory`. Accesses initiated from a debugger (for example, from a command processor) should always use these functions instead of the `read()`/`write()` or `put()`/`get()` functions.

```
void put_bits(value_type val,
              size_type index,
              size_type offset,
              size_type nBits);
value_type get_bits(size_type index,
                    size_type offset,
                    size_type nBits) const;
```

The bit-access functions `put_bits()` and `get_bits()` address the entry, described by the *`index`* argument, in the memory.
`offset` specifies the lowest bit to be accessed. This is to be interpreted as the second argument to an arithmetic shift operation.
`nBits` specifies the number of bits to be accessed. Hence it determines the size of the mask used in this access function.
These functions can also be used on aliases of size `1`. In this case, *`index`* needs to be set to `0`.

```
scml_memory_iterator<value_type> begin();
scml_memory_iterator<value_type> end();
```

Returns an iterator to the start or the end of the `scml_memory`. The returned iterator is a random access iterator that can be used with algorithms like `std::copy`. For more information, see "Algorithms" on page 21.

2.3.7 Properties

The following functions are available to set properties:

```
void set_endianness(scml_endian end);
scml_endian get_endianness() const ;
```

Set and get the endianness. The endianness that needs to be taken into account for accesses to this memory can be specified with the `scml_endian` enumeration type. The default is set to little endian.

```
void set_addressing_mode(unsigned int mode);
unsigned int get_addressing_mode() const ;
```

Set and get the addressing mode. Knowledge of the addressing mode of the bus is required in the memories. The addressing mode is specified in number of bits, so it is `8` for byte addresses. The default is `8`, corresponding to byte addressing.

```
uint64 getTotalOffset() const;
```

Returns the index position of this memory in the top-level memory.

```
bool store_after_read_call() const;
void set_store_after_read_call(bool);
```

By default, memories store the value produced by the read callback. This default behavior can be disabled by setting this property to `false`.

```
uint64 getSize() const
```

Returns the size of the memory in words.

```
unsigned int getWordSize() const
```

Returns the number of bytes in one word.

```
void set_default_response_latency(unsigned int l);
```

Sets the default to be used for the latency in the PV response and propagates this recursively to its children.

```
unsigned int get_default_response_latency() const;
```

Gets the default to be used for the latency in the PV response.

```
void set_default_response_status(PVResponse s);
```

Sets the default to be used for the response status in the PV response and propagates this recursively to its children.

```
PVResponse get_default_response_status() const;
```

Gets the default to be used for the response status in the PV response.

Note The default value for the latency and the status can be overridden by a transport call-back function attached to the memory.

2.3.8 Algorithms

Objects of type `scml_memory` can be used with algorithms like `std::copy`.

The begin() and end() methods of the memory object and the overloaded address operator of the index object return random access iterators that can be used with these algorithms. For more information, see "Access Functions" on page 19.

For example:

```
scml_memory<unsigned int> mem("mem", scml_memsize(100));
unsigned int a[100];
std::copy(mem.begin(), mem.end(), a);
std::copy(&mem[0], &mem[10], &mem[90]);
```

2.3.9 Behavior

The standard transport mechanism, proposed by the OSCI TLM WG, is used as the basis for communication in scml_memory objects.

```
response_type transport(const request_type& req);
```

where request_type and response_type can be one of the following types:

- scml_memory_pv_*addressWidth*_*dataSize*_if::request_type
- scml_memory_pv_*addressWidth*_*dataSize*_if::response_type

  where *addressWidth* can be 32 or 64 and *dataSize* can be 32, 64, 128, 256, or 512.

2.3.9.1 Default Behavior

Objects of type scml_memory implement the Programmer's View (PV) transport function. The transport implementation in the memory stores the data on write and retrieves the data on read

The transport implementation in the scml_memory objects supports bursts with the access size equal to the size of the underlying data type.

An error response is sent when an access is made to an `scml_memory` object with an access size larger than the size of the underlying data type.

Note The Synopsys PV implementation uses the following data types:

- Bit widths smaller than or equal to 32:

  `unsigned int`

- Bit widths greater than or equal to 33 and smaller than or equal to 64:

  `unsigned long long`

- Bit widths greater than or equal to 65 and smaller than or equal to 128:

  `sc_dt::sc_biguint<128>`

- Bit widths greater than or equal to 129 and smaller than or equal to 256:

  `sc_dt::sc_biguint<256>`

- Bit widths greater than or equal to 257 and smaller than or equal to 512:

  `sc_dt::sc_biguint<512>`

- Bit widths greater than 512 are not supported.

2.3.9.2 Overruling the Default Behavior by Means of Callbacks

To overrule the default behavior (store on write, retrieve on read), you can attach call-back functions.The implementation of a write callback is responsible for storing the new value. In the callback, the old value is available by reading from the memory object. The implementation of a read callback is responsible for returning the current value. After the read callback is executed, the default behavior of the memory object is to store the returned value (see the `store_after_read_call()` function in “Properties” on page 21).

- Attaching call-back functions

  Call-back functions can be attached to memory objects to overrule the default behavior. The following macros are available to achieve this:

```
MEMORY_REGISTER_TRANSPORT(mem, func);
MEMORY_REGISTER_READ(mem, func);
MEMORY_REGISTER_WRITE(mem, func);

MEMORY_REGISTER_NB_TRANSPORT(mem, func);
MEMORY_REGISTER_NB_READ(mem, func);
MEMORY_REGISTER_NB_WRITE(mem, func);

MEMORY_REGISTER_TRANSPORT_USER(mem, func, ID);
MEMORY_REGISTER_READ_USER(mem, func, ID);
MEMORY_REGISTER_WRITE_USER(mem, func, ID);

MEMORY_REGISTER_NB_TRANSPORT_USER(mem, func, ID);
MEMORY_REGISTER_NB_READ_USER(mem, func, ID);
MEMORY_REGISTER_NB_WRITE_USER(mem, func, ID);
```

where:

mem	Is an `scml_memory` object.

func	Is a pointer to a member function, having a transport-like signature (see "Behavior" on page 22) or the following signature in case of read or write callbacks:
	`value_type readLikeCB(unsigned int accessSize, unsigned int offset);`
	`void writeLikeCB(value_type writeData, unsigned int accessSize, unsigned int offset);`

ID	Specifies user-provided data. A type definition is provided in the `scml_memory_user` class for the user-provided ID. `typedef unsigned int user_id_data_type;` This user-provided ID data is set in the instance of the `scml_memory_user` class when the call-back function is triggered for the referenced memory object. The arguments to the callbacks are taken from the PV request structure.

By default, the implementations of these callbacks are allowed to call the System C `wait()` function. In case the implementations do not call the System C `wait()` function and do not write to signals or notify events, you are advised to register them as nonblocking callbacks, by using the `*_NB_*` variants of the macros.

Only one set of call-back functions is allowed per memory object. That means either one transport function, or a read and a write call-back function can be used. Hence, on one memory object a transport call-back function cannot be mixed with a read or a write call-back function, and blocking and nonblocking call-back functions cannot be mixed. It is acceptable to have a transport call-back function on a memory object and read and write call-back functions on an alias to a part of that memory. It is also acceptable to have a blocking call-back function on a memory object and a nonblocking call-back function on an alias to a part of that memory.

Two rules are applied to determine which callback is triggered. First of all, only one callback is triggered. Second, the call-back function (if any) of the most specialized object that can handle the request will be triggered. Hence, if a memory with a registered call-back function is partially aliased by another object with a registered call-back function, the latter is called for the aliased memory locations.

When a burst access cannot be handled by a single callback (if there are different aliases with callbacks or if there are some aliases without callbacks), the burst access is unrolled into single accesses. Each single access is forwarded to the correct callback (or the default behavior is executed if no callback is registered for this address).

Note

- It is not possible to mix different types of transport functions. The attached callbacks must be of the same type as the port to which the `scml_memory` is bound and all transport callbacks registered to an `scml_memory` must be of the type.
- By default, memories store the value returned by a read callback. This behavior can be disabled with `set_store_after_read_call()`.
- Debug access to `scml_memory` does not trigger the registered callbacks. For debug access, the default behavior is executed. If you want to trigger behavior on debug accesses, you should use `scml_router`.

- Removing call-back functions

  To remove attached callbacks from memory objects, the following functions are available:

  `void unregisterCB()`

  This removes the attached transport callback or the attached read and write callbacks.

  `void unregisterReadCB()`

  This removes the attached read callback; an attached write callback or transport callback is not removed.

  `void unregisterWriteCB()`

  This removes the attached write callback; an attached read callback or transport callback is not removed.

- Address to index conversion

  The following functions are available to convert addresses to indices and vice versa. In this context, an index is taken to indicate an argument to the `operator[]` to access array objects. The address is the address presented on the PV interface, from which the base address has been subtracted, so that locally index zero corresponds to address zero.
  The byte address is the address multiplied by the addressing mode divided by `8`.
  The base address is the offset of the first address of the alias relative to the start of the top-level `scml_memory`. The base address of the top-level `scml_memory` is always `0`.

```
// address & index relations
size_type addressToIndex(uint64) const;

uint64 indexToAddress(size_type) const;

uint64 indexToByteAddress(size_type) const;
```

  For subword accesses, the following function is available:

```
size_type addressToIndex(uint64,
                         unsigned int dataSize,
                         unsigned int& offset)
```

  where *`dataSize`* is the size in bits; this must be a multiple of `8`. After the call, the output parameter *`offset`* will contain the lowest bit to be accessed. This is to be interpreted as the second argument to an arithmetic shift operation. The calculations of *`offset`* take the endianness of the `scml_memory` into account.

2.3.9.3 Maintaining Control over Allowed Operations

Memory objects check whether accesses made over the bus are allowed operations. Allowed operations are read and write.

```
unsigned int get_allowed_operations() const;
void set_allowed_operations(unsigned int);
```

Gets and sets allowed operations controls.

```
void set_read_only(bool signal_error = true);
void set_write_only(bool signal_error = true);
```

Convenience functions to declare a memory read only or write only.
If `signal_error` is set to `true`, an error response is returned when the operation that is not allowed is

tried. The default value is `true`.
If `signal_error` is set to `false`, the error is not signaled, that is, an `OK` response is returned when the operation that is not allowed is tried, but the access is not performed (it is ignored).

Note Debug transactions, that is, transactions having `pvDebug` as mode in the PV request structure, are allowed. Hence debug read and debug write operations will be performed.

```
bool is_reading_allowed() const;
bool is_writing_allowed() const;
bool signal_error_on_disallowed_read() const;
bool signal_error_on_disallowed_write() const;
```

Convenience boolean test functions.

```
static const unsigned int
scml_memory_base::OPERATIONS_ALLOW_READ = 1;
static const unsigned int
scml_memory_base::OPERATIONS_ALLOW_WRITE = 2;
static const unsigned int scml_memory_base::OPERATIONS_ERROR_DISALLOWED_READ
                                                                        = 4;
static const unsigned int scml_memory_base::OPERATIONS_ERROR_DISALLOWED_WRITE
                                                                         = 8;
```

Symbolic constants.

2.3.10 Binding

An `scml_memory` object can be bound to a `PVTarget_port`. Accesses arriving on this port will be forwarded to the transport interface of the `scml_memory`.

2.4 scml_memory_user

The `scml_memory_user` class is a helper class to provide access to information that would otherwise not be available in call-back functions.

2.4.1 Types

The following type definitions are provided:

```
typedef unsigned int user_id_data_type;
typedef scml_memory_base::uint64 address_type;
typedef scml_memory_base::this_reference_type scml_memory_reference;
typedef scml_bitfield_base::scml_bitfield_reference scml_bitfield_reference;
```

2.4.2 Object Access

The following object access is provided:

```
static scml_memory_user & instance();
```

Provides a reference to the single instance of this class.

2.4.3 Access Functions

The following access functions are provided:

```
address_type get_last_top_level_address() const;
```

Provides the address presented to the top-level memory, that is, before address subtraction when propagating the request to an alias has been performed.

`user_id_data_type get_last_user_id() const;`

Provides the user ID data that was passed when the call-back function was registered with a memory object.

`scml_memory_reference get_last_memory_reference() const;`

Provides a reference to the most recently accessed memory object.

`scml_bitfield_reference get_last_bitfield_reference() const;`

Provides a reference to the most recently accessed bitfield object.

`bool is_debug_access() const;`

Checks whether a certain access is a debug access in case the PV request object is not directly available.

2.4.4 Example

In a callback attached to a bitfield object, you can retrieve the information related to the access to the current bitfield object as follows:

```
void
Peripheral::bitfieldWriteCB(scml_bitfield::value_type val)
{
  scml_memory_user & inst(scml_memory_user::instance());
  scml_memory_user::user_id_data_type id(inst.get_last_user_id());
  scml_memory_user::address_type topAddress = inst.get_last_top_level_address();
  scml_memory_user::scml_bitfield_reference
                                bitref(inst.get_last_bitfield_reference());
  // code dependent on ID to distinguish between different bitfields
}
```

2.5 scml_memsize

The `scml_memsize` class is a proxy class to identify a constructor argument in `scml_memory` as a size parameter rather than an initial value.

2.5.1 Types

`typedef unsigned long long size_type;`

2.5.2 Constructors

The following constructor is available:

`scml_memsize(size_type memSize);`

where *memSize* specifies the memory size.

This constructor is explicit.

2.5.3 Operators

The following operator is available:

`operator size_type() const;`

An `scml_memsize` can be converted to the underlying size type.

2.6 scml_pv_decoder

DEPRECATED: scml_pv_decoder is deprecated but is still available for backward compatibility. The same functionality is available by using scml_router with statically mapped ranges.

Objects of type scml_pv_decoder are address decoders for PV requests. They can be bound to multiple PV transport implementations. Based on the address, the request will be forwarded to the corresponding transport implementation. A typical use case is to bind an scml_pv_decoder to a PV target port and bind multiple scml_memory objects to the decoder. In SystemC terminology, an scml_pv_decoder is an sc_module with two ports, named bus_port and mem_port.

2.6.1 Types

The scml_pv_decoder class is templated with the width of the address and data type. Several type definitions are provided to support generic programming.

```
template<unsigned int address_width = 32,
         unsigned int data_width = 64>
class scml_pv_decoder;

typedef typename PV::dataWidth2Type<data_width>::result data_type;
typedef typename PV::dataWidth2Type<address_width>::result address_type;
typedef PVReq<data_type, address_type> request_type;
typedef PVResp<data_type> response_type;
typedef PV_if<data_type, address_type> interface_type;
typedef interface_type * interface_pointer_type ;
typedef data_type value_type;
typedef PVTarget_port<data_type, address_type> bus_port_type;
typedef scml_pv_decoder_port<address_width, data_width> peripheral_port_type;
```

The type used is unsigned int for widths up to 32 and unsigned long long for widths from 33 up to 64.

Note scml_pv_decoder does not support widths greater than 64 bits. For bigger widths, scml_router should be used.

2.6.2 Constructors

An explicit constructor taking a name is available:

```
explicit scml_pv_decoder(sc_module_name name);
```

2.6.3 Communication

```
virtual response_type transport(const request_type &);
```

Objects of type scml_pv_decoder implement the PV transport function. The transport implementation is to decode the incoming request and forward it to the corresponding server bound on mem_port.

2.6.4 Public Data Members

```
bus_port_type bus_port
peripheral_port_type mem_port
```

The module's two ports are publicly accessible.

2.6.5 Example

The following example shows a module using an `scml_pv_decoder` to bind multiple SCML memories to one PV port.

```
SC_MODULE(ExampleModule) {

  // pv port + scml_pv_decoder
  PVTarget_port<unsigned int, unsigned int> p;
  scml_pv_decoder<32, 32> d;
  scml_memory<unsigned int> mem1;
  scml_memory<unsigned int> mem2;

  ExampleModule(sc_module_name name) :
    p("p"),
    d("d"),
    mem1("mem1", scml_memsize(0x100)),
    mem2("mem2", scml_memsize(0x100))
  {
    //  bind the scml_pv_decoder to the pv port
    //  bind 2 memories to the scml_pv_decoder
    //      mem1: offset 0x0
    //      mem2: offset 0x10000
    p(d.bus_port);
    d.mem_port(mem1, 0x0, 0x100 * sizeof(unsigned int));
    d.mem_port(mem2, 0x10000, 0x100 * sizeof(unsigned int));
  }

  <...>

};
```

2.7 scml_pv_decoder_port

The `scml_pv_decoder_port` is the port that provides the functionality offered by the `scml_pv_decoder` module.

2.7.1 Types

The `scml_pv_decoder_port` class is templated with the width of the address and data type. Several type definitions are provided to support generic programming.

```
template<unsigned int ADDRESS_WIDTH_ = 32,
         unsigned int DATA_WIDTH_ = 64>
class scml_pv_decoder_port;

typedef typename PV::dataWidth2Type<DATA_WIDTH_>::result data_type;
typedef typename PV::dataWidth2Type<ADDRESS_WIDTH_>::result address_type;
typedef scml_pv_multi_port<typename PV::dataWidth2Type<DATA_WIDTH_>::result,
        typename PV::dataWidth2Type<ADDRESS_WIDTH_>::result > base_port_type;
typedef PVReq<data_type, address_type> request_type;
typedef PVResp<data_type> response_type;
typedef PV_if<data_type, address_type> interface_type;
typedef interface_type * interface_pointer_type ;
typedef data_type value_type;
```

The type used is `unsigned int` for widths up to 32 and `unsigned long long` for widths from 33 up to 64.

2.7.2 Constructors

A default constructor and an explicit constructor taking a name are available:

```
scml_pv_decoder_port();
explicit scml_pv_decoder_port(const char * name);
```

2.7.3 Port Binding

```
void bind(interface_type & i, address_type base, address_type size = 0);
void operator() (interface_type & i, address_type base, address_type size = 0);
```

Both the `bind()` function and the function call `operator()` are available for port binding. The additional functionality is that the base address and the size for the address region are added when the binding is performed.

The decoder is ignorant of the addressing mode. The base address (*base*) and the size (*size*) of the region need to be specified in terms of the addressing mode used in PV. It can be byte addresses or word addresses, but it needs to be consistent within the PV domain.

Caution Attaching memory objects with different addressing modes to the same decoder object may result in unexpected results.

The following example illustrates port binding.

Instantiate a decoder in the module:

```
scml_pv_decoder<32, 32> dec;
```

Bind its bus port hierarchically to the PV target port to connect to the bus:

```
module.pv_bus_port.bind(dec.bus_port);
```

Bind the different memories to the memory port of the decoder, providing the base address and the size:

```
const unsigned int aMemBaseAddress = 0x0;
const unsigned int aMemSize = 0x200;
const unsigned int bMemBaseAddress = 0x1000;
const unsigned int bMemSize = 0x400;
const unsigned int cMemBaseAddress = 0x2000;
const unsigned int cMemSize = 0x800;

dec.mem_port.bind(aMem, aMemBaseAddress, aMemSize);
dec.mem_port.operator()(bMem, bMemBaseAddress, bMemSize);
dec.mem_port(cMem, cMemBaseAddress, cMemSize);
```

2.8 scml_router

The `scml_router` object is similar to the `scml_memory` object, but it has no associated storage and no default behavior. You must attach a callback to the `scml_router` object that implements the desired behavior of accesses to this memory range.

scml_router objects can map a memory region to a region in an scml_memory, scml_post_port, or scml_tlm2_initiator_adaptor object. Accesses to mapped regions do not trigger the attached callback, but are automatically forwarded to the destination object.

2.8.1 Types

The scml_router class is templated with the underlying value type.

The following type definitions are available to support generic programming:

```
template <typename DT> class scml_router;
typedef DT value_type;
typedef value_type* pointer;
typedef const value_type* const_pointer;
typedef value_type* iterator;
typedef const value_type* const_iterator;
typedef value_type& reference;
typedef const value_type& const_reference;
typedef unsigned long long size_type;
typedef ptrdiff_t difference_type;
typedef scml_router<value_type> this_type;
typedef this_type & this_reference_type;
typedef scml_memory_pv32_if::request_type request32_type;
typedef scml_memory_pv32_if::response_type response32_type;
```

2.8.2 Enumeration Types

The following enumeration type is available:

```
enum scml_endian { scml_little_endian, scml_big_endian };
```

The scml_endian enumeration type is used to describe the endianness.

2.8.3 Constructors

Objects of type scml_router can be constructed using one of the following constructors:

```
scml_router(const char* name, scml_memsize size);
scml_router(const ::std::string& name, scml_memsize size);
```

where:

name	Specifies the name of the scml_router object. The name must be the same as the name of the target location that corresponds with this object. For more information, see "Specifying Target Locations" in the *Platform Creator User's Guide*.
size	Specifies the size in words. The word size depends on the template parameter of the scml_router object.

Unlike for scml_memory objects there is only one type of scml_router object. Aliases are not supported for scml_router objects.

The copy constructor is disabled.

2.8.4 Access Functions

The following access functions are available:

```
value_type read(size_type index = 0);
value_type read(size_type index, unsigned int acSiz, unsigned int offSet);
void write(value_type var, size_type index = 0);
void write(value_type var, size_type index, unsigned int acSiz,
           unsigned int offSet);
```

These functions trigger the registered callback for unmapped memory ranges. For mapped memory ranges, the functions forward the request to the real object that has the storage.
In case the access size (in bits) is not equal to the default access size (that is, eight times the size (in bytes) of the underlying `value_type`), the access size can be specified with the *`acSize`* variable. The access size (*`acSize`*) has to be a multiple of `8`.
`offSet` specifies the lowest bit to be accessed. This is to be interpreted as the second argument to an arithmetic shift operation.

Note The `read()` and `write()` functions trigger the registered call-back function (for nonmapped memory ranges). If these functions are called from the call-back function, this may result in an endless loop.

```
value_type readDebug(size_type index = 0);
value_type readDebug(size_type index,
                     unsigned int acSiz,
                    unsigned int offSet);
void writeDebug(value_type var,
               size_type index = 0);
void writeDebug(value_type var,
                size_type index,
                unsigned int acSiz,
               unsigned int offSet);
```

These functions do a debug access to the `scml_router`. Accesses initiated from a debugger (for example, from a command processor) should always use these functions instead of the `read()`/`write()` functions.

2.8.5 Properties

The following functions are available to set properties:

```
void set_endianness(scml_endian end);
scml_endian get_endianness() const;
```

Set and get the endianness. The endianness that needs to be taken into account for accesses to this memory can be specified with the `scml_endian` enumeration type. The default is set to little endian.

```
uint64 getSize() const
```

Returns the size of the memory in words.

```
unsigned int getWordSize() const
```

Returns the number of bytes in one word.

```
void allowOptimisedAccessForMappedRanges();
void disallowOptimisedAccessForMappedRanges();
```

Enable/disable optimized accesses to the `scml_router` for mapped regions without support for optimized accesses. The default behavior is to enable optimized accesses to the `scml_router` in case the destination of the mapped range does not support optimized accesses.

```
void allowOptimisedAccessForUnmappedRanges();
void disallowOptimisedAccessForUnmappedRanges();
```

Enable/disable optimized accesses to the `scml_router` for unmapped regions. The default behavior is to enable optimized accesses to the `scml_router` in case of unmapped regions.

2.8.6 Behavior

The standard transport mechanism, proposed by the OSCI TLM WG, is used as the basis for communication in `scml_router` objects.

```
response_type transport(const request_type& req);
```

where `request_type` and `response_type` can be one of the following types:

- `scml_memory_pv_`*`addressWidth_dataSize`*`_if::request_type`
- `scml_memory_pv_`*`addressWidth_dataSize`*`_if::response_type`

  where *`addressWidth`* can be `32` or `64` and *`dataSize`* can be `32`, `64`, `128`, `256`, or `512`.

Objects of type `scml_router` have no default behavior. You must attach a callback if accesses to unmapped memory are done. The implementation of a callback is responsible for handling the request (forwarding it to another port, storing it locally in another `scml_memory`, and so on.

- Attaching call-back functions

  The following macros are available to attach a callback:

```
ROUTER_REGISTER_TRANSPORT(router, func);
ROUTER_REGISTER_NB_TRANSPORT(router, func);
ROUTER_REGISTER_TRANSPORT_USER(router, func, ID);
ROUTER_REGISTER_NB_TRANSPORT_USER(router, func, ID);
```

where:

`router`	Is an `scml_router` object.
`func`	Is a pointer to a member function, having a transport-like signature. For more information, see "Behavior" on page 33.
`ID`	Specifies user-provided data. A type definition is provided in the `scml_memory_user` class for the user-provided ID. `typedef unsigned int user_id_data_type;` This user-provided ID data is set in the instance of the `scml_memory_user` class when the call-back function is triggered for the referenced router object. By default, the implementations of these callbacks are allowed to call the System C `wait()` function. In case the implementations do not call the System C `wait()` function and do not write to signals or notify events, you are advised to register them as nonblocking callbacks, by using the `*_NB_*` variants of the macros.

> **Note**
> - There should always be a callback attached to the `scml_router` object. If no callback is attached, this may result in undefined behavior at runtime.
> - It is not possible to mix different types of transport functions. The attached callback must be of the same type as the port to which the `scml_router` is bound.

- Removing call-back functions

  To remove attached callbacks from router objects, the following function is available:

  ```
  void unregisterCB()
  ```

  This removes the attached transport callback.

- Address to index conversion

  The following functions are available to convert addresses to indices and vice versa. In this context, an index is taken to indicate an argument to the `operator[]` to access array objects. The address is the address presented on the PV interface, from which the base address has been subtracted, so that locally index zero corresponds to address zero.
  The byte address is the address multiplied by the addressing mode divided by `8`.
  The base address is the offset of the first address of the alias relative to the start of the top-level `scml_memory`. The base address of the top-level `scml_memory` is always `0`.

  ```
  // address & index relations
  size_type addressToIndex(uint64) const;

  uint64 indexToAddress(size_type) const;

  uint64 indexToByteAddress(size_type) const;
  ```

  For subword accesses, the following function is available:

  ```
  size_type addressToIndex(uint64,
                           unsigned int dataSize,
                           unsigned int& offset)
  ```

  where *`dataSize`* is the size in bits; this must be a multiple of `8`. After the call, the output parameter *`offset`* will contain the lowest bit to be accessed. This is to be interpreted as the second argument to an arithmetic shift operation. The calculations of *`offset`* take the endianness of the `scml_router` into account.

- Debug accesses

  Debug accesses to the `scml_router` object trigger the attached callback for unmapped memory regions. Accesses to mapped memory regions are automatically forwarded by the `scml_router` object. This means that the attached callback must handle both regular accesses and debug accesses.

> **Note**
> - The implementation of the callback should not call the SystemC `wait()` function for debug accesses. This means that the `post()` function of the `scml_post_port` should not be used for debug accesses. It is advised to always use the `transport()` function of `scml_post_port` from the callback of an `scml_router` object. This function can be used both from a blocking and a nonblocking callback. For debug accesses, the `readDebug()`/`writeDebug()` or `put()`/`get()` methods must be used to access `scml_memory` or `scml_router` objects. Using `read()`/`write()` may result in undefined behavior.

2.8.7 Mapping/Unmapping Memory Ranges

The following functions are available to map/unmap memory regions:

```
bool map(size_type base, size_type size,scml_mapable_if& destination,
         size_type offset);
```

Maps the memory range (*base*, *base* and *size*) of the port to which the `scml_router` object is bound to the memory range (*offset*, *offset* and *size*) of the destination. Possible destinations are: `scml_post_port`, `scml_memory`, `scml_router`, `scml_tlm2_initiator_adaptor`.

All addresses/sizes are in bytes.

Returns `true` if the mapping succeeds and `false` if it fails.

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

Note

- It is not allowed to map the same or overlapping regions. Mapped ranges that are only mapped for read accesses may overlap with mapped ranges that are only mapped for write access.
- Mapped ranges must be word aligned and the size of the mapped range must be a multiple of the word size. The mapped range must fall inside the range of the `scml_router` object.

The `map()` function maps both write and read accesses.

```
bool map_read(size_type base, size_type size, scml_mapable_if& destination,
              size_type offset);
```

Same as `map()` (see above) but only for read accesses.

```
bool map_write(size_type base, size_type size, scml_mapable_if& destination,
               size_type offset);
```

Same as `map()` (see above) but only for write accesses.

```
bool unmap(size_type 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(size_type base);
```

Same as `unmap()` (see above) but only for read accesses.

```
bool unmap_write(size_type base);
```

Same as `unmap()` (see above) but only for write accesses.

```
void unmap_all();
```

Unmaps all mapped ranges.

Note

- Mapping and unmapping of memory regions can be done statically (from the constructor of the `sc_module`) or dynamically (from the attached callback).
- It is not possible to map to different types of post ports. The data type and address type of the post ports must be the same as the data type and address type of the registered transport callback of the `scml_router` object or the data type and address type of the port to which the `scml_router` is bound.
- If a memory region is mapped to an `scml_memory` object (or another `scml_router` object), the transport callbacks registered to both objects should be of the same type.

2.8.8 Binding

An `scml_router` object can be bound to a `PVTarget_port`. Accesses arriving on this port will be forwarded to the transport interface of the `scml_router`.

2.9 Code Example

The declarations in `Timer.h` look as follows:

```
SC_MODULE(Timer) {
  SC_HAS_PROCESS(Timer);
  Timer(sc_module_name);

  typedef scml_memory_pv32_if::request_type request_type;
  typedef scml_memory_pv32_if::response_type response_type;
  typedef scml_memory_pv32_if::address_type address_type;
  typedef scml_memory_pv32_if::data_type data_type;
  typedef scml_bitfield::value_type value_type;

  response_type transportLikeCB0(const request_type &);
  response_type transportLikeCB1(const request_type &);
  response_type transportLikeCB2(const request_type &);
  response_type transportLikeCB3(const request_type &);
  response_type transportLikeCB4(const request_type &);
  response_type transportLikeCB5(const request_type &);
  response_type transportLikeCB6(const request_type &);

  data_type readLikeCB0();
  data_type readLikeCB1();
  data_type readLikeCB2();
  data_type readLikeCB3();
  data_type readLikeCB4();
  data_type readLikeCB5();

  void writeLikeCB0(data_type);
  void writeLikeCB1(data_type);
  void writeLikeCB2(data_type);
  void writeLikeCB3(data_type);
  void writeLikeCB4(data_type);
  void writeLikeCB5(data_type);

  // data members
  scml_memory<int> timerRegs;
  scml_memory<int> load;
  scml_memory<int> value;
  scml_memory<int> control;
  scml_memory<int> intClr;
  scml_memory<int> RIS;
  scml_memory<int> MIS;
  scml_memory<int> BGLoad;

  scml_bitfield timerEnable;
  scml_bitfield timerMode;
  scml_bitfield interruptEnable;
  scml_bitfield timerPre;
  scml_bitfield timerSize;
  scml_bitfield oneShotCount;
};
```

The implementations in `Timer.cpp` look as follows:

```
Timer::Timer(sc_module_name n) :
sc_module(n),
  timerRegs("timerRegs", scml_memsize(7)),
  load(    "load",    timerRegs, 0, 1),
  value(   "value",   timerRegs, 1, 1),
  control( "control", timerRegs, 2, 1),
  intClr(  "intClr",  timerRegs, 3, 1),
  RIS(     "RIS",     timerRegs, 4, 1),
  MIS(     "MIS",     timerRegs, 5, 1),
  BGLoad(  "BGLoad",  timerRegs, 6, 1),
  timerEnable(     "Enable",          control, 7, 1),
  timerMode(       "Mode",            control, 6, 1),
  interruptEnable( "interruptEnable", control, 5, 1),
  timerPre(        "Pre",             control, 2, 2),
  timerSize(       "Size",            control, 1, 1),
  oneShotCount(    "oneShotCount",    control, 0, 1),
  ringerIsEnabled( false)
{
  timerRegs.set_addressing_mode(32);

  MEMORY_REGISTER_TRANSPORT(load,    transportLikeCB0);
  MEMORY_REGISTER_TRANSPORT(value,   transportLikeCB1);
  MEMORY_REGISTER_TRANSPORT(control, transportLikeCB2);
  MEMORY_REGISTER_TRANSPORT(intClr,  transportLikeCB3);
  MEMORY_REGISTER_TRANSPORT(RIS,     transportLikeCB4);
  MEMORY_REGISTER_TRANSPORT(MIS,     transportLikeCB5);
  MEMORY_REGISTER_TRANSPORT(BGLoad,  transportLikeCB6);

  BITFIELD_REGISTER_READ(timerEnable,     readLikeCB0);
  BITFIELD_REGISTER_READ(timerMode,       readLikeCB1);
  BITFIELD_REGISTER_READ(interruptEnable, readLikeCB2);
  BITFIELD_REGISTER_READ(timerPre,        readLikeCB3);
  BITFIELD_REGISTER_READ(timerSize,       readLikeCB4);
  BITFIELD_REGISTER_READ(oneShotCount,    readLikeCB5);

  BITFIELD_REGISTER_WRITE(timerEnable,     writeLikeCB0);
  BITFIELD_REGISTER_WRITE(timerMode,       writeLikeCB1);
  BITFIELD_REGISTER_WRITE(interruptEnable, writeLikeCB2);
  BITFIELD_REGISTER_WRITE(timerPre,        writeLikeCB3);
  BITFIELD_REGISTER_WRITE(timerSize,       writeLikeCB4);
  BITFIELD_REGISTER_WRITE(oneShotCount,    writeLikeCB5);
}

Timer::response_type
Timer::transportLikeCB0(const request_type & req) {
  const PVType reqType(req.getType());
  response_type resp(req.obtainResp());
  resp.setResponse(pvOk);
  if(pvWrite == reqType) {
  // ...
  } else if(pvRead == reqType) {
  // ...
  }
  return resp;
}

Timer::data_type
Timer::readLikeCB5()
{
// oneShotCount = 1;
return newValueOfThisBitField();
}

void Timer::writeLikeCB0(data_type d)
{
  value_type b = timerEnable;
  timerEnable = d;
  ::std::cout << " bitfield old value = " << b
          << " new value = " << d << ::std::endl;
}
```

Chapter 3
Clock Objects

This chapter describes the clock objects.

- Overview
- scml_clock
- scml_clock_counter
- scml_clock_gate
- scml_divided_clock
- Code Examples

Note If the following error message is issued when simulating:

```
ERROR : Derived clocks must be derived from a channel that
implements scml_clock_if !
```

you must use an object of type `scml_clock` for the master clock 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; // master clock object
```

3.1 Overview

The following table summarizes the clock objects.

Table 3-1 Clock Objects

Modeling Object	Summary
`scml_clock`	Implements `sc_clock_if`. It is an optimized version of `sc_clock`.
`scml_clock_counter`	Has to be attached to a clock and is used to get the number of clock edges that have happened in a certain period.
`scml_clock_gate`	Is a module which takes a clock and an enable as inputs and produces a gated clock as output.
`scml_divided_clock`	Is a clock derived from another clock by multiplying the start time and/or the period with specified integer factors.

3.2 scml_clock

An `scml_clock` object implements `sc_clock_if`.

3.2.1 Constructors

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:

name	Specifies a name for the clock object.
period	Specifies the clock period.
periodV	Specifies the value of the clock period.
periodTu	Specifies the time unit for the clock period.
dutyCycle	Specifies the duty cycle of the clock object.
startTime	Specifies the time of the first clock edge.
startTimeV	Specifies the value of the time of the first clock edge.
startTimeTu	Specifies the time unit for the time of the first clock edge.
posedgeFirst	Specifies if the first edge will be a `posedge` or a *negedge*.

3.2.2 Properties

The following functions are available to set/get properties of the clock object:

```
void set_period(const sc_core::sc_time &t);
const sc_core::sc_time& get_period() const;
```

Gets and sets the clock period.

```
double get_duty_cycle()const;
void set_duty_cycle(double d);
```

Gets and sets the duty cycle.

```
const sc_core::sc_time& get_start_time()const;
```

Returns the start time of the clock.

```
bool get_posedge_first()const;
void set_posedge_first(bool posedgeFirst);
```

Gets and sets the `posedge_first` property of the clock.

```
sc_dt::uint64 get_clock_count();
```

Queries the counter value of a clock. For more information, see “scml_clock_counter” on page 41.

```
void enable();
void disable();
```

Enables (that is, makes active) or disables (that is, makes inactive) the clock. When `scml_clock` is disabled, the output is `0`.

```
bool disabled();
```

Tests wether the clock is disabled.

```
void reset(int n);
```

Suspends the clock for an integer amount `n` of clock cycles. The output of `scml_clock` stays low or `0` during reset (independent of the value of `enable()` and `disable()`).

3.2.3 Events

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 wether a certain event occurred.

3.2.4 Tracing

For tracing purposes, a reference to the current value can be obtained:

```
const bool & get_data_ref() const;
```

Caution The `scml_clock` cannot be optimized when tracing is enabled. Enabling tracing will disable clock optimizations.

3.3 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`.

3.3.1 Types

The following type definitions are available:

```
typedef sc_dt::uint64 data_type;
```

3.3.2 Constructors

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:

name	Specifies a name for the clock object.
clk	Specifies the clock that should be used to determine this counter value.

The single-argument constructor is explicit.

3.3.3 Properties

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.

3.3.4 Binding

The following functions are provided to connect the clock counter to its input clock.

```
void bind(scml_clock_if &);
void operator()(scml_clock_if &);
```

3.4 scml_clock_gate

`scml_clock_gate` is a module which takes a clock and an enable 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.

3.4.1 Ports

A clock gate has a clock and an enable input port:

```
sc_in<bool> clk;
sc_in<bool> en;
```

3.4.2 Constructors

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.

3.4.3 Properties

The scml_clock_gate modeling object can be used as a normal clock. For an API reference, see "scml_divided_clock" on page 43.

3.5 scml_divided_clock

A divided clock is a clock 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.

3.5.1 Constructors

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:

name	Specifies a name for the clock object.
clk	Specifies the clock from which this object is derived.
periodMultiplier	Specifies the factor by which the period is multiplied.
startMultiplier	Specifies the factor by which the start time is multiplied.

The default values are such that a clone of the incoming clock is obtained.

3.5.2 Properties

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 2 times `p`.
`set_divider(4)` indicates that the new period is 4 times the original period, that is `8p`.

```
const sc_core::sc_time& get_period() const;
```

Returns the period of the clock.

```
sc_dt::uint64 get_clock_count();
```

Queries the counter value of a clock (see "scml_clock_counter" on page 41).

```
void enable();
void disable();
```

Enables (that is, makes active) or disables (that is, makes inactive) the clock. `scml_divided_clock` acts as a clock gate: When `enable()` is `true`, the clock signal is passed; when `enable()` is `false`, the output is `0`.

```
bool disabled();
```

Tests whether the clock is disabled.

```
void reset(int n);
```

Suspends the clock for an integer amount *n* of clock cycles.
The output of `scml_divided_clock` stays `low` or `0` during reset (independently of the value of `enable()` and `disable()`).

3.5.3 Events

Several events are available for clock objects. They 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() const;
bool posedge() const;
bool negedge() const;
```

Boolean functions to test whether a certain event occurred.

3.5.4 Tracing

For tracing purposes, a reference to the current value can be obtained:

```
const bool & get_data_ref() const;
```

3.5.5 Binding

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 &);
```

3.6 Code Examples

This sections shows the code of `Timer.h` and describes programmable clock peripherals, respectively.

3.6.1 Timer

The code in `Timer.h` look as follows:

```
SC_MODULE(Timer) {
  sc_in<bool> clk;
  sc_in<bool> clkEn;

  scml_divided_clock iClockMirror;
  scml_clock_gate    iClockGate;
  scml_divided_clock iClockAfterGate;
  scml_clock_counter iClkCount;
  scml_divided_clock iDivClock;
  scml_clock_counter iDivClkCount;

  Timer(sc_module_name);
}
```

The code in `Timer.cpp` look as follows:

```
Timer::Timer(sc_module_name n) :
  sc_module(n),
  clk("clock"),
  clkEn("clockEnable"),
  iClockMirror("internalClockMirror", clk, 1, 1),
  iClockGate("internalClockGate"),
  iClockAfterGate("internalClockAfterGate", iClockGate, 1, 1),
  iClkCount("iClkCount", iClockMirror),
  iDivClock("internalDividedClock", clk, 1, 1),
  iDivClkCount("iDivClkCount", iDivClock)
{
  iClockGate.clk(iClockMirror);
  iClockGate.en(clkEn);
}
```

3.6.2 Programmable Clock Peripherals

Programmable clock peripherals and timers can be coded easily, by calling the clock object functions from a call-back 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> END_VALUE_REG;
scml_divided_clock timer_out_clock;
scml_clock_counter counter;
```

A call-back function is registered with the `END_VALUE_REG` memory-mapped register:

```
MEMORY_REGISTER_WRITE(END_VALUE_REG, f_write_end_value);
```

The implementation of this call-back function is as follows: Writing to the `END_VALUE_REG` memory-mapped register results in the divided clock `timer_out_clock` getting a new value for the divisor and the separate counter being reset to zero.

```
void
simple_timer_PV::f_write_end_value(unsigned int new_value,
                                   unsigned int accsize,
                                   unsigned int offset)
{
  assert(accsize == 32);
  assert(offset == 0);
```

```
  END_VALUE_REG = new_value;
  timer_out_clock.set_divider(new_value);
  counter.set_count(0);
}
```

Chapter 4
Initiator-Side Objects

The goal of the initiator-side modeling objects is to support the creation of reusable and efficient models. In contrast to targets, initiators are characterized by the fact that they autonomously initiate transactions.

The focus of the initiator-side modeling objects is modeling of bus-protocol-agnostic, behavioral initiator models. Only a bus protocol agnostic modeling style enables the reuse of models across multiple abstraction levels and use models. Examples of bus protocol agnostic initiator peripherals are any PV or AV model or a generic DMA model.

The basic principle behind the communication architecture in the initiator-side modeling objects is that scheduling (posting) of transactions is nonblocking and data access can be blocking to provide a synchronization mechanism.

- Overview
- scml_array
- scml_array_base
- scml_pool
- scml_post_if
- scml_post_port
- scml_transaction_request

4.1 Overview

The following table summarizes the initiator-side objects.

Table 4-1 Initiator-Side Objects

Modeling Object	Summary
`scml_array`	Maintains information about locks placed on storage and validity of data stored in an array of a PODT.
`scml_array_base`	Is a template-free base class used by `scml_array`. It provides the claim and release protocol for synchronization and several functions for data access.
`scml_pool`	is a memory management helper class.
`scml_post_if`	Describes the interface used to submit a request for communication.
`scml_post_port`	Is an object of type `sc_port` instantiated with the corresponding post interface.
`scml_transaction_request`	Is used to describe a request for communication.

4.2 scml_array

An scml_array object maintains information about locks placed on storage and validity of data, stored in an array of a Plain Old Data Type (PODT). The scml_array object does not allocate, maintain, or release the actual storage.

A claim and release protocol is used to lock and unlock the storage position and to validate and invalidate data in the array.

The write cycle is as follows:

1. First claim the space, which locks the referenced positions in the array.
2. Next write the data into the array, for example using the [] and = operators.
3. Finally, release the data. This marks the referenced positions in the array as valid. As a result of releasing data, the ok to claim data event will be notified in the next delta cycle.

The read cycle is as follows:

1. First claim the data. This checks the validity of the referenced positions in the array.
2. Next, read the data from the array, for example using the [] and = operators.
3. Finally, release the space. This marks the referenced positions in the array as invalid and releases the lock on these positions. As a result of releasing data, the ok to claim space event will be notified in the next delta cycle.

4.2.1 Types

The scml_array class is templated with the data type of the underlying PODT array and provides the following type definitions to support generic programming.

```
template<typename DT> class scml_array;
  typedef DT value_type;
  typedef value_type * pointer;
  typedef const value_type * const_pointer;
  typedef value_type * iterator;
  typedef const value_type * const_iterator;
  typedef value_type & reference;
  typedef const value_type & const_reference;
  typedef scml_array_base::size_type size_type
```

4.2.2 Constructors

The following constructors are available:

```
explicit scml_array(size_type size);
scml_array(pointer p, size_type size);
scml_array(const char* name, pointer p, size_type n)
```

where:

size	Specifies the number of entries for which access control needs to be done.
p	Needs to point to the first element in the PODT array for which this element has to do access control.
name	Specifies the name of the scml_array object.

The following code shows an example.

```
typedef unsigned int value_type;
value_type rawStorage[20]; // PODT array
scml_array<value_type> com(rawStorage, 20); // access control on whole array
```

The single-argument constructor is explicit and protected. It is only to be used for custom storage elements inheriting from `scml_array`. For more information, see "Protected Access Functions" on page 49.

4.2.3 Access Functions

The following access functions are available:

```
const_reference operator[] (size_type) const;
reference operator[] (size_type index);
```

The access operator `[]` is provided for convenience and forward access to the underlying PODT array.

4.2.4 Protected Access Functions

The following protected access functions are available. In case you develop a custom storage element, the array features can be inherited. Since the single-argument constructor sets the storage pointer to `null`, a setter function for the storage pointer is required. The setter function is provided to obtain the pointer to the actual data store.

```
pointer getStoragePointer() const;
void setStoragePointer(pointer);
```

4.2.5 Example

See "Example" on page 56.

4.3 scml_array_base

The `scml_array_base` class is a template-free base class used by `scml_array`. It provides the claim and release protocol for synchronization and several functions for data access.

4.3.1 Constructors

The following constructors are available:

```
explicit scml_array_base(size_type size);
```

where *`size`* specifies the number of entries for which access control needs to be done.

Since the `scml_array_base` class is not meant to be used directly, it only has a protected constructor.

4.3.2 Access Functions

The following access functions are available:

```
typedef void * generic_data_pointer_type;
virtual generic_data_pointer_type get_storage_pointer() const;
```

Returns a pointer to the raw storage.

```
virtual bool get(generic_data_pointer_type destination, size_type index) const;
virtual bool get(generic_data_pointer_type destination, size_type index,
```

```
                  unsigned int acSiz, unsigned int offSet) const = 0;

virtual bool put(generic_data_pointer_type source, size_type index);
virtual bool put(generic_data_pointer_type source, size_type index,
                  unsigned int acSiz, unsigned int offSet);
```

The `get()` and `put()` access functions are nonblocking.
`destination` and *`source`* specify where the data is to be copied to and from, respectively.
`index` specifies the entry in the array to be addressed.
`acSiz` specifies the access size. Possible values are: `8,16,32,64,128,256,512`.
`offSet` specifies the shift amount to be used to access unaligned subwords. Possible values are integer multiples of the access size, smaller than the word size in bits.

```
void repackage(size_type destIndex,
                unsigned int destDataSize,
                this_reference_type srcArray,
                size_type srcIndex,
                unsigned int srcDataSize,
                unsigned int srcDataElements
);
```

The `repackage()` function repacks, that is, copies while changing the data layout, the content of one array into another. The destination array is the array related to this object, that is, the array this function is a member of. It is the array the data is to be copied to.
`destIndex` specifies the first position to be accessed in the destination array.
`destDataSize` specifies the access size to be used when putting, that is writing, the data. Possible values are: `8,16,32,64,128,256,512`.
`srcArray` indicates the array the data is to be taken from.
`srcIndex` specifies the first position to be accessed in the source array.
`srcDataSize` specifies the access size to be used when getting, that is reading, the data. Possible values are `8,16,32,64,128,256,512`.
`srcDataElements` specifies the number of elements in source format to be accessed.

The `repackage()` function automatically synchronizes on the data in the source array and the space in the destination array. To do this, it will:

- Claim data in the source array (`claim_data`)
- Claim space in the destination array (`claim_space`)
- Copy the data
- Release space in the source array
- Release data in the destination array

```
size_type get_size() const;
```

Returns the size of the array.

4.3.3 Synchronization Functions

Both blocking and nonblocking synchronization functions are available.

Following OSCI terminology and practices, *blocking* means that the implementation is allowed to call the SystemC `wait()` family of functions, and hence needs to be called from a System C thread (`SC_THREAD`). *Nonblocking* means that the implementation is not allowed to call the SystemC `wait()` family of functions, and hence can be called from a SystemC method (`SC_METHOD`).

The nonblocking variants return a boolean result and are complemented with a boolean inspection function and an event. The blocking variants never fail and return only when the requested operation has been completed successfully.

4.3.3.1 Nonblocking Synchronization

The following nonblocking synchronization functions are available:

```
bool nb_claim_data(size_type index,
                   size_type length = 1);
bool nb_can_claim_data(size_type index,
                       size_type length = 1) const;
const sc_event & ok_to_claim_data() const;
```

Claiming data is checking for the validity of data, which needs to be done before reading the referenced data.
`index` specifies the first index in the array which is claimed.
`length` specifies the number of entries claimed.
The functions return `false` if at least one data element in the requested range is not valid and hence cannot be claimed. The `ok to claim data` event is notified when the state changes, that is, when some data elements become valid (as a consequence of data being released).

```
bool nb_claim_space(size_type index, size_type length = 1);
bool nb_can_claim_space(size_type index, size_type length = 1) const;
const sc_event & ok_to_claim_space() const;
```

Claiming space is checking for the availability of free entries in the array, which needs to be done before writing into the referenced positions.
`index` specifies the first index in the array which is claimed.
`length` specifies the number of entries claimed.
The functions return `false` if at least one data element in the requested range is not available and hence cannot be claimed. The `ok to claim space` event is notified when the state changes, that is, when some positions become valid (as a consequence of space being released).

```
bool nb_release_data(size_type index, size_type length = 1);
bool nb_release_space(size_type index, size_type length = 1);
```

Releasing data is marking the referenced positions in the array as valid data, that is, the data becomes available to readers. *Releasing space* is marking the referenced positions in the array as invalid and unlocking these positions, that is, these memory locations become available to writers.
`index` specifies the first index in the array which is released.
`length` specifies the number of entries released.
Releasing is not blocking.

4.3.3.2 Blocking Synchronization

The following blocking synchronization functions are available. The blocking variants try to claim the requested data or space and return if this is successful. If it is not successful, the calling thread is suspended by waiting for the corresponding event to be notified. This is repeated until the requested operation can be completed successfully. Hence the blocking functions never fail and return only when the requested operation has been completed successfully.

```
void claim_data(size_type index, size_type length = 1);
void claim_space(size_type index, size_type length = 1);
```

4.3.4 Example

See "Example" on page 56.

4.4 scml_pool

The `scml_pool` class is a host memory management helper class. It is templated with the data type it needs to provide memory management for.
This data type needs to have a default constructor.

4.4.1 Types

The following type definitions are provided to support generic programming:

```
template<typename DT> class scml_pool;
  typedef DT value_type;
  typedef value_type * pointer;
  typedef const value_type * const_pointer;
  typedef value_type * iterator;
  typedef const value_type * const_iterator;
  typedef value_type & reference;
  typedef const value_type & const_reference;
  typedef size_t size_type;
```

4.4.2 Constructors

The following constructor is available:

```
explicit scml_pool(size_type initialPoolSize = 0);
```

where *`initialPoolSize`* specifies the number of objects that are allocated when the pool is created. Hence it is the size of the initial free pool. An object is free when it has not been claimed and hence is not in use.

4.4.3 Access Functions

The following access functions are available:

```
void allocate_additional_entries(size_type numberEntries);
```

Constructs *`numberEntries`* objects, adds them to the pool, and marks them as free.

```
pointer claim();
```

Returns a pointer to a free object and marks it as in use. In case the free pool was empty, additional objects are created and marked as free.

```
void release(reference);
void release(pointer);
```

Marks the object as free.

```
void release_all();
```

Marks all objects in the pool as free.

```
void clear();
```

Destroys all allocated objects. Hence the pool is completely empty.

```
size_type get_number_free() const;
```

Returns the number of free objects in the pool.

```
size_type get_number_in_use() const;
```

Returns the number of objects that are in use, that is, claimed but not yet released.

```
void reduce_pool();
```

Destroys all allocated, free objects. Claimed objects are not freed.

4.4.4 Example

```
typedef unsigned int data_type;
typedef unsigned int address_type;
typedef scml_transaction_request<data_type, address_type> transaction_request_type;
typedef scml_pool<transaction_request_type> pool_type;
typedef pool_type::pointer pointer;

pool_type memoryPool(10);
pointer p1 = memoryPool.claim();
memoryPool.release(p1);
```

4.5 scml_post_if

Describes the interface used to submit a request for communication.

This interface is templated with the address and data types in the same way as the PV interface.

```
template<typename DT = unsigned int, typename AT = unsigned int>
class scml_post_if;
```

4.5.1 Types

The following type definitions are provided to support generic programming:

```
typedef DT data_type;
typedef AT address_type;
```

4.5.2 Interface Functions

The following interface function is available:

```
virtual bool post(scml_transaction_request<DT, AT> &) = 0;
```

Posting a transaction request object is entering a request for communication. It can be done without the actual data being present (in case of a write transaction).
Posting a transaction request object is a nonblocking operation. This is the major difference with the transport in the PV interface.
The actual transaction corresponding to the posted request can be handled based on data availability, which is controlled by the data manager (the object of type `scml_array`) referenced in the transaction request.

The `post()` call can never fail; this means that the return value must always be `true`.

4.5.3 Example

See "Example" on page 56.

4.6 scml_post_port

The post port is an object of type `sc_port` instantiated with the corresponding post interface. The post port is templated with the address and data types in the same way as the post interface.

```
template<typename DT, typename AT > class scml_post_port;
```

4.6.1 Types

The following type definitions are provided to support generic programming:

```
public:
  typedef AT address_type;
  typedef DT data_type;
  typedef scml_transaction_request<data_type, address_type> transaction_request_type;
  typedef transaction_request_type * transaction_request_pointer_type;
  typedef transaction_request_type & transaction_request_reference_type;
  typedef PVReq<data_type, address_type> pv_request_type;
  typedef PVResp<data_type> pv_response_type;
  typedef scml_array_notify_if::array_base_pointer_type array_base_pointer_type;
  typedef array_base_pointer_type data_manager_pointer_type;
  typedef scml_array_base::size_type size_type;
  typedef scml_array<data_type> scml_array_type;
```

4.6.2 Constructors

The following constructor is available:

```
explicit scml_post_port(const char * name,
                        scml_endian endian = scml_little_endian););
```

where:

name	Specifies the name of the port.
endian	Specifies the endianness of the port. The default is `scml_little_endian`.

This constructor is explicit.

4.6.3 Convenience Functions

```
virtual bool post(scml_transaction_request<DT, AT> & tra);
```

Convenience function with the post signature.
The return value for this function is always `true` because this call can never fail.

```
virtual bool post_read(transaction_request_type & tra,
                       scml_array_type & dataStore,
                       size_type startIndex,
                       address_type address,
                       unsigned int dataSize = 8,
                       unsigned int burstCount = 1);
virtual bool post_write(transaction_request_type & tra,
                        scml_array_type & dataStore,
                        size_type startIndex,
                        address_type address,
                        unsigned int dataSize = 8,
                        unsigned int burstCount = 1);
```

Convenience functions to post a read transaction and a write transaction, respectively.
tra specifies the transaction to be posted.
dataStore specifies the array where the data is to be read from or written to.
startIndex specifies the first index in this array.
address specifies the address.
dataSize specifies the access size in bits. The default is `8`.

burstCount specifies the number of beats. The default is `1`.
The return value for these functions is always `true` because these calls can never fail.

```
virtual pv_response_type transport(const pv_request_type & req);
```

A PV transport convenience function.

4.6.4 Properties

The following function is available to set properties:

```
scml_endian get_endianness() const;
```

Returns the endianness of the port.

4.6.5 Debug Accesses

If the `transport()` function is called with a PV request whose mode is set to `pvDebug`, a debug access is done. If no debug access can be done, an error response is returned.

The `post()` function must never be called with a PV request whose mode is set to `pvDebug`. If `post()` is called for a debug access, an error is printed and the simulation is aborted.

```
bool debug_read(DT& data, address_type address) const;
bool debug_write(const DT& data, address_type address) const;
```

Convenience function for debug accesses. If the debug access cannot be done, `false` is returned, otherwise `true` is returned.

4.6.6 Example

```
// Allocate scml_array's for read/write
unsigned int r_raw_storage[100];
scml_array<unsigned int> r_array(r_raw_storage, 100);
unsigned int w_raw_storage[100];
scml_array<unsigned int> w_array(w_raw_storage, 100);

scml_transaction_request<unsigned int, unsigned int> trans;

//
// write
//

// post transaction
trans.setDataManager(w_array);
trans.setType(pvWrite);
trans.setAddress(0x0);
trans.setDataSize(32);
trans.setBurstCount(1);
trans.setOffset(0);
trans.setStartIndex(0);

p.post(trans);

// set up write data (index 0)
w_array.claim_space(0, 0);
w_array[0] = 0x01020304;
w_array.nb_release_data(0, 1);

// wait until transaction finished
wait(trans.end_event());

//
// read
//

// post transaction
trans.setDataManager(r_array);
trans.setType(pvRead);
trans.setAddress(0x0);
trans.setDataSize(32);
trans.setBurstCount(1);
trans.setOffset(0);
trans.setStartIndex(0);

p.post(trans);

// wait until transaction finished
wait(trans.end_event());

// blocking wait for read data (index 0)
r_array.claim_data(0, 1);
assert(r_array[0] == 0x01020304);
r_array.nb_release_space(0, 1);
```

4.7 scml_transaction_request

An object of type `scml_transaction_request` is used to describe a request for communication. It is compatible with the PV communication mechanism, since an object of type `scml_transaction_request` is a PV request object by public inheritance. For more information, see "PVReq Class" on page 72.

4.7.1 Types

The `scml_transaction_request` class is templated with the address and data types in the same way as the PV request.

```
template<typename DT = unsigned int, typename AT = unsigned int>
class scml_transaction_request;
```

The following type definitions are available to support generic programming:

```
typedef DT data_type;
typedef AT address_type;
typedef scml_array_base data_manager_type;
typedef data_manager_type * data_manager_pointer_type;
typedef data_manager_type & data_manager_reference_type;
typedef scml_array_base::size_type size_type;
```

Note The default data type for `scml_transaction_request` is `unsigned int` for data as well as for address. This is fully compatible with the default 32-bit PV interface which is for example implemented by the memory objects.

Since an object of type `scml_transaction_request` is also a PV request by public inheritance, and only PV connections with the same types in the template arguments are possible, it is very important that the same types are used.

4.7.2 Enumeration Types

The following enumeration type is available:

```
enum scml_endian { scml_little_endian, scml_big_endian };
```

The `scml_endian` enumeration type is used to describe the endianness. The default endianness is little endian.

4.7.3 Constructors

A default constructor is provided. All attributes can be accessed using the appropriate getter and setter functions.

```
scml_transaction_request();
```

4.7.4 Attribute-Access Functions

Besides all the attributes in the PV request structure (which is publicly inherited), the following attribute-access functions are available:

```
int getPriority() const;
void setPriority(const int);
```

Getter and setter for the priority attribute. The default priority is `0`.

```
data_manager_pointer_type getDataManager() const;
void setDataManager(data_manager_pointer_type);
void setDataManager(data_manager_reference_type);
```

Getter and setter for the data manager. The data manager is an object of type `scml_array` to handle the access control for the data.

```
const sc_event & end_event() const;
void notifyEndEvent();
```

The end event is notified in the next delta cycle after the transaction initiated by this request structure has ended. The `end_event()` function returns a reference to this event. The `notifyEndEvent()` function can be used by transactor developers to signal the end of the transaction. The event notification happens in the next delta cycle.

```
scml_endian getEndianness() const;
void setEndianness(scml_endian a);
```

The transaction request object carries an endianness attribute. The `scml_endian` enumeration type is used to describe the endianness.

```
size_type getStartIndex() const;
void setStartIndex(size_type a);
```

Getter and setter for the start index. The start index is the index position of the first element in the array to be used in this transaction. It is typically used as the index parameter of a claim function provided by an object of type `scml_array`.

```
virtual void setBurstCount(unsigned int burstCount);
```

Sets the burst count in the inherited PV request structure and resets the acquired burst count progress indicator to zero.

4.7.5 Progress-Indicator Functions

The following progress-indicator functions are available. These functions are typically used by transactor developers. They control the progress of the transaction and are most useful when the transaction request needs to be broken up into several smaller bus accesses or transfers.

```
address_type getAcquiredAddress() const;
```

Returns the updated address. It is equal to the base address (which is obtained by the `getAddress()` function) incremented with the acquired burst count multiplied by the data size in bytes.

```
size_type getAcquiredBurstCountTodo() const;
```

Returns the remainder of the burst that needs to be done.

```
size_type acquireBurstCount(size_type beats);
```

Needs to be called to indicate that the transaction is making progress. The `size_type` *`beats`* argument specifies the number of beats by which the transactor can advance the transaction. The return value is the actual number of burst beats the transaction can advance. The return value is bounded from above by the `size_type` *`beats`* argument and by the remainder of the burst that needs to be done (as obtained by the `getAcquiredBurstCountTodo()` function).

```
size_type resetAcquireBurstCount(size_type beats);
```

Sets the acquired burst beat counter to a specific value, which is specified as the `size_type` *`beats`* argument. The return value of this function is the new value of the burst beat counter.

4.7.6 Example

See “Example” on page 56.

Chapter 5
TLM2 Adapters

The TLM2 adapters allow a *TLM2 module* (which is a module with TLM2 sockets) to use `scml_memory` and `scml_router` objects. For more information on TLM2, see the *OSCI TLM-2.0 User Manual*.

SCML does not yet support the TLM2 data structures and TLM2 APIs. The adapters will either:

- convert the TLM2 data structures and API calls into the PV data structures and API calls that are supported by the `scml_memory` and `scml_router` objects,
- or convert PV data structures forwarded by an `scml_router` to TLM2 transactions that can be forwarded over a TLM2 initiator socket.

Typical use cases for the adapters are:

- to allow `scml_memory` and `scml_router` objects to bind (indirectly) to a TLM2 target socket,
- or to allow a TLM2 initiator socket to be the destination for a mapped memory region of an `scml_router` object.

5.1 Overview

The following table summarizes the TLM2 adapters.

Table 5-1 TLM2 Adapters

Modeling Object	Summary
`scml_tlm2_target_adaptor`	Converts incoming TLM2 transactions (debug and regular) on its target socket into PV requests and forward them to its output port.
`scml_tlm2_initiator_adaptor`	Converts incoming PV requests into TLM2 transactions and forward them to its initiator socket.

5.2 scml_tlm2_target_adaptor

`scml_tlm2_target_adaptor` converts incoming TLM2 transactions (debug and regular) on its target socket (`pIn`) into PV requests and forwards them to its output port (`pOut`). If the TLM2 transaction cannot be converted to one PV request (for example, byte enables, unaligned access, and so on), the transaction will be unrolled into single-byte PV requests.

The target socket of the adapter can be bound to a standard TLM2 target socket `tlm::tlm_target_socket<BUSWIDTH>`. The output port of the adapter can be bound to an `scml_memory`, `scml_router`, or `scml_post_port` object.

5.2.1 Types

The `scml_tlm2_target_adaptor` class is templated with the `BUSWIDTH` of its target socket.

```
template <unsigned int BUSWIDTH> class scml_tlm2_target_adaptor;
typedef tlm::tlm_target_socket<BUSWIDTH> socket_type;
```

5.2.2 Enumeration Types

The following enumeration type is available:

```
enum scml_endian { scml_little_endian, scml_big_endian };
```

The `scml_endian` enumeration type is used to describe the endianness.

5.2.3 Public Data Members

```
socket_type pIn;
sc_core::sc_port<scml_mapable_if> pOut;
```

The module's two ports are publicly accessible.

5.2.4 Properties

The following functions are available to set properties:

```
void set_endianness(scml_endian e);
scml_endian get_endianness() const;
```

Set and get the endianness. The default is little endian.

If the configured endianness is different from the host endianness, the adapter will perform "address swizzling," as described in "Endianness" of the *OSCI TLM-2.0 User Manual*.

In most cases, the adapter should be configured with the same endianness as the `scml_memory` or `scml_router` object that is bound to the output port.

The endianness should be configured at construction time and should not change after end of elaboration.

5.2.5 Example

```
class ExampleModule : public sc_module {
  typedef tlm::tlm_target_socket<32> socket_type;
  socket_type s;
  scml_tlm2_target_adaptor<32> adaptor;
  scml_memory<unsigned int> mem;

  ExampleModule(sc_module_name name) :
    s("s"),
    adaptor("adaptor"),
    mem("mem", scml_memsize(0x100))
  {
    // bind the scml_tlm2_target_adaptor to the TLM2 target socket
    s(adaptor.pIn);
    // bind the scml_memory to the scml_tlm2_target_adaptor
    adaptor.pOut(mem);
  }

  <...>
};
```

5.3 scml_tlm2_initiator_adaptor

`scml_tlm2_initiator_adaptor` converts incoming PV requests into TLM2 transactions and forwards them to its initiator socket (`pOut`).

`scml_tlm2_initiator_adaptor` can be used as a destination for a mapped memory region of an `scml_router` object. The initiator socket can be bound to a TLM2 initiator socket.

5.3.1 Types

The `scml_tlm2_initiator_adaptor` class is templated with the `BUSWIDTH` of its initiator socket.

```
template <unsigned int BUSWIDTH> class scml_tlm2_initiator_adaptor;
typedef tlm::tlm_initiator_socket<BUSWIDTH> socket_type;
```

Note `BUSWIDTH` should be larger than the data size of the incoming PV accesses. If this is not the case, the data will be truncated when the PV request is converted to a TLM2 transaction.

5.3.2 Enumeration Types

The following enumeration types are available:

```
enum Mode { MODE_LT, MODE_AT };
```

The `Mode` enumeration type is used to specify whether the blocking or nonblocking transport call will be used by the adaptor.

```
enum scml_endian { scml_little_endian, scml_big_endian };
```

The `scml_endian` enumeration type is used to describe the endianness.

5.3.3 Public Data Members

```
socket_type pOut;
```

The module's output port is publicly accessible.

5.3.4 Properties

The following functions are available to set properties:

```
void set_endianness(scml_endian e);
scml_endian get_endianness() const;
```

Set and get the endianness. The default is little endian.

If the configured endianness is different from the host endianness, the adapter will perform "address swizzling," as described in "Endianness" of the *OSCI TLM-2.0 User Manual*.

The endianness should be configured at construction time and should not change after end of elaboration.

```
bool set_at_mode();
bool set_lt_mode();
bool set_mode(Mode mode);
bool is_at_mode() const;
bool is_lt_mode() const;
Mode get_mode() const;
```

Set and get the mode:

- In Loosely-Timed (LT) mode, the adaptor uses `b_transport`. This is the default mode.
- In Approximately-Timed (AT) mode, the adaptor uses `nb_transport`.

The mode should be configured at construction time and should not change after end of elaboration. For detailed information about these modes, see “Coding Styles” in the *OSCI TLM-2.0 User Manual*.

5.3.5 Example

```
class ExampleModule : public sc_module {
  typedef tlm::tlm_initiator_socket<32> socket_type;
  PVTarget_port<unsigned int> p;
  scml_router<unsigned int> router;
  scml_tlm2_initiator_adaptor<32> adaptor;
  socket_type s;

  ExampleModule(sc_module_name name) :
    p("p"),
    router("router", scml_memsize(0x100)),
    adaptor("adaptor"),
    s("s")
  {
    // bind the scml_router to the PV target socket
    p(router);
    // bind the scml_tlm2_initiator_adaptor to the TLM2 initiator socket
    adaptor.pOut(s);
    // map all access to the adaptor
    router.map(0x0, 4 * 0x100, adaptor, 0x0);
  }

  <...>
};
```

Chapter 6
Modeling Utilities

This chapter describes modeling utilities.

- Overview
- scml_property
- scml_property_registry
- scml_property_server_if
- scml_simple_property_server

6.1 Overview

The following table summarizes the modeling utilities.

Table 6-1 Modeling Utilities

Modeling Utility	Summary
scml_property	Is a property object which can hold an object of type int, bool, double, or string.
scml_property_registry	Is a class that manages the properties of a system and provides mechanisms to load the property values.
scml_property_server_if	Is a class that defines the interface a property server should implement.
scml_simple_property_server	Is a class that defines an example property server that implements the scml_property_server_if interface.

6.2 scml_property

Properties are objects that represent a property or parameter of a systemC module. The values of these properties are read from an XML file when the module is constructed (before the constructor's body code is executed).

Since these properties are read from an XML file at run time, there is no need to recompile the simulation when the values of these properties change. As such, you can quickly run multiple simulations with different parameters.

6.2.1 Kinds

The following scml_property classes are available:

```
scml_property<int>
scml_property<unsigned int>
scml_property<double>
```

```
scml_property<bool>
scml_property<std::string>
```

6.2.2 Types

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;
```

6.2.3 Constructors

The following constructors are available:

```
scml_property(const ::std::string& name);
scml_property(const ::std::string& name, T defaultValue);
```

where:

name	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.
defaultValue	Specifies the default value of the property. This default value is only used if the property is not found in the XML file.

6.2.4 Operators

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;
```

6.2.5 Access Functions

`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`.

6.2.6 Example

```
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.

6.3 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

`scml_property_registry` offers an API to read the values of these parameters.

6.3.1 Enumeration Types

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.

6.3.2 Access Functions

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.

Note 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 67.

6.3.3 Example

```
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;
};
```

6.4 scml_property_server_if

This class defines the interface a property server should implement.

6.4.1 Interface Functions

```
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.

6.4.2 Example

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;
  }
```

6.5 scml_simple_property_server

This class defines an example property server that implements the `scml_property_server_if` interface.

6.5.1 Interface Functions

```
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.

6.5.2 Property File Syntax

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:

name	Specifies the hierarchical name of the property.
value	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
```

Chapter 7
Using the PV Abstraction Level

This chapter describes the Programmer's View (PV) abstraction level and the PV API functions in detail. It includes coding guidelines, description of the modeling of the different elements that make up a system, and examples.

7.1 About the PV Abstraction Level

The purpose of modeling a system at the PV abstraction level is to enable the designer to create a model of a system for use by embedded software developers. Hence, simulation speed is a major factor.

A system modeled at the PV abstraction level contains the detail necessary to run the software on the processing elements. The PV abstraction level supports the following:

- Bit-true behavior
- Synchronization (interrupts, polling mechanisms)
- Modeling of all memory-mapped registers in the system
- Basic communication and bus architecture

The PV abstraction level is characterized by the following:

- Synchronous protocol - blocking function.
- No concept of time exists; that is, no delay is modeled.
- Target modules are untimed.
- One `sc_thread` per true initiator exists.
- Enough synchronization is available to model correct functionality.
- No explicit generic channel is visible; that is, it is hidden.
- Ports are bound to `sc_export`.
- The communication mechanism has no concept of timing.
- The transport mechanism is blocking.

The PV abstraction level defines precisely what the transport mechanism is but it does not put a lot of hard requirements on the coding style. However, since simulation speed is a very important factor, you must achieve a good trade-off between the level of the functionality that gets modeled, and abstraction of behavior to achieve high simulation speeds. This chapter provides suggestions to determine this trade-off.

The PV abstraction level is not suited to provide accurate timing information. For this, a more accurate bus model is required. It can, however, be used to determine optimal cache and memory sizes.

7.2 PV Transport API Syntax

The PV abstraction level uses a single “transport” function to transfer data between an initiator and target. The transport function transfers information stored in a `PVReq` object to the target. The target returns information by means of a `PVResp` object.

The initiator calls the transport function through a `PVInitiator_port`.

The target implements a transport function, and registers it to a `PVTarget_port`. The port will then call the target’s implementation.

In between the initiator and target, a PV node (or several) can be present. This node enables the designer to create systems with multiple masters and slaves. The node handles address decoding.

You can also use the PV transport API to create point-to-point connections. In that case, the initiator’s port is connected directly to the target’s port.

Note The PV transport occurs in zero time.

7.2.1 Transport Call

The PV transport call is defined as follows:

```
PVResp<DT> transport(const PVReq<DT, AT>&)
```

Note This definition is compliant with the proposal of the OSCI TLM WG.

7.2.2 PVReq Class

To use the `PVReq` class, include the `PV.h` header file.

The `PVReq` class template is declared as follows:

```
PVReq<DT,AT>
```

where:

DT	Is the data type. You can specify any data type or class.
AT	Is the address type. You can specify any data type or class that can be cast to `unsigned int`.

Note All initiators, targets, and nodes must use the same address and data type throughout the system.

The following sections describe the information and related member functions which an object of the `PVReq` class contains.

7.2.2.1 Address

Specifies the address to which the transport will be done.

Note	The address member of the `PVReq` class can be changed by a PV node or a PV target. An initiator module should not expect that the value of the address is the same after calling the transport.

7.2.2.1.1 Type

Taken from `PVReq` template parameter *`AT`*.

7.2.2.1.2 Related Member Functions

- `setAddress(ATaddress)`

  Value returned: None.

  Usage: This function is typically called by the initiator to set the address associated with the transaction.

- *`AT`* `getAddress() const`

  Value returned: Address.

  Usage: This function is typically called by the slave to retrieve the address associated with the transaction.

7.2.2.2 Write Data

One or more data items will be transported from initiator to target. The `PVReq` object contains a pointer to the data. The data is owned by the initiator. The `PVReq` object only transports the pointer to the slave, not the actual data. This approach boosts simulation speed. In the case of a burst transfer, the data will be an array of items of data type *`DT`*. The pointer should point to the first item in the array.

7.2.2.2.1 Type

Taken from `PVReq` template parameter *`DT`*.

7.2.2.2.2 Related Member Functions

- `setWriteDataSource(`*`DT* arg_writeData`*`)`

  Value returned: None.

  Usage: This function is typically called by the initiator to set the pointer to the write data.

- *`DT`* `getWriteData(unsigned int` *`arg_index`* `= 0) const`

  Value returned: Data of type *`DT`*.

  Usage: This function is typically called by the target to retrieve the write data from the initiator. One optional argument can be specified to get the *n*th element in case of a burst transfer. Specifying an *`arg_index`* which is larger than the actual burst size can cause a core dump, as no checking is done to maintain simulation speed.

7.2.2.3 Type

Specifies whether this access is a read or a write access.

7.2.2.3.1 Type

`PVType`

7.2.2.3.2 Possible Values

`pvRead`, `pvWrite`

7.2.2.3.3 Related Member Functions

- `setType(PVType` *`arg_type`*`)`

  Value returned: None.

  Usage: This function is typically called by the initiator to set the transfer type.

- `PVType getType() const`

  Value returned: Transfer type of type `PVType`.

  Usage: This function is typically called by the target to retrieve the transfer type.

7.2.2.4 Data Size

Specifies the width of the transferred data in bits, or the size of each data item in bits in case of burst transfers.

7.2.2.4.1 Type

`unsigned int`

7.2.2.4.2 Related Member Functions

- `setDataSize(unsigned int` *`arg_dataSize`*`)`

  Value returned: None.

  Usage: This function is typically called by the initiator to set the data size of the transfer.

- `unsigned int getDataSize() const`

  Value returned: Data size of the transfer.

  Usage: This function is typically called by the target to retrieve the data size of the transfer.

7.2.2.5 Offset

Specifies the offset of the first data item in bits. This can be useful when transporting items which have a smaller width than the width available in data type *`DT`*.

7.2.2.5.1 Type

`unsigned int`

7.2.2.5.2 Related Member Functions

- `setOffset(unsigned int` *`arg_offset`*`)`

  Value returned: None.

  Usage: This function is typically called by the initiator to set the offset of the transfer.

- `unsigned int getOffset() const`

  Value returned: `unsigned int`

  Usage: This function is typically called by the target to get the offset of the transfer.

7.2.2.6 Burst Count

Specifies the number of items of type *DT* that will be transported. For a single access, it should be set to `1`. If it is set to `0`, no data items should be accessed.

7.2.2.6.1 Type

`unsigned int`

7.2.2.6.2 Related Member Functions

- `setBurstCount(unsigned int` *`arg_burstCount`*`)`

  Value returned: None.

  Usage: This function is typically called by the initiator to set the burst count of the transfer.

- `unsigned int getBurstCount() const`

  Value returned: `unsigned int`

  Usage: This function is typically called by the target to retrieve the burst count of the transfer.

7.2.2.7 Burst Type

Specifies the type of burst:

- `pvIncr` indicates that the address is incremented by *`dataSize`*`/8` each transfer.
- `pvWrap` is the same as `pvIncr` except a wrap happens at the address boundary aligned with *`burstCount`*` * `*`dataSize`*`/8`.
- `pvStream` indicates that the address stays constant each transfer.
- `pvUnknown` indicates that the address sequence is not specified.
- `pvXor`: Suppose that:
  - *`BASE`* is the lowest byte address in the burst (it must be aligned with the total burst size).
  - *`FIRST_OFFSET`* is the byte offset (from *`BASE`*) of the first transfer in the burst, starting at `0`.
  - *`WORD_SHIFT`* is the `log2` of *`dataSize`*`/8`.

  Then the current address of the transfer is:
  `BASE`` | (`*`FIRST_OFFSET`*` ^ (`*`CURRENT_COUNT`*` << `*`WORD_SHIFT`*`)).`

The `pvWrap` and `pvXor` burst types can only be used with a power-of-two *`burstCount`*.

7.2.2.7.1 Type

`PVBurstType`

7.2.2.7.2 Possible Values

`pvIncr, pvWrap, pvStream, pvUnknown, pvXor`

7.2.2.7.3 Related Member Functions

- `void setBurstType(PVBurstType);`

  Value returned: None.

  Usage: This function is typically called by the initiator to set the burst type of the transfer.

- `PVBurstType getBurstType() const;`

  Value returned: `PVBurstType`

Usage: This function is typically called by the target to retrieve the burst type of the transfer.

7.2.2.8 Mode

Specifies whether this access is a regular or a debug access.

7.2.2.8.1 Type

`PVMode`

7.2.2.8.2 Possible Values

`pvDefault,pvDebug`

7.2.2.8.3 Related Member Functions

- `void setMode(PVMode);`

  Value returned: None.

  Usage: This function is typically called by the initiator to set the mode of the transfer.

- `PVMode getMode() const;`

  Value returned: `PVMode`

  Usage: This function is typically called by the target to retrieve the mode of the transfer.

7.2.2.9 Thread ID

Specifies the ID of the thread in the initiator performing this transaction.

7.2.2.9.1 Type

`unsigned int`

7.2.2.9.2 Related Member Functions

- `void setThreadID(unsigned int);`

  Value returned: None.

  Usage: This function is typically called by the initiator to set the thread ID of the transfer.

- `unsigned int getThreadID() const;`

  Value returned: `unsigned int`

  Usage: This function is typically called by the target to retrieve the thread ID of the transfer.

7.2.2.10 Read Data Destination

Specifies the memory location to which the target should write its data in case of a read transaction. The memory is owned by the master, and as such, it must be allocated by the master before the transaction is initiated. The `PVReq` object only carries the pointer to this memory, not the data itself.

The target will use this information through the `PVResp` object by calling `setReadData`, which is a `PVResp` member function. For more information on the `setReadData` member function, see "Read Data" on page 78.

7.2.2.10.1 Type

Pointer to *DT*, where *DT* is taken from the data type template of the `PVReq` object.

7.2.2.10.2 Related Member Functions

- `setReadDataDestination(`*`DT* arg_readData`*`)`

  Value returned: None.

  Usage: This function is typically called by the initiator to set the destination memory address for read data.

7.2.2.11 Obtaining the Response

When creating the `PVReq` object, a `PVResp` object is also created. A reference to this `PVResp` object is stored within the `PVReq` object. The target must use the `PVResp` object to return status information or read data. For more information on the use of the `PVResp` object, see "PVResp Class" on page 78. You should only use the `PVResp` objects that are obtained through the `PVReq` object, as some critical initializations of the `PVResp` object can only be performed by a `PVReq` object.

7.2.2.11.1 Type

`PVResp<DT>`

7.2.2.11.2 Related Member Functions

- `obtainResp()`

  Value returned: `PVResp` object.

  Usage: This function must be called by the target to obtain a `PVResp` object.

7.2.2.12 Custom Data

When you want to transport information which is not available in the address or data type, it is possible to pass along a pointer to a user-defined child of the `PVCustomReq` class. In this user-defined class, extra custom data fields can be created.

When using the `PVCustomReq` class at the target, a cast from the `PVCustomReq` class is required.

7.2.2.12.1 Type

Any class inherited from the `PVCustomReq` class.

7.2.2.12.2 Related Member Functions

- `void setCustomData(PVCustomReq& `*`arg_customData`*`)`

  Value returned: `void`

  Usage: This function is typically called by the initiator to attach custom data to the `PVReq` object.

- `PVCustomReq& getCustomData() const`

  Value returned: `const PVCustomReq&`

  Usage: This function is typically used by the target to read the data attached to the `PVReq` by the initiator. A cast will be needed to the appropriate user-defined child of `PVCustomReq` to access the members.

7.2.2.13 Protocol Data

When you want to transport protocol-specific information which is not available in the address or data type, it is possible to pass along a pointer to a user-defined child of the `PVProtocolReq` class. In this user-defined class, extra data fields can be created.

When using the `PVProtocolReq` class at the target, a cast from the `PVProtocolReq` class is required.

7.2.2.13.1 Type

Any class inherited from the `PVProtocolReq` class.

7.2.2.13.2 Related Member Functions

- `void setProtocolData(PVProtocolReq&` *`arg_protocolData`*`)`

  Value returned: void

  Usage: This function is typically called by the initiator to attach protocol data to the `PVReq` object.

- `PVProtocolReq& getProtocolData() const`

  Value returned: `const PVProtocolReq&`

  Usage: This function is typically used by the target to read the data attached to the `PVReq` by the initiator. A cast will be needed to the appropriate user-defined child of `PVProtocolReq` to access the members.

7.2.3 PVResp Class

To use the `PVResp` class, include the `PV.h` header file.

The `PVResp` class template is declared as follows:

`PVResp<DT>`

where *`DT`* is the data type. You must specify the data type or class specified in the `PVReq` class.

Note

- All initiators, targets, and nodes must use the same data type throughout the system.
- You must always obtain a `PVResp` instance by calling the `obtainResp` member function of a `PVReq` object.

The following sections describe the information and related member functions which an object of the `PVResp` class contains.

7.2.3.1 Read Data

One or more data items will be transported from initiator to target. The `PVResp` contains the pointer to the read data destination in memory. This pointer should be set by the initiator by calling the function `setReadDataDestination` of the `PVReq` object. More than one data item can be transferred with one transport.

7.2.3.1.1 Type

`DT`, as taken from the data type template of the `PVResp` object.

7.2.3.1.2 Related Member Functions

- `setReadData(`*`DT arg_data`*`, unsigned int` *`arg_index`* `= 0)`

  Value returned: None.

  Usage: This function is typically called by the target to set the read data for the initiator. The first argument is the data, a second optional argument *`arg_index`* can be specified to set the *n*th element in case of a burst transfer.

  Specifying an *`arg_index`* that is larger than the actual burst size can cause a core dump, as no checking is done to maintain simulation speed.

7.2.3.2 Response

Can be used by the target to flag signal status information to the initiator.

Note A `pvError` response will be generated by the default slave of a node.

7.2.3.2.1 Type

`PVResponse`

7.2.3.2.2 Possible Values

`pvOk, pvError`

7.2.3.2.3 Related Member Functions

- `setResponse(PVResponse` *`arg_response`*`)`

  Value returned: None.

  Usage: This function is typically called by the target to set the response.

- `getResponse()`

  Value returned: `PVResponse`

  Usage: This function is typically called by the initiator to retrieve the status information set by the target, after completion of the transport.

7.2.3.3 Latency

Can be used by the target to communicate to the initiator how much time is consumed in the slave. This information is currently ignored by the bus.

The latency has to be specified in number of clock cycles.

7.2.3.3.1 Type

`unsigned int`

7.2.3.3.2 Related Member Functions

- `setLatency(unsigned int` *`arg_latency`*`)`

  Value returned: `void`

  Usage: This function is typically called by the target to set the latency for the transaction.

- `unsigned int getLatency() const`

  Value returned: `unsigned int`

  Usage: This function is typically called by the initiator to retrieve the latency information set by the target, after completion of the transport.

7.2.3.4 Custom Data

If you want to transport information which is not available in the address or data type, you can pass along a pointer to a user-defined child of the `PVCustomResp` class. In this user-defined class, extra custom data fields can be created.

7.2.3.4.1 Type

Any class inherited from the `PVCustomResp` class.

7.2.3.4.2 Related Member Functions

- `void setCustomData(PVCustomResp& arg_customData)`

  Value returned: None.

  Usage: This function is typically called by the target to attach custom data to the `PVResp` object.

- `PVCustomResp& getCustomData() const;`

  Value returned: `const PVCustomResp&`

  Usage: This function is typically used by the target to read the data attached to the `PVResp` by the target. A cast is needed to the appropriate user-defined child of `PVCustomResp` to access the members.

7.2.3.5 Protocol Data

When you want to transport protocol-specific information which is not available in the address or data type, it is possible to pass along a pointer to a user-defined child of the `PVProtocolResp` class. In this user-defined class, extra data fields can be created.

7.2.3.5.1 Type

Any class inherited from the `PVProtocolResp` class.

7.2.3.5.2 Related Member Functions

- `void setProtocolData(PVProtocolResp& arg_protocolData)`

  Value returned: None.

  Usage: This function is typically called by the target to attach protocol data to the `PVResp` object.

- `PVProtocolResp& getProtocolData() const;`

  Value returned: `const PVProtocolResp&`

  Usage: This function is typically used by the initiator to read the data attached to the `PVResp` by the target. A cast is needed to the appropriate user-defined child of `PVProtocolResp` to access the members.

7.3 Coding a PV Initiator

Two major subclasses of initiator blocks exist:

1. The first class is an initiator block which is free running. This means that all transfers are initiated by the initiator without the block having to be accessed by another peripheral first. A typical example of this is a core model.
2. The second class is an initiator block that will only initiate transfers when it has a target port which is accessed by another initiator. A typical example of this could be a DMA controller.

Both classes will require at least one instance of the `PVInitiator_port` class. This class is defined in the header file `PV.h`. Note that the class `PVInitiator_port` has template arguments for address type and data type.

```
template <DT, AT>
     class PVInitiator_port: public sc_port<PV_if<DT, AT>>
```

The transport function will be called through that `PVInitiator_port`. As an argument to the transport function a `PVReq` object needs to be passed. The address and data type template arguments of the `PVReq` object must match the address and data type template arguments of the `PVInitiator_port` instance. The easiest way to ensure consistency is to have type definitions with the specific *`DT`* and *`AT`* in one place. For example, put the following in a header file and include it where needed:

```
typedef PVInitiator_port<unsigned long, unsigned int> MyPVInitiatorPort;
typedef PVTarget_port<unsigned long, unsigned int> MyPVTargetPort;
```

Then, you can use the types `MyPVInitiatorPort` and `MyPVTargetPort` without worrying about the types for *`DT`* and *`AT`*. Changes to *`DT`* and *`AT`* now only have to be made in one place.

The members of this `PVReq` object must be initialized before calling the transport function.

The transport function will return a `PVResp` object. The data type template argument of the `PVResp` object must match the data type template argument of the `PVInitator_port` instance. From that `PVResp` object, the response data, read data, or both can be retrieved.

The basic steps to set up a transport are the same, independent from type of initiator you are implementing.

The following code shows an example of how to model a read transaction.

```
#include "PV.h"

...
//instantiate necessary classes
PVInitiatorPort<unsigned int, unsigned int> thePort;
PVReq<unsigned int, unsigned int> theReq;
PVResp<unsigned int> theResp;
//allocate memory to store read data
unsigned int readData[4] = {1, 2, 3, 4};

...
//initialize PVReq members.
theReq.setAddress(0x1000);
theReq.setReadDataDestination(&readData[0]);
theReq.setOffset(0);
theReq.setBurstCount(4);
theReq.setDataSize(32);
theReq.setType(pvRead);

//do the transport
theResp = thePort.transport(theReq);
...
//check the response and read data
if (theResp.getResponse() == pvOk) {
  cout << "received data : " << endl;
  for (int i = 0; i<=4 ; i++){
    cout << " " << readData[i] << endl;
  }
}
else{
  cout << "received error response !" << endl;
}
```

7.4 Coding a PV Target

When writing a PV target, the target must contain at least one `PVTarget_port`.

The `PVTarget_port` has the following template arguments:

```
template <DT, AT>
      class PVTarget_port : public sc_export<PV_if<AT, DT> >
```

The PV target must also contain an implementation of the transport function. The name of the function is not important, but the signature must match the signature of the transport call definition. (See "Transport Call" on page 72.)

The `PVResp` object returned by the transport function must be initialized using the `obtainResp()` member function of the `PVReq` function argument. For more information, see "Obtaining the Response" on page 77.

The `PVTarget_port` template arguments and the template arguments of the `PVReq` and `PVResp` arguments of the transport function must match.

The transport function must be registered, or basically bound to the target port, by calling the `REGISTER_PVSLAVE` macro. This macro call takes two arguments, a reference to the target port, and the name of the transport implementation function. Only one function per port can be registered. The following example shows the implementation of a PV target peripheral.

```
#include <systemc.h>
#include "PV/PV.h"

class PVSlave : public sc_module{

public:

  unsigned int m_readData;

  PVTarget_port<unsigned int, unsigned int> p_slave;
  PVSlave(sc_module_name name);
  SC_HAS_PROCESS(PVSlave);
  PVResp<unsigned int>
  theTransportfunc(const PVReq<unsigned int, unsigned int> &
                   arg_Req);
};

PVSlave::PVSlave(sc_module_name name) : sc_module(name), p_slave(){
  cout << sc_time_stamp() << " Slave constructor called " << endl;
  REGISTER_PVSLAVE(p_slave, theTransportfunc);
};

PVResp<unsigned int>
  PVSlave::theTransportfunc(
    const localPVReq<unsigned int, unsigned int>& arg_Req){
  cout << sc_time_stamp() << " "
       << name()
       << " transport function is called with address "
       << hex << (unsigned int) arg_Req.getAddress() << dec <<
       << endl;

  PVResp<unsigned int> theResp = arg_Req.obtainResp();
  if (arg_Req.getType() == pvRead){
    if (arg_Req.getBurstCount() <= 1){
      theResp.setReadData(1);
    }
    else{
      for (int i = 0; i<= arg_Req.getBurstCount(); i++){
        theResp.setReadData(i*2, i);
      }
    }
  }
  else {
    if (arg_Req.getBurstCount() <= 1){
      cout << "received " <<hex<< arg_Req.getWriteData() <<dec << endl;
    }
    else{
      cout << "received burst write with data : " << endl;
      for (int i = 0; i<= arg_Req.getBurstCount(); i++){
        cout << "    " << hex << arg_Req.getWriteData(i) << dec << endl;
      }
    }
  }
  theResp.setResponse(pvOk);

  return theResp;
};
```

7.5 SCML and PV

Note An initiator should not issue accesses bigger than the data width of its PV port.

Note The actual SystemC description of the peripherals does not contain any information regarding memory-map size. So all memory-map information of initiators and peripherals must be set manually in Platform Creator by selecting appropriate values for *address_width*, and by creating memory regions for the target peripherals, if this is required.

SCML only supports a subset of the PV protocol. If the PV protocol is used in combination with SCML, the following limitations have to be taken into account:

- The DT template parameter in the PVReq or PVResp structures can be one of the following types:
  - Data widths smaller than or equal to 32:

    `unsigned int`

  - Data widths greater than 32 and smaller than or equal to 64:

    `unsigned long long`

  - Data widths greater than 64 and smaller than or equal to 128:

    `sc_dt::sc_biguint<128>`

  - Data widths greater than 128 and smaller than or equal to 256:

    `sc_dt::sc_biguint<256>`

  - Data widths greater than 256 and smaller than or equal to 512:

    `sc_dt::sc_biguint<512>`

  The AT template parameter in the PVReq structure can be one of the following types:
  - Address width smaller than or equal to 32:

    `unsigned int`

  - Address width greater than 32 and smaller than or equal to 64:

    `unsigned long long`

- The address in the PVReq structure must be a byte address.
- The offset in the PVReq structure must always be 0.
- The burstCount in the PVReq structure must be 1 for single accesses and larger than 1 for burst accesses. A burstCount of 0 is illegal and may result in undefined behavior.
- SCML only supports incremental bursts. The only valid value for burstType is pvIncr.
- The data in the PVReq or PVResp structure must **not** be aligned. The data pointer stores the arithmetic value (in host endianness) of the data that will be transmitted.

  For example, to write a word of 4 bytes to address 0x0:

```
<...>
  req.setAddress(0x0);
  unsigned int data = 0x03020100;
  req.setWriteDataSource(&data);
  req.setDataSize(32);
  req.setBurstCount(1);
  req.setOffset(0);
<...>
```

To write only the two most significant bytes of the word:

```
<...>
  req.setAddress(0x2);
  unsigned int data = 0x0302;
  req.setWriteDataSource(&data);
  req.setDataSize(16);
  req.setBurstCount(1);
  req.setOffset(0);
<...>
```

- The `data size` attribute of a burst access must be equal to the data width of the destination `scml_memory`. Burst accesses with a data size that is different as the data width of the `scml_memory` may result in undefined behavior.

Index