SYNOPSYS®

Synopsys Virtualizer Product Family:

SystemC Modeling Library 2 Manual

Version 2.1.3
March 2013

Copyright Notice and Proprietary Information

Destination Control Statement

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

Disclaimer

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

Trademarks

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

http://www.synopsys.com/Company/Pages/Trademarks.aspx.

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

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

www.synopsys.com

Contents

Preface

The preface of the *SystemC Modeling Library 2 Manual* describes:

- About This Manual
- Documentation Conventions
- Terminology

About This Manual

This manual describes SystemC Modeling Library 2 (SCML2) modeling objects.

Note

- This release supports SCML2 as well as SCML1. The latter is described in the *SystemC Modeling Library Manual*.
- This version of the source code SCML kit is aligned with the SCML delivered in the Product Version G-2012.06-SP3 of Platform Architect MCO and Virtualizer.

It is assumed that you have some knowledge of SystemC.

This manual is organized as follows:

- Introduction gives an overview of the modeling objects.
- memory describes `memory` objects.
- memory_alias describes `memory_alias` objects.
- reg describes `reg` objects.
- bitfield describes `bitfield` objects.
- router describes `router` objects.
- tlm2_gp_target_adapter describes `tlm2_gp_target_adapter` objects.
- tlm2_gp_initiator_adapter describes `tlm2_gp_initiator_adapter` objects.
- memory_index_reference describes `memory_index_reference` objects.
- mappable_if describes `mappable_if` objects.
- dmi_handler describes `dmi_handler` objects.
- Callback Base Classes describes callback base classes.
- initiator_socket describes `initiator_socket` objects.
- status describes `status` objects.
- stream describes `stream` objects.
- severity describes `severity` objects.
- Pin Callback Functions describes the pin callback functions available for registering user callbacks on changes of input pins.

- Convenience Functions describes the convenience functions available in `scml2/utils.h`.

Documentation Conventions

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

Table 1-1 Documentation Conventions

Convention	Description and Examples
italic	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:

`courier`	Is used for:
	■ Code text. For example:
	`list_library_configurations`
	`myConfig`
	In this example, `myConfig` is used.
	■ System messages. For example:
	`JVM not found.`
	■ Text you must type literally. For example:
	At the prompt, type `go`.
	■ Names (of environment variables, commands, utilities, prompts, paths, macros, and so on). For example:
	The `build-options` command sets build parameters.

`courier italic`	Indicates variables. For example: *`scope`* specifies a module, a channel, or a refined port.
bold	Serves to draw your attention to the text in question. For example: `coreId = cwrSAGetCoreId(`**`"ARM7"`**`);`

Table 1-1 Documentation Conventions

Convention	Description and Examples

`[ ]`	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']'`

`{ }`	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`
	`'}'`

`|`	A vertical bar separates items in a list of choices. For example: `autoflush {on | off}`
>	A right angle bracket separates menu commands. For example: The *Library* > *Update System Library* menu command is available.

`...`	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 *`

Terminology

API	Application Programmer's Interface
DMI	Direct Memory Interface
DT	Data Type
IP	Intellectual Property

MRU	Most-Recently Used
SCML1	SystemC Modeling Library 1. SCML1 is described in the *SystemC Modeling Library Manual*.
SOC	Stands for System-On-a-Chip.
SCML2	SystemC Modeling Library 2. SCML2 is described in this manual.
TLM2	Transaction-Level Modeling 2

Chapter 1
Introduction

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

Table 1-1 SCML2 Modeling Objects

Modeling Object	Description
memory	Models memories and register files.
memory_alias	Models an alias for a memory region of another memory or memory_alias object.
reg	A register; it models a memory_alias object of size 1.
bitfield	Models an alias for a number of consecutive bits in a reg object.
router	Models a dynamic address decoder that can map a memory region to a region in another memory, router, or tlm2_gp_initiator_adapter object.
tlm2_gp_target_adapter	Allows a memory object to bind to a tlm2_target_socket.
tlm2_gp_initiator_adapter	Allows a router object to map a memory region to a region on a tlm2_initiator_socket.
dmi_handler	Is a convenience object to do the bookkeeping of DMI pointers.
initiator_socket	Is a convenience socket that first tries to do a DMI access before doing a bus access.
status	Is a simple object that holds a status value in string format.
stream	Is the front-end object of SCML2 logging library. It formats the output and sends it to the back-end logger objects for processing.
severity	Holds a severity name and a value. Lower severity values mean a higher severity level.

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

All SCML2 header files can be included by including scml2.h.

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

Chapter 2
memory

Objects of type `memory` can be used to model memories and register files.

The `memory` object has the following properties:

- Allocates the storage.
- Can be bound to TLM2 target sockets via an adapter (see "tlm2_gp_target_adapter" on page 57).
- Is a top-level object, that is, it cannot be an alias for another `memory` object.
- Can have aliases and/or registers.
- Can have callbacks to change the default memory behavior.

The `memory` object implements the `mappable_if` object, which means that it can be the destination for a mapped range of a `router` object.

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

The following sections describe:

2.1 Types

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

```
template <typename DT> class memory
```

The following types are supported:

- `unsigned char`
- `unsigned short`
- `unsigned int`
- `unsigned long long`

- sc_dt::sc_biguint<128>
- sc_dt::sc_biguint<256>
- sc_dt::sc_biguint<512>

The following type definitions are available:

```
typedef DT data_type
typedef memory_index_reference<DT> reference
typedef memory_iterator<DT> iterator
typedef memory_const_iterator<DT> const_iterator
```

2.2 Constructors

The following constructor is available:

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

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

2.3 Initialization

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

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

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

2.4 Properties

The following methods are available to set properties:

`const std::string& get_name() const`

Returns the full hierarchical name of the `memory` object.

`unsigned long long get_size() const`

Returns the size of the `memory` object in words.

`unsigned int get_width() const`

Returns the width in bytes of the underlying data type of the `memory` object.

```
void set_default_read_latency(const sc_core::sc_time& t)
const sc_core::sc_time& get_default_read_latency() const
void set_default_write_latency(const sc_core::sc_time& t)
const sc_core::sc_time& get_default_write_latency() const
```

Set/get the latency returned in the `tlm::tlm_dmi` structure of the `get_direct_mem_ptr()` call. If no callback is attached, this latency is also added to the timing annotation argument of the `b_transport()` call.

`bool is_dmi_enabled()`

Returns `true` if DMI accesses are allowed for the object, `false` otherwise. DMI is enabled by default.

```
void enable_dmi()
void disable_dmi()
```

Enables/disables DMI accesses for the object.

2.5 TLM2 API Methods

The `memory` object implements the following TLM2 methods:

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

The `memory` object does not implement the `nb_transport()` call. The target adapter to which the memory is bound should convert `nb_transport()` calls into `b_transport()` calls before forwarding them to the `memory` object.

The TLM2 transport methods trigger the callbacks registered to the `memory` object or its aliases.

2.6 Access Methods

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

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

These transport methods do - unlike the TLM2 transport methods (`b_transport()` and `transport_dbg()`) - not trigger any callbacks. They access the current content of the `memory` object. The debug methods also do not trigger any watchpoints on the `memory` object.

Next to the transport methods, the `memory` objects have some convenience put and get methods. There are four classes of put/get methods with the following properties:

- `put()/get()`
  - Access the content of the memory
  - Do not trigger callbacks
  - May trigger watchpoints
- `put_debug()/get_debug()`
  - Access the content of the memory
  - Do not trigger callbacks
  - Do not trigger watchpoints
- `put_with_triggering_callbacks()/get_with_triggering_callbacks()`
  - Trigger callbacks
  - May trigger watchpoints
- `put_debug_with_triggering_callbacks()/get_debug_with_triggering_callbacks()`
  - Trigger debug callbacks
  - Do not trigger watchpoints

For each class, there are four different types of put/get methods with the following arguments:

- TLM2 style
  - `unsigned long long` *`address`*, where *`address`* specifies the byte address
  - `(const) unsigned char*` *`data`*, where *`data`* specifies the data array

- `unsigned int` *`dataLength`*, where *`dataLength`* specifies the data length in bytes
- `const unsigned char*` *`byteEnablePtr`*, where *`byteEnablePtr`* specifies a byte enable array (it may be `0`)
- `unsigned int` *`byteEnableLength`*, where *`byteEnableLength`* specifies the byte enable length in bytes

- TLM2 style without byte enables
  - `unsigned long long` *`address`*, where *`address`* specifies the byte address
  - `(const) unsigned char*` *`data`*, where *`data`* specifies the data array
  - `unsigned int` *`dataLength`*, where *`dataLength`* specifies the data length in bytes
- Word access
  - `unsigned long long` *`index`*, where *`index`* specifies the word index
  - `(const) DT&` *`data`*, where *`data`* specifies the data
- Subword access
  - `unsigned long long` *`index`*, where *`index`* specifies the word index
  - `(const) DT&` *`data`*, where *`data`* specifies the data
  - `unsigned int` *`size`*, where *`size`* specifies the size in bytes
  - `unsigned int` *`offset`*, where *`offset`* specifies the offset in bytes

The `put_with_triggering_callbacks()` and `get_with_triggering_callbacks()` methods also take an `sc_time` argument that is passed to the callback. They return the TLM2 response status, returned by the triggered callback.

The `get()` and `get_debug()` methods for word or subword accesses return the read data instead of passing it as an argument.

The `put_debug_with_triggering_callbacks()` and `get_debug_with_triggering_callbacks()` calls must not use byte enables (*`byteEnablePtr`* must be `0`) since TLM2 does not support byte enables for debug calls.

The following put and get methods are defined on the `memory` object:

```
void put(unsigned long long address,
         const unsigned char* data,
         unsigned int dataLength,
         const unsigned char* byteEnablePtr,
         unsigned int byteEnableLength);
void put(unsigned long long address,
         const unsigned char* data,
         unsigned int dataLength);
void get(unsigned long long address,
         unsigned char* data,
         unsigned int dataLength,
         const unsigned char* byteEnablePtr,
         unsigned int byteEnableLength) const;
void get(unsigned long long address,
         unsigned char* data,
         unsigned int dataLength) const;

void put(unsigned long long index,
         const DT& data,
         unsigned int size,
         unsigned int offset);
void put(unsigned long long index,
         const DT& data);
DT get(unsigned long long index,
       unsigned int size,
       unsigned int offset) const;
DT get(unsigned long long index) const;

void put_debug(unsigned long long address,
               const unsigned char* data,
               unsigned int dataLength,
               const unsigned char* byteEnablePtr,
               unsigned int byteEnableLength);
void put_debug(unsigned long long address,
               const unsigned char* data,
               unsigned int dataLength);
void get_debug(unsigned long long address,
               unsigned char* data,
               unsigned int dataLength,
               const unsigned char* byteEnablePtr,
               unsigned int byteEnableLength) const;
void get_debug(unsigned long long address,
               unsigned char* data,
               unsigned int dataLength) const;

void put_debug(unsigned long long index,
               const DT& data,
               unsigned int size,
               unsigned int offset);
void put_debug(unsigned long long index,
               const DT& data);
DT get_debug(unsigned long long index,
             unsigned int size,
             unsigned int offset) const;
DT get_debug(unsigned long long index) const;

tlm::tlm_response_status
put_with_triggering_callbacks(unsigned long long address,
                              const unsigned char* data,
                              unsigned int dataLength,
                              const unsigned char* byteEnablePtr,
                              unsigned int byteEnableLength,
                              sc_core::sc_time& t);
tlm::tlm_response_status
put_with_triggering_callbacks(unsigned long long address,
                              const unsigned char* data,
                              unsigned int dataLength,
                              sc_core::sc_time& t);
tlm::tlm_response_status
get_with_triggering_callbacks(unsigned long long address,
                              unsigned char* data,
                              unsigned int dataLength,
                              const unsigned char* byteEnablePtr,
                              unsigned int byteEnableLength,
                              sc_core::sc_time& t);

tlm::tlm_response_status
get_with_triggering_callbacks(unsigned long long address,
                              unsigned char* data,
                              unsigned int dataLength,
                              sc_core::sc_time& t);
```

```
tlm::tlm_response_status
put_with_triggering_callbacks(unsigned long long index,
                              const DT& data,
                              unsigned int size,
                              unsigned int offset,
                              sc_core::sc_time& t);
tlm::tlm_response_status
put_with_triggering_callbacks(unsigned long long index,
                              const DT& data,
                              sc_core::sc_time& t);
tlm::tlm_response_status
get_with_triggering_callbacks(unsigned long long index,
                              DT& data,
                              unsigned int size,
                              unsigned int offset,
                              sc_core::sc_time& t);
tlm::tlm_response_status
get_with_triggering_callbacks(unsigned long long index,
                              DT& data,
                              sc_core::sc_time& t);

unsigned int
put_debug_with_triggering_callbacks(unsigned long long address,
                                    const unsigned char* data,
                                    unsigned int dataLength,
                                    const unsigned char* byteEnablePtr,
                                    unsigned int byteEnableLength);
unsigned int
put_debug_with_triggering_callbacks(unsigned long long address,
                                    const unsigned char* data,
                                    unsigned int dataLength);
unsigned int
get_debug_with_triggering_callbacks(unsigned long long address,
                                    unsigned char* data,
                                    unsigned int dataLength,
                                    const unsigned char* byteEnablePtr,
                                    unsigned int byteEnableLength);
unsigned int
get_debug_with_triggering_callbacks(unsigned long long address,
                                    unsigned char* data,
                                    unsigned int dataLength);

unsigned int
put_debug_with_triggering_callbacks(unsigned long long index,
                                    const DT& data,
                                    unsigned int size,
                                    unsigned int offset);
unsigned int
put_debug_with_triggering_callbacks(unsigned long long index,
                                    const DT& data);
unsigned int
get_debug_with_triggering_callbacks(unsigned long long index,
                                    DT& data,
                                    unsigned int size,
                                    unsigned int offset);
unsigned int
get_debug_with_triggering_callbacks(unsigned long long index,
                                    DT& data);
```

2.7 Operators

The following assignment operators are available:

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

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

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

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

`iterator end()`

```
const_iterator end() const
```

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

2.8 Callbacks

The default behavior of the memory object can be changed by registering a callback to the object. Callbacks can be registered for read accesses or write accesses and for debug accesses or regular accesses.

For single-word accesses, the callback of the most specialized memory object that can handle the access is triggered. If this memory object does not have a callback, the callback of the parent object is triggered.

When a burst access cannot be handled by a single callback (for example, if there are aliases with different callbacks), the burst access is unrolled into single-word accesses. Each unrolled access is forwarded to the correct callback.

The address of the transaction passed to the execute method of the callback object is relative to the start of the memory object to which the callback is registered (the offset of the memory object is subtracted from the address).

The following methods are available to register or unregister callbacks to a memory object:

```
void set_callback(memory_callback_base* cb)
void set_read_callback(memory_callback_base* cb)
void set_write_callback(memory_callback_base* cb)

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

void set_debug_callback(memory_debug_callback_base* cb)
void set_debug_read_callback(memory_debug_callback_base* cb)
void set_debug_write_callback(memory_debug_callback_base* cb)

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

Memory callbacks must inherit from the memory_callback_base class.
Memory debug callbacks must inherit from the memory_debug_callback_base class.

Registering a callback (a regular callback or a debug callback) to a memory object will disable DMI access to this memory object.

2.9 Convenience Callback Functions

A number of convenience functions is defined to register predefined callbacks to a memory object. These functions are defined in the scml2/memory_callback_functions.h and scml2/memory_debug_callback_functions.h file, respectively.

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

```
set_callback(mem, object, callback, syncType)
set_read_callback(mem, object, callback, syncType)
set_write_callback(mem, object, callback, syncType)
set_callback(mem, object, callback, syncType, tag)
set_read_callback(mem, object, callback, syncType, tag)
set_write_callback(mem, object, callback, syncType, tag)
set_word_read_callback(mem, object, callback, syncType)
set_word_write_callback(mem, object, callback, syncType)
set_word_read_callback(mem, object, callback, syncType, tag)
set_word_write_callback(mem, object, callback, syncType, tag)
```

and the following functions are available to register a member method as a debug callback to a `memory` object:

```
set_debug_callback(mem, object, callback)
set_debug_read_callback(mem, object, callback)
set_debug_write_callback(mem, object, callback)
set_debug_callback(mem, object, callback, tag)
set_debug_read_callback(mem, object, callback, tag)
set_debug_write_callback(mem, object, callback, tag)
```

where:

mem	Is the `memory` object to which the callback will be registered.
object	Is a pointer to the class containing the callback method.

callback	Is a pointer to a member function of the `object` class. For regular callbacks, *`callback`* must have one of the following signatures:
	`void transportCallback(tlm::tlm_generic_payload&, sc_core::sc_time&)`
	`void transportCallback(tlm::tlm_generic_payload&, sc_core::sc_time&, int tag)`
	`void transportCallback(tlm::tlm_generic_payload&)`
	`void transportCallback(tlm::tlm_generic_payload&, int tag)`
	Callbacks of these types can be registered to all memory objects. The transaction is passed unmodified to the callback.
	`bool readCallback(DT& data, const DT& byteEnables, sc_core::sc_time&)`
	`bool readCallback(DT& data, const DT& byteEnables, sc_core::sc_time&, int tag)`
	`bool readCallback(DT& data, const DT& byteEnables)`
	`bool readCallback(DT& data, const DT& byteEnables, int tag)`
	These types of read callbacks can only be used with the `set_read_callback()` function.
	`bool wordReadCallback(DT& data, sc_core::sc_time&)`
	`bool wordReadCallback(DT& data, sc_core::sc_time&, int tag)`
	`bool wordReadCallback(DT& data)`
	`bool wordReadCallback(DT& data, int tag)`
	These types of read callbacks can only be used with the `set_word_read_callback()` function.
	`bool writeCallback(const DT& data, const DT& byteEnables, sc_core::sc_time&)`
	`bool writeCallback(const DT& data, const DT& byteEnables,`
	`sc_core::sc_time&, int tag)`
	`bool writeCallback(const DT& data, const DT& byteEnables)`
	`bool writeCallback(const DT& data, const DT& byteEnables, int tag)`
	These types write callbacks can only be used with the `set_write_callback()` function.
	`bool wordWriteCallback(const DT& data, sc_core::sc_time&)`
	`bool wordWriteCallback(const DT& data, sc_core::sc_time&, int tag)`
	`bool wordWriteCallback(const DT& data)`
	`bool wordWriteCallback(const DT& data, int tag)`
	These types write callbacks can only be used with the `set_word_write_callback()` function.
	The boolean return value indicates whether or not the access was successful.
	Except for the `transportCallback`, each callback can have an extra `const scml2::tlm2_gp_extensions& extensions` parameter, which can be used to query any extensions that were present in the original TLM payload. For example:
	`MyExtension* my_extension = extensions.get_extension<MyExtension>();`
	The callbacks of type `readCallback`, `writeCallback`, `wordReadCallback`, and `wordWriteCallback` can only be registered to a memory object of size `1`. Streaming burst accesses are unrolled into word accesses and subword accesses are converted into word accesses with byte enables. The `byteEnables` mask will contain `0xff` for enabled bytes and `0x0` for disabled bytes. The `wordReadCallback` and `wordWriteCallback` types are for word accesses only. For unaligned accesses or subword accesses, an error response is returned. The untimed callbacks (without the `sc_time` parameter) cannot be `SELF_SYNCING` callbacks.
	For debug callbacks, *`callback`* must have one of the following signatures:
	`unsigned int transportCallback(tlm::tlm_generic_payload&)`
	`unsigned int transportCallback(tlm::tlm_generic_payload&, int tag)`
	The return value is the number of consecutive bytes successfully read or written. If the access cannot be executed, `0` must be returned. For details, see the *IEEE Std 1666 TLM-2.0 Language Reference Manual.*

syncType	Can be one of the following: • NEVER_SYNCING indicates that the callback is nonblocking and must never call wait(). • SELF_SYNCING indicates that the callback is blocking and may call wait(). The timing annotation is passed unmodified to the callback. • AUTO_SYNCING indicates that the callback is blocking and may call wait(). The memory object synchronizes before calling the callback. The timing annotation passed to the callback is always SC_ZERO_TIME. These types are defined in the scml2/types.h file.
tag	Is a user-provided integer that is passed to the callback.

The read data which is returned by a callback is stored in the memory object after the callback returns. It is not necessary to store the read data to the memory in the callback implementation.

- The following macros are available to register a post write callback. The post write callback will be called for a write access, after the value is written to the memory:

```
set_post_write_callback(mem, object, callback, synType);
set_post_write_callback(mem, object, callback, synType, tag);
```

where:

mem	Is the memory object to which the callback will be registered.
object	Is a pointer to the class containing the callback method.

callback	Is a pointer to a member function of the object class. It must have one of the following signatures:
	`void postWriteCallback()`
	`void postWriteCallback(int tag)`

syncType	Can be one of the following: • NEVER_SYNCING indicates that the callback is nonblocking and must never call wait(). • AUTO_SYNCING indicates that the memory object synchronizes before calling the callback. These types are defined in the scml2/types.h file.
tag	Is a user-provided integer that is passed to the callback.

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

```
set_clear_on_read(mem): Clears all bits of the memory when the memory is read
set_set_on_read(mem): Sets all bits of the memory when the memory is read
set_clear_on_write_0(mem): Clears all bits to which the bit '0' is written
set_clear_on_write_1(mem): Clears all bits to which the bit '1' is written
set_write_once(mem): Sets all bits to which the bit '0' is written
set_set_on_write_1(mem): Sets all bits to which the bit '1' is written
```

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

```
set_clear_on_read_callback(mem, object, callback, syncType)
set_clear_on_read_callback(mem, object, callback, syncType, tag)
set_word_clear_on_read_callback(mem, object, callback, syncType)
set_word_clear_on_read_callback(mem, object, callback, syncType, tag)
set_set_on_read_callback(mem, object, callback, syncType)
set_set_on_read_callback(mem, object, callback, syncType, tag)
set_word_set_on_read_callback(mem, object, callback, syncType)
set_word_set_on_read_callback(mem, object, callback, syncType, tag)
set_clear_on_write_0_callback(mem, object, callback, syncType)
set_clear_on_write_0_callback(mem, object, callback, syncType, tag)
set_word_clear_on_write_0_callback(mem, object, callback, syncType)
set_word_clear_on_write_0_callback(mem, object, callback, syncType, tag)
set_clear_on_write_1_callback(mem, object, callback, syncType)
set_clear_on_write_1_callback(mem, object, callback, syncType, tag)
set_word_clear_on_write_1_callback(mem, object, callback, syncType)
set_word_clear_on_write_1_callback(mem, object, callback, syncType, tag)
set_write_once_callback(mem, object, callback, syncType)
set_write_once_callback(mem, object, callback, syncType, tag)
set_word_write_once_callback(mem, object, callback, syncType)
set_word_write_once_callback(mem, object, callback, syncType, tag)
set_set_on_write_1_callback(mem, object, callback, syncType)
set_set_on_write_1_callback(mem, object, callback, syncType, tag)
set_word_set_on_write_1_callback(mem, object, callback, syncType)
set_word_set_on_write_1_callback(mem, object, callback, syncType, tag)
```

where:

mem	Is the `memory` object to which the callback will be registered.
object	Is a pointer to the class containing the callback method.
callback	Is a pointer to a member function of the `object` class. It must have one of the `transportCallback`, `readCallback`, or `wordReadCallback` signatures as listed above.
syncType	Can be one of the following: • `NEVER_SYNCING` indicates that the callback is nonblocking and must never call `wait()`. • `SELF_SYNCING` indicates that the callback is blocking and may call `wait()`. The timing annotation is passed unmodified to the callback. • `AUTO_SYNCING` indicates that the callback is blocking and may call `wait()`. The `memory` object synchronizes before calling the callback. The timing annotation passed to the callback is always `SC_ZERO_TIME`. These types are defined in the `scml2/types.h` file.
tag	Is a user-provided integer that is passed to the callback.

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

```
set_post_clear_on_write_0_callback(mem, object, callback, syncType)
set_post_clear_on_write_0_callback(mem, object, callback, syncType, tag)
set_post_clear_on_write_1_callback(mem, object, callback, syncType)
set_post_clear_on_write_1_callback(mem, object, callback, syncType, tag)
set_post_write_once_callback(mem, object, callback, syncType)
set_post_write_once_callback(mem, object, callback, syncType, tag)
set_post_set_on_write_1_callback(mem, object, callback, syncType)
set_post_set_on_write_1_callback(mem, object, callback, syncType, tag)
```

  where:

mem	Is the memory object to which the callback will be registered.
object	Is a pointer to the class containing the callback method.

callback	Is a pointer to a member function of the `object` class. It must have one of the following signatures:
	`void postWriteCallback()`
	`void postWriteCallback(int tag)`

syncType	Can be one of the following: • `NEVER_SYNCING` indicates that the callback is nonblocking and must never call `wait()`. • `AUTO_SYNCING` indicates that the memory object synchronizes before calling the callback. These types are defined in the `scml2/types.h` file.
tag	Is a user-provided integer that is passed to the callback.

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

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

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

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

- The following functions are available to disallow the debug access to a memory. These callbacks register a debug callback of type `memory_disallow_debug_access_callback`. The callback ignores the access and returns `0`.

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

where *mem* is the memory object to which the callback will be registered.

Note

- SCML2_CALLBACK can be used as a convenience macro when registering a member function of the SC_CURRENT_USER_MODULE class as a callback. The macro takes the name of the member function and replaces the *object* and *callback* arguments in the convenience functions. For example:

```
scml2::set_callback(myMemory, SCML2_CALLBACK(myCallbackMethod),
                    scml2::NEVER_SYNCING);"
```

  The include file for this macro is scml2/callback_macro.h.

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

2.10 Binding

A memory object can be bound to a TLM2 target socket via a tlm2_gp_target_adapter object.

The following methods can be used by the adapter to register and unregister a pointer to the tlm_bw_direct_mem_if interface:

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

When the memory object has to invalidate the DMI pointers, it calls invalidate_direct_mem_ptr() on each registered interface.

Chapter 3
memory_alias

A `memory_alias` object is an alias for a memory region of a memory or another `memory_alias` object.

A `memory_alias` object has the following properties:

- Has no own storage (the top-level memory has storage).
- Must have a parent (`memory` or another `memory_alias` object).
- Can have other aliases and/or registers.
- Can have callbacks to change the default memory behavior.
- Can**not** be bound to TLM2 target sockets.

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

The following sections describe:

3.1 Types

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

```
template <typename DT> class memory_alias
```

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

The following type definitions are available:

```
typedef DT data_type
typedef memory_index_reference<DT> reference
typedef memory_iterator<DT> iterator
typedef memory_const_iterator<DT> const_iterator
```

3.2 Constructors

The following constructors are available:

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

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

3.3 Initialization

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

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

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

3.4 Properties

The following methods are available to set properties:

```
const std::string& get_name() const
```

Returns the full hierarchical name of the of the `memory_alias` object.

```
unsigned long long get_offset() const
```

Returns the offset in words relative to the top-level `memory` object.

```
unsigned long long get_size() const
```

Returns the size of the `memory_alias` object in words.

```
unsigned int get_width() const
```

Returns the width in bytes of the underlying data type of the `memory_alias` object.

```
memory_base* get_parent() const
```

Returns a pointer to the parent `memory` object.

```
bool is_dmi_enabled()
```

Returns `true` if DMI accesses are allowed for the object, `false` otherwise. DMI is enabled by default.

```
void enable_dmi()
void disable_dmi()
```

Enables/disables DMI accesses for the object.

3.5 TLM2 API Methods

The `memory_alias` object implements the following TLM2 methods:

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

The memory_alias object forwards all operations to its parent memory object.

The memory_alias object does not implement the nb_transport() call.

The TLM2 transport methods trigger the callbacks registered to the memory_alias object, its aliases, or its ancestors.

3.6 Access Methods

The following transport methods are available on the memory_alias objects:

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

These transport methods do - unlike the TLM2 transport methods ((b_transport() and transport_dbg()) - not trigger any callbacks. They access the current content of the memory_alias object. The debug methods also do not trigger any watchpoints on the memory_alias object.

Next to the transport methods, the memory_alias objects have some convenience put and get methods. The following methods are available (see also "Access Methods" on page 15):

```
void put(unsigned long long address,
         const unsigned char* data,
         unsigned int dataLength,
         const unsigned char* byteEnablePtr,
         unsigned int byteEnableLength);
void put(unsigned long long address,
         const unsigned char* data,
         unsigned int dataLength);
void get(unsigned long long address,
         unsigned char* data,
         unsigned int dataLength,
         const unsigned char* byteEnablePtr,
         unsigned int byteEnableLength) const;
void get(unsigned long long address,
         unsigned char* data,
         unsigned int dataLength) const;

void put(unsigned long long index,
         const DT& data,
         unsigned int size,
         unsigned int offset);
void put(unsigned long long index,
         const DT& data);
DT get(unsigned long long index,
       unsigned int size,
       unsigned int offset) const;
DT get(unsigned long long index) const;

void put_debug(unsigned long long address,
               const unsigned char* data,
               unsigned int dataLength,
               const unsigned char* byteEnablePtr,
               unsigned int byteEnableLength);
void put_debug(unsigned long long address,
               const unsigned char* data,
               unsigned int dataLength);
void get_debug(unsigned long long address,
               unsigned char* data,
               unsigned int dataLength,
               const unsigned char* byteEnablePtr,
               unsigned int byteEnableLength) const;
void get_debug(unsigned long long address,
               unsigned char* data,
               unsigned int dataLength) const;

void put_debug(unsigned long long index,
               const DT& data,
               unsigned int size,
               unsigned int offset);
void put_debug(unsigned long long index,
               const DT& data);
DT get_debug(unsigned long long index,
             unsigned int size,
             unsigned int offset) const;
DT get_debug(unsigned long long index) const;

tlm::tlm_response_status
put_with_triggering_callbacks(unsigned long long address,
                              const unsigned char* data,
                              unsigned int dataLength,
                              const unsigned char* byteEnablePtr,
                              unsigned int byteEnableLength,
                              sc_core::sc_time& t);
tlm::tlm_response_status
put_with_triggering_callbacks(unsigned long long address,
                              const unsigned char* data,
                              unsigned int dataLength,
                              sc_core::sc_time& t);
tlm::tlm_response_status
get_with_triggering_callbacks(unsigned long long address,
                              unsigned char* data,
                              unsigned int dataLength,
                              const unsigned char* byteEnablePtr,
                              unsigned int byteEnableLength,
                              sc_core::sc_time& t);
tlm::tlm_response_status
get_with_triggering_callbacks(unsigned long long address,
                              unsigned char* data,
                              unsigned int dataLength,
                              sc_core::sc_time& t);
```

```
tlm::tlm_response_status
put_with_triggering_callbacks(unsigned long long index,
                              const DT& data,
                              unsigned int size,
                              unsigned int offset,
                              sc_core::sc_time& t);
tlm::tlm_response_status
put_with_triggering_callbacks(unsigned long long index,
                              const DT& data,
                              sc_core::sc_time& t);
tlm::tlm_response_status
get_with_triggering_callbacks(unsigned long long index,
                              DT& data,
                              unsigned int size,
                              unsigned int offset,
                              sc_core::sc_time& t);
tlm::tlm_response_status
get_with_triggering_callbacks(unsigned long long index,
                              DT& data,
                              sc_core::sc_time& t);

unsigned int
put_debug_with_triggering_callbacks(unsigned long long address,
                                    const unsigned char* data,
                                    unsigned int dataLength,
                                    const unsigned char* byteEnablePtr,
                                    unsigned int byteEnableLength);
unsigned int
put_debug_with_triggering_callbacks(unsigned long long address,
                                    const unsigned char* data,
                                    unsigned int dataLength);
unsigned int
get_debug_with_triggering_callbacks(unsigned long long address,
                                    unsigned char* data,
                                    unsigned int dataLength,
                                    const unsigned char* byteEnablePtr,
                                    unsigned int byteEnableLength);
unsigned int
get_debug_with_triggering_callbacks(unsigned long long address,
                                    unsigned char* data,
                                    unsigned int dataLength);

unsigned int
put_debug_with_triggering_callbacks(unsigned long long index,
                                    const DT& data,
                                    unsigned int size,
                                    unsigned int offset);
unsigned int
put_debug_with_triggering_callbacks(unsigned long long index,
                                    const DT& data);
unsigned int
get_debug_with_triggering_callbacks(unsigned long long index,
                                    DT& data,
                                    unsigned int size,
                                    unsigned int offset);
unsigned int
get_debug_with_triggering_callbacks(unsigned long long index,
                                    DT& data);
```

3.7 Operators

The following assignment operators are available:

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

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

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

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

`iterator end()`

```
const_iterator end() const
```

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

3.8 Callbacks

The default behavior of the `memory_alias` object or the behavior of a callback on an ancestor can be changed by registering a callback to the object. Callbacks can be registered for read accesses or write accesses and for debug accesses or regular accesses (see "Callbacks" on page 19).

The following methods are available to register or unregister callbacks to a `memory_alias` object:

```
void set_callback(memory_callback_base* cb)
void set_read_callback(memory_callback_base* cb)
void set_write_callback(memory_callback_base* cb)

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

void set_debug_callback(memory_debug_callback_base* cb)
void set_debug_read_callback(memory_debug_callback_base* cb)
void set_debug_write_callback(memory_debug_callback_base* cb)

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

Memory callbacks must inherit from the `memory_callback_base` class.
Memory debug callbacks must inherit from the `memory_debug_callback_base` class.

Registering a callback (a regular callback or a debug callback) to a `memory_alias` object will disable DMI access to the memory region of this `memory_alias` object.

3.9 Convenience Macros

The same convenience macros are available as for `memory` objects. For detailed information, see "Convenience Callback Functions" on page 19.

3.10 Binding

A `memory_alias` object cannot be bound to a TLM2 target socket.

Chapter 4
reg

A register (`reg`) object is a `memory_alias` object of size `1`.

A `reg` object has the following properties:

- Has no own storage (the top-level memory has storage).
- Must have a parent (`memory` or `memory_alias`).
- Can**not** have other aliases and/or registers.
- Can have callbacks to change the default memory behavior.
- Can**not** be bound to TLM2 target sockets.

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

The following sections describe:

4.1 Types

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

```
template <typename DT> class reg
```

When instantiating a `reg` object, it must have the same template value as its parent `memory`/`memory_alias` object.

The following type definitions are available:

```
typedef DT data_type
typedef memory_iterator<DT> iterator
typedef memory_const_iterator<DT> const_iterator
```

4.2 Constructors

The following constructors are available:

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

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

4.3 Initialization

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

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

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

4.4 Properties

The following methods are available to set properties:

```
const std::string& get_name() const
```

Returns the full hierarchical name of the `reg` object.

```
unsigned long long get_offset() const
```

Returns the offset in words relative to the top-level `memory` object.

```
unsigned int get_width() const
```

Returns the width in bytes of the underlying data type of the `reg` object.

```
memory_base* get_parent() const
```

Returns a pointer to the parent `memory`/`memory_alias` object.

```
bool is_dmi_enabled()
```

Returns `true` if DMI accesses are allowed for the object, `false` otherwise. DMI is enabled by default.

```
void enable_dmi()
void disable_dmi()
```

Enables/disables DMI accesses for the object.

4.5 TLM2 API Methods

The `reg` object implements the following TLM2 methods:

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

The `reg` object forwards all operations to its parent `memory`/`memory_alias` object.

Note The `reg` object does not implement the `nb_transport()` call.

The TLM2 transport methods trigger the callbacks registered to the `reg` object or its ancestors.

4.6 Access Methods

The following transport methods are available on the `reg` object:

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

These transport methods will - unlike the TLM2 transport methods (`b_transport()` and `transport_dbg()`) - not trigger any callbacks. They access the current content of the `reg` object. The debug methods also do not trigger any watchpoints on the `reg` object.

Next to the transport methods, the `reg` objects have some convenience put and get methods (see also "Access Methods" on page 15). Unlike the `memory` object, the put and get methods of the `reg` object do not take an *index* argument (since this should always be *0*). The following methods are available:

```
void put(unsigned long long address,
         const unsigned char* data,
         unsigned int dataLength,
         const unsigned char* byteEnablePtr,
         unsigned int byteEnableLength);
void put(unsigned long long address,
         const unsigned char* data,
         unsigned int dataLength);
void get(unsigned long long address,
         unsigned char* data,
         unsigned int dataLength,
         const unsigned char* byteEnablePtr,
         unsigned int byteEnableLength) const;
void get(unsigned long long address,
         unsigned char* data,
         unsigned int dataLength) const;

void put(const DT& data,
         unsigned int size,
         unsigned int offset);
void put(const DT& data);
DT get(unsigned int size,
       unsigned int offset) const;
DT get() const;

void put_debug(unsigned long long address,
               const unsigned char* data,
               unsigned int dataLength,
               const unsigned char* byteEnablePtr,
               unsigned int byteEnableLength)
void put_debug(unsigned long long address,
               const unsigned char* data,
               unsigned int dataLength)
void get_debug(unsigned long long address,
               unsigned char* data,
               unsigned int dataLength,
               const unsigned char* byteEnablePtr,
               unsigned int byteEnableLength) const
void get_debug(unsigned long long address,
               unsigned char* data,
               unsigned int dataLength) const

void put_debug(const DT& data,
               unsigned int size,
               unsigned int offset)
void put_debug(const DT& data)
DT get_debug(unsigned int size,
             unsigned int offset) const
DT get_debug() const

tlm::tlm_response_status
put_with_triggering_callbacks(unsigned long long address,
                              const unsigned char* data,
                              unsigned int dataLength,
                              const unsigned char* byteEnablePtr,
                              unsigned int byteEnableLength,
                              sc_core::sc_time& t)
tlm::tlm_response_status
put_with_triggering_callbacks(unsigned long long address,
                              const unsigned char* data,
                              unsigned int dataLength,
                              sc_core::sc_time& t)
tlm::tlm_response_status
get_with_triggering_callbacks(unsigned long long address,
                              unsigned char* data,
                              unsigned int dataLength,
                              const unsigned char* byteEnablePtr,
                              unsigned int byteEnableLength,
                              sc_core::sc_time& t)
tlm::tlm_response_status
get_with_triggering_callbacks(unsigned long long address,
                              unsigned char* data,
                              unsigned int dataLength,
                              sc_core::sc_time& t)

tlm::tlm_response_status
put_with_triggering_callbacks(const DT& data,
                              unsigned int size,
                              unsigned int offset,
                              sc_core::sc_time& t)
tlm::tlm_response_status
```

```
put_with_triggering_callbacks(const DT& data,
                              sc_core::sc_time& t)
tlm::tlm_response_status
get_with_triggering_callbacks(DT& data,
                              unsigned int size,
                              unsigned int offset,
                              sc_core::sc_time& t)
tlm::tlm_response_status
get_with_triggering_callbacks(DT& data,
                              sc_core::sc_time& t)

unsigned int
put_debug_with_triggering_callbacks(unsigned long long address,
                                    const unsigned char* data,
                                    unsigned int dataLength,
                                    const unsigned char* byteEnablePtr,
                                    unsigned int byteEnableLength)
unsigned int
put_debug_with_triggering_callbacks(unsigned long long address,
                                    const unsigned char* data,
                                    unsigned int dataLength)
unsigned int
get_debug_with_triggering_callbacks(unsigned long long address,
                                    unsigned char* data,
                                    unsigned int dataLength,
                                    const unsigned char* byteEnablePtr,
                                    unsigned int byteEnableLength)
unsigned int
get_debug_with_triggering_callbacks(unsigned long long address,
                                    unsigned char* data,
                                      unsigned int dataLength)

unsigned int
put_debug_with_triggering_callbacks(const DT& data,
                                    unsigned int size,
                                    unsigned int offset)
unsigned int
put_debug_with_triggering_callbacks(const DT& data)
unsigned int
get_debug_with_triggering_callbacks(DT& data,
                                    unsigned int size,
                                    unsigned int offset)
unsigned int
get_debug_with_triggering_callbacks(DT& data)
```

4.7 Operators

The following assignment operators are available:

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

Returns a random access iterator pointing to the `reg` object.

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

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

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

```
operator DT() const
```

The following assignment operators are available:

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

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

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

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

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

4.8 Callbacks

The default behavior of the `reg` object or the behavior of a callback on an ancestor can be changed by registering a callback to the `reg` object. Callbacks can be registered for read accesses or write accesses and for debug accesses or regular accesses (see "Callbacks" on page 19).

The following methods are available to register or unregister callbacks to a `reg` object:

```
void set_callback(memory_callback_base* cb)
void set_read_callback(memory_callback_base* cb)
void set_write_callback(memory_callback_base* cb)

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

void set_debug_callback(memory_debug_callback_base* cb)
void set_debug_read_callback(memory_debug_callback_base* cb)
void set_debug_write_callback(memory_debug_callback_base* cb)

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

Memory callbacks must inherit from the `memory_callback_base` class.
Memory debug callbacks must inherit from the `memory_debug_callback_base` class.

Registering a callback (a regular callback or a debug callback) to a `reg` object will disable DMI access to the memory region of this `reg` object.

4.9 Convenience Macros

The same convenience macros are available as for `memory` objects. For detailed information, see "Convenience Callback Functions" on page 19.

4.10 Binding

A `reg` object cannot be bound to a TLM2 target socket.

Chapter 5
bitfield

`bitfield` objects can be attached to `reg` objects to alias some of the bits in the original register.

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

The following sections describe:

- Types
- Constructors
- Properties
- Access Methods
- Operators
- Callbacks
- Convenience Callback Functions

5.1 Types

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

```
template <typename DT> class bitfield
```

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

The following type definition is available:

```
typedef DT data_type
```

5.2 Constructors

The following constructor is available:

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

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

5.3 Properties

The following methods are available to set properties:

```
const std::string& get_name() const
```

Returns the full hierarchical name of the `bitfield` object.

```
unsigned int get_offset() const
```

Returns the offset (in bits) of the `bitfield` object in the register.

```
unsigned int get_size() const
```

Returns the size of the `bitfield` object in number of bits.

5.4 Access Methods

The following access methods are defined for `bitfield` objects:

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

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

bool put_with_triggering_callbacks(const DT& value, sc_core::sc_time& t)
bool get_with_triggering_callbacks(DT& value, sc_core::sc_time& t) const

bool put_debug_with_triggering_callbacks(const DT& value)
bool get_debug_with_triggering_callbacks(DT& value) const
```

The debug versions of these methods do not trigger watchpoints.

The versions with `_with_triggering_callbacks` trigger the callback registered to the `bitfield` object.

5.5 Operators

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

```
operator DT() const
```

The following assignment operators are available:

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

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

```
bitfield& operator+=(DT value)
bitfield& operator-=(DT value)
bitfield& operator/=(DT value)
bitfield& operator*=(DT value)
bitfield& operator%=(DT value)
bitfield& operator^=(DT value)
bitfield& operator&=(DT value)
bitfield& operator|=(DT value)
```

```
bitfield& operator<<=(DT value)
bitfield& operator>>=(DT value)
```

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

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

5.6 Callbacks

The default behavior of the `bitfield` object can be changed by registering a callback to the object. Callbacks can be registered for read accesses or write accesses and for debug accesses or regular accesses.

It is not possible to register both a memory callback to the parent `register` object and bitfield callbacks to the `bitfield` object. If there are bitfield callbacks, no memory callbacks can be registered to the `register` object.

The following methods are available to register or unregister callbacks to a `bitfield` object:

```
void set_read_callback(bitfield_read_callback_base<DT>* cb)
void set_write_callback(bitfield_write_callback_base<DT>* cb)

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

void set_debug_read_callback(bitfield_debug_read_callback_base<DT>* cb)
void set_debug_write_callback(bitfield_debug_write_callback_base<DT>* cb)

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

Bitfield read callbacks must inherit from the `bitfield_read_callback_base` class.
Bitfield write callbacks must inherit from the `bitfield_write_callback_base` class. Bitfield debug read callbacks must inherit from the `bitfield_debug_read_callback_base` class.
Bitfield debug write callbacks must inherit from the `bitfield_debug_write_callback_base` class.

5.7 Convenience Callback Functions

A number of convenience functions are defined to register predefined callbacks to a `bitfield` object. These functions are defined in the `scml2/bitfield_callback_functions.h` file.

- The following functions are available to register a member method as a callback to a `bitfield` object:

  ```
  set_read_callback(bitfield, object, callback, syncType)
  set_write_callback(bitfield, object, callback, syncType)
  set_read_callback(bitfield, object, callback, syncType, tag)
  set_write_callback(bitfield, object, callback, syncType, tag)
  ```

  where:

bitfield	Is the `bitfield` object to which the callback will be registered.

object	Is a pointer to the class containing the callback method.

callback	Is a pointer to a member function of the `object` class.
	For regular callbacks, *callback* must have one of the following signatures:
	`bool readCallback(DT& value, sc_core::sc_time& t)`
	`bool readCallback(DT& value, sc_core::sc_time& t, int tag)`
	`bool readCallback(DT& value)`
	`bool readCallback(DT& value, int tag)`
	`bool writeCallback(const DT& value, sc_core::sc_time& t)`
	`bool writeCallback(const DT& value, sc_core::sc_time& t, int tag)`
	`bool writeCallback(const DT& value)`
	`bool writeCallback(const DT& value, int tag)`
	For debug callbacks, *callback* must have one of the following signatures:
	`bool readCallback(DT& value)`
	`bool readCallback(DT& value, int tag)`
	`bool writeCallback(const DT& value)`
	`bool writeCallback(const DT& value, int tag)`
	The boolean return value indicates whether or not the access was successful.
	Except for the`transportCallback`, each callback can have an extra `const scml2::tlm2_gp_extensions& extensions` parameter, which can be used to query any extensions that were present in the original TLM payload. For example:
	`MyExtension* my_extension = extensions.get_extension<MyExtension>();`

syncType	Can be one of the following: • `NEVER_SYNCING` indicates that the callback is nonblocking and should never call `wait()`. • `SELF_SYNCING` indicates that the callback is blocking and may call `wait()`. The timing annotation is passed unmodified to the callback. • `AUTO_SYNCING` indicates that the callback is blocking and may call `wait()`. The `bitfield` object synchronizes before calling the callback. The timing annotation passed to the callback is always `SC_ZERO_TIME`. These types are defined in the `scml2/types.h` file.
tag	Is a user-provided integer that is passed to the callback.

- The following macros are available to register a post write callback. The post write callback will be called for a write access, after the value is written to the bitfield:

```
set_post_write_callback(bitfield, object, callback, synType);
set_post_write_callback(bitfield, object, callback, synType, tag);
```

where:

bitfield	Is the bitfield object to which the callback will be registered.
object	Is a pointer to the class containing the callback method.

callback	Is a pointer to a member function of the `object` class. It must have one of the following signatures:
	`void postWriteCallback()`
	`void postWriteCallback(int tag)`

syncType	Can be one of the following: • `NEVER_SYNCING` indicates that the callback is nonblocking and must never call `wait()`. • `AUTO_SYNCING` indicates that the `bitfield` object synchronizes before calling the callback. These types are defined in the `scml2/types.h` file.
tag	Is a user-provided integer that is passed to the callback.

- The following functions are available to disallow the access to a `bitfield` object:

  `set_ignore_access(`*bitfield*`)`
  `set_ignore_read_access(`*bitfield*`)`
  `set_ignore_write_access(`*bitfield*`)`
  `set_disallow_access(`*bitfield*`)`
  `set_disallow_read_access(`*bitfield*`)`
  `set_disallow_write_access(`*bitfield*`)`
  `set_read_only(`*bitfield*`)`
  `set_write_only(`*bitfield*`)`

  These callbacks register a callback of type `bitfield_disallow_read_access_callback` or `bitfield_disallow_write_access_callback`.

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

- The following functions are available to disallow the debug access to a `bitfield` object:

  `set_disallow_debug_access(`*bitfield*`)`
  `set_disallow_debug_read_access(`*bitfield*`)`
  `set_disallow_debug_write_access(`*bitfield*`)`

  These callbacks register a debug callback of type `memory_disallow_debug_read_access_callback` or `memory_disallow_debug_write_access_callback`. The callback ignores the access and returns `false`.

Registering a callback (a regular callback or a debug callback) to a `bitfield` object will disable DMI access to the parent `reg` object.

Note

`SCML2_CALLBACK` can be used as a convenience macro when registering a member function of the `SC_CURRENT_USER_MODULE` class as a callback. The macro takes the name of the member function and replaces the *`object`* and *`callback`* arguments in the convenience functions. For example:

```
scml2::set_callback(myBitfield, SCML2_CALLBACK(myCallbackMethod),
                    scml2::NEVER_SYNCING);"
```

The include file for this macro is `scml2/callback_macro.h`.

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

```
set_clear_on_read(bitfield): Clears all bits of the bitfield when the bitfield is read
set_set_on_read(bitfield): Sets all bits of the bitfield when the bitfield is read
set_clear_on_write_0(bitfield): Clears all bits to which the bit '0' is written
set_clear_on_write_1(bitfield): Clears all bits to which the bit '1' is written
set_set_on_write_0(bitfield): Sets all bits to which the bit '0' is written
set_set_on_write_1(bitfield): Sets all bits to which the bit '1' is written
```

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

```
set_clear_on_read_callback(bitfield, object, callback, syncType)
set_clear_on_read_callback(bitfield, object, callback, syncType, tag)
set_word_clear_on_read_callback(bitfield, object, callback, syncType)
set_word_clear_on_read_callback(bitfield, object, callback, syncType, tag)
set_set_on_read_callback(bitfield, object, callback, syncType)
set_set_on_read_callback(bitfield, object, callback, syncType, tag)
set_word_set_on_read_callback(bitfield, object, callback, syncType)
set_word_set_on_read_callback(bitfield, object, callback, syncType, tag)
set_clear_on_write_0_callback(bitfield, object, callback, syncType)
set_clear_on_write_0_callback(bitfield, object, callback, syncType, tag)
set_word_clear_on_write_0_callback(bitfield, object, callback, syncType)
set_word_clear_on_write_0_callback(bitfield, object, callback, syncType, tag)
set_clear_on_write_1_callback(bitfield, object, callback, syncType)
set_clear_on_write_1_callback(bitfield, object, callback, syncType, tag)
set_word_clear_on_write_1_callback(bitfield, object, callback, syncType)
set_word_clear_on_write_1_callback(bitfield, object, callback, syncType, tag)
set_set_on_write_0_callback(bitfield, object, callback, syncType)
set_set_on_write_0_callback(bitfield, object, callback, syncType, tag)
set_word_set_on_write_0_callback(bitfield, object, callback, syncType)
set_word_set_on_write_0_callback(bitfield, object, callback, syncType, tag)
set_set_on_write_1_callback(bitfield, object, callback, syncType)
set_set_on_write_1_callback(bitfield, object, callback, syncType, tag)
set_word_set_on_write_1_callback(bitfield, object, callback, syncType)
set_word_set_on_write_1_callback(bitfield, object, callback, syncType, tag)
```

where:

`bitfield`	Is the `bitfield` object to which the callback will be registered.
`object`	Is a pointer to the class containing the callback method.

callback	Is a pointer to a member function of the `object` class. It must have one of the `readCallback` or `writeCallback` signatures described above.
syncType	Can be one of the following: • `NEVER_SYNCING` indicates that the callback is nonblocking and should never call `wait()`. • `SELF_SYNCING` indicates that the callback is blocking and may call `wait()`. The timing annotation is passed unmodified to the callback. • `AUTO_SYNCING` indicates that the callback is blocking and may call `wait()`. The `bitfield` object synchronizes before calling the callback. The timing annotation passed to the callback is always `SC_ZERO_TIME`. These types are defined in the `scml2/types.h` file.
tag	Is a user-provided integer that is passed to the callback.

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

```
set_post_clear_on_write_0_callback(bitfield, object, callback, syncType)
set_post_clear_on_write_0_callback(bitfield, object, callback, syncType, tag)
set_post_clear_on_write_1_callback(bitfield, object, callback, syncType)
set_post_clear_on_write_1_callback(bitfield, object, callback, syncType, tag)
set_post_set_on_write_0_callback(bitfield, object, callback, syncType)
set_post_set_on_write_0_callback(bitfield, object, callback, syncType, tag)
set_post_set_on_write_1_callback(bitfield, object, callback, syncType)
set_post_set_on_write_1_callback(bitfield, object, callback, syncType, tag)
```

  where:

bitfield	Is the bitfield object to which the callback will be registered.
object	Is a pointer to the class containing the callback method.
callback	Is a pointer to a member function of the `object` class. It must have one of the `readCallback` or `writeCallback` signatures described above.
syncType	Can be one of the following: • `NEVER_SYNCING` indicates that the callback is nonblocking and must never call `wait()`. • `AUTO_SYNCING` indicates that the `bitfield` object synchronizes before calling the callback. These types are defined in the `scml2/types.h` file.
tag	Is a user-provided integer that is passed to the callback.

Chapter 6
router

Objects of type `router` can be used to model a dynamic address decoder.

A `router` object is similar to a `memory` object, but it has no associated storage and no default behavior. A callback must be registered to the `router` object that implements the desired behavior of the accesses to this memory range.

A `router` object can map a memory region to a region into a `memory` object, another `router` object, a `tlm2_gp_initiator_adapter` object, or an object that implements the `mappable_if` interface. Accesses to mapped regions do not trigger the attached callback, but are automatically forwarded to the destination object.

The `router` object has the following properties:

- Has no associated storage.
- Can be bound to TLM2 target sockets via an adapter (see "tlm2_gp_target_adapter" on page 57).
- Can**not** have aliases and/or registers.
- Must have a callback to implement the default behavior of accesses to its memory range.

The `router` object implements the `mappable_if` object, which means that it can be the destination for a mapped range of a `router` object.

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

The following sections describe:

6.1 Types

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

```
template <typename DT> class router
```

The following types are supported:

- `unsigned char`

- `unsigned short`
- `unsigned int`
- `unsigned long long`
- `sc_dt::sc_biguint<128>`
- `sc_dt::sc_biguint<256>`
- `sc_dt::sc_biguint<512>`

The following type definitions are available:

```
typedef DT data_type
```

6.2 Constructors

The following constructor is available:

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

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

6.3 Properties

The following methods are available to set properties:

```
const std::string& get_name() const
```

Returns the full hierarchical name of the `router` object.

```
unsigned long long get_offset() const
```

Always returns 0 for a `router` object.

```
unsigned long long get_size() const
```

Returns the size of the `router` object in words.

```
unsigned int get_width() const
```

Returns the data width of the `router` object in bytes.

```
bool is_dmi_enabled()
```

Returns `true` if DMI accesses are allowed for the object, `false` otherwise. DMI is enabled by default.

```
void enable_dmi()
void disable_dmi()
```

Enables/disables DMI accesses for the object.

6.4 Mapping Memory Regions

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

```
void unmap_all();
```

Unmaps all previously mapped memory regions.

Mapping memory regions can be done:

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

6.5 TLM2 API Methods

The `router` object implements the following TLM2 methods:

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

The `router` object does not implement the `nb_transport()` call. The target adapter to which the router is bound should convert `nb_transport()` calls into `b_transport()` calls before forwarding them to the `router` object.

The TLM2 transport methods trigger the callback registered to the `router` object.

6.6 Access Methods

Next to the transport methods described in TLM2 API Methods, the `router` objects have some convenience put and get methods. There are two classes of put/get methods with the following properties:

- `put_with_triggering_callbacks()/get_with_triggering_callbacks()`
  - Trigger callbacks
  - May trigger watchpoints
- `put_debug_with_triggering_callbacks()/get_debug_with_triggering_callbacks()`
  - Trigger debug callbacks
  - Do not trigger watchpoints

The `router` object does **not** have put/get methods that do **not** trigger the callback since the `router` object does not have a default behavior.

For each class, there are four different types of put/get methods with the following arguments:

- TLM2 style
  - `unsigned long long` *`address`*, where *`address`* specifies the byte address
  - `(const) unsigned char*` *`data`*, where *`data`* specifies the data array
  - `unsigned int` *`dataLength`*, where *`dataLength`* specifies the data length in bytes
  - `const unsigned char*` *`byteEnablePtr`*, where *`byteEnablePtr`* specifies the byte enable array (it may be `0`)
  - `unsigned int` *`byteEnableLength`*, where *`byteEnableLength`* specifies the byte enable length in bytes
- TLM2 style without byte enables
  - `unsigned long long` *`address`*, where *`address`* specifies the byte address
  - `(const) unsigned char*` *`data`*, where *`data`* specifies the data array
  - `unsigned int` *`dataLength`*, where *`dataLength`* specifies the data length in bytes
- Word access
  - `unsigned long long` *`index`*, where *`index`* specifies the word index
  - `(const) DT&` *`data`*, where *`data`* specifies the data
- Subword access
  - `unsigned long long` *`index`*, where *`index`* specifies the word index
  - `(const) DT&` *`data`*, where *`data`* specifies the data

- `unsigned int` *`size`*, where *`size`* specifies the size in bytes
- `unsigned int` *`offset`*, where *`offset`* specifies the offset in bytes

The `put_with_triggering_callbacks()` and `get_with_triggering_callbacks()` methods also take an `sc_time` argument that is passed to the callback. They return the TLM2 response status, returned by the triggered callback.

The `put_debug_with_triggering_callbacks()` and `get_debug_with_triggering_callbacks()` calls must not use byte enables (*`byteEnablePtr`* must be `0`) since TLM2 does not support byte enables for debug calls.

The following put and get methods are defined on the `router` object:

```
tlm::tlm_response_status
put_with_triggering_callbacks(unsigned long long address,
                              const unsigned char* data,
                              unsigned int dataLength,
                              const unsigned char* byteEnablePtr,
                              unsigned int byteEnableLength,
                              sc_core::sc_time& t);
tlm::tlm_response_status
put_with_triggering_callbacks(unsigned long long address,
                              const unsigned char* data,
                              unsigned int dataLength,
                              sc_core::sc_time& t);
tlm::tlm_response_status
get_with_triggering_callbacks(unsigned long long address,
                              unsigned char* data,
                              unsigned int dataLength,
                              const unsigned char* byteEnablePtr,
                              unsigned int byteEnableLength,
                              sc_core::sc_time& t);
tlm::tlm_response_status
get_with_triggering_callbacks(unsigned long long address,
                              unsigned char* data,
                              unsigned int dataLength,
                              sc_core::sc_time& t);

tlm::tlm_response_status
put_with_triggering_callbacks(unsigned long long index,
                              const DT& data,
                              unsigned int size,
                              unsigned int offset,
                              sc_core::sc_time& t);
tlm::tlm_response_status
put_with_triggering_callbacks(unsigned long long index,
                              const DT& data,
                              sc_core::sc_time& t);
tlm::tlm_response_status
get_with_triggering_callbacks(unsigned long long index,
                              DT& data,
                              unsigned int size,
                              unsigned int offset,
                              sc_core::sc_time& t);
tlm::tlm_response_status
get_with_triggering_callbacks(unsigned long long index,
                              DT& data,
                              sc_core::sc_time& t);

unsigned int
put_debug_with_triggering_callbacks(unsigned long long address,
                                    const unsigned char* data,
                                    unsigned int dataLength,
                                    const unsigned char* byteEnablePtr,
                                    unsigned int byteEnableLength);
unsigned int
put_debug_with_triggering_callbacks(unsigned long long address,
                                    const unsigned char* data,
                                    unsigned int dataLength);
unsigned int
get_debug_with_triggering_callbacks(unsigned long long address,
                                    unsigned char* data,
                                    unsigned int dataLength,
                                    const unsigned char* byteEnablePtr,
                                    unsigned int byteEnableLength);
unsigned int
get_debug_with_triggering_callbacks(unsigned long long address,
                                    unsigned char* data,
                                    unsigned int dataLength);

unsigned int
put_debug_with_triggering_callbacks(unsigned long long index,
                                    const DT& data,
                                    unsigned int size,
                                    unsigned int offset);
unsigned int
put_debug_with_triggering_callbacks(unsigned long long index,
                                    const DT& data);
unsigned int
get_debug_with_triggering_callbacks(unsigned long long index,
                                    DT& data,
                                    unsigned int size,
                                    unsigned int offset);
unsigned int
```

```
get_debug_with_triggering_callbacks(unsigned long long index,
                                    DT& data);
```

6.7 Callbacks

The default behavior of the `router` object can be changed by registering a callback to the object.

The following methods are available to register callbacks to a `router` object:

```
void set_callback(router_callback_base* cb)
void set_debug_callback(router_debug_callback_base* cb)
```

Router callbacks must inherit from the `router_callback_base` class.
Router debug callbacks must inherit from the `router_debug_callback_base` class.

6.8 Convenience Callback Functions

A number of convenience functions are defined to register predefined callbacks to a router. These functions are defined in the `scml2/router_callback_functions.h` and `scml2/router_debug_callback_functions.h` file, respectively.

- The following functions are available to register a member method as a callback to a `router` object:

  ```
  set_callback(router, object, callback, syncType)
  set_callback(router, object, callback, syncType, tag)
  ```

  and the following functions are available to register a member method as a debug callback to a `router` object:

  ```
  set_debug_callback(router, object, callback)
  set_debug_callback(router, object, callback, tag)
  ```

  where:

router	Is the `router` object to which the callback will be registered.
object	Is a pointer to the class containing the callback method.

callback	Is a pointer to a member function of the `object` class.
	For regular callbacks, *callback* must have the following signatures:
	`void transportCallback(tlm::tlm_generic_payload&, sc_core::sc_time&)`
	`void transportCallback(tlm::tlm_generic_payload&, sc_core::sc_time&, int tag)`
	For debug callbacks, *callback* must have the following signature:
	`unsigned int transportCallback(tlm::tlm_generic_payload&)`
	`unsigned int transportCallback(tlm::tlm_generic_payload&, int tag)`
	The return value is the number of consecutive bytes successfully read or written. If the access cannot be executed, `0` must be returned. For details, see the *IEEE Std 1666 TLM-2.0 Language Reference Manual*.

syncType	Can be one of the following: • NEVER_SYNCING indicates that the callback is nonblocking and must never call wait(). • SELF_SYNCING indicates that the callback is blocking and may call wait(). The timing annotation is passed unmodified to the callback. • AUTO_SYNCING indicates that the callback is blocking and may call wait(). The router object synchronizes before calling the callback. The timing annotation passed to the callback is always SC_ZERO_TIME. These types are defined in the scml2/types.h file.
tag	Is a user-provided integer that is passed to the callback.

SCML2_CALLBACK can be used as a convenience macro when registering a member function of the SC_CURRENT_USER_MODULE class as a callback. The macro takes the name of the member function and replaces the *object* and *callback* arguments in the convenience functions. For example:

```
scml2::set_callback(myMemory, SCML2_CALLBACK(myCallbackMethod),
                    scml2::NEVER_SYNCING);"
```

The include file for this macro is scml2/callback_macro.h.

6.9 Binding

A router object can be bound to a TLM2 target socket via a tlm2_gp_target_adapter object.

The following methods can be used by the adapter to register and unregister a pointer to the tlm_bw_direct_mem_if interface:

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

When the router object has to invalidate the DMI pointers, it calls invalidate_direct_mem_ptr() on each registered interface.

Chapter 7
tlm2_gp_target_adapter

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

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

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

The following sections describe:

7.1 Types

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

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

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

7.2 Constructors

The following constructor is available:

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

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

7.3 Binding

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

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

Binds an object to the adapter class.

7.4 Custom Forwarding

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

For example:

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

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

Chapter 8
tlm2_gp_initiator_adapter

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

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

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

The following sections describe:

8.1 Types

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

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

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

8.2 Constructors

The following constructor is available:

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

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

Chapter 9
memory_index_reference

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

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

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

The following sections describe:

- Types
- Access Methods
- Operators

9.1 Types

The following type definitions are available:

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

9.2 Access Methods

The following access methods are available:

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

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

9.3 Operators

A memory_index_reference object can be converted to the underlying data type of the referenced memory object:

```
operator DT() const
```

The following assignment operators are available:

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

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

```
reference& operator+=(DT value)
reference& operator-=(DT value)
reference& operator/=(DT value)
reference& operator*=(DT value)
```

```
reference& operator%=(DT value)
reference& operator^=(DT value)
reference& operator&=(DT value)
reference& operator|=(DT value)
reference& operator>>=(DT value)
reference& operator<<=(DT value)
```

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

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

Chapter 10
mappable_if

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

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

The following section describes:

- TLM API Methods

10.1 TLM API Methods

The following methods must be implemented:

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

Should return the name of the mapped destination. For a `memory` or `router` object, this is the name of the object. For a `tlm2_gp_target_adapter` object, this is the name of the TLM2 initiator socket.

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

Is called to register/unregister a pointer to a `tlm_bw_direct_mem_if` object. When the object that inherits from the `mappable_if` has to invalidate the DMI pointers, it has to call `invalidate_direct_mem_ptr()` on each registered interface.

If a `tlm_bw_direct_mem_if` object is registered multiple times, it must only be stored once and the invalidate call must only be called once.

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

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

Chapter 11
dmi_handler

The `dmi_handler` object is a convenience object to do the bookkeeping for DMI regions.

The object requests DMI pointers and stores them in a MRU-ordered list.

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

The following sections describe:

- Configuration
- Access Methods
- Other Methods

11.1 Configuration

The following methods are available to configure the `dmi_handler` object:

```
void set_interface(tlm::tlm_fw_direct_mem_if<tlm::tlm_generic_payload>* ifs)
```

Sets the forward DMI. This interface is used to request the DMI pointers.

```
bool is_dmi_enabled()
```

Returns `true` if DMI accesses are allowed for the object, `false` otherwise. DMI is enabled by default.

```
void enable_dmi()
void disable_dmi()
```

Enables/disables DMI accesses for the object.

11.2 Access Methods

The `dmi_handler` object has the following access methods:

```
bool read(unsigned long long address,
          unsigned char* data,
          unsigned int dataLength,
          const unsigned char* byteEnables,
          unsigned int byteEnableLength,
          sc_core::sc_time& t)
bool write(unsigned long long address,
          const unsigned char* data,
          unsigned int dataLength,
          const unsigned char* byteEnables,
          unsigned int byteEnableLength,
          sc_core::sc_time& t)
bool read(unsigned long long address,
          unsigned char* data,
          unsigned int dataLength,
          sc_core::sc_time& t)
bool write(unsigned long long address,
```

```
        const unsigned char* data,
        unsigned int dataLength,
        sc_core::sc_time& t)
```

Try to do a DMI access. If a DMI access is not possible or if the access does not fit into one DMI range, `false` is returned. Otherwise the data is copied and `true` is returned. The *t* argument is incremented with the read or write latency, respectively.

```
bool read_debug(unsigned long long address,
                unsigned char* data,
                unsigned int dataLength)
                bool write_debug(unsigned long long address,
                const unsigned char* data,
                unsigned int dataLength)
```

Tries to do a DMI access. If a DMI access is not possible or if the access does not fit into one DMI range, `false` is returned. Otherwise the data is copied and `true` is returned.

```
bool transport(tlm::tlm_generic_payload& trans, sc_core::sc_time& t)
bool transport_debug(tlm::tlm_generic_payload& trans)
```

Try to do a DMI access. If a DMI access is not possible or if the access does not fit into one DMI range, `false` is returned. Otherwise the data is copied and `true` is returned.

11.3 Other Methods

```
void invalidate_direct_mem_ptr(sc_dt::uint64 startRange,
                               sc_dt::uint64 endRange)
```

Must be called when the DMI pointers have to be invalidated.

Chapter 12
Callback Base Classes

This chapter describes:

- memory_callback_base
- memory_debug_callback_base
- router_callback_base
- router_debug_callback_base
- bitfield_read_callback_base
- bitfield_write_callback_base
- bitfield_debug_read_callback_base
- bitfield_debug_write_callback_base

12.1 memory_callback_base

Base class for regular callbacks of `memory`, `memory_alias`, or `reg` objects.

The following virtual methods must be implemented:

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

Implementation of the callback behavior.

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

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

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

12.2 memory_debug_callback_base

Base class for debug callbacks of `memory`, `memory_alias`, or `reg` objects.

The following virtual method must be implemented:

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

Implementation of the callback behavior.

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

12.3 router_callback_base

Base class for regular callbacks of `router` objects.

The following virtual methods must be implemented:

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

Implementation of the callback behavior.

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

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

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

12.4 router_debug_callback_base

Base class for debug callbacks of `router` objects.

The following virtual method must be implemented:

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

Implementation of the callback behavior.

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

12.5 bitfield_read_callback_base

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

The following virtual methods must be implemented:

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

Implementation of the callback behavior.

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

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

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

12.6 bitfield_write_callback_base

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

The following virtual methods must be implemented:

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

Implementation of the callback behavior.

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

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

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

12.7 bitfield_debug_read_callback_base

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

The following virtual method must be implemented:

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

Implementation of the callback behavior.

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

12.8 bitfield_debug_write_callback_base

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

The following virtual method must be implemented:

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

Implementation of the callback behavior.

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

Chapter 13
initiator_socket

The `initiator_socket` object is a convenience TLM2 socket that first tries to do a DMI access before doing a bus access. The socket uses the `dmi_handler` object to do the DMI access.

The `initiator_socket` object implements the `mappable_if` object, which means that it can be the destination for a mapped range of a `router` object.

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

The following sections describe:

13.1 Types

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

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

13.2 Configuration

The following methods are available to configure the `initiator_socket` object:

```
template <typename T>
void set_quantumkeeper(T& quantumKeeper)
```

Sets the quantum keeper the socket should use. The registered class must implement the following methods (see the section on `tlm_quantumkeeper` in the *IEEE Std 1666 TLM-2.0 Language Reference Manual*):

```
void inc(const sc_core::sc_time& t)
void set(const sc_core::sc_time& t)
bool need_sync() const
void sync()
sc_core::sc_time get_local_time() const
```

If a quantum keeper is set, the socket will pass the local time when doing a bus access and increment the local time when the timing annotation was incremented by the DMI access or bus access. If needed (`need_sync()` returns `true`), the socket will synchronize the quantum keeper after incrementing the local time.

If no quantum keeper is set, `sc_core::SC_ZERO_TIME` will be passed and `wait()` will be called if the timing annotation was incremented.

```
void set_endianness(tlm::tlm_endianness endianness)
```

Sets the endianness of the initiator mode. If the endianness is different from the host endianness, the socket converts the address and data before doing the access.

```
bool is_dmi_enabled()
```

Returns `true` if DMI accesses are allowed for the object, `false` otherwise. DMI is enabled by default.

```
void enable_dmi()
void disable_dmi()
```

Enables/disables DMI accesses for the object.

13.3 Access Methods

The following access methods are available:

```
template <typename DT>
bool read(unsigned long long address, DT& data)
template <typename DT>
bool write(unsigned long long address, const DT& data)
```

Access methods to do single-word or subword accesses. The data passed must be in arithmetic format (host endianness). If the endianness of the socket is different from the host endianness, the address and data are converted before doing the access. First a DMI access is done. If this fails, a bus access (`b_transport()`) is done. If this access fails with an error response, `false` is returned, otherwise `true` is returned. If a quantum keeper is set, the local time is passed with the bus access and the local time of the quantum keeper is incremented with the returned timing annotation. If no quantum keeper is set, `SC_ZERO_TIME` is passed and `wait()` is called if the timing annotation was incremented.

```
template <typename DT>
bool read(unsigned long long address, DT* data, unsigned int count)
template <typename DT>
bool write(unsigned long long address, const DT* data, unsigned int count)
```

Access methods for burst accesses. The passed data pointer should contain an array of words in arithmetic format (host endianness). If the endianness of the socket is different from the host endianness, the address and data are converted before doing the access. First a DMI access is done. If this fails, a bus access (`b_transport()`) is done. If this access fails with an error response, `false` is returned; otherwise `true` is returned. If a quantum keeper is set, the local time is passed with the bus access and the local time of the quantum keeper is incremented with the returned timing annotation. If no quantum keeper is set, `SC_ZERO_TIME` is passed and `wait()` is called if the timing annotation was incremented.

```
template <typename DT>
bool read(unsigned long long address, DT& data, sc_core::sc_time& t)
template <typename DT>
bool write(unsigned long long address, const DT& data, sc_core::sc_time& t)
```

Access methods to do single-word or subword accesses. The data passed must be in arithmetic format (host endianness). If the endianness of the socket is different from the host endianness, the address and data are converted before doing the access. First a DMI access is done. If this fails, a bus access (`b_transport()`) is done. If this access fails with an error response, `false` is returned, otherwise `true` is returned.
The time argument is passed with the `b_transport()` call. If a quantum keeper was set in the socket, it will be ignored.

```
template <typename DT>
bool read(unsigned long long address, DT* data, unsigned int count,
                                                sc_core::sc_time& t)
```

```
template <typename DT>
bool write(unsigned long long address, const DT* data, unsigned int count)
```

Access methods for burst accesses. The passed data pointer should contain an array of words in arithmetic format (host endianness). If the endianness of the socket is different from the host endianness, the address and data are converted before doing the access. First a DMI access is done. If this fails, a bus access (`b_transport()`) is done. If this access fails with an error response, `false` is returned; otherwise `true` is returned. The time argument is passed with the `b_transport()` call. If a quantum keeper was set in the socket, it will be ignored.

```
template <typename DT>
bool read_debug(unsigned long long address, DT& data)
template <typename DT>
bool write_debug(unsigned long long address, const DT& data)
```

Access methods to do single-word or subword debug accesses. The data passed must be in arithmetic format (host endianness). If the endianness of the socket is different from the host endianness, the address and data are converted before doing the access. First a DMI access is done. If this fails, a bus access (`b_transport()`) is done. If the debug bus access did not succeed, `false` is returned; otherwise `true` is returned.

```
template <typename DT>
bool read_debug(unsigned long long address, DT* data, unsigned int count)
template <typename DT>
bool write_debug(unsigned long long address, const DT* data, unsigned int count)
```

Access methods to do burst debug accesses. The passed data pointer should contain an array of words in arithmetic format (host endianness). If the endianness of the socket is different from the host endianness, the address and data are converted before doing the access. First a DMI access is done. If this fails, a bus access (`b_transport()`) is done. If the debug bus access did not succeed, `false` is returned; otherwise `true` is returned.

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

TLM2 access methods. First a DMI access is tried. If this fails, a bus access is done. No endianness conversions are done; the passed transaction should already be in the correct format.

13.4 TLM2 Backward Path Interfaces

The following methods are available to register or unregister a backward path interface to the `initiator_socket`. For more information, see the *Accellera IEEE 1666 LRM Language Reference Manual*.

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

Register or unregister `tlm::tlm_bw_direct_mem_if` to the `initiator_socket`. The `invalidate_direct_mem_ptr` method of all registered interfaces will be called in case the `invalidate_direct_mem_ptr` call is done on the backward path of the `initiator_sockets`. Multiple interfaces can be registered. In such cases, the call will be forwarded to all registered interfaces.

```
typedef tlm::tlm_bw_nonblocking_transport_if<tlm::tlm_generic_payload, tlm::tlm_phase>
                                                                        BwTransportIf
virtual void register_bw_transport_if(BwTransportIf* bwInterface)
virtual void unregister_bw_transport_if(BwTransportIf* bwInterface)
```

Register or unregister `tlm::tlm_bw_nonblocking_transport_if` to the `initiator_socket`. The `nb_transport_bw` method of the registered interface will be called in case the `nb_transport_bw` call is done on the backward path of the `initiator_socket`s. Only one `tlm::tlm_bw_nonblocking_transport_if` can be registered to the `initiator_socket`.

Chapter 14
status

The `status` object is a very simple object that holds a status value in string format. It can be used as the base of other higher level modeling objects or it can be used to enable debugging and analysis for a module.

The include file of the status object is `scml2/status.h`.

The following sections describe:

- Constructors
- Properties

14.1 Constructors

The following constructors are available for the `status` object.

```
explicit status(const std::string& name)
```

Creates a new status object with the specified name

14.2 Properties

The following are the properties of the `status` object.

```
std::string get_name() const
```

Returns the name of the `status` object.

```
void set_status(const std::string& status)
```

Sets the new value of the `status` object.

```
const std::string& get_status() const
```

Returns the current value of the `status` object.

Chapter 15
stream

The `stream` object is the front-end object of the SystemC Modeling Library 2 (SCML2) logging library. This is a logging library based on stream objects, similar to `std::ostream` objects (for example, `std::cout` or `std::cerr`). The `stream` object is the front-end object of this library. It formats the output and sends it to the back-end logger objects for processing (for example, sends it to `std::cout`, or writes it to a file, and so on.).

The back-end is configured (for example, by debuggers) to enable streams based on name or severity level, or a combination of both.

The include file for this object is `scml2/stream.h`.

The following sections describe:

- Constructor
- Properties
- Access Methods

15.1 Constructor

The following constructor is available for the `stream` object.

```
stream(const std::string& name, const severity& severity)
```

Creates a new stream with the specified name and severity level. For information on severity, see "severity" on page 79.

```
stream(const severity& severity)
```

Creates a new stream with the specified severity level. For information on severity, see "severity" on page 79. The name of the stream will be the name of the current `sc_module`.

15.2 Properties

The following are the properties of the `stream` object.

```
std::string get_name() const
```

Returns the name of the `stream` object.

```
const severity& get_severity() const
```

Returns the `severity` object of the `stream` object.

```
bool is_enabled() const
```

Returns `true` if the `stream` object is enabled, or returns `false` otherwise. The `stream` object will be enabled in case at least one back-end object requests output from this stream.

15.3 Access Methods

All methods that are defined on `std::ostream` are also defined on the `scml2::stream` object. A `stream` object can be used as a replacement of an `std::ostream` object like `std::cerr` or `std::cout`. The `scml2::stream` object will send the output to the back-end logger objects only when the stream is flushed. This is done when `std::endl` or `std::flush` is written to the stream.

`SCML2_LOG` can be used as a convenience macro while checking if a `stream` object is enabled.

For example:

```
SCML2_LOG(myStream) << "Debug output for myStream" << std::endl;
```

In case the stream is disabled, the macro will evaluate to one boolean check. There will be no performance impact caused by example, the operator `<<` or the implicit conversion operators in the debug output.

For performance reasons, the `SCML2_LOG` macro should always be used, or the `is_enabled` flag should be checked before sending output to the stream.

Similarly, the `SCML2_LOG_ASSERT` macro conditionally writes to a stream if its argument evaluates to `false`.

For example:

```
SCML2_LOG_ASSERT(value == 0x1234, mStream) << "Value can not be
0x1234" << std::endl;
```

Chapter 16
severity

The `severity` object holds a severity name and value. Each `stream` object has an associated `severity` object. For information on the `stream` object, see "stream" on page 77.

Lower severity level values mean a higher severity.

The include file for this object is `scml2/severity.h`.

The following sections describe:

- Constructor
- Properties
- Pre-Defined Severity Levels

16.1 Constructor

The following constructor is available for the `severity` object.

```
severity(const std::string& name, unsigned int level)
```

Creates a new `severity` object with the specified name and severity level value.

16.2 Properties

The following are the available properties of the `severity` object.

```
const std::string& get_name() const
```

Returns the name of the `severity` object.

```
unsigned int get_level() const
```

Returns the severity level value of the `severity` object.

16.3 Pre-Defined Severity Levels

The following severity levels are pre-defined by the logging library:

- `internal_error (5)`
- `error (10)`
- `warning (100)`
- `note (1000)`
- `debug (10000)`

The `severity` object has the following static methods to create the predefined `severity` objects:

```
static severity internal_error()
static severity error()
static severity warning()
static severity note()
```

```
static severity debug()
```

Chapter 17
Pin Callback Functions

The following functions are available for registering user callbacks on changes of input pins:

- The second enhancement we propose is adding the following convenience functions for registering user callbacks on pins:

```
set_change_callback(pin, object, callback);
set_change_callback(pin, object, callback, tag);
set_posedge_callback(pin, object, callback)
set_posedge_callback(pin, object, callback, tag)
set_negedge_callback(pin, object, callback);
set_negedge_callback(pin, object, callback, tag);
```

  where:

`pin`	Specifies the pin of type `sc_in<T>`. For `set_posedge_callback` and `set_negedge_callback`, the pin has to be of type `sc_in<bool>`.
`object`	Is a pointer to the class containing the callback method.

`callback`	Is a pointer to a member function of the object class. It must have one of the following signatures:
	`void changeCallback()`
	`void changeCallback(int tag)`

`tag`	Is an user-provided integer that is passed to the callback.

The `SCML2_CALLBACK` macro can be used as a convenience macro for registering a member function as a callback. For details on this macro, see "Convenience Callback Functions" on page 19.

Chapter 18
Convenience Functions

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

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

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

Little endian bit ordering is used (the offset of the `lsb` is `0`.

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

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

Little endian bit ordering is used (the offset of the `lsb` is `0`).

Index

I

M

O

R

S

T