A SystemC™ OCP Transaction Level Communication Channel

V2.2 – February 6, 2007

Document version 2.3

DISCLAIMER

OCP-IP
www.ocpip.org
E-mail: admin@ocpip.org
Phone: +1 503-291-2560
Fax: +1 503-297-1090
OCP-IP Technical Support
techsupport@ocpip.org

Revision History

Version	Date	Comment
1.0	1/15/03	Initial Generic Transaction Channel
1.0.1	3/31/03	First revision for OCP 1.0 channel
1.1	7/18/03	OCP 1.0 Sideband and layer adapters included
2.0	11/26/03	Updated generic channel, and OCP data class. Added new OCP 2.0 specific API on the generic channel.
2.0.1	2/15/04	Patched TL1 ports
2.0.2	5/17/04	Updated with pre-emptive accept methods, clocked blocking methods, made OCP monitor and protocol checker optional, modified constructors. TL2 Reset methods, added the reset case for 'return false' conditions.
2.03	10/14/04	New performance OCP TL2 channel model with timing points.
2.04	10/20/04	New chapter on differences between TL1 and TL2 as well as the differences between the two channel models.
2.1.0	2/9/05	Added OCP 2.1 support. New chapter referring to OCP Performance Monitor.
2.1.1	7/14/05	Removed Generic channel references. Removed old TL2 channel. Cleaned API descriptions. Added data type section. Removed references to self-timing. Added documentation of TL1 timing distribution. Added documentation of TL1 Thread busy event access
2.1.2	1/30/06	Removed OCP thread restriction from TL2 blocking request. Modified constructors. Added TL3 chapter.
2.1.3	9/10/06	Added monitor interfaces and configuration from cores interfaces.
2.2	2/6/07	Changed the TL1 example to catch up the channel/interface evolution.

Table of Contents

1 Introduction

This document describes the SystemC channel model for Open Core Protocol (OCP) . This model is meant for system simulation of cores that use the OCP to connect to one another. A System on a Chip (SOC) with processors, memory, an interconnect, and I/O devices could use OCP channels to handle the connections from core to core as well as between the cores and the interconnect(s).

The OCP channel models were designed with the goals of OCP correctness and ease of use. These models are useful for cores that require a model of the OCP that is close to cycle accurate. As a group, the OCP API commands are powerful and mask some of the complexity of the channel. The earlier versions (upto 2.1.0) of these models were based on a generic channel model. As Open SystemC Initiative (OSCI) has released a generic TLM package, which can be used for creating models of arbitrary interface protocols, we do not see a need for an OCP-provided generic channel, and generic transaction API.

This document covers Transaction Level One (TL1), Transaction Level Two (TL2) and Transaction Level Tree (TL3). The communication abstraction levels are categorized according to those introduced in the white paper “SystemC™ based SoC Communication Modeling for the OCP™ Protocol.” (You can obtain a copy of this paper at www.ocpip.org.) The abstraction levels of the models described in this document are as follows:

Transaction Level

Layer-3: Generic Transactions

Model approximately-timed functionality

Bus-protocol-agnostic SoC architecture

Layer-2: OCP Transactions

Model approximately-timed functionality

SoC architecture with details of OCP configurations

Layer-1: OCP Transfers

Cycle true but faster than RTL

Layer-0: Signals and signal groups

Register Transfer Level

“TLx” and Layer-x are used for Transaction Level, Layer-x interchangeably. For example, the acronym “TL1” stands for Transaction Level One.

SystemC is a C++ modeling environment designed for both cycle based and higher level modeling of systems. This document assumes a basic understanding of the SystemC language. For more information on SystemC, go to www.systemc.org.

The OCP is a non-proprietary, openly licensed, core-centric protocol for on-chip communications. To use the OCP channel model correctly, the user would be well served to have a solid understanding of the OCP protocol. The protocol is described in the *Open Protocol Specification* manual, which is available at: www.ocpip.org. The chapters on "Overview," "Theory of Operation," "Signals and Encoding," and "Protocol Semantics" are essential for understanding the OCP protocol and for using the OCP channel model.

1.1 Overview of Transaction Channels

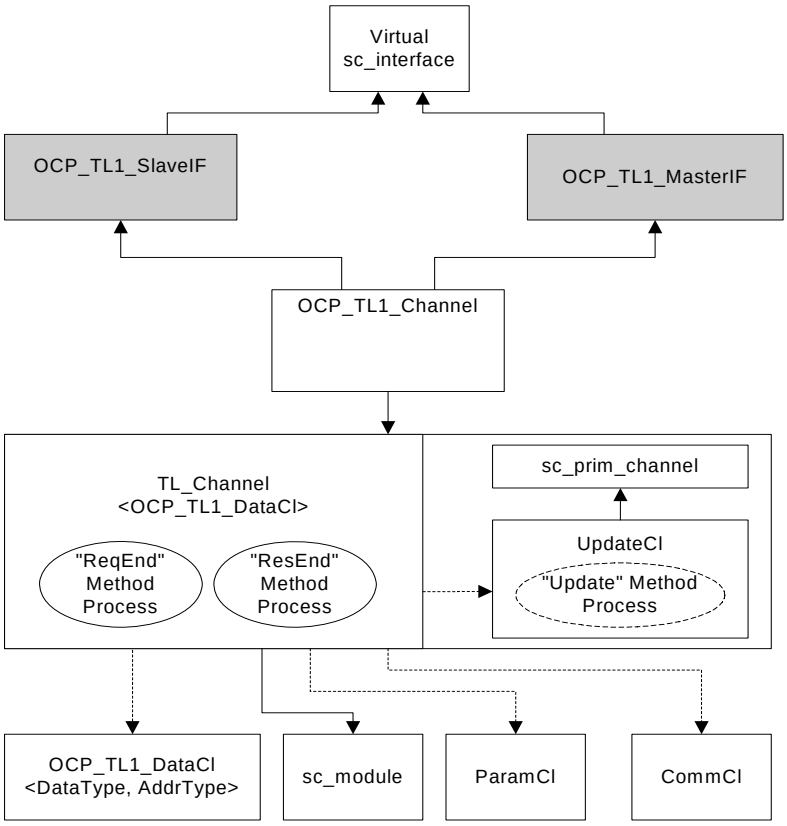
The OCP channel models are built specifically to implement the OCP. The channels are OCP correct and follow the definitions in the OCP standard. In addition, the OCP models were tailored to be easy for the core writer to use while still maintaining full OCP functionality.

Each different channel interface is meant to be a stand-alone set of commands for implementing that particular channel model. Commands should not be mixed from multiple APIs. For example, a core that uses the OCP-specific TL1 API should only use commands from that API.

1.2 Directory structure and Class Hierachy

The OCP TL1 channel is a SystemC module (`sc_module`) that uses "request/update" methods for delta cycle delayed updates of the channel state. Figure 1 shows the principal features of the internal class hierarchy for the channel. The TL_Channel contains a pointer to the type of data that moves through the channel. In this case, the data is in the Open Core Protocol (OCP) Transaction Layer One (TL1) format.

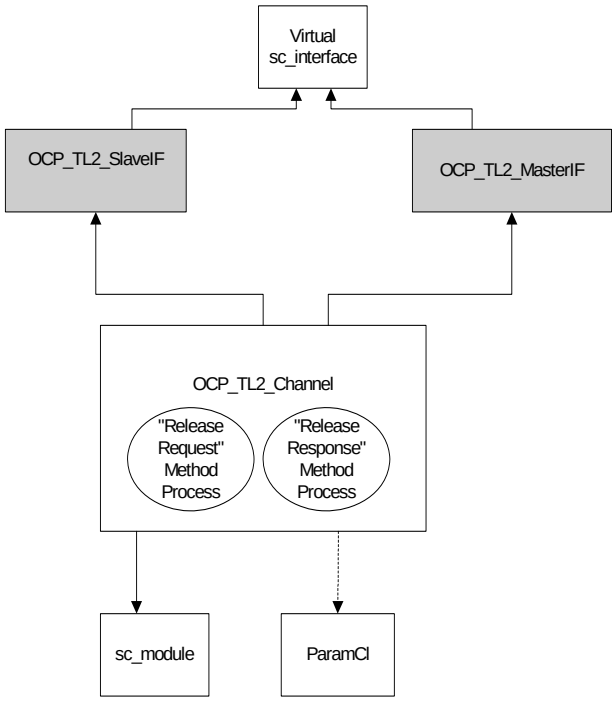
Figure 1 OCP TL1 Channel Class Hierarchy



The OCP TL1 channel is derived from the TL_Channel class. The `OCP_TL1_Channel` class implements the OCP API commands that process requests, responses, and data handshakes. In addition, the OCP TL1 channel is built to ensure that the timing and the behavior of the channel is OCP-correct. Other commands in the `OCP_TL1_Channel` provide direct access to the events in the channel (CommCl) as well as the commands of the `OCP TL1` Data Class.

The interfaces `OCP_TL1_SlaveIF` and `OCP_TL1_MasterIF` provide port access to all of the OCP API commands. There are also OCP ports for the master and slave to provide OCP specific event finders so that methods in the user's SystemC core model may be statically sensitive events in the channel.

Figure 2 OCP TL2 Channel Class Hierarchy

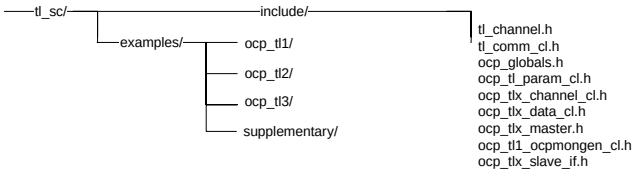


The performance oriented OCP TL2 channel is not layered upon the TL_Channel class. For more information on the TL2 channel, please see the OCP TL2 chapter.

The OCP TL3 channel is built on OSCI TLM package, which is delivered with OCP code.

Figure 3 illustrates the installed directory structure for the OCP SystemC channel models. Only a subset of the files included in the distribution is shown in figure 3.

Figure 3 OCP Channel Directory Tree



1.3 Datatypes

The OCP transaction channels use C++ templates to set datatypes for OCP address and data fields. The templating allows different bus widths to be supported in an efficient way but it does introduce a risk of incompatibility between models. The same template parameters must be used for the OCP master port and OCP slave port if they are to be bound to the same OCP channel.

The following choices of template parameters are recommended. C++ and SystemC implementations in some cases offer alternative 'type names' for these and there is no problem using a data type which is compatible with one listed in this table, where 'compatible' means that the compiler considers them the same:

OCP addr_wdth or data_wdth	C++/SystemC types recommended for templates

0 < wdth ≤ 32	`unsigned int`
	`uint32_t`
	other compatible types

32 < wdth ≤ 64	`unsigned long long`
	`uint64`
	`uint64_t`
	other compatible types

64 < wdth ≤ 128	`sc_biguint<128>` no other options
128 < wdth ≤ 256	`sc_biguint<256>` no other options
2^{N-1} < wdth ≤ 2^N	`sc_biguint<` 2^N`>` no other options

Use of other non-compatible datatypes is strongly discouraged, including SystemC datatypes with non-power-of-2 widths, for example sc_biguint<192> or similar. Use of types more precise than specified in the above table, for example uint64 for a 32-bit address bus, may sometimes be difficult to avoid, but users are warned that this may lead to compatibility problems.

2 OCP TL1 Channel Model

The OCP TL1 channel implements OCP TL1 API commands for sending and accepting OCP requests, data, and responses.

2.1 OCP TL1 Channel Constructors

There are four constructors available. The main difference is whether the instantiated channel is using an external clock or not. If the master and slave use only non-blocking methods, no timing is required in the channel, and the default constructor can be used. This is the fastest configuration of the channel. The master and slave use a clock and channel events to simulate progression of time, but the channel itself does not know of the time.

The other timing mode, clocked channel, can be used with blocking methods. (The older releases of the OCP channel had also self-timed mode, which is not compatible with clocked mode.)

The constructors in this release are greatly simplified from earlier versions, and not always compatible with old models. This is unavoidable as the self-timed mode is not included anymore. Masters and slaves that use blocking methods should still work, but there are small differences in cycle to cycle behavior, so self-timed systems should be converted to clocked with great care. This is anyway a smaller price to pay than supporting two incompatible timing modes. The timing of any TL1 models is interoperable by construction when all models are clocked.

The version 2.1.1 clocked constructor with monitor file name has been deprecated, because the monitors are now implemented completely outside the channel.

Default Constructor

The default constructor can configure non-timed channels. Normally, this constructor would need only name as a parameter, the other parameters can be left as defaults.

```
OCP_TL1_Channel(std::string name,
                bool use_event = true,
                bool use_default_event = true
               )
```

`name`
: specifies the name of the module (channel) instance.

`use_event`
: specifies whether the channel's events for the synchronization of `Mput*()` and `Sget*()` methods as well as `Sput*()` and `Mget*()` methods are triggered (`use_event = true`) or not (`use_event = false`). Always set `use_event` to `true`. This parameter may be used in future for simulation speed optimization.

`use_default_event`
: specifies whether the channel should trigger the default event. The channel may be faster if no default event is triggered. `use_default_event` can be false if none of the attached modules are sensitive to port events. This speeds up the simulation a little.

Simple Clocked Constructor

OCP_TL1_Channel(std::string name,
<clock_object> * clk)

`name`
specifies the name of the module (channel) instance.

`<clock_object> ::= "sc_in_clk" | "sc_clock" | "sc_signal<bool>"`
A pointer to the object giving clock events.

The OCP TL1 channel provides a number of constructors with different parameter combinations. The use models of these options are not clear and most of the constructors are anyway deprecated. To clarify the intended use-case of the officially supported constructors, two wrapper classes for the TL1 channel have been added.

A corresponding trace monitor is provided for each wrapper class. These trace monitors generate CoreCreator-compliant trace files for the OCP traffic.

2.1.1 OCP TL1 Channel Clock Wrapper

The OCP TL1 channel gets the clock through a constructor pointer-argument. This makes is easier to instantiate the same channel class for clocked and non-clocked applications, since no sc_port-members are needed in the channel. Unfortunately, it also makes the use of the TL1 channel more difficult with EDA tools, which depend on sc_port in binding. To alleviate this situation, a wrapper class with a clock port is provided for the TL1 channel. This class is called OCP_TL1_Channel_Clocked, and it inherits the OCP_TL1_Channel.

The version 2.1.1 constructor with monitor file name has been deprecated, because the monitors are now implemented completely outside the channel.

The clock wrapper has one constructor:

OCP_TL1_Channel_Clocked(sc_module_name name)

`name`
specifies the name of the module (channel) instance. (Notice that the type of this parameter is sc_module_name, as appropriate.)

The clock port is defined as:

sc_in<bool> p_clk;

In addition, the clock wrapper channel overloads the setConfiguration-method (see section 2.2) with a new method that reads the OCP configuration from a file:

void setConfiguration(std::string configFileName)

2.1.2 OCP TL1 Channel Untimed Wrapper

The purpose of the untimed channel is to represent a simple interface for channels which are not attached to a clock. This class is called OCP_TL1_Channel_Untimed, and inherits the OCP_TL1_Channel. Modules attached to an untimed channel are not allowed to call the ocpWait() method.

The version 2.1.1 constructor with monitor file name has been deprecated because the monitors are now implemented completely outside the channel.

The untimed wrapper has one constructor:

OCP_TL1_Channel_Untimed(std::string name)

name
specifies the name of the module (channel) instance.

In addition, the untimed wrapper channel overloads the setConfiguration method (see section 2.2) with a new method that reads the OCP configuration from a file:

Void setConfiguration(std::string configFileName)

2.2 Configuration of OCP TL1 Channel

The OCP TL1 can be configured using the standard OCP configuration parameters. These describe aspects of the OCP interface such as bus widths, flow control options and transactions supported. For the complete list of parameters and their meanings, refer to the *Open Core Protocol Specification* document. The parameters of the OCP channel have the exact same names and function as the parameters in that document. Some of these parameters affect the behaviour of the OCP channel, such as the parameters respaccept and sthreadbusyexact. Others do not, although they might affect the behaviour of an attached monitor. All parameters are stored in the channel and can be accessed from the master, the slave, and any other C++ object which has a reference to the OCP channel.

If the channel is configured, this must happen during the elaboration phase of the SystemC simulation or at end-of-elaboration. If the channel is not configured then a default configuration is adopted, which is basically the set of defaults for the OCP parameters as specified in the *Open Core Protocol Specification.* The default configuration is not normally useful.

Internally the OCP TL1 channel stores the parameters in an object of the class ParamCl which is derived from the class OCPParameters. The OCP parameters are public data members of the OCPParameters class and hence of the ParamCl class.

2.2.1 Configuration from Cores

The channel can be configured by the master and slave modules bound to it. This happens at end-of-elaboration. If both the master and slve configure the channel, the channel analyses the two configurations and determines:

- If they are compatible, according to the compatibility rules from *Open Core Protocol Specification.*
- The single OCP configuration resulting from their combination

Note that this functionality may not be complete in release 2.2. The constraints on master/slave compatibility may be tighter than absolutely necessary.

If only one of the master and slave configures the channel, then the other may register itself in the channel as a configuration listener. The channel will then inform it when any changes to the configuration are made.

If the master or the slave tries simply to read the configuration from the channel at end-of-elaboration, there is a risk that the configuration will subsequently change. Therefore the method GetParameters() should not be used until after the simulation has

started, unless the user is certain that the channel configuration is done from the environment before end-of-elaboration.

2.2.2 Configuration from Environment

The channel may be configured directly from the environment. Such a configuration is ignored if the channel is configured also by either the OCP master or the OCP slave.

If the channel is configured from the environment, both the master and the slave may register as configuration listeners.

2.2.3 Parameter Map Format

The channel may be configured using a MAP object that contains all of the parameter settings, for example:

```
setOCPMasterConfiguration( map<string,string>& parameterMap );
```

The MAP object is a C++ Standard Template Library (STL) object that is an associative array. In this case, the MAP is string-to-string with the key string being the name of the parameter and the value string being the parameter value. This parameter MAP may be automatically generated by a configuration tool. It may be hand coded in the source code for the master or slave, or in the `main.cc` program, or it may be built by reading in parameter data from a file.

Each entry in the parameter map is a pair of strings. The left side (the key side) of the pair is the parameter name. The right side (the value side) is the parameter value. The parameter name is a string, and it must exactly match the OCP standard parameter name. For example, "`cmdaccept`" is the OCP parameter to indicate that the SCmdAccept signal is part of the OCP channel. You must be careful in the use of case or nonstandard spellings (such as "`CMDAccept`" or "`SCommandAccept`"), which will not give you the desired result.

The value side of the parameter map has the following format:

type_char:value

Where `type_char` is a single character is one of the following:

"`i`" specifies an integer or Boolean

"`f`" specifies a floating point value

"`s`" specifies a string.

Note that a colon (:) is required, and the value is the value of the parameter. Also, the value should not contain any spaces. For example:

"`i:1`" An integer value 1 or the Boolean value TRUE.

"`f:3.14159`" The floating point value for PI.

"`s:little`" The string value "little."

For a usage example see section 6.1.

2.2.4 Building the Parameter Map from a File

The channel may also be configured by using a text file. Additionally this can be useful because the file name may be passed to the main program that builds the simulation.

Also, the file name may be changed on the command line so the parameters are changed without having to recompile the model.

In the example below, the parameters are in a file as lines of pairs of space separated strings:

```
cmdaccept i:1
addr_wdth i:40
endian s:both
```

The user's code then reads the strings from the file and stores them into an STL map. The map is then passed to the channel's `setConfiguration` function.

2.3 OCP TL1 Enum Types and Template Classes

The OCP TL1 API commands pass requests, responses and data handshakes through as single structures. This section describes those structures (actually template classes) as well as the Enum types used by elements of those structures.

2.3.1 OCPMCmdType Enum

The `OCPMCmdType` enumerator defines the master command names. The enumerator values are listed in Table 1. This Enum type is defined as Enum OCPMCmdType

Table 1 OCPMCmdType Enum Labels and Values

Label	Value	Description
OCP_MCMD_IDLE	0	Idle command
OCP_MCMD_WR	1	Write command
OCP_MCMD_RD	2	Read command
OCP_MCMD_RDEX	3	Exclusive read command
OCP_MCMD_RDL	4	Read linked command
OCP_MCMD_WRNP	5	Non-posted write command
OCP_MCMD_WRC	6	Write conditional command
OCP_MCMD_BCST	7	Broadcast command

2.3.2 OCPRespType Enum

The `OCPSRESPType` enumerator defines the slave response names. The enumerator values are listed in Table 2. This Enum type is defined as Enum OCPSRESPType.

Table 2 OCPRespType Enum Labels and Values

Label	Value	Description
OCP_SRESP_NULL	0	Null response
OCP_SRESP_DVA	1	Data valid/accept response
OCP_SRESP_FAIL	2	Request failed
OCP_SRESP_ERR	3	Error response

2.3.3 OCPMBurstSeqType Enum

The OCPMBurstSeqType enumerator defines the OCP master burst sequence types. The enumerator values are listed in Table 3. This Enum type is defined as Enum OCPMBurstSeqType

Table 3 OCPMBurstSeqType Enum Labels and Values

Label	Value	Description
OCP_MBURSTSEQ_INCR	0	Incrementing
OCP_MBURSTSEQ_DFLT1	1	Custom (packed)
OCP_MBURSTSEQ_WRAP	2	Wrapping
OCP_MBURSTSEQ_DFLT2	3	Custom (not packed)
OCP_MBURSTSEQ_XOR	4	Exclusive OR
OCP_MBURSTSEQ_STRM	5	Streaming
OCP_MBURSTSEQ_UNKN	6	Unknown
OCP_MBURSTSEQ_BLCK	7	Block (2-dimensional)

2.3.4 OCPRequestGrp Template Class

The `OCPRequestGrp` class is used for sending and receiving requests. All the signals that make up the request group are to be found here. This template class is defined as

```
Template<class Td, class Ta>
class OCPRequestGrp
```

2.3.4.1 Data Type and Address Type

The class template parameters `Td` and `Ta` indicate the data type and address type of the MData and MAddr signals, respectively. By making this a template, any sized data or address width may be supported.

2.3.4.2 Members

Some configurations of the OCP will not use all the members in the class. In that case, the unused members are invalid and should not be referenced or used. Table 4 lists the member names and their data types for `OCPRequestGrp`.

Table 4 OCPRequestGrp Member Types

Name	Data Type	Description
MCmd	OCPMCmdType	Master command
MAddr	AddrType	Master address
MAddrSpace	unsigned int	Master address space
MData	DataType	Master data, when no data handshake
MDataInfo	Unsigned int	Extra information sent with the write data
MByteEn	unsigned int	Master byte enable
MThreadID	unsigned int	Master thread identifier

Name	Data Type	Description
MConnId	unsigned int	Master connection identifier
MTagID	unsigned int	Master tag identifier (See OCP 2.1 specification)
MTagInOrder	bool	Force tag-in-order (See OCP 2.1 specification)
MReqInfo	unsigned int	Extra information sent with the response.
MAtomicLength	unsigned int	Length of atomic burst
MBurstLength	unsigned int	Burst length
MBurstPrecise	bool	Given burst length is precise
MBurstSeq	OCPMBurstSeqType	Address sequence of burst
MBurstSingleReq	bool	Burst uses single request/multiple data protocol
MReqLast	bool	Last response in burst
MBlockHeight	unsigned int	For 2-D burst support
MBlockStride	unsigned int	For 2-D burst support
MReqRowLast	bool	For 2-D burst support

2.3.4.3 Constructor

OCPRequestGroup(bool has_mdata = true)

OCPRequestGroup(const OCPRequestGrp& src)

The first form constructs a default OCPRequestGrp object and uses the `has_mdata` parameter to indicate whether or not there is a data handshake. The value for `has_mdata` should be true for channels without data handshaking where all data is transmitted with the request. It should be false for write requests when data handshaking is enabled because the data will come through the data handshake, not the request.

The second form is the copy constructor, which copies the `src` into a new OCPRequestGroup object.

2.3.4.4 Assignment Operator (=)

OCPRequestGroup& operator=(const OCPRequestGroup& rhs)

The assignment operator assigns one OCPRequestGroup object to another.

2.3.4.5 copy

void copy(const OCPRequestGrp& src)

Copies one OCPRequestGrp object to another.

2.3.5 OCPResponseGrp Template Class

The `OCPResponseGrp` class is used to send and receive responses with the OCP TL1 channel. All of the signals that make up the response group are to be found in this class. This template class is defined as

```
Template<class Td>
OCPResponseGrp
```

2.3.5.1 Data Type

The class template parameter Td indicates the data type of the SData signal. This allows the response to contain any width of data. Note that the type of the response data must match the type of request and data handshake data.

2.3.5.2 Members

Some configurations of the OCP will not use all of the members in the class. This corresponds to the fact that some OCP implementations do not use all of the OCP signals. In that case, the unused members are invalid and should not be referenced or used. Table 5 lists the names and their data types of `OCPResponseGrp`.

Table 5 OCPResponseGrp Member Types

Name	Type	Description
SResp	OCPSRespType	Slave response
SData	DataType	Data returned by slave
SThreadID	unsigned int	Slave thread identifier
STagID	unsigned int	Slave tag identifier (See OCP 2.1 specification)
STagInOrder	bool	Force tag-in-order (See OCP 2.1 specification)
SdataInfo	unsigned int	Extra information sent with the response data.
SrespInfo	unsigned int	Extra information sent out with the response.
SrespLast	bool	Last response in burst

2.3.5.3 Constructor

OCPResponseGrp(void)

OCPResponseGrp(const OCPResponseGrp& src)

The first form constructs a default OCPResponseGrp object. The second form is the copy constructor which copies the `src` into a new OCPResponseGrp object.

2.3.5.4 Assignment Operator (=)

OCPResponseGrp& operator=(const OCPResponseGrp& rhs)

The assignment operator assigns one `OCPResponseGrp` object to another.

2.3.5.5 copy

void copy(const OCPResponseGrp& src)

Copies one `OCPResponseGrp` object to another.

2.3.6 OCPDataHSGrp Template Class

The `OCPDataHsGrp` class is a structure used to send and receive data handshake data. All of the OCP signals that make up the data group are to be found in this class. This template class is defined as

Template<class Td>
Class OCPDataHSGrp

2.3.6.1 Data Type

The class template parameter `Td` indicates the data type of the `Mdata` signal. For instance, it can be `int` or `uint64` to represent a data width of up to 32 bits and 64 bits, respectively. Note that the data type used for the `DataHSGrp` should match the data type used for the request and response group.

2.3.6.2 Members

Some configurations of the OCP will not use all of the members in the class. This is due to the fact that not every OCP configuration uses all of the OCP signals. In that case, the unused fields are invalid and should not be referenced or used. Table 6 lists the member names and their data types of `OCPDataHSgrp`.

Table 6 OCPDataHSGrp Member Types

Name	Type	Description
Mdata	DataType	The master data being sent to the slave
MdataThreadID	unsigned int	The thread identifier for the write data
MDataTagID	unsigned int	Data tag identifier (See OCP 2.1 specification)
MDataByteEn	unsigned int	The data byte enable field
MDataInfo	unsigned int	The data info field.
MDataLast	bool	Is this the last data transfer in a burst?
MDataValid	bool	Synchronization bit. True when the master places the data onto the channel. False after the slave has accepted the data.

2.3.6.3 Constructor

OCPDataHSGrp(void)

OCPDataHSGrp(const OCPDataHSGrp& src)

The first form constructs a default (empty) data handshake structure. The second form copies the passed datahandshake data into the new object.

2.3.6.4 Assignment Operator (=)

OCPDataHSGrp& operator=(const OCPDataHSGrp& rhs)

The assignment operator assigns one `OCPDataHSGrp` object to another.

2.3.6.5 copy

void copy(const OCPDataHsGrp& src)

Copies one `OCPDataHSGrp` object to another.

2.3.7 OCP_TL1_Master_TimingCl Class

This class contains a set of sc_time members, which store information about the timing characteristics of an OCP TL1 master.

2.3.7.1 Members

Table 7 OCP_TL1_Master_TimingCl Member Types

Name	Type	Description
RequestGrpStartTime	sc_time	Time after cycle start when startOCPRequest() is called
DataHSGrpStartTime	sc_time	Time after cycle start when startOCPDataHS() is called
MThreadBusyStartTime	sc_time	Time after cycle start when putMThreadBusy() is called
MRespAcceptStartTime	sc_time	Time after cycle start when putMRespAccept(bool x) is called

2.3.7.2 Equality Operator ==

The operator "==" is available for 2 objects of the class OCP_TL1_Master_TimingCl.

2.3.8 OCP_TL1_Slave_TimingCl Class

This class contains a set of sc_time members, which store information about the timing characteristics of an OCP TL1 slave.

2.3.8.1 Members

Table 8 OCP_TL1_Slave_TimingCl Member Types

Name	Type	Description
ResponseGrpStartTime	sc_time	Time after cycle start when startOCPResponse() is called
SThreadBusyStartTime	sc_time	Time after cycle start when putSThreadBusy() is called
SDataThreadBusyStartTime	sc_time	Time after cycle start when putSDataThreadBusy() is called

SCmdAcceptStartTime	sc_time	Time after cycle start when putSCmdAccept(bool x) is called
SDataAcceptStartTime	sc_time	Time after cycle start when putSDataAccept(bool x) is called

2.3.8.2 Equality Operator ==

The operator "==" is available for 2 objects of the class OCP_TL1_Slave_TimingCl.

2.4 TL1 Master Interface Methods (ocp_tl1_master.if.h)

The methods described in this section handle the OCP TL1 master's transaction request phase, response phase, and data handshake. There are also methods for OCP configuration management and cycle-accurate timing information distribution.

All methods return immediately if the channel is in reset state. The non-void methods return false if called during reset. It is advisable to make sure that the threads trusting blocking methods for sequencing call a wait if a blocking method returns false, to avoid infinite loops.

2.4.1 Reset

This section describes the methods for the master's reset phase.

bool getReset()

Purpose: Check if channel is in reset state.

Return: Returns *true* if the channel is in reset, false otherwise.

Events: No event.

void MResetAssert()

Purpose: Puts channel in reset state. Resets all channel state variables, and calls data class reset. All in-band methods will return immediately with false return value while reset is active. All blocking methods are released, and return with false.

Events: All start and end events fire (to release all waits in the system).

void MResetDeassert()

Purpose: Removes reset state from the channel.

Events: ResetEndEvent.

sc_event& ResetStartEvent()

Purpose: This event is triggered when channel reset starts.

Return: Reset start event.

sc_event& ResetEndEvent()

Purpose: This event is triggered when channel reset ends.

Return: Reset end event.

2.4.2 Request Phase

This section describes the methods for the master's TL1 request phase.

bool getSBusy()const

Purpose: Used to check whether a new request can be placed on the channel.

Return: Returns *true* if the channel is not free for a new request. This function does not check the threadbusy signal (if any). See also `getSThreadBusy()`.

Events: No event.

bool startOCPRequest(
const OCPRequestGrp<Td,Ta>& newRequest)

Purpose: Places the passed request onto the channel.

Return: Returns false if there is already a request on the channel which has not yet been accepted by the slave, or if the OCP is a configured as sthreadbusy-exact and the OCP thread is busy, or if the channel is in reset.

Events: RequestStartEvent. RequestEndEvent, if the putSCmdAccept(1) has been called before, or if the SCmdAccept is not part of the channel. No event if return value is false.

Notice: Behavior changed from release 2.1.

bool startOCPRequestBlocking(
const OCPRequestGrp<Td,Ta>& newRequest)

Purpose: Repeat - try request - Wait for a rising clock edge - until successful.

`startOCPRequestBlocking()` returns once the request has started but before the slave has accepted the request.

Notice: Not to be used for modeling OCP interfaces with multiple threads. Use non-blocking instead. Not to be called from multiple SC_TREADs.

Return: Returns an immediate false if the channel is not clocked. Returns false after a clock if the channel is in reset state. Reset is synchronous.

Events: RequestStartEvent. RequestEndEvent, if the putSCmdAccept(1) has been called before, or if the SCmdAccept is not part of the channel. No event if return value is false.

bool getSCmdAccept() const

Purpose: Get state of SCmdAccept.

Note

Despite the name, this behaves like an RTL version of SCmdAccept signal only after

a request is put into the channel, and only at rising clock edge, that is only when SCmdAccept is not don't-care according to OCP standard.

Return: Returns always true if parameter cmdaccept is 0, and `!getSBusy()` otherwise.

Event: None.

unsigned int getSThreadBusy( ) const

Purpose: Returns the current value of the SThreadBusy signal in the channel, or the value from the previous cycle if the OCP parameter sthreadbusy_pipelined is set to true.
Note that if sthreadbusy_pipelined is set to false, then this method can never be safely used in the delta cycle immediately after the clock rising edge (for a clocked channel). This is because at that time it is impossible to be confident that the SThreadBusy has stabilised for the new clock cycle.

Return: The `unsigned int` returned contains the SThreadBusy signals for each of the threads in the channel. If a bit position is "1" then that thread is busy.

Event: SThreadBusyEvent( )

sc_event& SThreadBusyEvent( ) const

Purpose: This event is triggered when the value returned by getSThreadBusy( ) changes.

Return: The event associated with a change in SThreadBusy's value

sc_event& CurrentSThreadBusyEvent( ) const

Purpose: This event is triggered when the slave changes the value of the SThreadBusy signal. No difference from SThreadBusyEvent() if sthreadbusy_pipelined is set to false.

Return: The event associated with a change in SThreadBusy's value

sc_event& RequestStartEvent()

Purpose: This event is triggered when a new request has been placed on the channel. A slave could wait this event so that it would restart when a new request was available.

Return: RequestStartEvent.

sc_event& RequestEndEvent()

Purpose: This event is triggered when the request is accepted.

Return: RequestEndEvent.

void waitSCmdAccept(void)

Purpose: If there is a current request on the channel, `waitSCmdAccept()` waits until the request has been accepted by the slave. This method returns immediately if there is no request on the channel or if that request has already been accepted. Note that if SCmdAccept is not part of the channel, this command will wait until request is automatically accepted by the channel (one delta cycle after the request is submitted.)

Return: None.

Event: None.

2.4.3 Response Phase

This section describes the methods for the master's TL1 response phase.

bool getOCPResponse(OCPResponseGrp<Td>& myResponse,
bool acceptResponse = false)

Purpose: If there is an unread response available on the channel, the response is read and returned as `myResponse`. If `acceptResponse` is true, `putMRespAccept()` is called. Note that if MRespAccept is not part of the OCP channel, the response is always automatically accepted, and the value of the `acceptResponse` parameter is ignored.

Return: Returns false if there is no response available or if the response has already been read by a `getResponse` command or if there is a `getResponseBlocking` command in progress.

Event: ResponseEndEvent, if the response has not been pre-accepted, and is accepted with this call.

bool getOCPResponseBlocking(OCPResponseGrp<Td>& myResponse,
bool acceptResponse = false)

Purpose: Waits for a new, unread response to become available on the channel. The response is then read and returned as `myResponse`. If `acceptResponse` is true, `putMRespAccept()` is called. Note that if MRespAccept is not part of the OCP channel, the response is always automatically accepted, and the value of the parameter `acceptResponse` is ignored.

Notice: Not to be used for modeling OCP interfaces with multiple threads. Use non-blocking instead. Not to be called from multiple SC_TREADs.

Return: Returns false if channel is in reset.

Notice that if a false can be expected (reset is used), this must be treated as a special case in the responding thread so no infinite loop is created. (The SC_THREAD must yield by using a wait statement.)

Event: ResponseEndEvent, if the response has not been pre-accepted, and is accepted with this call.

bool putMRespAccept()

Purpose: Sets the MRespAccept signal in the OCP channel and releases the response.

Return: Returns false if there is no response to accept or if the current response has already been accepted. Otherwise, `putMRespAccept()` returns true and the response will be accepted on the next delta cycle. Note that after the response has been accepted, the OCP channel signal SResp is then automatically reset to "`OCP_SRESP_NULL`".

Event: ResponseEndEvent, if there is an active response on the channel

void putMRespAccept(bool accept = false)

Purpose: Sets or unsets the MRespAccept signal in the OCP channel. Set can be called at any time during clock cycle, unset only at clock edge. Persistent once called.

Event: ResponseEndEvent, if there is an active response on the channel.

void putMThreadBusy(unsigned int nextMThreadBusy)

Purpose: At the next delta cycle, the OCP signal MThreadBusy will be set to the passed value

Return: None.

Event: None

// Deprecated

void putNextMThreadBusy()

sc_event& ResponseStartEvent()

Purpose: This event is triggered when a new response has been placed on the channel.

Return: ResponseStartEvent.

sc_event& ResponseEndEvent()

Purpose: This event is triggered when the response is accepted.

Return: ResponseEndEvent.

2.4.4 Data Handshake

This section describes the methods for the master's TL1 data handshake.

bool getSBusyDataHS() const

Purpose: Used to check whether a new data handshake can be started on the channel.

Return: Returns true if the channel is not free for a new data handshake. This function does not check the `threadbusy` signal (if any). See also `getSDataThreadBusy()`.

Events: No event.

bool startOCPDataHS(const OCPDataHSGrp<Td>& newData)

Purpose: Places the passed data onto the channel and automatically sets the OCP signal MDataValid to true.

Return: Returns false if there is already a data-handshake on the channel which has not yet been accepted by the slave, or if the OCP is a configured as datathreadbusy-exact and the OCP thread is busy, or if the channel is in reset.

Events: DataHSStartEvent. DataHsEndEvent, if the putSDataAccept(1) has been called before, or if the SDataAccept is not part of the channel. No event if return value is false.

Notice: Behavior changed from release 2.1.

bool startOCPDataHSBlocking(
const OCPDataHSGrp<Td>& newData)

Purpose: Repeat - try request - wait for a rising clock edge - until successful.

`startOCPDataHSBlocking()` returns once the handshake has started but before the slave has accepted the handshake.

Notice: Not to be used for modeling OCP interfaces with multiple threads. Use non-blocking instead. Not to be called from multiple SC_TREADs.

Return: Returns an immediate false if the channel is not clocked. Returns a false after a clock if the channel is in reset state. Reset is synchronous.

Notice that if a false can be expected (reset is used), this must be treated as a special case in the requesting thread so no infinite loop is created.

Events: DataHSStartEvent. DataHsEndEvent, if the putSDataAccept(1) has been called before, or if the SDataAccept is not part of the channel. No event if return value is false.

bool getSDataAccept() const

Purpose: Get state of SDataAccept.

Note

Despite the name, this behaves like an RTL version of SDataAccept signal only after a data request is put into the channel, and only at rising clock edge, that is only when SDataAccept is not don't-care according to OCP standard.

Return: Returns true, if dataaccept parameter is 0, and `!getSBusyDataHS()` otherwise.

Event: No event.

unsigned int getSDataThreadBusy( ) const

Purpose: Returns the current value of the SDataThreadBusy signal in the channel, unless sdatathreadbusy_pipelined is true in the OCP configuration, in which case the SDataThreadBusy signal from the

previous cycle is returned.
If sdatathreadbusy_pipelined is false then this method should not be called in the delta cycle immediately following a clock rising edge.

Return: The `unsigned int` returned has one bit for each thread on the channel. If a bit is "1", that thread is busy and no more data transfers should be sent to that thread.

Event: SDataThreadBusyEvent( )

sc_event& SDataThreadBusyEvent( ) const

Purpose: This event is triggered when the channel changes the value returned by getSDataThreadBusy().

Return: The event associated with a change in SDataThreadBusy's value

sc_event& CurrentSDataThreadBusyEvent( ) const

Purpose: This event is triggered when the slave changes the value of the SDataThreadBusy signal. No difference from SDataThreadBusyEvent() if sdatathreadbusy_pipelined is set to false.

Return: The event associated with a change in SDataThreadBusy's value

sc_event& DataHSStartEvent()

Purpose: This event is triggered whenever a new data handshake transfer is started on the channel.

Return: DataHSStartEvent.

sc_event& DataHSEndEvent()

Purpose: This event is triggered when the current data handshake transfer has been accepted by the slave.

Return: DataHSEndEvent.

void waitSDataAccept(void)

Purpose: If there a current data handshake on the channel, `waitSDataAccept()` waits until the data has been accepted by the slave. This method returns immediately if there is no data handshake on the channel or if that data has already been accepted. Note that if SDataAccept is not part of the channel, this command will wait until the data handshake is automatically accepted by the channel (one delta cycle after the data is submitted).

Return: None.

Event: None.

2.4.5 Timing Distribution Methods

This section describes methods implemented in the OCP TL1 channel to support timing distribution at end-of-elaboration.

void setOCPTL1MasterTiming(OCP_TL1_Master_TimingCl master_timing)

Purpose: OCP master must use this method to inform the channel of its timing parameters at end-of-elaboration, unless it conforms to default TL1 timing.

Return: None.

void registerTimingSensitiveOCPTL1Master(OCP_TL1_Slave_TimingIF *master)

Purpose: Timing-sensitive OCP masters may use this method to register themselves with the channel at end-of-elaboration. Once this has been done, all timing information provided by the slave to the channel will be forwarded to the master by the channel.

The pure virtual class OCP_TL1_Slave_TimingIF contains only the single method setOCPTL1SlaveTiming() which is also part of the OCP_TL1_Slave_IF (see below).

Return: None.

2.4.6 OCP Configuration Management Methods

virtual void setOCPMasterConfiguration(MapStringType& passedMap)

Purpose: OCP master may use this method at end of elaboration to pass the configuration of its OCP port to the channel.

Return: None.

virtual void addOCPConfigurationListener(OCP_TL_Config_Listener& listener)

Purpose: OCP master may use this method at end of elaboration to register itself as a configuration listener. After registration any changes to the OCP configuration of the channel, for example because the slave sets the channel's configuration, are passed on to the listener (see definition of OCP_TL_Config_Listener class below).
Warning: if the channel has already been configured by the slave when this is called, the listener will be informed (called-back) of the configuration before this method returns.
Warning: this method can be called multiple times during end of elaboration. The listener needs to ignore all but the last time it is called-back.
This listener should not be called-back after end-of-elaboration, if the channel is being correctly used.
This method is provided so that 'generic' OCP masters can be implemented. A generic OCP master is an OCP master without a fixed OCP configuration, whose behaviour will adapt to the OCP configuration of the slave.

Return: None.

virtual const std::string peekChannelName()

Purpose: Allows the master to find out the name of the channel, which simplifies the implementation of a 'generic' OCP master with more than one OCP port.

Return: Channel name as std::string.

virtual OCPParameters *GetParameters()

Purpose: Simple access to the OCP parameters of the channel. This method should not be used until after end-of-elaboration, unless it is certain that the channel has been configuredhand.

Return: Pointer to Parameters object of the channel. This object contains all the OCP parameters, with the expected names and types as in the OCP protocol documentation, for example

```
int t = my_OCP_port->GetParameters()->threads;
```

2.5 OCP TL1 Slave Interface Methods (ocp_tl1_slave_if.h)

The methods described in this section handle the slave's transaction level 1 request phase, response phase, and data handshake. There are also methods for OCP configuration management and cycle-accurate timing information distribution.

All methods return immediately if the channel is in reset state. The non-void methods return false if called during reset. It is advisable to make sure that the threads trusting blocking methods for sequencing call a wait if a blocking methods returns false, to avoid infinite loops.

2.5.1 Reset

This section describes the methods for the slave's reset phase.

bool getReset()

Purpose: Check if channel is in reset state.

Return: Returns *true* if the channel is in reset, false otherwise.

Events: No event.

void SResetAssert()

Purpose: Puts channel in reset state. Resets all channel state variables, and calls data class reset. All in-band methods will return immediately with false return value while reset is active. All blocking methods are released, and return with false.

Events: All start and end events fire (to release all waits in the system).

void SResetDeassert()

Purpose: Removes reset state from the channel.

Events: ResetEndEvent.

sc_event& ResetStartEvent()

Purpose: This event is triggered when channel reset starts.

Return: Reset start event.

sc_event& ResetEndEvent()

Purpose: This event is triggered when channel reset ends.

Return: Reset end event.

2.5.2 Request Phase

This section describes the methods for the slave's TL1 response phase.

bool getOCPRequest(OCPRequestGrp<Td,Ta>& myRequest,
bool acceptRequest = false)

Purpose: If there is an unread request available on the channel, the request is read and returned as "myRequest." And if `acceptRequest` is true, `putSCmdAccept()` is called. Note that if the SCmdAccept signal is not part of the OCP channel, the request is always automatically accepted, and the value of the `acceptRequest` parameter is ignored.

Return: Returns false if there is no request available or if the request has already been read by a `getOCPRequest` command or if there is a `getOCPRequestBlocking` command in progress.

Event: RequestEndEvent, if the response has not been pre-accepted, and is accepted with this call.

bool getOCPRequestBlocking(
OCPRequestGrp<Td,Ta>& myRequest,
bool acceptRequest = false)

Purpose: Waits for a new, unread request to become available on the channel, then reads the request and returns it as `myRequest`. If `acceptRequest` is true then `putSCmdAccept()` is called to accept the request at the end of the delta cycle. Note that this function waits only until it has the new request. Also note that if the SCmdAccept signal is not part of the OCP channel, the request is always automatically accepted, and the value of the `acceptRequest` parameter is ignored.

Notice: Not to be used for modeling OCP interfaces with multiple threads. Use non-blocking instead. Not to be called from multiple SC_TREADs.

Return: Returns false if the channel is in reset.

Notice that if a false can be expected (reset is used), this must be treated as a special case in the responding SC_THREAD so no infinite loop is created.

Event: RequestEndEvent, if the response has not been pre-accepted, and is accepted with this call.

bool putSCmdAccept()

Purpose: Sets the SCmdAccept signal in the OCP channel and “releases” the request.

Return: Returns false if there is no request to accept or if the current request has already been accepted. Otherwise, `putSCmdAccept()` returns true and the request will be accepted on the next delta cycle. Note that after the command has been accepted, the OCP channel signal MCmd is then automatically reset to "`OCP_MCMD_IDLE`".

Event: RequestEndEvent, if there is an active request on the channel.

Void putSCmdAccept(bool accept = false)

Purpose: Sets or unsets the SCmdAccept signal in the OCP. Set can be called at any time during clock cycle, unset only at clock edge. Persistent once called.

Event: RequestEndEvent, if there is an active request on the channel.

void putSThreadBusy(unsigned int nextSThreadBusy)

Purpose: Sets the next value of the OCP signal SThreadBusy. This signal is updated at the end of the current delta cycle.

Return: None.

Event: None.

//Deprecated

void putNextSThreadBusy()

2.5.3 Response Phase

This section describes the methods for the slave’s TL1 response phase.

bool getMBusy() const

Purpose: Used to check whether a new response can be placed on the channel.

Return: Returns *true* if the channel is not free for a new response. This function does not check the threadbusy signal (if any). See also `getMThreadBusy()`.

Events: No event.

bool startOCPResponse(
const OCPResponseGrp<Td>& newResponse)

Purpose: Places the passed response onto the channel.

Return: Returns false if there is already a response on the channel which has not yet been accepted by the master, or if the OCP is a configured as mthreadbusy-exact and the OCP thread is busy, or if the channel is in reset.

Event: ResponseStartEvent. ResponseEndEvent, if the putMRespAccept(1) has been called before, or if the SRespAccept is not part of the channel. No event if return value is false.

Notice: Behavior changed from release 2.1.

bool startOCPResponseBlocking(
const OCPResponseGrp<Td>& newResponse)

Purpose: Repeat - try response - Wait for a rising clock edge until successful.

`startOCPResponseBlocking()` returns once the response has started but before the master has accepted the response.

Notice: Not to be used for modeling OCP interfaces with multiple threads. Use non-blocking instead. Not to be called from multiple SC_TREADs.

Return: Returns an immediate false if the channel is not clocked. Returns a false after a clock if the channel is in reset state. Reset is synchronous.

Notice that if a false can be expected (reset is used), this must be treated as a special case in the requesting thread so no infinite loop is created.

Event: ResponseStartEvent. ResponseEndEvent, if the putMRespAccept(1) has been called before, or if the SRespAccept is not part of the channel. No event if return value is false.

sc_event& RequestStartEvent()

Purpose: This event is triggered when a new request has been placed on the channel.

Return: RequestStartEvent.

sc_event& RequestEndEvent()

Purpose: This event is triggered when the request is accepted.

Return: RequestEndEvent.

unsigned int getMThreadBusy()

Purpose: Returns the current value of the MThreadBusy signal. This allows the slave to determine if a thread is busy before sending a response on that thread.
If mthreadbusy_pipelined is true in the OCP configuration, this method will return the MThreadBusy from the previous clock cycle.
For a clocked channel with mthreadbusy_pipelined false, this method

should never be called in the delta cycle immediately following a clock rising edge.

Return: The `unsigned int` returned has one bit for each thread in the channel. If a bit position is "1", that thread is busy.

Event: MThreadBusyEvent(void)

sc_event& MThreadBusyEvent( ) const

Purpose: This event is triggered when the channel changes the value returned by getMThreadBusy().

Return: The event associated with a change in MThreadBusy's value

sc_event& CurrentMThreadBusyEvent( ) const

Purpose: This event is triggered when the master changes the value of the MThreadBusy signal. The same as MThreadBusyEvent( ) unless mthreadbusy_pipelined is true.

Return: The event associated with a change in MThreadBusy's value

bool getMRespAccept()

Purpose: Get state of MRespAccept signal.

Note

Despite the name, this behaves like an RTL version of MRespAccept signal only after a request is put into the channel, and only at rising clock edge, that is only when MRespAccept is not don't-care according to OCP standard.

Return: Returns true, if respaccept parameter is 0, and `!getMBusy()` otherwise.

Event: No event.

sc_event& ResponseStartEvent()

Purpose: This event is triggered when a new response has been placed on the channel.

Return: ResponseStartEvent.

sc_event& ResponseEndEvent()

Purpose: This event is triggered when the response is accepted.

Return: ResponseEndEvent.

void waitMRespAccept(void)

Purpose: If there a current response on the channel, `waitMRespAccept()` waits until the response has been accepted by the master. This method returns immediately if there is no response on the channel or if that response has already been accepted. Note that if MRespAccept is not part of the

channel, this command will wait until the response is automatically accepted by the channel (one delta cycle after the response is submitted).

Return: None.

Event: None.

2.5.4 Data Handshake

This section describes the methods for the slave's TL1 data handshake.

bool getOCPDataHS(OCPDataHSGrp<Td>& myData,
bool acceptData = false)

Purpose: If there is an unread data handshake available on the channel, the data group is read and returned as `myData`. If `acceptData` is true then `putSDataAccept()` is called. Note that if `SDataAccept` is not part of the OCP channel, data is always automatically accepted during the next delta cycle, and the value of the `acceptData` parameter is ignored.

Return: Returns false if there is no data available or if the data has already been read by a getData command or if there is a `getDataBlocking` command in progress.

Event: None.

bool getOPCDataHSBlocking(OCPResponseGrp<Td>& myData,
bool acceptData = false)

Purpose: Waits for new, unread data to become available on the channel. The data is then read and returned as "myData." And if `acceptData` is true then `putSDataAccept()` is called. `getOPCDataHSBlocking()` returns once the data has been placed on the channel. Note that if the SDataAccept signal is not part of the OCP channel, data is always automatically accepted, and the value of the `acceptData` parameter is ignored.

Notice: Not to be used for modeling OCP interfaces with multiple threads. Use non-blocking instead. Not to be called from multiple SC_TREADs.

Return: Returns false if channel is in reset.

Notice that if a false can be expected (reset is used), this must be treated as a special case in the data thread so no infinite loop is created. (The thread must yield by using a wait statement.)

Event: None.

bool putSDataAccept()

Purpose: Sets the SDataAccept signal in the OCP channel and "releases" the data handshake.

Return: Returns false if there is no data to accept or if the current data has already been accepted. Otherwise, `putSDataAccept()` returns true and the data handshake will be accepted on the next delta cycle. Note that after the data has been accepted, the OCP channel signal MDataValid is automatically reset to false.

Event: DataHSEndEvent, if there is an active request on the channel.

bool putSDataAccept(bool accept = false)

Purpose: Sets or unsets the SDataAccept signal in the OCP channel. Set can be called at any time during clock cycle, unset only at clock edge. Persistent once called.

Return: Returns false if there is no data to accept or if the current data has already been accepted. Otherwise, `putSDataAccept()` returns true and the data handshake will be accepted on the next delta cycle. Note that after the data has been accepted, the OCP channel signal MDataValid is automatically reset to false.

Event: DataHSEndEvent, if there is an active request on the channel.

sc_event& DataHSStartEvent()

Purpose: This event is notified whenever any new data handshake data is placed on the channel.

Return: DataHSStartEvent.

sc_event& DataHSEndEvent()

Purpose: This event is notified when the current data handshake data is accepted by the slave.

Return: DataHSEndEvent.

void putSDataThreadBusy(unsigned int nextSDataThreadBusy)

Purpose: Sets the next value of the SDataThreadBusy signal on the channel. Each bit in the `nextSDataThreadbusy` parameter represents one thread in the channel. If a bit is “1” that means that the corresponding thread is now busy.

Return: No return value.

Event: None.

// Deprecated

void putNextSDataThreadBusy()

2.5.5 Timing Distribution Methods

This section describes methods implemented in the OCP TL1 channel to support timing distribution at end-of-elaboration.

void setOCPTL1SlaveTiming(OCP_TL1_Slave_TimingCl slave_timing)

Purpose: OCP slave must use this method to inform the channel of its timing parameters at end-of-elaboration, unless it conforms to default TL1 timing.

Return: None.

void registerTimingSensitiveOCPTL1Slave(OCP_TL1_Master_TimingIF *slave)

Purpose: Timing-sensitive OCP slaves may use this method to register themselves with the channel at end-of-elaboration. Once this has been done, all timing information provided by the master to the channel will be forwarded to the slave by the channel.

The pure virtual class OCP_TL1_Master_TimingIF contains only the single method setOCPTL1MasterTiming() which is also part of the OCP_TL1_Master_IF (see above).

Return: None.

2.5.6 OCP Configuration Management Methods

virtual void setOCPSlaveConfiguration(MapStringType& passedMap)

Purpose: OCP slave may use this method at end of elaboration to pass the configuration of its OCP port to the channel.

Return: None.

virtual void addOCPConfigurationListener(OCP_TL_Config_Listener& listener)

Purpose: OCP slave may use this method at end of elaboration to register itself as a configuration listener. After registration any changes to the OCP configuration of the channel, for example because the master sets the channel's configuration, are passed on to the listener (see definition of OCP_TL_Config_Listener class below).
Warning: if the channel has already been configured by the master when this is called, the listener will be informed of the configuration (called-back) before this method returns.
Warning: this method can be called multiple times during end of elaboration. The listener needs to ignore all but the last time it is called-back.
This listener should not be called-back after end-of-elaboration, if the channel is being correctly used.
This method is provided so that 'generic' OCP slaves can be implemented. A generic OCP slave is an OCP slave without a fixed OCP configuration, whose behaviour will adapt to the OCP configuration of the master.

Return: None.

virtual const std::string peekChannelName()

Purpose: Allows the master to find out the name of the channel, which simplifies the implementation of a 'generic' OCP slave with more than one OCP port.

Return: Channel name as std::string.

virtual OCPParameters *GetParameters()

Purpose: Simple access to the OCP parameters of the channel. This method should not be used until after end-of-elaboration, unless it is certain that the channel has been configuredhand.

Return: Pointer to Parameters object of the channel. This object contains all the OCP parameters, with the expected names and types as in the OCP protocol documentation, for example

```
bool wrap = my_OCP_port->GetParameters()->burstseq_wrap_enable;
```

2.6 OCP TL1 Timing Interface Classes

The pure virtual classes OCP_TL1_Slave_TimingIF and OCP_TL1_Master_TimingIF exist. These classes are defined in the header files ocp_tl1_slave_timing_if.h and ocp_tl1_master_timing_if.h respectively.

Each contains a single method.

OCP_TL1_Slave_TimingIF contains the method setOCPTL1SlaveTiming(), which is also part of the OCP_TL1_SlaveIF and documented in section 2.5.5 above.

OCP_TL1_Master_TimingIF contains the method setOCPTL1MasterTiming(), which is also part of the OCP_TL1_MasterIF and documented in section 2.4.5 above.

2.7 OCP TL1 Configuration Management Classes

In normal operation, an OCP master is connected to an OCP slave, both of which know their configuration of their OCP ports. Both of them inform the channel of their configuration and the channel verifies that they are compatible (see section 2.2). In some cases however it is convenient to connect a 'generic' master or slave; one that adapts its configuration and behaviour to the configuration of whatever it is connected to. For example a SystemC model of an OCP master might be able to generate WRAP bursts, but when connected to an OCP slave that does not support WRAP bursts it would instead use single accesses.

In such a case the generic core registers itself in the channel as a configuration-listener. The channel si then able to call it back to inform it of changes to the OCP configuration. A pure virtual class is provided for this purpose. A generic core should derive from this class and implement the method set_configuration, which is called by the channel whenever the configuration is revised by another core or the environment.

Example code for extracting parameter values from the passedMap can be found in the distribution.

```
class OCP_TL_Config_Listener {

public:

        virtual void set_configuration(MapStringType& passedMap,
                                       std::string channelName)=0;

};
```

2.8 OCP TL1 Monitor Interface

The OCP TL1 channel implements the OCP TL1 monitor interface. This allows monitors to be connected to the channel, for performance analysis, trace dumping, protocol checking and so on.

The methods of the monitor interface are listed below. Multiple monitors may be used in parallel on a single OCP TL1 channel. The basic principle is that the monitors poll the channel to find out what is happening. This implies that the monitors are synchronized with the OCP clock cycles. As there are some options (see section 12) around OCP clock cycle synchronization, a single monitor design may not be compatible with all uses of the OCP TL1 channel. In particular different monitors may be needed for the untimed and timed channels.

The methods of the interface are merely listed here. More detailed documentation of their meaning is available in the documentation of the available example monitors, in the monitor release package from OCP-IP.

```
template <typename TdataCl>
class OCP_TL1_MonitorIF :  virtual public sc_interface
{
public:

  typedef typename TdataCl::DataType    Td;
  typedef typename TdataCl::AddrType    Ta;
  typedef OCPRequestGrp<Td,Ta>          request_type;
  typedef OCPDataHSGrp<Td>              datahs_type;
  typedef OCPResponseGrp<Td>            response_type;
  typedef ParamCl<TdataCl>              paramcl_type;

  // Monitor access
  virtual  const OCPMCmdType getMCmdTrace ()   const = 0;
  virtual  const bool getMDataValidTrace ()    const = 0;
  virtual  const OCPSRespType getSRespTrace () const = 0;

  // port names
  virtual const std::string   peekChannelName()       const = 0;
  virtual const std::string   peekMasterPortName()    const = 0;
  virtual const std::string   peekSlavePortName()     const = 0;

  // transactions
  virtual const request_type&   peekOCPRequest()        const = 0;
  virtual const datahs_type&    peekDataHS()            const = 0;
  virtual const response_type&  peekOCPResponse()       const = 0;

  virtual const bool            peekRequestEnd()        const = 0;
  virtual const bool            peekRequestStart()      const = 0;
  virtual const bool            peekRequestEarlyEnd()   const = 0;

  virtual const bool            peekResponseEnd()       const = 0;
  virtual const bool            peekResponseStart()     const = 0;
  virtual const bool            peekResponseEarlyEnd()  const = 0;

  virtual const bool            peekDataRequestEnd()    const = 0;
  virtual const bool            peekDataRequestStart()  const = 0;
  virtual const bool            peekDataRequestEarlyEnd() const = 0;
```

```
  // thread busy
  virtual const unsigned int    peekSThreadBusy()       const = 0;
  virtual const unsigned int    peekSDataThreadBusy()   const = 0;
  virtual const unsigned int    peekMThreadBusy()       const = 0;

  // reset
  virtual const bool            peekMReset_n()          const = 0;
  virtual const bool            peekSReset_n()          const = 0;

  // sideband signals
  virtual const bool            peekMError()            const = 0;
  virtual const unsigned int    peekMFlag()             const = 0;
  virtual const bool            peekSError()            const = 0;
  virtual const unsigned int    peekSFlag()             const = 0;
  virtual const bool            peekSInterrupt()        const = 0;
  virtual const unsigned int    peekControl()           const = 0;
  virtual const bool            peekControlWr()         const = 0;
  virtual const bool            peekControlBusy()       const = 0;
  virtual const unsigned int    peekStatus()            const = 0;
  virtual const bool            peekStatusRd()          const = 0;
  virtual const bool            peekStatusBusy()        const = 0;
  virtual const bool            peekExitAfterOCPMon()   const = 0;


  // OCP paramertes
  virtual paramcl_type*         GetParamCl()                  = 0;
};
```

3 Overview of the OCP TL2

The OCP Transaction Level Two channel model is designed for architectural evaluation and modeling. The OCP TL2 channel works at a higher abstraction level than the TL1 channel. Instead of clocked cycle accurate support for all of the OCP signals, the OCP TL2 channel provides estimated timing as well as some signal abstraction to improve channel throughput and ease-of-use.

This chapter is an overview of OCP TL2 channel and of the two SystemC channel models that have been built to implement it. The sections below cover the differences between OCP TL1 and OCP TL2, and when to use the TL2 channel.

3.1 OCP TL1 vs OCP TL2

The OCP TL2 channel is meant to run faster and to be easier to use than the OCP TL1 channel. To achieve this goal, the OCP TL2 channel lacks the exact cycle accuracy of TL1, lacks timing enforcement, and simplifies the phase ordering of the channel. In addition, the OCP TL2 channel allows for burst-at-once which can greatly increase performance. Each of these topics is each covered below.

3.1.1 Event Driven Models

Unlike the OCP TL1 channel, the OCP TL2 channel is not explicitly clocked. A new TL2 request may be placed on the channel as soon as the previous request has been accepted. Thus, if the slave accepts each request immediately, the master is free to send a new request immediately. Other than the request/accept flow, the channel model does not enforce any timing. Instead, it is up to the core models attached to the OCP TL2 channel to provide the correct timing by sending their commands at the appropriate time.

For example, the slave should wait an appropriate amount of time before accepting a request to allow for the request and data to cross the channel. The OCP TL2 channel provides some helper functions to make this calculation easier for the cores.

One advantage of no channel clocking is that it allows a TL2 core to be completely event driven. A slave can be written “passively” with a SystemC SC_METHOD that is sensitive to the channel’s RequestStartEvent. Thus, the slave would only be activated when there is a new request on the channel to be processed. Such event driven models are more efficient and run much faster in SystemC than clocked models or models based on SC_THREADS.

3.1.2 No Separate Data Handshake

The OCP TL2 channel simplifies the interface by combining the request path with the data handshake path. In the OCP TL1 channel, these two paths are separate. As a result, a TL1 slave core must have three processes: one to handle incoming requests, one for incoming data, and a third to send back responses. In addition, the TL1 slave must buffer the incoming requests and then match the incoming data to the corresponding request.

This is simplified with the OCP TL2 channel. Requests and data are always sent together. This means that a TL2 slave need only have two processes: one to receive requests and another to send responses. Additionally, there is no longer any overhead in trying to match data back to a request.

The downside of sending data and responses together is that some OCP timing information may be lost. Specifically, the actual hardware master may send data one or more cycles after a request. This behavior may be modeled directly in OCP TL1 by having the TL1 master model send a request in one cycle and data in the next. However, this cannot be modeled directly with the OCP TL2 channel since data and request are always sent together. The OCP TL2 channel partially solves this problem by providing a timing point “RqDL” set by the master that specifies the latency “L” between when the request “Rq” starts on the channel and when the data “D” starts on the channel.

3.1.3 Simpler Phase Timing

The OCP TL1 channel employs a set of checks to ensure that the data flow through the channel exactly follows the ordering rules of the OCP specification. In addition, the OCP TL1 channel uses delta cycles and delayed request/update schemes to give each phase its own delta cycle. This careful phase tracking is not needed for most OCP communications, especially when there is no separate data handshake phase. Thus, it is not used for OCP TL2.

In the OCP TL2 channel, the next request may be sent as soon as the previous request has finished. The channel does not check and does not enforce that the next request should wait until the next OCP cycle. This makes the OCP TL2 channel faster, but it does put some of the timing burden on the TL2 core writer.

3.1.4 Burst at Once

The greatest performance advantage of the OCP TL2 channel over the TL1 channel is the ability to send bursts with a single command. In order to send a write burst of length eight over the OCP TL1 channel, the master must send eight individual write requests (and possibly eight individual write data handshakes) in order to get the full burst across the channel. With the OCP TL2 channel, a single command sends the whole burst request at once. This eliminates much of the overhead and greatly improves the throughput of the channel.

3.1.5 Passing Pointers

The OCP TL2 channel achieves its “burst at once” commands through the use of pointers. While write data and read data responses are sent one at a time in OCP TL1, they are sent as an array in the OCP TL2. Thus, instead of a data word, the OCP TL2 channel passes a pointer to the first data word in an array of data words. Of course, the OCP TL2 channel can also be used to send writes and reads of single data words. In this case, the data pointer is still used but it only points to a single data word (or an array of one).

Any time pointers are used, it is important to establish who owns the memory pointed to by the pointer. For the OCP TL2 channel, the memory is owned by the core sending the request or the response. The values pointed to by the pointer are to remain valid until the request (or response) is accepted by the other side.

For example, the master wants to send an eight-word burst write command over the OCP TL2 channel. The master creates an array of data at least eight words long. The Master then copies the data to be written into the new array. The master then makes a request group and sets the “MDataPtr” value to the data array. The Master then calls “sendOCPRequest” to place the request on the channel. At this point, the master is committed to keeping the data array valid and constant until the request is over and the master receives the RequestEndEvent from the channel.

One “gotcha” to look out for: avoid using data arrays that are automatic variables as they will get automatically deleted at the end of the function call they were defined in. Rather, it is safer to use an array just for sending data that is a class data member. That way the array will not be deleted and can be reused for each request.

3.2 Using the OCP TL2 Channel

When used as intended, the OCP TL2 channel can give increased performance with close to cycle accurate timing. The following guidelines will help to get the best results from the channel.

3.2.1 Timing

The OCP TL2 channel does not have cycle accurate timing as the OCP TL1 channel does nor does it have the timing and ordering checks that are built into TL1. However, it is possible to get quite accurate timing when using the TL2 channel, as long as the underlying OCP connection is understood and followed.

The timing of the channel is set by the two cores that are connected to it. Anytime that there is no request on the channel, the channel allows the master to send a new request. The channel will then not allow another request until the current request has been accepted by the slave. Thus, it is “accept” functions that drive the timing of the channel.

It is up to the slave to calculate how long it would take the request to cross the channel and how long it would then take the slave to process the request. The slave should then accept the request after that length of time has elapsed. Similarly, it is up to the master to calculate how long it would take a response to cross the OCP connection and additionally how long it would take for the master to process it and be ready for a new response. The master should then accept the response after that length of time.

The OCP TL2 channel provides delayed accept functions to make this easier. In addition, there are timing variables and helper functions that can automatically calculate the OCP timing of a request or response.

3.2.2 Events

In order to get the best performance from the OCP TL2 channel, it is advisable to make the cores connected to it event-driven. In general, an OCP TL2 core should have an SC_METHOD for sending and another for receiving. Each method should be sensitive to an OCP TL2 event.

For example, a master would have a method for sending new requests that is sensitive to the channel’s RequestEndEvent. When the previous request has been accepted by the slave, the channel triggers the RequestEndEvent. A method in master that is sensitive to this event will then be activated and it can send a new request to the channel.

The OCP TL2 channel also supports blocking calls that must be used with an SC_THREAD process. However, an SC_THREAD process is slower than an SC_METHOD and developers interested in greater model performance should aim for an event driven simulation.

3.2.3 OCP Burst Signals

The OCP specification includes a collection of signals that specify the details of each individual transfer of an OCP burst transaction. While these signals are certainly useful at the hardware level and at the cycle accurate TL1 level, their use is less clear for an OCP TL2 connection which allows an entire burst to be sent as a single command. This section covers the OCP burst signals as a group and then gives guidelines for specific burst signals as well.

As a group, the OCP Burst signals are meant to help through the individual transfers of a burst. Many of them change with each of individual request or response of the burst. For example, for imprecise bursts, the MBurstLength may count down as there are fewer and fewer requests left to send in the burst. Since the OCP TL2 channel allows an entire burst in a single command, how does one set a burst signals that changes throughout the burst?

One simple solution is to ignore the burst signals altogether when using the OCP TL2 channel. The OCP TL2 API provides the basic signaling needed for burst-at-once transactions. The request group has a pointer to the entire burst of data, there is a field for how many requests are in this burst command (DataLength) as well as a flag to indicate whether or not this command is the last OCP TL2 command in the burst (LastOfBurst). These fields are enough to send bursts over the TL2 channel either as a single command or as a set of commands.

But the OCP burst signals do have there place in the OCP TL2 channel, especially when the OCP TL2 channel is used to send bursts one request/response at a time. Here are the guidelines for each of the OCP burst signals.

3.2.4 DataLength

This is the number of write data words in the OCP TL2 write request, the number of data words to read in an OCP TL2 read request, and the number of data words in an OCP TL2 response. The DataLength field gives the number of data words in the array pointed to by MDataPtr or SDataPtr.

Note that DataLength applies to the command and not necessarily to the whole burst. For example, say that master wanted to send at 16 burst read request to the slave. If the master sent it as a single command, then DataLength=16. If the master sends the burst request in two parts, the first OCP TL2 burst request might have a DataLength=8, and the second might have DataLength=8 as well. If the master wanted to send the burst as sixteen separate requests, then each of the requests would have DataLength=1.

The DataLength field is required, even when its value is one. This is because the DataLength field is needed to dereference the pointers that passed with the OCP TL2 request or response.

3.2.5 LastOfBurst

The LastOfBurst field indicates that this command is the last command of the burst. It is part of both the request group and the response group. If the entire burst is being sent as a single command, then LastOfBurst=true as this is the first and last command of the burst. If the burst is sent in two commands, then LastOfBurst=false for the first part and LastOfBurst=true for the second part.

3.2.6 MBurstSeq

This field sets how the addressing is to be done for each data word in the burst. It is has the same meaning in the OCP TL2 channel as it does in the hardware.

3.2.7 MBurstPrecise & MBurstLength

This field is indicates whether the total length of the burst is known. If MBurstPrecise=true, then the MBurstLength field contains the total length of the burst and if MBurstPrecise=false then MBurstLength indicates how many data words might be remaining in the burst.

If entire bursts are being sent as single commands, then these fields are not useful for the TL2 core writer as the total burst length is know when the command is received. However, these fields may be useful when a burst is sent as a set of several commands. In the case of MBurstPrecise=true, the field MBurstLength contains the total number of data words in the whole burst, while DataLength contains the number of data words in a particular individual request or response.

For example, the master wants to send a precise 16 word write request to the slave through the OCP TL2 channel. Instead of sending the whole request at once, the master instead sends it as three separate request commands: the first with 6 data words, the second with 6 data words and the last with 4 data words. Here are the values for MBurstPrecise, MBurstLength, DataLength, and LastOfBurst fields for these three TL2 request commands that together make up the 16-word write burst:

Request #1: MBurstPrecise=true, MBurstLength=16, DataLength=6, LastOfBurst=false.

Request #2: MBurstPrecise=true, MBurstLength=16, DataLength=6, LastOfBurst=false.

Request #3: MBurstPrecise=true, MBurstLength=16, DataLength=4, LastOfBurst=true.

3.2.8 MBurstSingleReq

This OCP field specifies that the transfer is to be sent with a single request phase for a burst of length N (SRMD semantics). An SRMD (Single Request, Multiple Data) read crosses in only one transfer while an MRMD (Multiple Requests, Multiple Data) read will have one transfer per data word.

Note that when MBurstSingleReq=1 (SRMD semantics), sending a burst in chunks smaller than MBurstLength can still make sense for a write as it would specify different timing to issue data, but for a read the burst must be sent as a single request with DataLength matching MBurstLength. Responses to an SRMD read request can still be issued by the slave in multiple chunks amounting to the request's MBurstLength. This is not applicable for write requests with response, as OCP specifies that a single response is expected for an SRMD write burst. Thus the response to any write request with MBurstSingleReq=1 must have DataLength=1.

The timing helper functions in the OCP TL2 channel take the SRMD or MRMD nature of the burst into account.

3.2.9 MAtomicLength

This OCP field is not used in the OCP TL2 channel.

3.2.10 MReqLast

This field indicates that this is the last write word request or the last read word request of the burst. This signal is not very helpful when whole bursts and chunks of bursts may be sent at once. In the OCP TL2 channel, the field LastOfBurst is used instead.

3.2.11 SRespLast

This field indicates that this is the last response word of the burst. This signal is not very helpful when whole bursts and chunks of bursts may be sent at once. In the OCP TL2 channel, the field LastOfBurst is used instead.

3.3 Benchmarking the Channels

The benchmark tests below show that the OCP TL2 channel gets its greatest performance boost over the TL1 channel when bursts are sent as single commands.

3.3.1 Overview of the Benchmark Tests

In each of these tests, a simple master is connected to a simple core through the OCP channel model. The channel and the cores use Td (data type) & Ta (address type) = unsigned int. The OCP Channel has data handshake with command, data, and response accept. For writes, the command goes first and then the data goes in the next cycle.

The TL1 master uses one thread method (to send requests), the rest of the TL1 master and all of the TL1 slave model is event driven. The TL2 master and slave models are all event driven.

These tests were run on a dual processor Pentium III 1.26GHz machine. All simulations ran on a single processor. The tests were compiled under Linux with gcc 2.96 using the "-O" flag and the standard OSCI SystemC library.

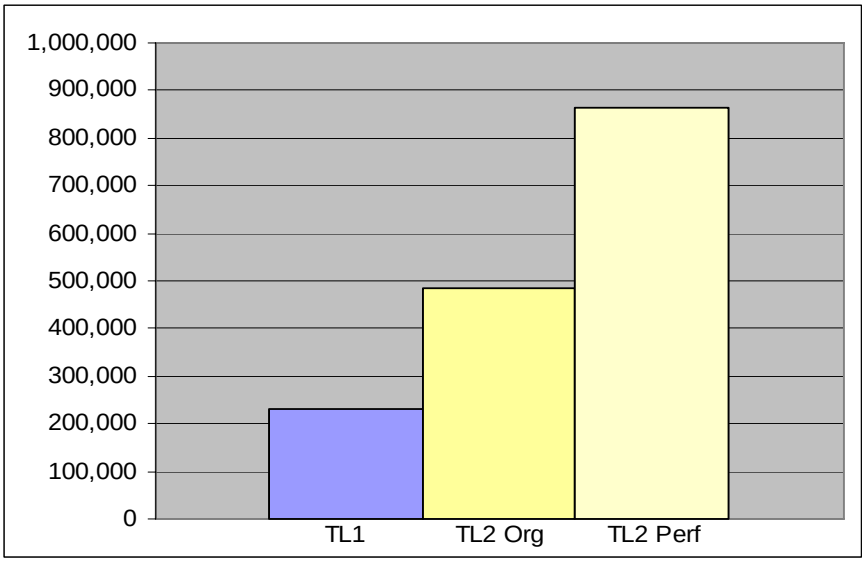
3.3.2 Single Data Word Writes and Reads

The first test is a single data word write command followed by a single data word read. This sequence is looped through 10,000,000 times.

Table 9 Single word reads and writes

Model	Run 1 (s)	Run 2 (s)	Run 3 (s)	Avg Time (s)	Data Words / sec
TL1	86.37	86.26	85.80	86.14	232,171
TL2 original	41.11	41.14	41.14	41.13	486,263
TL2 performance	23.19	23.13	23.08	23.13	864,553

Figure 4 Throughput (Data Words/sec) for single writes and reads



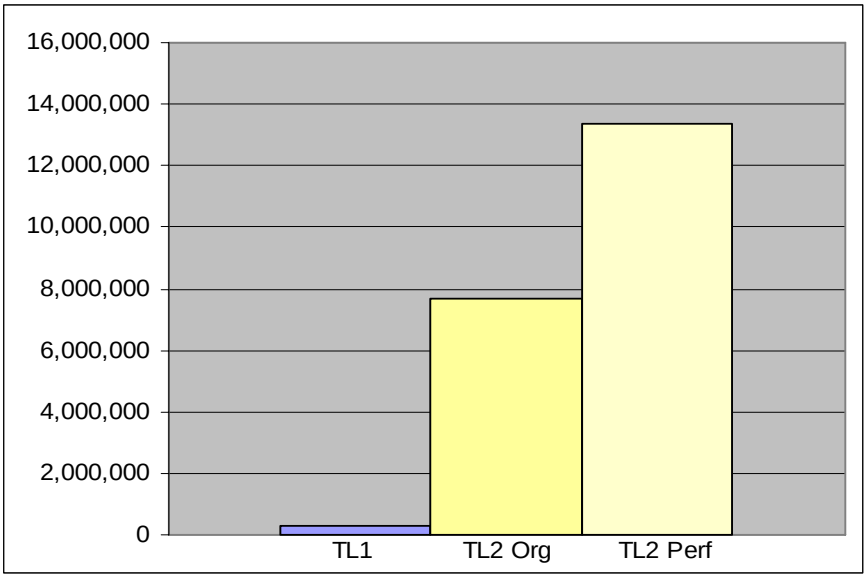
3.3.3 Burst Writes and Reads

The second test is a burst write of 16 data words followed by a burst read of 16 data words. The sequence is looped through 1,000,000 times for TL1 and 10,000,000 times for the TL2 channels.

Table 10 Burst writes and reads

Model	Run 1	Run 2	Run 3	Avg Time	Data Words / sec	Notes
TL1	115.54	116.07	115.36	115.66	276,681	1,000,000 loops
TL2 original	41.58	41.57	41.72	41.62	7,687,996	10,000,000 loops
TL2 performance	23.96	23.90	23.88	23.91	13,381,656	10,000,000 loops

Figure 5 Throughput (Data Words/sec) for burst 16 writes and reads



4 OCP TL2 Channel Model

4.1 Data Structures for the OCP TL2 Channel

The following data classes are used to pass requests and responses through the OCP TL2 channel. These classes also contain conversion functions and constructors for compatibility with the original TL2 channel.

4.1.1 OCPTL2RequestGrp Template Class

The `OCPTL2RequestGrp` class is used for sending and receiving OCP TL2 burst requests. This template class is defined as

```
Template<class Td, class Ta>
class OCPTL2RequestGrp
```

Where Td is the data type and Ta is the address type.

4.1.1.1 Data Type and Address Type

The class template parameters `Td` and `Ta` indicate the data type and address type of the MDataPtr and MAddr signals, respectively. By making this a template, any sized data or address width may be supported.

4.1.1.2 Members

Some configurations of the OCP will not use all of the members in the class. In that case, the unused members are invalid and should not be referenced or used. The table below lists the member names and their data types for OCPTL2RequestGrp.

Table 11 OCPTL2RequestGrp Members

Name	Data Type	Description
MCmd	OCPMCmdType	Master command
MAddr	AddrType	Address of first data word of the request. According to the OCP specification MAddr is supposed to be a byte address that must be aligned with the OCP word size.
MAddrSpace	unsigned int	Master address space
MDataPtr	DataType*	Pointer to the first word of write data for the request. A burst of write data should be an array. If this is a read request, then MDataPtr = NULL. According to section 3.1.5 the data memory is owned by the master and only guaranteed to be valid until the slave has accepted the request.

Name	Data Type	Description
DataLength	unsigned int	The number of data words in this request. If this is a write request, then DataLength is the number of data words in the array pointed to by MDataPtr. If this is a read request then DataLength is the number of data words to be read.
MByteEn	unsigned int	Master byte enable field. Use this value if the (optional) MByteEnPtr is set to NULL.
MByteEnPtr	unsigned int*	Pointer to an array of byte enable fields. The length of the array pointed to should be DataLength long. Each ByteEn field is to be used for the corresponding data word in the MDataPtr array. If the byte enable remains constant throughout the burst, set MByteEnPtr=NULL and use MByteEn instead.
MDataInfo	unsigned long long int	Extra information sent with the write data
MDataInfoPtr	unsigned long long int*	Pointer to an array of data info fields. The length of the array pointed to should be DataLength long. Each MDataInfo field in the array is to be used for the corresponding data word in the MDataPtr array. If the DataInfo remains constant throughout the burst, set MDataInfoPtr=NULL and use MDataInfo instead.
MThreadID	unsigned int	Master thread identifier
MConnID	unsigned int	Master connection identifier
MTagID	unsigned int	Master tag identifier (see OCP 2.1 standard)
MTagInOrder	bool	If true, force tag-in-order (see OCP 2.1 standard)
MReqInfo	unsigned long long int	Extra information sent with the response.
MBurstLength	unsigned int	If MBurstPrecise=true, this is the total length of the OCP burst. Note that is the OCP burst is sent as several requests then MBurstLength will be equal to the sum of the DataLength's of each of the requests.
MBlockHeight	unsigned int	For 2-D bursts
MBlockStride	unsigned int	For 2-D Bursts
MBurstPrecise	bool	Given burst length is precise
MBurstSeq	OCPMBurstSeqType	Address sequence of burst
LastOfBurst	bool	Is this burst request the last request of the OCP burst?

Some notes on the usage of these fields:

DataLength

This is always the length of this chunk of the request burst. DataLength gives the length of the arrays pointed to by each of the pointer variables in the request structure.

For read requests, the DataLength field indicates how many data words are to be read as part of this TL2 request.

MByteEn & MbyteEnPtr

The MByteEnPtr is an array of byte enable fields, one byte enable for each OCP data word. The MByteEnPtr allows accurate simulation of a channel where the byte enable value is changed with each data word of the burst. If MByteEn changes with each data word, then the MByteEnPtr should be set and the MByteEn field should be ignored. If, however, the MByteEn stays constant then the MByteEnPtr should be set to NULL (the default) and the MByteEn field should be used.

MDataInfo & MdataInfoPtr

The MDataInfoPtr is an array of data info fields, one data info value for each OCP data word. Just as with the MByteEnPtr, the MDataInfoPtr allows accurate simulation of a channel where the data info value is changed with each data word of the burst. If data info changes with each data word, then the MDataInfoPtr should be set and the MDataInfo field should be ignored. If, however, the MDataInfo stays constant then the MDataInfoPtr should be set to NULL (the default) and the MDataInfo field should be used.

MBurstPrecise & MBurstLength

These fields allow for the specification of precise bursts in TL2. When MBurstPrecise is true, the MBurstLength field should contain the total length of the OCP burst request.

For example, if a precise OCP burst write request of 16 data words were sent as three TL2 requests, each request could have the following fields:

Request #1: MBurstPrecise = true; MBurstLength = 16; DataLength = 8; LastOfBurst=false;

Request #2: MBurstPrecise = true; MBurstLength = 16; DataLength = 4; LastOfBurst=false;

Request #3: MBurstPrecise = true; MBurstLength = 16; DataLength = 4; LastOfBurst=true;

If the same OCP burst write of 16 data words were sent as one TL2 request, then the TL2 request would be:

Request: MBurstPrecise = true; MBurstLength = 16; DataLength = 16; LastOfBurst=true;

Note the difference between MBurstLength and DataLength: DataLength is required and specifies that number of data words in this TL2 request, MBurstLength is used for precise bursts and always holds the total number of data words in all of the TL2 requests that make up the burst.

4.1.2 OCPTL2ResponseGrp Template Class

The `OCPTL2ResponseGrp` class is used to send and receive OCP TL2 burst responses with the OCP TL2 channel. This template class is defined as

```
Template<class Td>
OCPTL2ResponseGrp
```

4.1.2.1 Data Type

The class template parameter Td indicates the data type of the SDataPtr signal. This allows the response to contain any size of data. Note that the type of the response data must match the type of request data.

4.1.2.2 Members

Some configurations of the OCP will not use all of the members in the class. This corresponds to the fact that some OCP implementations do not use all of the OCP signals. In that case, the unused members are invalid and should not be referenced or used. The table below lists the names and their data types of `OCPTL2ResponseGrp`.

Table 12 OCPTL2ResponseGrp Member Types

Name	Type	Description
SResp	OCPSRespType	Slave response code
SDataPtr	DataType*	Pointer to the data words returned by slave. This should be an array of data words that is DataLength long. Note that for responses without data, such as write acknowledgement responses, SDataPtr=NULL. According to section 3.1.5 the data memory is owned by the slave and only guaranteed to be valid until the master has accepted the response.
DataLength	unsigned int	The number of data words in this response. If this response does not contain any data words, then DataLength=0.
SThreadID	unsigned int	Slave thread identifier
STagID	unsigned int	Slave tag identifier (see OCP 2.1 standard)
STagInOrder	bool	Force tag-in-order (see OCP 2.1 standard)
SDataInfo	unsigned long long int	Extra information about the response data.
SRespInfo	unsigned long long int	Extra information sent out with the response.
LastOfBurst	bool	Is this the last response of the OCP burst? The OCP burst may be sent as one or as several separate responses.

DataLength is always the length of this chunk of the response burst. DataLength gives the length of the array pointed to the SDataPtr.

4.1.3 Timing Values

For ease of use, the timing values are organized into two groups: the master timing group and the slave timing group. As the name implies, the values in the master timing group are set by the master and the values in the slave timing group are set by the slave.

4.1.3.1 Master Timing Variables

These timing values are set by the master side of the OCP connection.

Table 13 OCP TL2 Master Timing Variables Structure

MTimingGrp:	
int RqDL	Request Data Latency
int RqSndI	Request Send Interval
int DSndI	Data Send Interval

int RpAL	Response Accept Latency

The Master Timing Variables are defined as follows. The Request Data Latency (RqDL) is the number of cycles between the start of a write request and the start of the corresponding data that is a associated with that write request. This variable only applies to write requests on channels with data handshake.

The Request Send Interval (RqSndI) is the number of cycles between read requests in a read burst when the master is connected to a very fast slave. This is the fastest that the master can send read requests. If RqSndI is set to 1, this means that the master is capable of sending out a read request every cycle. A RqSndI of 3 means that the master is much slower and only able to issue a read request every three cycles.

The Data Send Interval (DSndI) is the number of cycles between the data words in a burst write request. In the case of data handshake, this is the distance between the data words on the data path through the channel. In the case of no data handshake, this is the number of cycles between the write requests (that contain the data words). A DSndI of 1 means the master can send a new write data word every single cycle. A DSndI of 2 means that the master can send a new write request only ever other cycle.

The Response Accept Latency (RpAL) indicates how long the master will wait to accept a response from the slave after the response arrives on the channel. An RpAL of 1 means the master can accept a response every single cycle. An RpAL of 10 means that the master is much slower and only able to process a new response every 10 cycles.

4.1.3.2 Slave Timing Variables

Table 14 OCP TL2 Slave Timing Variables Structure

STimingGrp	
int RqAL	Request Accept Latency
int DAL	Data Accept Latency
int RpSndI	Response Send Interval

The Slave Timing Variables are similar to the Master's timing variables and are defined as follows. The Request Accept Latency (RqAL) is the minimum number of cycles between read requests required by the slave. A very fast slave would have an RqAL of 1 which would meant that the slave could process and accept a read request every cycle. A slower slave might have an RqAL of 4 which means that the slave can only handle a new read request every 4 cycles.

The Data Accept Latency (DAL) variable defines the minimum interval between the data words of a write request burst. It specifies how many cycles the slave requires to accept each data word of a write request burst. A DAL of 1 means the slave is capable of processing a new write request every 1 cycles (every cycle).

Finally, the Response Send Interval (RpSndI) gives the number of cycles between responses if the slave were connected to a very fast master. That is, if the slave were able to run at full speed, how many cycles would there be between responses? A RpSndI of 4 indicates that the slave could send a new response every 4 cycles, while a RpSndI of 1 means that the slave is capable of sending a new response every cycle.

4.2 Building the OCP TL2 Channel

4.2.1 Constructor

The OCP TL2 channel has the following constructor:

OCP_TL2_Channel(sc_module_name name,
ostream *traceStreamPtr=NULL)

Name
Name of the module (channel) instance.

TraceStreamPtr
Pointer to an output stream to use to print debugging information.

4.2.2 Configuring the Channel Clock Period

The OCP TL2 channel operates in integer cycles. To set the length of the clock period of an OCP channel cycle, use the following command:

void setPeriod(const sc_time& clkPer)

Sets the time taken by one OCP cycle period. Only called from the “outside.” Not called by master or slave.

4.2.3 Setting the Parameters

The Tl2 channel is configured in the same way as the TL1 channel. The same options are available, namely configuration from the environment or configuration from the master and slave cores, with the same callback mechanism to allow the implementation of generic cores. In fact the classes and methods used are exactly the same as for TL1, with one exception. For full details see sections 2.2 and 2.7. The exception is that in TL2, the channel method getParamCl returns a pointer to OCPParameters rather than a pointer to ParamCl as it does in TL1.

Briefly, the parameters of the channel are set using a string to string map.

void setConfiguration(MapStringType& passedMap)

void setOCPMasterConfiguration(MapStringType& passedMap)

void setOCPSlaveConfiguration(MapStringType& passedMap)

Where passedMap is a map< string, string> where the left side string is the parameter name (as defined in the OCP specification) and the right side is in the form of “type:value” where type is “i” for integer or “s” for string.

4.3 OCP TL2 Master Interface Methods (ocp_tl2_master_if.h)

The methods described in this section handle the OCP TL2 channel interface for a master core model.

API Function	Description
Request Commands	

bool sendOCPRequest(OCPTL2RequestGrp Rq)	Puts an OCP TL2 request on the channel. Returns true if the request was successfully placed on the channel. False otherwise.
bool sendOCPRequestBlocking(OCPTL2RequestGrp Rq);	Puts an OCP TL2 request on the channel, waiting until the channel is free if necessary. Waits until the slave accepts the request and then returns. Blocking calls may only be called from SC_THREAD processes.
bool requestInProgress()	True is there is currently an active request on the channel.
Response Commands	
bool getOCPResponse(OCPTL2ResponseGrp& Resp)	Gets a new response from the channel and returns true. Returns false if no new response transaction available.
bool getOCPResponseBlocking(OCPTL2ResponseGrp& Resp)	Waits for a new, unread OCP TL2 response to come on to the channel and then gets it. Can only be called from an SC_THREAD process. Notice: Not to be used for modeling OCP interfaces with multiple threads. Use non-blocking instead. Not to be called from multiple SC_TREADs.
bool acceptResponse()	Accepts the response immediately and returns true. Returns false if no response to accept.
bool acceptResponse(const sc_time& accept_time)	Accepts the response in the future, accept_time SystemC time units from now. Returns false if no response to accept.
bool acceptResponse(int cycles)	Accepts the response in the future, cycles OCP cycle periods from now. If cycles=0 then the accept is immediate. If cycles=-1 then the response is accepted after the current values of the timing variables indicate that it should have completed. That is, if cycles < 0 then cycles = getTL2RespDuration(); Returns false if no response to accept.
bool responseInProgress()	True if there is currently an active response on the channel.
ThreadBusy Commands	
putMThreadBusyBit(bool value, unsigned int ThreadID);	Sets MThreadBusy thread bit # ThreadID to value.
bool getSThreadBusyBit(unsigned int ThreadID);	Returns the value of SThreadBusy bit # ThreadID.
Channel Timing Functions	
const sc_time& getPeriod(void) const	Get the time taken by one OCP cycle period in the channel.
Timing Value Functions	
putMasterTiming(MTimingGrp mTimes);	Set new values for all of the master timing variables.
getMasterTiming(MTimingGrp& mTimes);	Get the current values for all of the master timing variables.

getSlaveTiming(STimingGrp& sTimes);	Get the current values for all of the slave timing variables.
Timing Helper Functions	
int getWDI();	Gets the Write Data Interval, the number of cycles between data words in a write request. Called by master or slave. If the channel does not have a data handshake path, this function returns the number of cycles between write requests. (Note that this value is calculated from Master Data Send Interval and Slave Data Accept Interval).
int getRqI();	Gets the Read Request Interval, the number of cycles between the individual read requests in a read request burst. Called by master or slave.
int getTL2ReqDuration();	The estimated minimum number of cycles the current request will be on the channel. This value is computed from the timing values as well as from the channel configuration. Called by master or slave.
int getRDI();	Gets the Response Data Interval, the number of cycles between data words in a read response. Called by master or slave. Note that this value is computed from timing values set by the master and slave as well as by from the channel configuration.
int getTL2RespDuration();	The estimated minimum number of cycles the current response will be on the channel. This value is computed from the timing values, the number of data words in the response and the channel configuration. Called by master or slave.
void setOCPMasterConfiguration(MapStringType& passedMap)	Method for the master to inform the channel of the configuration of its OCP port during end-of-elaboration
void addOCPConfigurationListener(OCP_TL_Config_Listener& listener)	Method for the master to register itself to be called-back when the channel configuration changes, for example when set by the slave.
const std::string peekChannelName()	Method to get the channel name, needed when a multiple-port configuration listener is called back, to distinguish the OCP port to which the callback refers.

4.4 OCP TL2 Slave Interface Methods (ocp_tl2_slave_h)

The methods described in this section handle the OCP TL2 channel interface for a slave core model.

API Function	**Description**
Request Commands	
bool getOCPRequest(OCPTL2RequestGrp& Rq)	Gets a new request from the channel and returns true, otherwise returns false if no new request available.
bool getOCPRequestBlocking(OCPTL2RequestGrp& Rq)	Gets a new request from the channel if available, otherwise waits for a new request and then gets it. Notice: Not to be used for modeling OCP interfaces with multiple threads. Use non-blocking instead. Not to be called from multiple SC_TREADs.

bool acceptRequest(void)	Accepts the request immediately and returns true. Returns false if no request to accept.
bool acceptRequest(const sc_time& accept_time)	Accepts the request in the future, accept_time SystemC time units from now. Returns false if no request to accept.
bool acceptRequest(int cycles)	Accepts the request in the future, cycles OCP cycle times from now. If cycles=0 then the accept is immediate. If cycles=-1 then the request is accepted after the timing points indicate that it should have completed. That is, if cycles < 0 then cycles = getTL2ReqDuration(); Returns false if no request to accept.
bool requestInProgress()	True is there is currently an active request on the channel.
Response Commands	
bool sendOCPResponse(OCPTL2ResponseGrp Resp)	Puts an OCP TL2 response on the channel. Called by slave. Returns true if channel was open for a new response. False otherwise.
bool sendOCPResponseBlocking(OCPTL2ResponseGrp Resp)	Waits for the OCP TL2 response channel to become free. Puts an OCP TL2 response on the channel.
bool responseInProgress()	True if there is currently an active response on the channel.
ThreadBusy Commands	
bool getMThreadBusyBit(unsigned int ThreadID);	Returns the value of MThreadBusy bit # ThreadID.
putSThreadBusyBit(bool value, unsigned int ThreadID);	Sets SThreadBusy bit # ThreadID to value.
bool getSThreadBusyBit(unsigned int ThreadID);	Returns the value of SThreadBusy bit # ThreadID.
Channel Timing Functions	
const sc_time& getPeriod(void) const	Get the time taken by one OCP cycle period in the channel.
Timing Value Functions	
getMasterTiming(MTimingGrp& mTimes);	Get the current values for all of the master timing variables.
putSlaveTiming(STimingGrp sTimes);	Set new values for all of the slave timing variables.
getSlaveTiming(STimingGrp& sTimes);	Get the current values for all of the slave timing variables.
Timing Helper Functions	

int getWDI();	Gets the Write Data Interval, the number of cycles between data words in a write request. Called by master or slave. If the channel does not have a data handshake path, this function returns the number of cycles between write requests. (Note that this value is calculated from Master Data Send Interval and Slave Data Accept Interval).
int getRqI();	Gets the Read Request Interval, the number of cycles between the individual read requests in a read request burst. Called by master or slave.
int getTL2ReqDuration();	The estimated minimum number of cycles the current request will be on the channel. This value is computed from the timing values as well as from the channel configuration. Called by master or slave.
int getRDI();	Gets the Response Data Interval, the number of cycles between data words in a read response. Called by master or slave. Note that this value is computed from timing values set by the master and slave as well as by from the channel configuration.
int getTL2RespDuration();	The estimated minimum number of cycles the current response will be on the channel. This value is computed from the timing values, the number of data words in the response and the channel configuration. Called by master or slave.
void setOCPSlaveConfiguration(MapStringType& passedMap)	Method for the slave to inform the channel of the configuration of its OCP port during end-of-elaboration
void addOCPConfigurationListener(OCP_TL_Config_Listener& listener)	Method for the slave to register itself to be called-back when the channel configuration changes, for example when set by the master.
const std::string peekChannelName()	Method to get the channel name, needed when a multiple-port configuration listener is called back, to distinguish the OCP port to which the callback refers.

4.5 OCP TL2 Channel Events

The methods described in this section handle give access to the events generated by the OCP TL2 channel. While most events are available to both the master and the slave, some events are meant for only one side or the other and when this is the case it is indicated in the table below.

API Event Function	**Description**
DataFlow Events	
sc_event& RequestStartEvent()	Event finder for the channel event that is triggered when a new request is placed on the channel.
sc_event& RequestEndEvent()	Event finder for the channel event that is triggered when the request is accepted by the slave and the channel is released.
sc_event& ResponseStartEvent()	Event finder for the channel event that is triggered when a new response is placed on the channel.

sc_event& ResponseEndEvent()	Event finder for the channel event that is triggered when the response is accepted by the master and the channel is released.
ThreadBusy Events	
sc_event& MThreadBusyEvent()	Event finder for the channel event that is triggered whenever MThreadBusy signal changes. This event finder is available to the Slave only.
sc_event& SThreadBusyEvent()	Event finder for the channel event that is triggered whenever SThreadBusy signal changes. This event finder is available to the Master only.
Channel Timing Events	
sc_event& MasterTimingEvent()	Event finder for the channel event that is triggered whenever the master's timing variables are changed. This event finder is available to the slave only.
sc_event& SlaveTimingEvent()	Event finder for the channel event that is triggered whenever the slave changes its timing variables on the channel. This event finder is available to the master only.
Sideband Signal Events	
sc_event& SidebandMasterEvent()	Event finder for the event that is triggered whenever the master changes one of its sideband signals. This event finder is available to the slave only.
sc_event& SidebandSlaveEvent()	Event finder for the event that is triggered whenever the slave changes one of its sideband signals. This event finder is available to the master only.
sc_event& SidebandCoreEvent()	Event finder for the event that is triggered whenever the "Core" side of the OCP connection changes one of its sideband signals. This event finder should be used by the "System" side only.
sc_event& SidebandSystemEvent()	Event finder for the event that is triggered whenever the "System" side of the OCP connection changes one of its sideband signals. This event finder should be used by the "Core" side only.

4.6 Reset

The OCP TL2 channel has limited reset support. The reset commands set and unset the reset flags in the channel. *They do not change or reset the current state of the channel.* Nor do they interrupt blocking commands. If a reset signal is desired, then it is up to the master and slave cores to take appropriate action by immediately accepting outstanding requests and responses and refraining from sending any new requests or responses until the reset is over.

Reset API Function	**Description**
sc_event& ResetStartEvent()	Event finder for the channel event that is triggered when a reset is asserted on the channel.
sc_event& ResetEndEvent()	Event finder for the channel event that is triggered when a reset is ended on the channel.
bool getReset()	Checks if channel is in reset state. Returns *true* if the channel is in reset, false otherwise. Called by the master or slave.

void MResetAssert()	Called by the master only. Sets the MReset_n flag to false. Triggers the ResetStartEvent.
void MResetDeassert()	Called by the master only. Sets the MReset_n flag to true. Triggers the ResetEndEvent.
void SResetAssert()	Called by the slave only. Sets the SReset_n flag to false. Triggers the ResetStartEvent.
void SResetDeassert()	Called by the slave only. Sets the SReset_n flag to true. Triggers the ResetEndEvent.

4.7 Timing Model for the OCP TL2 Channel

The timing model for the OCP TL2 channel aims to reap the benefits of increased channel speed due to OCP burst transaction granularity while mitigating the trade-off by providing sub granularity timing information that can be used to more accurately estimate the timing of the individual OCP transfers that underlie each OCP burst transaction.

4.7.1 Time in the OCP TL2 Channel

For speed and efficiency, the OCP TL2 channel runs un-clocked with the timing taken care of in the master and slave core modules that are connected to it. The timing of the OCP channel is determined by the when the channel's transaction functions (send and accept) are called. This in turn is determined by the cores connected to the channel as they are the ones that call the channel's functions. The channel itself operates passively without a notion of time. The channel is only active when one of its functions has been called by an attached core. Once a function has been called, the channel will do its processing and may also generate events.

The starting time and ending time of each OCP burst request and response are available to the core modules in the course of the simulation. In addition to the start and end timing information, the core modules may also need the timing of the underlying OCP transfers that the burst transaction represents. In the following section, a method is described for doing the above by utilizing the latency definitions listed in the *OCP 2.0 Specification* and additional timing variables added to the channel.

4.7.2 Timing for Different Burst Types

The timing model covers OCP write bursts, read bursts, and non-posted write bursts, where the burst size can be 1 or any other number. The timing model works with a combination of different MRMD (Multiple Requests, Multiple Data) and SRMD (Single Request, Multiple Data) burst transaction types. For instance, an OCP connection (and the channel model representing it) can be configured at elaboration time to deliver any combination of the following burst types:

Imprecise MRMD burst

Precise MRMD burst

SRMD burst

The most complicated case is an OCP connection that allows imprecise MRMD burst delivery, precise MRMD burst delivery, and SRMD burst delivery at the same time[1]. Size of 3 bursts are used as examples in Figure 3 to Figure 10 to demonstrate what kind of TL2 timing information can be important to both the master and slave core modules.

4.7.3 A Guide to the Timing Figures

In these TL2 timing figures, activities for the OCP request phase (Req), the datahandshake phase (DHS), and the response phase (Resp) within a burst are represented horizontally -- simulation time goes from left to right. Each dashed, vertical line indicates a timing point (can be an estimated one) that happens inside a burst transaction and can be used by the TL2 master (on the top of the figure) and slave (on the bottom of the figure) modules to improve timing accuracy. A timing point usually represents either the beginning or the end of an OCP phase activity inside a burst. The alphabetical order among letters shown inside the two dashed boxes attached to a timing point line tells which one needs to happen before the other. The number shown inside a dashed box, if any, indicates the OCP transfer count. Latency between two interesting timing points is shown by a horizontal, double arrow line segment tagged with a fixed latency or a latency estimation function.

Each triangle represents a TL2 channel (API) call that may need to be issued by the master module or the slave module to the OCP TL2 channel model. Note that the times when these calls are made to the OCP TL2 channel model are associated with actual simulation times given by the operation of the simulation. The other timing points are then estimated using both the actual timing points from the API calls and the timing variables passed to the channel.

For each OCP burst, there can be many interesting timing points and latency numbers associated with the underlying transfers. The following is a summary list of these variables used (details are given later):

Triangle 1. This is the last chance for the master to set the OCP TL2 timing variables for this transaction. This is the start time of the TL2 burst request. This is also the start time of the first request of the burst.

Dashed box A is the starting point (the send time) of a write or read OCP request.

Dashed box B is the ending point (the accept time) of a write OCP data or a read OCP request.

Triangle 2 is the end of the OCP TL2 burst request transaction. This is the time when the TL2 slave accepts the OCP TL2 burst request. This is also the accept time of the last OCP data word transfer of the burst. This is the last chance for the slave to set the OCP TL2 timing variables for the next request.

Triangle 3 is the start time of the TL2 burst response. This is also the start time of the first response of the burst. This is also the last time for the slave to set its timing variables for this response.

Dashed box C is the starting point of an OCP response phase.

Triangle 4 is the end of the OCP TL2 burst response. This is the time that the TL2 master accepts the OCP TL2 response. This is also the last time for the master to set its timing variables for the next response.

---

1 If SRMD burst is allowed on an OCP connection, the datahandshake is always turned on.

Dashed box D is the ending point of an OCP response phase.

Fixed latency numbers are defined in the *OCP Specification*

RqAL Request accept latency

RqDL Request-data latency

DAL Data accept latency

RpAL Response accept latency

Expected rates:

RqSndR
Master's send rate of the read requests in a burst. The request send interval, RqSndI = 1/RqSndR.

DSndR
Master's send rate of the write OCP data words in a burst. The data send interval, DSndI = 1/DSndR.

RpSndR
Slave's send rate of the write responses in a burst. The response send interval, RpSndI = 1/RpSndR.

Latency estimation functions:

avgWDI = max(DSndI,DAL)
Estimated average write data interval, given the master's write data send rate (DSndR) and the slave's data accept latency (DAL)

avgRRqI = max(RqSndI,RqAL)
Estimated average read request interval, given the master's read request send interval (RqSndR) and the slave's request accept latency (RqAL)

avgRDI = max(RpSndR,RpAL)
Estimated average read data interval, given the slave's response send interval (RpSndR) and the master's response accept latency (RpAL)

avgWRpI = max(RpSndR,RpAL)
Estimated average write response interval, given the slave's response send rate (RpSndR) and the master's response accept latency (RpAL) <note: same as avgRDI>

4.7.4 Write Requests

Figure 6 Timing information for MRMD posted Write Burst with datahandshake

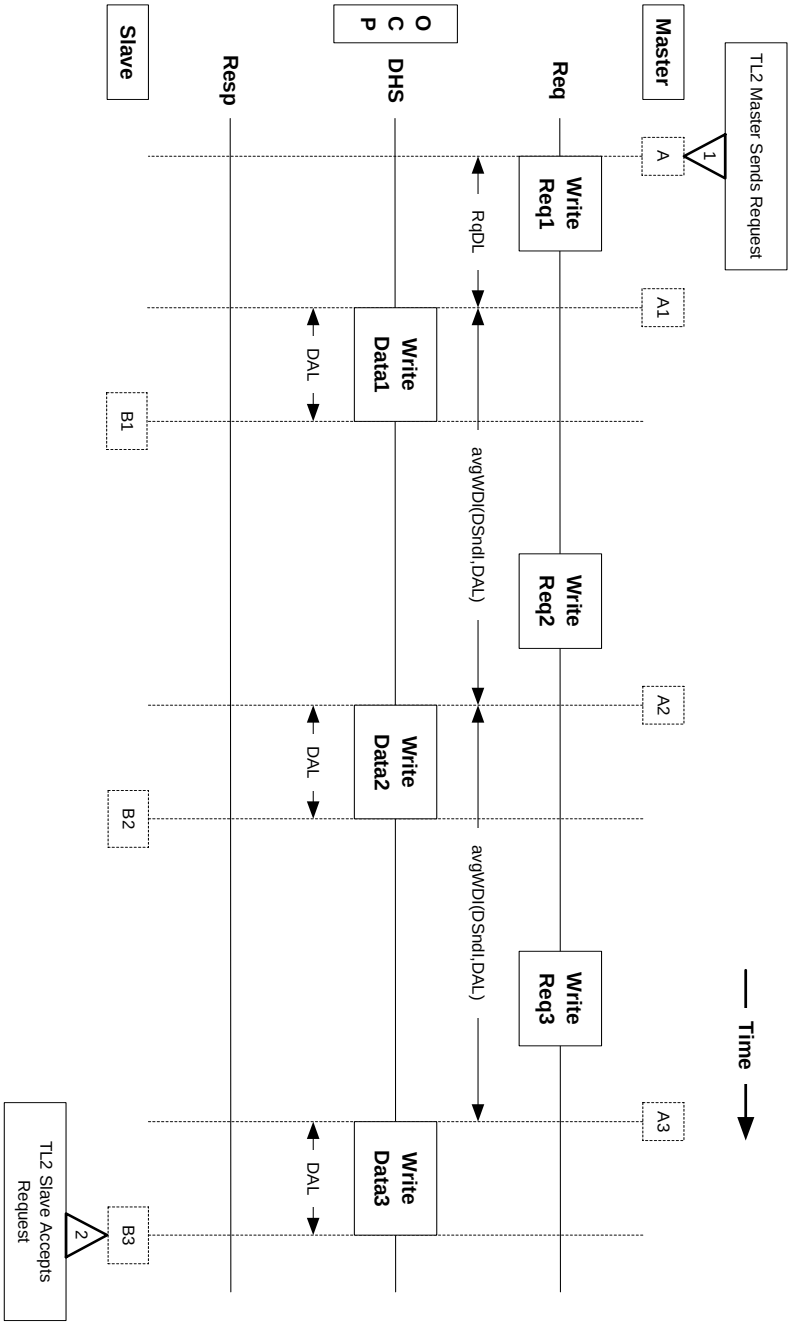
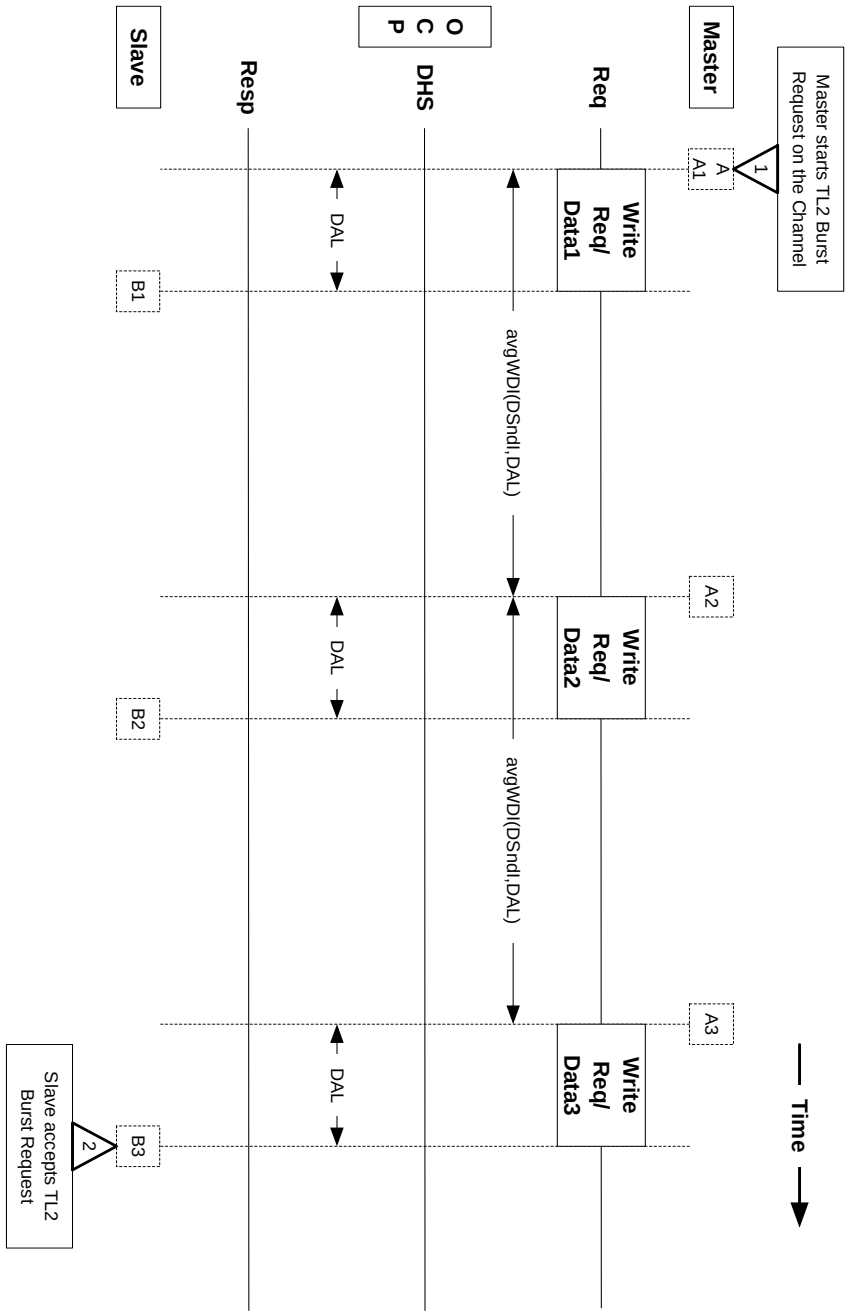
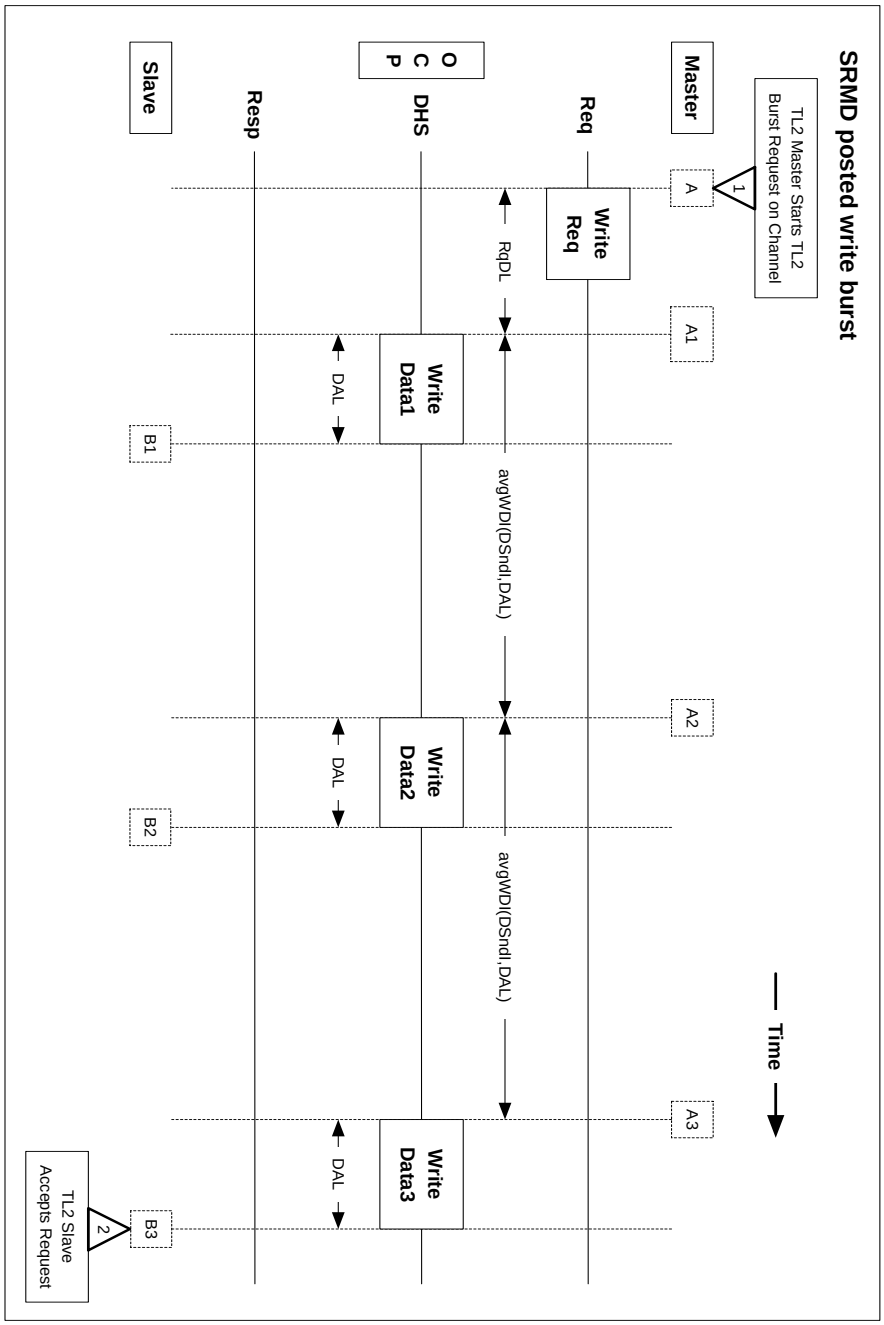


Figure 7 Timing information for MRMD posted Write Burst w/o datahandshake



4.7.5 OCP Posted Write Burst Timing

Figure 8 Timing information for SRMD posted Write Burst



The current OCP TL2 channel models only the OCP burst transaction-level timing but no individual OCP transfer-level timing. For a posted write burst of size 3, it gives us only two timing points:

The starting time of the burst given by the master module when the master issues a posted write burst request unto the OCP TL2 channel – corresponding to the timing point “Triangle 1” shown in Figure 6

The ending time of the burst given by the slave module when the slave accepts the whole burst and releases the request path of the OCP TL2 channel – corresponding to the timing point “Triangle 2” shown in Figure 6

In order to have a more accurate timing in the master and slave modules, the approximate start times and ending times of each of the write data words become valuable. For instance, the master module and the slave module can use these timing estimations to mimic the releasing and allocating resources, respectively. Details on how to compute these OCP transfer-level timing points are described below.

4.7.5.1 Start Time of the First Data Word

Timing point A1 can be determined by the master’s RqDL. This is the interval (in cycles) between the time when the master places the request on the channel and the time that master places the corresponding data word on to the channel. When the slave receives the OCP TL2 write burst request from the master at time A1, the slave knows the start time of the first OCP write request of the burst and can compute the start time of the first data word as:

A1 = A + RqDL

When data handshake is turned off on the OCP connection, the value of RqDL is 0; therefore, timing point A and A1 always happen in the same time (as shown in Figure 7.

4.7.5.2 Time between Two Data Write Words

Another important timing information between OCP transfers is the average time between the i-th data word and the (i+1)-th data word of a burst; i.e., the average Write Data Interval (avgWDI). The start time of the i-th data word, B_i, can be computed approximately as:

Define: $A_i := A_{i-1} + avgWDI$

The avgWDI is determined by two factors: how fast the master can send data down the channel (Data Send Rate, DSndR), and how long the slave waits to accept the data (DAL). Since the master cannot send a new data word until the slave accepts the previous data word, both the master and slave have a hand in determining this value. As shown in Figure 6, we represent the write data interval value by the following function:

```
avgWDI = max(DSndI, DAL)
```

To make this tractable, the DSndR (Data Send Rate) is defined to be the data rate the master can send data down the channel if the slave were to instantly accept all data. And DSndI, the data send interval, is simply 1/DSndR. Thus DSndI is the interval between the data words if the master were connected to a perfectly fast slave. If a master could send data over the channel every single cycle, then the DSndI would be 1. If the master could only send data every other cycle, then the DSndI would be 2.

The DAL, data accept interval, is number of cycles the slave will take to accept each data word. If the slave does not need to use backpressure to delay acceptance of data words, the DAL would be set to 1 (meaning that the slave could accept a new data word every cycle).

4.7.5.3 End Time of the OCP Write Burst

This is the time (B_3 on Figure 6) when the last data word has accepted by the slave. The slave needs to decide this timing point and after this time the channel is free to start a new burst. A master can use the avgWDI formula as described in the previous sub-section to determine, approximately, when an OCP data word within a burst is consumed by the slave.

Note that the slave must accept the OCP TL2 Burst transaction even if the OCP SCmdAccept (or SDataAccept) signal is not part of the OCP channel. In the OCP TL2 model, accepting a request indicates that the correct amount of time has passed for the slave to have processed the data and also indicates that the slave is ready to receive another TL2 burst request from the master. In the lower level OCP TL1 channel model, the slave must toggle the SCmdAccept or SDataAccept for each individual request transfer and data word if those signals are part of the OCP connection. At the TL2 level, the slave accepts the whole OCP burst transaction at once and must do so regardless of the OCP signals used to send that burst.

Note that because the OCP TL2 channel does not explicitly model the data handshake path, some of the parallelism available in an OCP connection can be lost. For instance, the first OCP request of a burst can be sent after the last request of a previous OCP burst has been accepted -- even if the data word associated with this last request of the previous burst has not yet been accepted. In the OCP TL2 model described in the previous paragraph, the first request of a new burst cannot be sent until both the previous OCP burst's last request and data have been accepted by the slave. This difference can contribute to timing inaccuracy especially when the master tightly interlaces write requests (which send data) with reads (which do not) over an OCP channel with data handshake turned. The problem can be overcome by careful bookkeeping in the slave combined with early accepts of read requests that follow write bursts.

4.7.5.4 SRMD Posted Write Burst

The difference between a SRMD (Single Request / Multiple Data) posted write burst (as shown in Figure 8] and a MRMD (Multiple Requests / Multiple Data) one (as shown in Figure 6) is to send only one request instead of N request phases.

4.7.5.5 Posted Write with Responses

Posted writes also have responses. We will skip this topic now and cover it when the non-posted write burst is discussed later.

4.7.6 Read Requests

Figure 9 Timing Information for MRMD Read Burst

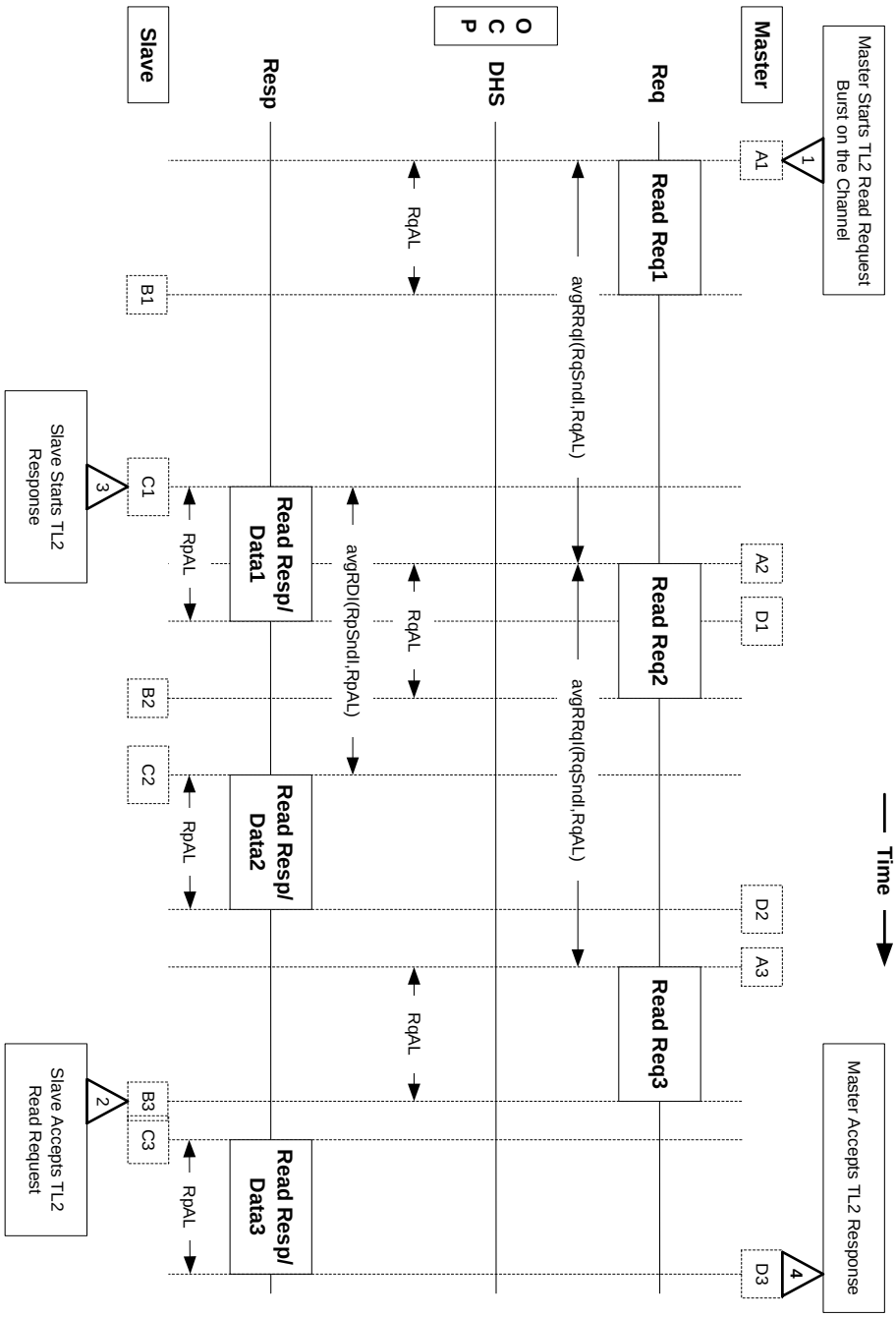
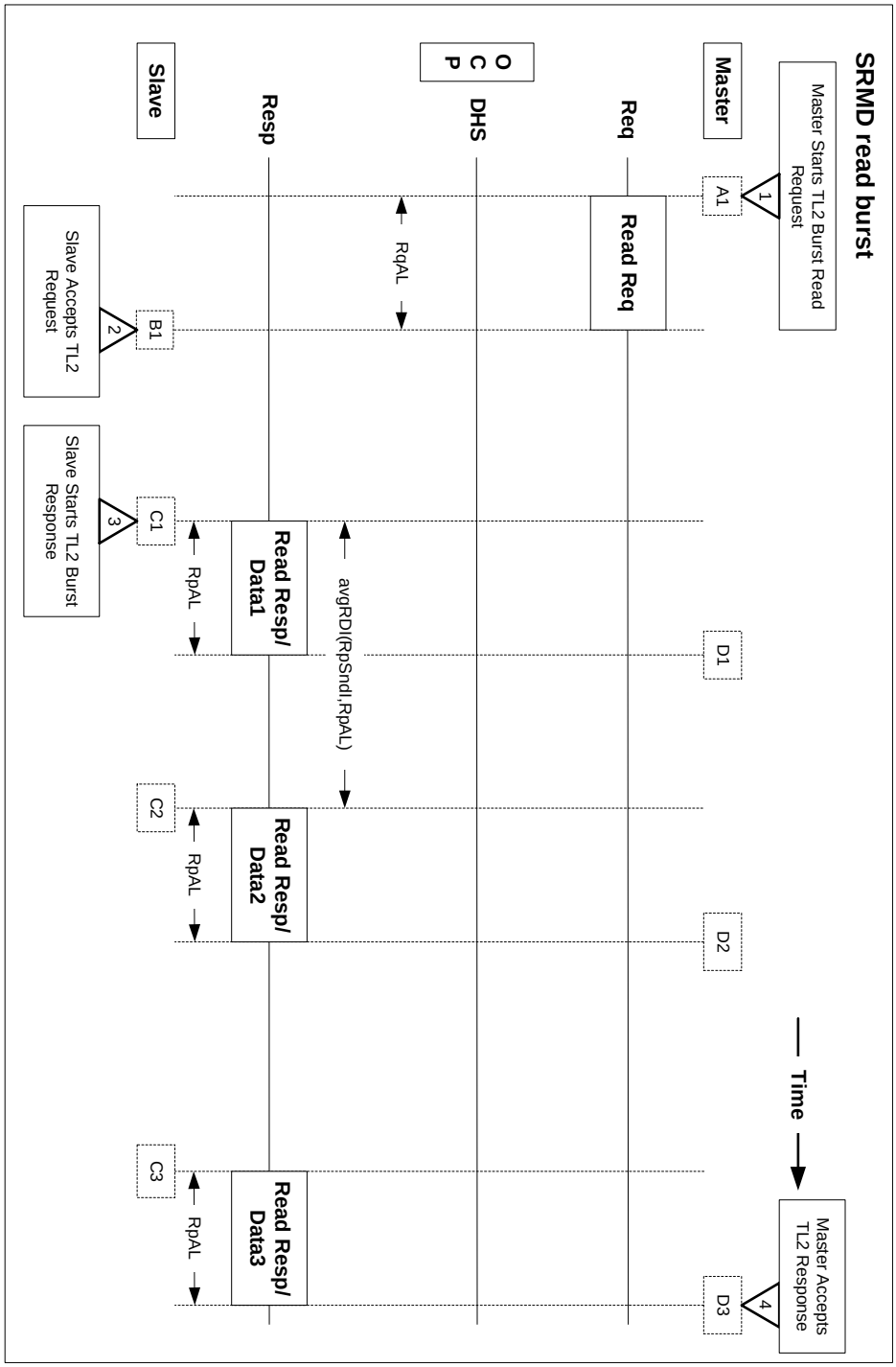


Figure 10 Timing information for SRMD Read Burst



4.7.7 OCP Read Burst Timing

Unlike a posted write burst (a write without a response) described in the previous section, a read burst is modeled using a read burst, request-side transaction and a read burst, response-side transaction in parallel. Thus, a read burst's response-side transaction can be overlapped with another read burst's request-side transaction, in terms of simulation timing.

Similar timing points described for a posted write burst are also listed for a read burst (as shown in Figure 9); except the following:

- There is no request-side data word delivery (i.e., no data handshake)
- There are new terms of RqSndR, RqAL, and avgRRqI

Details about new timing points/variables for the read burst are described below.

4.7.7.1 Time between Two Read Requests

For a MRMD read burst and on the request side, timing information about the individual OCP requests that make up the burst request can be calculated and is represented by the average time between the i-th read request and the (i+1)-th read request of a burst; i.e., the average Read Request Interval (avgRRqI). The start time of the i-th read request, A_i, can be computed approximately as:

$$\text{Define: } A_i := A_{i-1} + \text{avgRRqI}$$

The avgRRqI is determined by two factors: how fast the master can send requests down the channel (Request Send Rate, RqSndR or Request Send Interval, RqSndI = 1/RqSndR), and how long the slave waits to accept the request (RqAL). Thus, both the master and slave have a hand in determining this value. We represent the read request interval value by the following function:

$$\text{avgRRqI} = \max(\text{RqSndI}, \text{RqAL})$$

To make this tractable, RqSndI is defined to be the interval between requests if the master were connected to a perfectly fast slave, which could instantly accept all requests. If the master could send a request every cycle, then RqSndI would be one. If the master could send requests every third cycle then RqSndI would be 3. If the slave does not need to use backpressure to delay acceptance of requests, the RqAL would be set to 1 (meaning that the slave could accept a new read request every cycle).

4.7.7.2 Different Chunk Sizes for the Request Burst and Data Response Burst

Read requests and read data responses are processed independently on different paths; thus, it is possible that the master could send a size of read burst requesting 3 data words and the slave could respond with two separate read response bursts of size 2 and size 1, respectively.

4.7.7.3 Time of the First OCP Data Response

The timing point C_1 is the time of the first read data response sent over the OCP connection. This is also the same time as the start time of an OCP TL2 read burst data response. Note that the interval between A and C_1 is known as the "First Read Latency".

4.7.7.4 Time between Two Read Data Words

Timing information between two consecutive OCP read data responses can be important and is represented by the average time between the i-th read data (and response) and the (i+1)-th read data (and response) of a burst; i.e., the average Read Data Interval (avgRDI). The start time of the i-th read data response, C_i, can be computed approximately as:

Define: C_i := C_{i-1} + avgRDI, where i >2

The avgRDI is determined by two factors: how fast the slave can send response data words (RpSndI), and how long the master waits to accept the response data word (RpAL). Thus, both the slave and master have a hand in determining the avgRDI. As shown in Figure 9, we represent the read data (and response) interval value by the following function:

```
avgRDI = max(RpSndI, RpAL)
```

To make this tractable, RpSndI (Response Send Interval) is defined to be the number of cycles there would be between response data words if the master were to instantly accept response. A fast slave that could send a new response data word every cycle would have a RpSndI of 1. I slower slave that take 3 cycles to send each data word response would have a RpSndI of 3. If the master does not need to use backpressure to delay acceptance of data words and responses, the RpAL would be set to 1 (meaning that the master could accept a new read data response every cycle).

4.7.7.5 SRMD Read Burst

The difference between a SRMD read burst (as shown in Figure 10) and a MRMD one (as shown in Figure 9) is to only send one read request instead of N read request phases.

4.7.8 Non-Posted Writes

Figure 11 MRMD Non-Posted Write Burst with data handshake

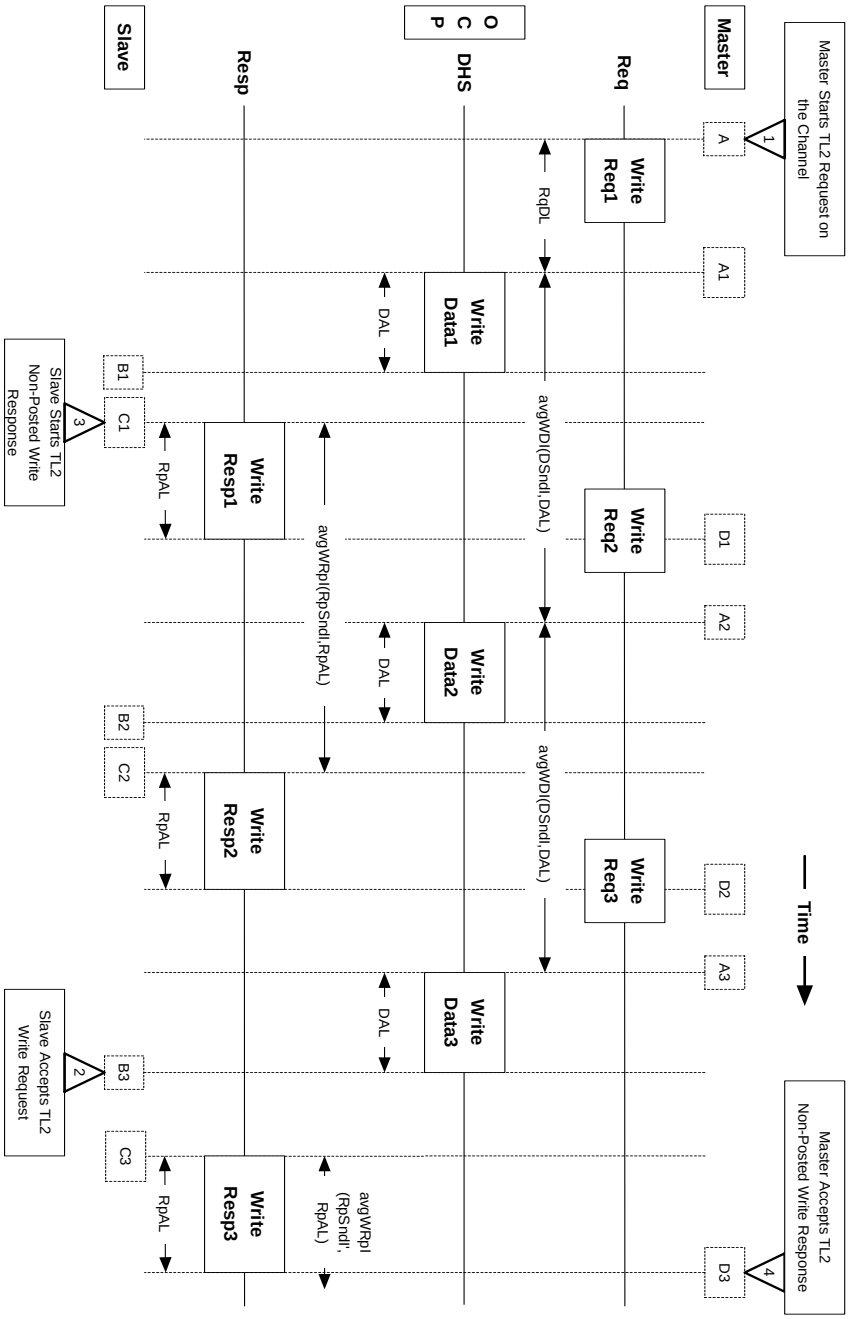


Figure 12 Timing information for MRMD non-posted Write Burst w/o datahandshake

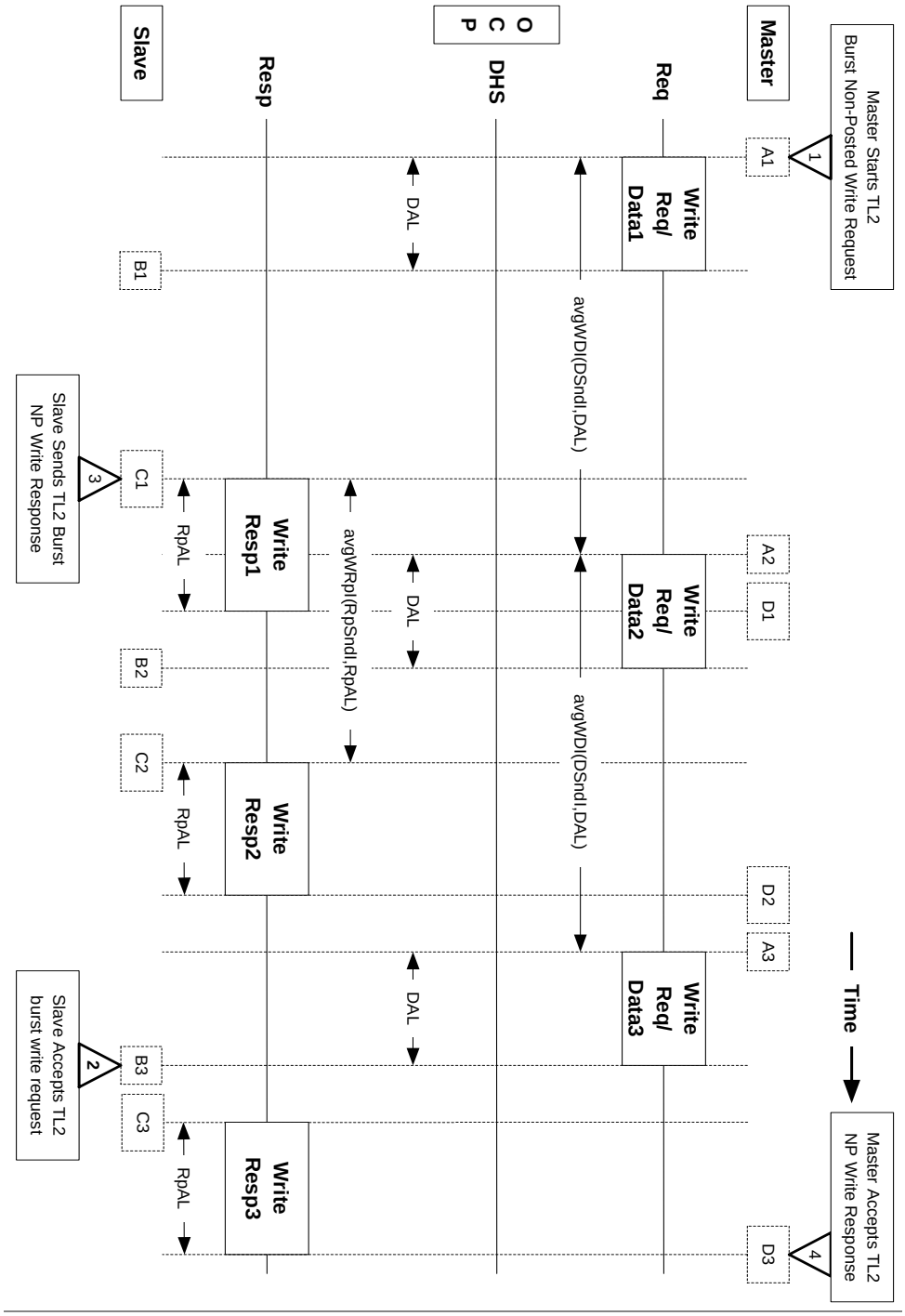
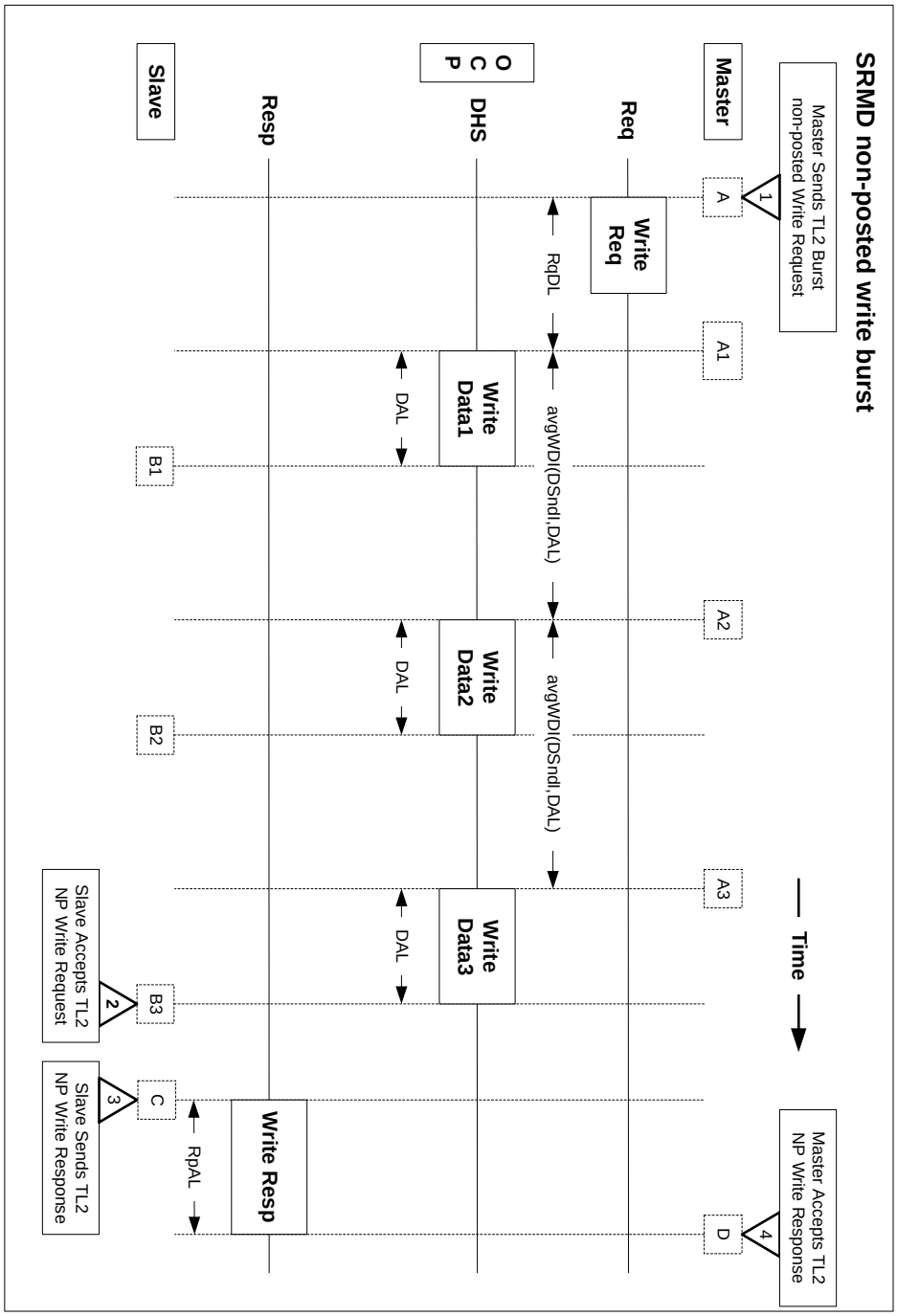


Figure 13 Timing information for SRMD non-posted Write Burst



4.7.9 Non-Posted Write Timing

A non-posted write is a request that receives an acknowledgement response from the slave. The non-posted write's timing model is like a composition of the posted write burst's timing model and the response side of the read burst's timing model (see Figure 11 for details). On the request side, timing points and variables described in the posted write section apply here also. On the response side, the differences compared to the read burst one are as follows:

- Even though no data words are delivered, the avgRDI function is used for the interval between responses (avgWRpI).
- For a SRMD non-posted write burst, only one write response is sent back for a write burst transaction (see Figure 13, the Resp line).

Details about the new timing points and variables for the non-posted write burst are described below.

4.7.9.1 Time between Two MRMD Write Responses

Timing information between two consecutive OCP write responses of a MRMD write burst can be important and is represented by the average time between the i-th write response and the (i+1)-th write response of a MRMD write burst; i.e., the average Write Response Interval (avgWRpI). The start time of the i-th write response, E_i, can be computed approximately as:

$$\text{Define: } C_i := C_{i-1} + \text{avgWRpI}, \quad \text{where } i > 2$$

In order to reduce the complexity of the timing variables, it is assumed that the avgWRpI is the same as the avgRDI for read responses and this is used instead. Thus:

$$C_i := C_{i-1} + \text{avgRDI}, \quad \text{where } i > 2 \ \&\ \text{avgRDI} = \text{avgWRpI}$$

Where avgRDI is calculated exactly as with read responses.

4.7.9.2 Posted Write Burst with Responses

Note that the above response-side timing model can be applied to posted write burst with responses.

4.7.10 OCP TL2 Timing Variables

In order for the master and slave core models attached to an OCP TL2 channel to calculate the approximate timing points regarding individual OCP transfers of the bursts that they receive, the core models need to set and read the basic channel timing variables. Table 15 lists timing variables that are stored in the channel to help derive the timing points of the corresponding transfers.

Table 15 TL2 Channel Timing Variables

Timing Variables	Set by	Description
RqAL	Slave	Request accept latency
RqDL	Master	Request-data latency. The number of cycles between the start of the first request of a write burst and the start of the first write data word of the burst. Note that this variable is zero in

		the case where there is no data handshake in the channel.
DAL	Slave	Write data accept latency. The number of cycles it takes the slave to accept a write data word (for data handshake) or to accept a write request (when data handshake is not part of the channel).
RpAL	Master	Response accept latency. How many cycles it take the master to accept a response.
RqSndI	Master	Request Send Interval. Number of cycles between read requests when the master is sending to a very fast slave. If the master could send every cycle, RqSndI=1. If the master can only send a new request every other cycle, RqSndI=2.
DSndI	Master	Data Send Interval. Number of cycles between write data words when the master is sending to a very fast slave. If the master could send a new data word every cycle, DSndI=1. If the master can only send a new write data word every other cycle, DSndI=2.
RpSndI	Slave	Response Send Interval. Number of cycles between responses when the slave is sending to a very fast master. If the slave could send a new response every cycle, RpSndI=1. If the slave can only send a new response every third cycle, RpSndI=3.

These timing variables are stored in the master and slave timing structures described in the trimming structures section above.

4.7.11 OCP TL2 Timing Functions

In addition to providing the timing variables, the TL2 channel also provides timing helper functions that calculate derived timing information commonly needed by core models. The functions are further described in the master and slave interface sections above.

4.8 OCP TL2 Channel Monitor Interface

The OCP TL2 channel implements the OCP TL2 monitor interface. This allows monitors to be connected to the channel, for performance analysis, trace dumping, protocol checking and so on.

The methods of the monitor interface are listed below. Multiple monitors may be used in parallel on a single OCP TL2 channel. A TL2 monitor support the OCP TL2 observer interface. The monitor registers itself with the channel as observing certain aspects of the traffic, such as request-start-events. The channel informs the monitor by call-back when observed events occur and the monitor is able in turn to poll (peek) the associated data values (eg the OCP request group) from the channel.

The methods of the interfaces are merely listed here. More detailed documentation of their meaning is available in the documentation of the example TL2 monitors, in the monitor release package from OCP-IP. There are four C++ interfaces:

- Peek interface, for getting data values from channel transactions
- Register interface, for registering a monitor with the channel
- Monitor interface, which is simply the union of the peek and register interfaces

- Observer interface, from which the monitor is derived, to allow the channel to call it back. In this interface the methods have default implementations (not shown below) which means that the monitor is not obliged to implement all methods anew.

```
template <class Tdata, class Taddr>
class OCP_TL2_Monitor_ObserverIF
{
public:
  typedef OCP_TL2_MonitorPeekIF<Tdata,Taddr> tl2_peek_type;

  virtual void registerChannel(tl2_peek_type *,
                        bool master_is_node=false,
                        bool slave_is_node=false);

  virtual void start_of_simulation();

  virtual void NotifyRequestStart(tl2_peek_type *);
  virtual void NotifyRequestEnd(tl2_peek_type *);
  virtual void NotifyResponseStart(tl2_peek_type *);
  virtual void NotifyResponseEnd(tl2_peek_type *);

  virtual void NotifyMThreadBusy(tl2_peek_type *);
  virtual void NotifySThreadBusy(tl2_peek_type *);

  // timing
  virtual void NotifyMasterTiming(tl2_peek_type *);
  virtual void NotifySlaveTiming(tl2_peek_type *);

  // reset
  virtual void NotifyResetStart(tl2_peek_type *);
  virtual void NotifyResetEnd(tl2_peek_type *)  ;

  // sideband signals
  virtual void NotifySidebandMaster(tl2_peek_type *);
  virtual void NotifySidebandSlave(tl2_peek_type *);
  virtual void NotifySidebandCore(tl2_peek_type *);
  virtual void NotifySidebandSystem(tl2_peek_type *);
};



template <class Tdata, class Taddr>
class OCP_TL2_MonitorPeekIF :  virtual public sc_interface
{
public:
  typedef OCPTL2RequestGrp<Tdata,Taddr>              request_type;
  typedef OCPTL2ResponseGrp<Tdata>             response_type;

  // port names
  virtual const string        peekChannelName() const = 0;
  virtual const string        peekMasterPortName()    const = 0;
  virtual const string        peekSlavePortName()     const = 0;

  // transactions
  virtual const request_type& peekOCPRequest()  const = 0;
```

```
  virtual const response_type&      peekOCPResponse() const = 0;
  virtual bool                requestInProgress()     const = 0;
  virtual bool                responseInProgress()    const = 0;

  // thread busy
  virtual const unsigned int  peekMThreadBusy() const = 0;
  virtual const unsigned int  peekSThreadBusy() const = 0;

  // timing
  virtual const MTimingGrp&   peekMasterTiming()      const = 0;
  virtual const STimingGrp&   peekSlaveTiming() const = 0;

  // timing helper
  virtual int                 getWDI()          const = 0;
  virtual int                 getRqI()          const = 0;
  virtual int                 getTL2ReqDuration()     const = 0;
  virtual int                 getRDI()          const = 0;
  virtual int                 getTL2RespDuration()    const = 0;

  // reset
  virtual bool                getReset()              = 0;

  // sideband signals
  virtual const OCPSidebandGrp&     peekSideband()          const = 0;

  // OCP paramertes
  virtual OCPParameters*      GetParamCl()                = 0;
};



template <class Tdata, class Taddr>
class OCP_TL2_MonitorRegisterIF :  virtual public sc_interface
{
public:
  typedef OCP_TL2_Monitor_ObserverIF<Tdata,Taddr>     observer_type;

  // transactions
  virtual void                RegisterRequestStart (observer_type *) = 0;
  virtual void                RegisterRequestEnd   (observer_type *) = 0;
  virtual void                RegisterResponseStart(observer_type *) = 0;
  virtual void                RegisterResponseEnd  (observer_type *)  = 0;
  // thread busy
  virtual void                RegisterMThreadBusy(observer_type *)  = 0;
  virtual void                RegisterSThreadBusy(observer_type *)  = 0;

  // timing
  virtual void                RegisterMasterTiming(observer_type *) = 0;
  virtual void                RegisterSlaveTiming (observer_type *) = 0;

  // reset
  virtual void                RegisterResetStart(observer_type *) = 0;
  virtual void                RegisterResetEnd  (observer_type *) = 0;

  // sideband signals
  virtual void                RegisterSidebandMaster(observer_type *) = 0;
  virtual void                RegisterSidebandSlave (observer_type *) = 0;
```

```
  virtual void                RegisterSidebandCore  (observer_type *) = 0;
  virtual void                RegisterSidebandSystem(observer_type *) = 0;

};



template <class Tdata, class Taddr>
class OCP_TL2_MonitorIF :
  virtual public OCP_TL2_MonitorPeekIF<Tdata,Taddr>,
  virtual public OCP_TL2_MonitorRegisterIF<Tdata,Taddr>
{};
```

5 OCP TL3 Channel Model

5.1 OCP TL3 Communication API

The OCP TL3 implements the Architects View use model, which is defined in OCP TLM for Architectural Modeling white paper (www.ocpip.org). The Architects View use model requires a communication API, which supports timing approximate modeling of the platform architecture. On the other hand the API has to be agnostic to any particular bus protocol to enable the flexible and unbiased exploration of different communication architectures. In principle, these capabilities are delivered by the unidirectional non-blocking transfer API in the OSCI TLM standard. The OCP TL3 can be seen as a convenience layer for architectural modeling.

The TL3 API was first presented in the OCP-IP White Paper for SoC Communication Modeling and implemented in the first release of the OCP-IP SystemC models.

The TL3 Master API definition is listed and explained in the following table:

API Function	Description
Regular Request Commands	
bool sendRequest(const REQ& req)	Puts a request on the channel. Returns true if the request was successfully placed on the channel. False otherwise.
bool sendRequestBlocking(const REQ& req);	Puts a request on the channel, waiting until the channel is free if necessary. Waits until the slave accepts the request and then returns. Blocking calls may only be called from SC_THREAD processes.
bool requestInProgress() const	True if there is currently an active request on the channel.
const sc_event RequestStartEvent()	Returns the event that is triggered when the master places a new request on the channel.
const sc_event RequestEndEvent()	Returns the event that is triggered when the slave accepts the current request.
Timed Request Commands	
bool sendRequest(const REQ&, const sc_time& t)	Delays the sending of the request for time t. Otherwise identical to sendRequest.
bool sendRequest(const REQ&, const int cycles)	Delays the sending of the request for cycles number of clock ticks. Otherwise identical to sendRequest.
Regular Response Commands	
Bool getResponse(RESP& resp)	Gets a new response from the channel and returns true. Returns false if no new response transaction available.

Kommentar: Are these absolutely necessary? This is a major departure from the TL2, and the original TL3 definition. Or should we introduce these to TL3 also?

bool getResponseBlocking(RESP& resp)	Waits for a new, unread OCP TL2 response to come on to the channel and then gets it. Can only be called from an SC_THREAD process.
bool acceptResponse()	Accepts the response immediately and returns true. Returns false if no response to accept.
bool responseInProgress() const	True if there is currently an active response on the channel.
const sc_event ResponseStartEvent()	Returns the event that is triggered when the slave places a new response on the channel.
const sc_event ResponseEndEvent()	Returns the event that is triggered when the master accepts the current response.
Timed Response Commands	
bool acceptResponse(const sc_time& t)	Delays accept by t SystemC time units from now. Returns false if no response to accept.
bool acceptResponse(const int cycles)	Delays accept by cycles OCP clock periods from now. If cycles=0 then the accept is immediate. Returns false if no response to accept.

Table 16: OCP TL3 Master Interface Definition

The conversion of integer cycles into SystemC time is based on the clock-period, which is a member of the channel. The slave interface is not depicted as it is perfectly symmetrical to the master interface, with just Request and Response interchanged.

5.2 Mapping TL3 onto OSCI TLM

All the functions and events in the TL3 API can be implemented on top of the non-blocking unidirectional OSCI TLM standard. This section demonstrates the mapping of the TL3 primitives onto the OSCI TLM API. In a similar way, the complete TL2 API including thread-busy, handshake-timing, and reset, can be mapped onto the TLM API. The TL2-TLM mapping is also included in the methodology example package, but not discussed in detail in this document.

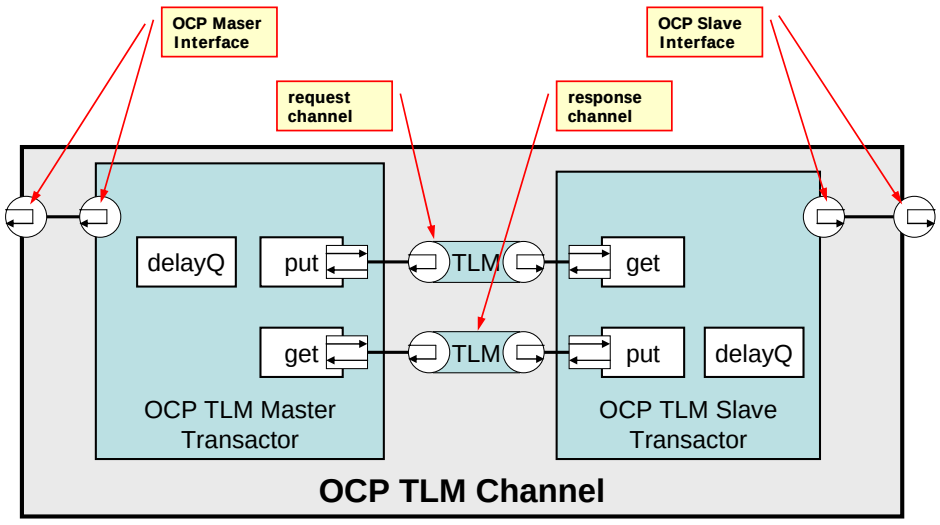


Figure 14: Mapping the TL3 API onto the OSCI TLM standard

Kommentar: Could

The overall structure of the OSCI-TLM based OCP channel is depicted in the figure. From the outside perspective, the TLM based OCP channel looks just like any other OCP channel, i.e. it implements a master and a slave interface. As the only minor difference with the original OCP-TLM-channel does not implement master and a slave interfaces but uses the sc_export feature of SystemC 2.1.

Inside the OCP TLM channel the interfaces are implemented by two separate modules, the master-transactor and the slave-transactor. Most importantly the master- and slave-transactor are completely separated, i.e. they communicate only via two OSCI TLM FIFOs. In that the mapping of the OCP API onto the TLM API is 100% complete. Each of these FIFOs is of course a size of 1, which corresponds to the current transaction in the OCP channel.

The master- and slave-transactor are composed of policy-classes, which each implement one aspect of the OCP protocol. The put-policy implements everything related to the transmitting of data and the get-policy implements everything related to the receiving of data. Both policies are templatized with the transaction data structure, so they can be used for both master and slave side. Apart from reusing the code in the transactor this structure nicely emphasizes the symmetry of the OCP protocol: the sending of request is handled in the exact same way as the sending of response.

The implicit timing annotation features of the TL3 API require additional functionality. The timed sendRequest (and sendResponse) methods are implemented using a special delay queue called ChronoQueue, which delays the sending of transactions by a specific amount of time. The timed requestAccept (and responseAccept) methods are implemented by the get policy itself by means of delayed event notification.

The detailed mapping of OCP TL3 methods and events on the OSCI TLM standard is depicted in the following table:

OCP TL3 Master API Function	**OSCI TLM Standard API**
Regular Request Commands	
bool sendRequest(const REQ& req)	bool nb_put(const T &t

bool sendRequestBlocking(const REQ& req);	derived from sendRequest and RequestEndEvent
bool requestInProgress() const	!(bool can_nb_put() const)
const sc_event RequestStartEvent()	local event
const sc_event RequestEndEvent()	const sc_event &ok_to_put() const
Timed Request Commands	
bool sendRequest(const REQ&, const sc_time& t)	derived from sendRequest and local delay queue
bool sendRequest(const REQ&, const int cycles)	
Regular Response Commands	
bool getResponse(RESP& resp)	bool nb_peek(T &t) and local flag
bool getResponseBlocking(RESP& resp)	derived from getResponse
bool acceptResponse()	bool nb_get(T &t)
bool responseInProgress() const	bool nb_can_peek() const
const sc_event ResponseStartEvent()	const sc_event &ok_to_peek() const
const sc_event ResponseEndEvent()	local event
Timed Response Commands	
bool acceptResponse(const sc_time& t)	derived from acceptResponse and delayed event notification
bool acceptResponse(const int cycles)	

Table 17: TLM mapping of OCP TL3 master interface

The regular request commands and events are implemented in the put-policy. These TL3 primitives have almost a one-to-one correspondence with the tlm_nonblocking_put_if of the TLM API. Only the RequestStartEvent is missing in the TLM API, but the occurrence of this event is of course known at the master side. Hence a local event in the put-policy is notified whenever a new request is put into the TLM FIFO.

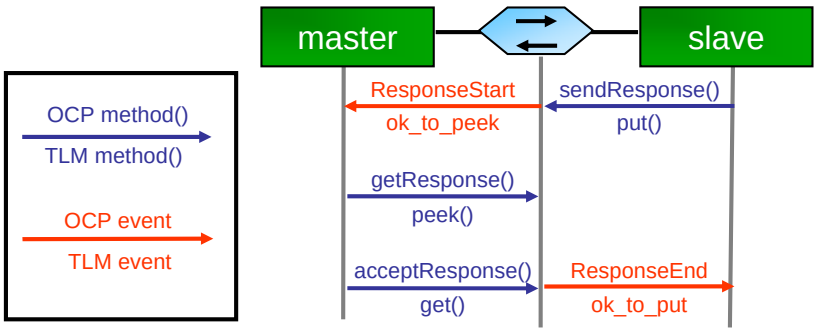


Figure 15: Sequence of methods and events

The get policy is slightly more sophisticated, because we need to mimic the get-accept mechanism of the OCP protocol. As illustrated in the figure, we cannot use the TLM-get method for the OCP-getResponse method, since the TLM-get is destructive and

immediately notifies the ok_to_put event on the producer side. According to the OCP protocol, this event is not supposed to be notified until the master has released the response channel by calling the acceptResponse method. Hence we need the full expressiveness of the tlm_nonblocking_get_peek_if, which provides with the required non-destructive peek method.

In analogy to the put-policy, the get-policy provides a local ResponseEndEvent, which is notified upon the acceptance of the current response. Additionally, the get-policy needs a local SC_METHOD together with a separate event to implement the delayed accept methods.

Please refer to the online documentation of the methodology package for detailed information on the implementation of the tlm_tl3_transactor_channel implementing the mapping.

5.3 TL3 Timing

The major purpose of the Architects View use-case is to investigate the performance of a given HW/SW partitioning and platform architecture based on an approximate timing model. Hence special care needs to be given to the modeling of timing.

In principle the implicit timing annotation primitives in the TL3 API operate at the interval-level, i.e. the granularity is limited to the boundaries of transactions. Two timing parameters characterize the performance of any activity in the system:

- The *accept-delay* Δt_{accept} specifies the minimum time between two consecutive start request events or two consecutive start response events. In essence the accept-delay constraints the bandwidth of a block, i.e. during this period a slave module is busy with the processing of a request or a master module is busy with the processing of a response.

- The *latency* $\Delta t_{latency}$ specifies the time between the process activation and the sending of the transaction. At the target side, this parameter is called *response-delay* $\Delta t_{response}$ and denotes the duration between request start event and the response start event.

In that way the timing requirements of arbitrary platform building blocks can be roughly specified. For example a pipelined ASIC block will exhibit an accept delay smaller than the response delay, whereas for a task executed on a programmable core it will be the other way around. Please note, that the accept-delay and latency are not specific for the performance modeling of a communication node or a processing element.

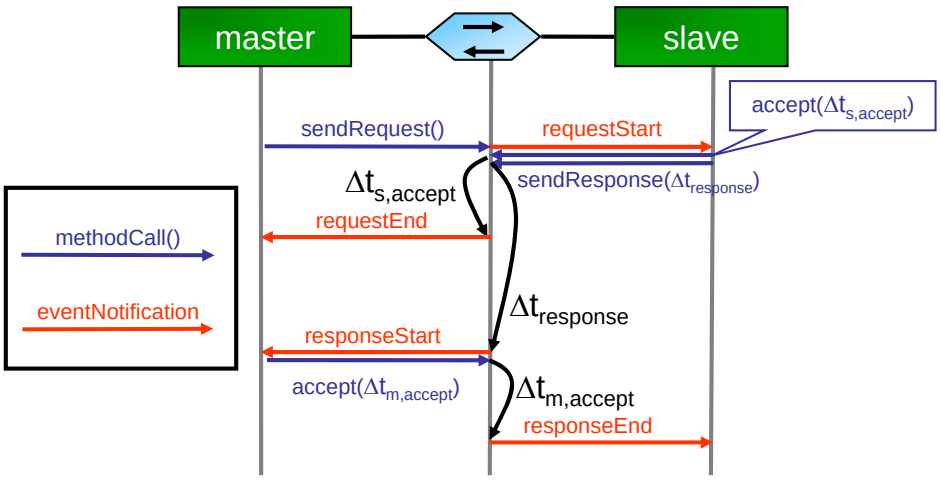


Figure 16: TL3 point-to-point timing annotation

The diagram depicted above shows the implicit timing annotation methodology of the TL3 channel applied to a typical sequence of method calls (blue) and events (red):

- The master initiates a transaction using the sendRequest API call
- The channel immediately triggers the RequestStartEvent
- The slave reads the data from the channel (not depicted) and at the same time uses the delayed acceptRequest method to specify the slave accept delay $\Delta t_{s,accept}$. The OCP TL3 slave interface also supports implicit timing annotation for the response delay in the sendResponse method, so even here the slave does not have to call wait(). (This feature is not implemented in the TL2 API, but is under consideration for the later versions.)
- After $\Delta t_{s,accept}$ the OCP channel self-acting releases the request path and notifies the RequestEndEvent
- After $\Delta t_{s,response}$ the OCP channel self-acting initiates the response phase by notifying the ResponseStartEvent. The remainder of the response phase is completely symmetric to the request phase.

Of course both delays could also be specified explicitly by calling `wait(`Δt_{accept}`)` and `wait(`$\Delta t_{response}$`)`. However the explicit way of modeling timing is unfavorable in terms of simulation speed and orthogonalization of timing and behavior. The latter limitation is not obvious and shall be further explained by means of a simplified example depicted in the next figure.

```
thread(){
  ocp->getRequest(req);
  wait(Δt_1);
  ocp->acceptRequest();
  wait(Δ_2);
  ocp->sendResponse(resp);
}
```

a) ASIC

```
thread(){
  ocp->getRequest(req);
  wait(Δt_1);
  ocp->sendResponse(resp);
  wait(Δt_2);
  ocp->acceptRequest();
}
```

b) SW task

```
thread(){
  ocp->getRequest(req);
  ocp->acceptRequest(Δt_1);
  ocp->sendResponse(resp,Δt_2);
}
```

c) implicit timing model

Figure 17: AV timing annotation

In case a) the anticipated implementation of a SystemC thread is an ASIC block, so the accept delay is smaller than the response delay. Modeled explicitly the request must be accepted before the response is sent. To change the implementation into a SW task, the source code of the SystemC model needs to be modified according to b). The implicit timing model depicted in c) is more flexible, in that only the values of Δt_1 and Δt_2 need to be modified to change the performance characteristic from ASIC to SW.

The actual value of the timing parameters can either be a (configurable) constant, a data-dependent variable, or can be drawn from a stochastic distribution function. The TL2 and TL3 APIs allow specifying the implicit delay in terms of cycles as well as in terms of time. In case cycles are used, the channel calculates the effective delay by multiplying the cycle count with its clock period parameter. In principle this cycle-based annotation is favorable, because it enables a more efficient exploration of the impact of clock frequency on performance. The frequency could even be modified during runtime to investigate e.g. the impact of dynamic voltage scaling on system performance. A delay specification based on cycles is also more re-usable than a timing value, as it does not tie a TLM model to a certain technology.

In principle we advocate the concept of individual timing annotation, i.e. the timing parameters should be maintained by the respective model. For example, a communication node should own the delay parameters related to communication latency and bandwidth. On the other hand, the computational element should own the delay parameters related to processing latency and bandwidth. It is absolutely discouraged to mix up the ownership of delay parameters, for example using the accept delay in a target processing element to annotate the communication latency.

The major advantage of individual timing annotation is that it is modular and compositional, i.e. components can be successively composed to systems without reworking the timing annotation. Figure below shows the deployment of the individual interval-level timing annotation parameters in a typical request phase of a simple platform model. The platform comprises a shared bus connected to one or multiple initiators and one or multiple targets.

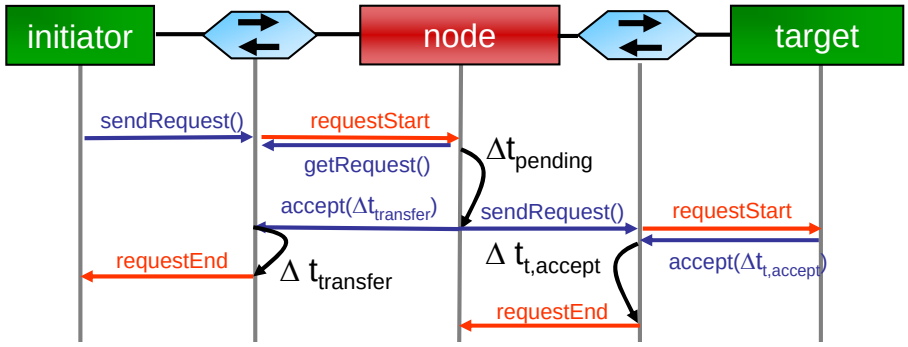


Figure 18: AV platform timing

In case of a computational element like the target module on the right hand side, the assignment of accept- and response-delays to processing delays is straight-forward. For the communication node however, the accept delay representing the bandwidth is split into a pending delay and a transfer delay. The former corresponds to the send-delay of the request and has to be modeled explicitly, because it depends on the current traffic situation. In contrast the transfer delay can be arithmetically derived from the performance parameters and is therefore applicable for implicit timing annotation.

The point of the figure is that neither the initiator nor the targets need to change their timing annotation, when a bus node is plugged between them. Of course the system-level timing changes, but this merely turns out as a consequence of all the individual timing annotations.

In summary, this section has introduced the AV timing model by means of timing annotation:

- Implicit timing annotation refers to the modeling of timing by means of TLM communication API parameters in favor of using wait or delayed event notification.
- Interval level timing annotation refers to the transaction granularity of the AV timing model, in that the timing resolution is limited to the start and end of transactions.
- Individual timing annotation refers to a well-defined ownership policy of the timing annotation parameters in order to achieve a compositional performance model.

The OCP TL2 API also supports implicit timing annotation at the word-level, which specifies in more detail the timing of individual beats in a burst. This is done by a set of timing parameters, which are defined in the master and slave timing-groups. Word-level annotation may increase the timing accuracy in case the data-handshaking feature of the OCP protocol is used (see section 6.8 of the OCP channel documentation [5]). However it is rather specific for OCP and hence not part of the protocol agnostic TL3 API.

5.3.1 Scalable Accuracy

There is not necessarily a one-to-one relation between an OCP burst and a TL2 or TL3 request. The LastOfBurst attribute in the TL2 data structures allows the user defined

segmentation of an OCP protocol burst into multiple chunks. This degree of freedom enables a scalable accuracy of the timing model:

The user can transfer an OCP burst as a single TL2 request, i.e. LastOfBurst is always true, and the DataLength equals the total OCP burst size. However the granularity of TL2 timing annotation is tied to the TL2 transfers, and no predication can be made the timing within the transfer. The parameters in the TL2 timing group specify the timing of individual beats in a burst, but this assumes these timing parameters are constant during the complete burst. In summary, this is for sure the fastest way to transfer the data, but on the other the least accurate.

On the other had, the user can decide to segment an OCP burst into multiple TL2 request, e.g. a burst of 8 word is split into 2 TL2 requests of DataLength 4 each. Now the timing information can be applied to each of the requests, so the performance model is more accurate. On the other hand, more events are required to transfer the data, so naturally the simulation speed degrades.

The decision on the granularity depends very much on the required accuracy of the performance model as well as on the dynamic in the system. As long as the state of a module does not change during an OCP burst, a finer granularity would not increase the accuracy.

5.4 TL3 Channel Monitor Interface

The TL3 channel implements the TL3 monitor interface. This allows monitors to be connected to the channel, for performance analysis, trace dumping, protocol checking and so on.

The methods of the monitor interface are listed below. Multiple monitors may be used in parallel on a single TL3 channel. A TL3 monitor supports the TL3 observer interface. The monitor registers itself with the channel as observing certain aspects of the traffic, such as request-start-events. The channel informs the monitor by call-back when observed events occur and the monitor is able in turn to poll (peek) the associated data values (eg the request group) from the channel.

The methods of the interfaces are merely listed here. More detailed documentation of their meaning is available in the documentation of the example monitors, available from OCP-IP as part of the monitor release package. There are four C++ interfaces:

- Peek interface, for getting data values from channel transactions
- Register interface, for registering a monitor with the channel
- Monitor interface, which is simply the union of the peek and register interfaces
- Observer interface, from which the monitor is derived, to allow the channel to call it back. In this interface the methods have default implementations (not shown below) which means that the monitor is not obliged to implement all methods anew.

```
template
< typename REQ,
  typename RESP
>
class OCP_TL3_Monitor_ObserverIF
{
public:
```

```
  typedef OCP_TL3_MonitorPeekIF<REQ,RESP> tl3_peek_type;

  virtual ~OCP_TL3_Monitor_ObserverIF() {};

  virtual void registerChannel(tl3_peek_type *,
                       bool master_is_node = false,
                       bool slave_is_node = false)  = 0;

  virtual void NotifyRequestStart(tl3_peek_type *)          = 0;
  virtual void NotifyRequestEnd(tl3_peek_type *)            = 0;
  virtual void NotifyResponseStart(tl3_peek_type *)         = 0;
  virtual void NotifyResponseEnd(tl3_peek_type *)           = 0;

};



template
<
  typename REQ,
  typename RESP
>
class OCP_TL3_MonitorPeekIF :  virtual public sc_interface
{
public:
  typedef REQ                 request_type;
  typedef RESP                response_type;

  // port names
  virtual const std::string   peekChannelName() const = 0;
  virtual const std::string   peekMasterPortName()    const = 0;
  virtual const std::string   peekSlavePortName()     const = 0;

  // transactions
  virtual const request_type& peekRequest()     const = 0;
  virtual const response_type&      peekResponse()    const = 0;
  virtual bool                requestInProgress()     const = 0;
  virtual bool                responseInProgress()    const = 0;
};



template
<
  typename REQ,
  typename RESP
>
class OCP_TL3_MonitorRegisterIF :  virtual public sc_interface
{
public:
  typedef OCP_TL3_Monitor_ObserverIF<REQ,RESP>  observer_type;

  // transactions
  virtual void                RegisterRequestStart (observer_type *) = 0;
  virtual void                RegisterRequestEnd   (observer_type *) = 0;
  virtual void                RegisterResponseStart(observer_type *) = 0;
  virtual void                RegisterResponseEnd  (observer_type *)  = 0;
```

```
};



template
<
  typename REQ,
  typename RESP
>
class OCP_TL3_MonitorIF :
  virtual public OCP_TL3_MonitorPeekIF<REQ,RESP>,
  virtual public OCP_TL3_MonitorRegisterIF<REQ,RESP>
{};
```

6 Example Using OCP TL1 Channel and API

The example described in this section demonstrates the use of the OCP TL1 channel in a simple reference master and slave. The first part of the example shows how the configuration parameters can be set in the OCP TL1 channel. This technique is expanded upon to configure a master and a slave core.

The second part of the example shows a configurable reference master core that uses the OCP TL1 API. The third part of the example is a configurable slave core that also uses the OCP TL1 API.

This example makes a heavy use of blocking TL1 methods. There are other examples included in the release package that use non-blocking methods.

6.1 Configuring the OCP TL1 Simulation

As described in section 2.2, in this example the OCP TL1 channel is configured using some of the available parameters. It does not use all the OCP configuration options. This example is of configuration by the master, during elaboration.

To configure the channel, the master calls the `setOCPMasterConfiguration()` function with a MAP object that contains all of the parameter settings:

```
setOCPMasterConfiguration( map<string,string>& parameterMap );
```

In this example the configuration of the channel is passed from the environment to the master who passes it to the channel, and the slave adapts to it.The master in this example contains a `provideChannelConfiguration` function, by which the environment can pass the channel configuration to the master. For models of "real" cores that do something more than simply excite the bus interface, the OCP parameters would often be known in advance and fixed. In such cases configuration of the channel would be done directly from the cores, without information from the environment.

The slave in this example uses the addOCPConfigurationListener() method of the channel, at end of elaboration, to receive the channel's configuration and adapt its behaviour. The master only configures the channel and does not register at the channel as a configuration listener. This style of implementation is not recommended because it is potentially not compatible with configuration of the channel from the slave core. Thus this master and slave work well together, but this master might not work properly with a slave using setOCPSlaveConfiguration().

6.1.1 Configurable Master and Slave

The same parameter map scheme described in section 2.2 is used to configure the reference master and reference slave. The following table gives the parameters for the reference master.

Table 18 Reference Master Parameters

Parameter Name	Type	Default Value	Description
mrespaccept_delay	i	1	The number of cycles to delay before accepting a response from the slave.
mrespaccept_fixeddelay	i	1	MRespAccept Delay Style. If this parameter is true (1), the master always waits for

	"mrespaccept_delay" cycles before accepting a response. If this parameter is false (0), the master waits for a random number of cycles before accepting the response. This random number of cycles will vary uniformly from 0 to mrespaccept_delay.

To configure the reference master, create a parameter map using the parameters above and then send it to the reference master using the following command:

```
void Master<TdataCl>::setModuleConfiguration( MapStringType&
passedMap )
```

The following table gives the parameters for the reference slave:

Table 19 Reference Slave Parameters

Name	Type	Default Value	Description
latencyX	i	3	This is actually a set of parameters, one for each thread in the channel. Each parameter sets the latency for one thread. The latency is the minimum number of cycles between when the request arrives and when the response is sent. As an example, the parameter latency0 will set how many cycles the slave will wait before accepting a request on thread number zero, while latency5 will set the latency cycles for thread 5. In this example only latency0 is used.
limitreq_max	i	4	The maximum number of requests that the slave can have outstanding at any one time on any one thread. Note that this parameter is not used if limitreq_enable is false.

Once the parameter map for the reference slave has been built, it can be sent to the slave with the following commands:

```
void Slave<TdataCl>::setModuleConfiguration( MapStringType&
passedMap )
```

6.1.2 Building a Custom Configurable Core

A user core may also be configurable and of course the core writer is free to use the parameter map scheme presented here to configure their own custom core.

6.2 A Configurable Master Model

This section provides an example of a configurable master model that has a single-threaded master OCP interface and that can generate simple OCP traffic to mimic an initiator core. This master model not only has its own parameters but can also deal with different OCP parameter settings. For instance, the master model can talk to an OCP channel with the following settings:

- cmdaccept == 1, sthreadbusy == 0 or 1, and sthreadbusy_exact == 0

- cmdaccept == 0, sthreadbusy == 1, and sthreadbusy_exact == 1
- respaccept == 0, mthreadbusy == 0, and mthreadbusy_exact == 0
- respaccept == 1, mthreadbusy == 0 or 1, and mthreadbusy_exact == 1
- respaccept == 0, mthreadbusy == 1, and mthreadbusy_exact == 1

The address, the request type (WR or RD), and the write data of a request can also be specified.

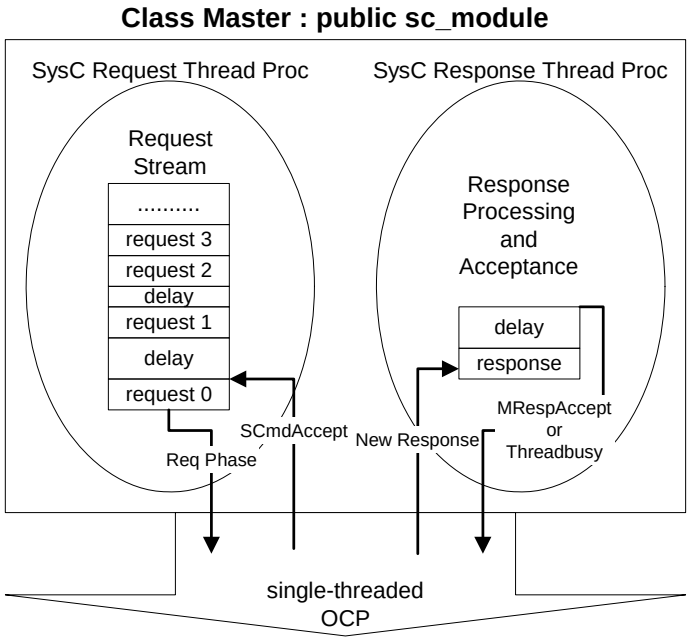
Since the supported configurations include sthreadbusy and mthreadbusy, the timing distribution mechanism as described in section 2.6 is also used by this master.

In addition, the latency between the acceptance of a previous request and sending of a current request can be controlled. Also, the latency between receiving a response and accepting the response can be controlled.

Figure 8 shows a diagram of the configurable master model. This master model implements two SystemC thread processes (represented by the two ovals in the figure). (The master model is a derived class of the SystemC sc_module class.) The request thread process handles the sending of requests for the master core. The response thread process handles the receiving of responses for the master core.

In the following sections, the source code (with explanations) of the master model is described to help you understand the implementation of the model.

Figure 19 Master Model



6.2.1 Header File

You must follow a few rules in defining the master core template class so that it can communicate with the OCP Channel. The following are comments on the code followed by the full master header file.

First, include the OCP TL1 channel header files:

```
// OCP-IP Channel header files
#include "globals.h"
#include "ocp_tl1_master_port.h"
#include "ocp_tl_param_cl.h"
```

The file globals.h contains the definitions of the types used in the channel. This also includes the file ocp_tl1_data_cl.h that defines the data class used by the OCP TL1 channel, which then includes ocp_globals.h. The header file ocp_globals.h in turn is used to define the structures used to pass requests and responses to the channel. If this core did not have a header file such as globals.h, it would need to directly include the header file ocp_globals.h.

The header ocp_tl1_master_port.h contains the master port to the OCP TL1 channel. In addition to providing the master interface to the channel, the port also provides event finders for all of the master and sideband events of the channel.

The ocp_tl_param_cl.h header file contains the definition of the parameter class. The configurable master uses this class to read the channel's configuration and then uses that information to set up its own configuration to match.

The master class is a template class and the parameter of the template is the data class that the master will support over the OCP connection. A data class with 32-bit data width and a 32-bit address is specified as follows:

```
OCP_TL1_DataCl<OCPCHANNELBit32, OCPCHANNELBit32>
```

Where `OCPCHANNELBit32` is defined as follows in the file `globals.h`:

```
typedef unsigned int OCPCHANNELBit32;
```

After including the header files, you must declare a SystemC port (`sc_port`). Specifically, you need to declare an OCP TL1 master port (`ipP`) for the Master class to communicate with an OCP SystemC TL1 channel. This is accomplished with the following statement:

```
OCP_TL1_MasterPort<TdataCl> ipP;
```

The master port provides event finders for the channel events (such as `RequestStart` and `RequestEnd`). If these event finders are not needed, they could be declared as follows, which would also work:

```
sc_port< OCP_TL1_MasterIF<TdataCl> > ipP;
```

The example master also possesses a second port:

```
sc_in_clk clk;
```

This is the master's clock port, through which it can access the system clock, which is provided by the environment.

Next, declare functions that define SystemC thread or method processes used in your model. For example, in this master core model, the following functions are defined:

SC_HAS_PROCESS(Master);

void requestThreadProcess();

void responseThreadProcess();

void exerciseSidebandThreadProcess();

The macro `SC_HAS_PROCESS(Master)` tells SystemC that the master core is a SystemC module with its own processes. In this case, the thread processes that follow. Each of these processes are explained in detail in later sections.

After declaring the functions for the thread or method processes, define a SystemC `end_of_elaboration function`. For example,

void end_of_elaboration(); // SystemC method

Now define a member of type OCPParameters within which the master may store the configuration of the channel (details of the use of this memeber can be found in section 6.2.3:

```
OCPParameters  m_parameters;
```

The rest of the data members hold the parameter and configuration values of the master.

The following is the complete header file for the master.

```
#ifndef _SIMPLE_MASTER_H
#define _SIMPLE_MASTER_H

// OCP-IP Channel header files
#include "globals.h"
#include "ocp_tl1_master_port.h"
#include "ocp_tl_param_cl.h"

// define the Master transactor class
template <typename TdataCl>
class Master : public sc_module, public OCP_TL1_Slave_TimingIF
{
  public:
    // --------------------------
    // public members and methods
    // --------------------------

    // type definitions
    typedef typename TdataCl::DataType Td;
    typedef typename TdataCl::AddrType Ta;

    // member definitions

    // channel port
    OCP_TL1_MasterPort<TdataCl> ipP;
    sc_in_clk clk;

    // SystemC macros
    // has SystemC processes
    SC_HAS_PROCESS(Master);
```

```
    // constructor and destructor
    Master(sc_module_name,
           int, ostream* debug_os_ptr = NULL);
    ~Master();

    // methods
    void provideChannelConfiguration( MapStringType&);
    void setOCPTL1SlaveTiming(OCP_TL1_Slave_TimingCl);
    void setModuleConfiguration(MapStringType&);

    // process methods
    void requestThreadProcess();
    void responseThreadProcess();
    void exerciseSidebandThreadProcess();

private:
    // ---------------------------
    // private members and methods
    // ---------------------------

    // SystemC methods
    void end_of_elaboration();

    // member definitions

    // master identification
    int          m_ID;

    //
    ostream* m_debug_os_ptr;

    // Parameters from the OCP Channel:

    // The number of threads
    int m_threads;

    // is MAddrSpace part of the OCP channel?
    bool m_addrspace;

    // is SThreadBusy part of the channel?
    bool m_sthreadbusy;

    // Is SThreadBusy compliance required?
    bool m_sthreadbusy_exact;

    // is MThreadBusy part of the channel?
    bool m_mthreadbusy;

    // Is MThreadBusy compliance required?
    bool m_mthreadbusy_exact;

    // is MRespAccept part of the channel?
    bool m_respaccept;

    // is Data Handshake part of the channel?
    bool m_datahandshake;
```

```
    // is write response part of the channel?
    bool m_writeresp_enable;

    // is the READ-EX command part of the channel
    bool m_readex_enable;

    // Are non-posted writes (write commands that receive responses)
    // part of the channel?
    bool m_writenonpost_enable;

    //----------------------------------------
    //  Master Specific Parameters
    //----------------------------------------

    // Response delay style - fixed or random
    bool m_respaccept_fixeddelay;

    // Delay in accepting responses (max delay for random)
    int m_respaccept_delay;

    // Map of string to string that holds the Master's paramter
values
    OCPParameters m_parameters;

    sc_time m_sthreadbusy_sample_time;
};

#endif // _SIMPLE_MASTER_H
```

6.2.2 Constructor

In the master core model's constructor, the following items are implemented:

- The base `sc_module` class is initialized using the name parameter passed to the Master class.
- The OCP master interface port (`ipP`) is also initialized and named "`ipPort`".
- The clock port (clk) is also initialized and named "`clkPort`".
- The master's configuration and parameters are given their initial default values.
- Functions for sending a request from the master, processing a response from the slave, and for setting sideband signals on the channel are registered using the SystemC `SC_THREAD` macro.
- The time after which getSThreadBusy returns a stable value is guessed.

The following is the code for the constructor of the master core model:

```
// ----------------------------------------------------------------
// constructor
// ----------------------------------------------------------------
template<typename TdataCl>
Master<TdataCl>::Master(
    sc_module_name name,
    int            id,
    ostream*       debug_os_ptr
```

```
) : sc_module(name),
    ipP("ipPort"),
    clk("clkPort"),
    m_ID(id),
    m_debug_os_ptr(debug_os_ptr),
    m_threads(1),
    m_addrspace(false),
    m_sthreadbusy(false),
    m_sthreadbusy_exact(false),
    m_mthreadbusy(false),
    m_mthreadbusy_exact(false),
    m_respaccept(true),
    m_datahandshake(false),
    m_writeresp_enable(false),
    m_writenonpost_enable(false),
    m_respaccept_fixeddelay(1),
    m_respaccept_delay(1)
{
    // setup a SystemC thread process, sensitive to the clock
    SC_THREAD(requestThreadProcess);
    sensitive<<clk.pos();

    // setup a SystemC thread process, sensitive to the clock
    SC_THREAD(responseThreadProcess);
    sensitive<<clk.pos();

    // setup a SystemC thread process to drive any connected
    // sideband signals
    SC_THREAD(exerciseSidebandThreadProcess);
    sensitive<<clk.pos();

    //assuming default timing the master will expect sthreadbusy to
    //be stable 1 ps after the clock edge
    m_sthreadbusy_sample_time=sc_time(1, SC_PS);
}
```

6.2.3 The provideChannelConfiguration() Method

Within the provideChannelConfiguration() method, the master converts the passed String map into an OCPParameters class object, that can be used to configure the connected channel at end_of_elaboration.

The following shows the code for the provideChannelConfiguration() method:

```
template<typename TdataCl>
void Master<TdataCl>::provideChannelConfiguration(MapStringType&
passedMap){
  m_parameters.setOCPConfiguration(name(), passedMap);
}
```

6.2.4 The setModuleConfiguration() Method

The setModuleconfiguration() method is called by the environment sometime after the construction of the master module, but before the end_of_elaboration() method gets called.

Within this method the master uses functions in the `OCPParameters` class that extract integers and Booleans from string formatted parameter maps. For example, the complex looking function call

```
OCPParameters::getBoolOCPConfigValue(myPrefix, paramName,
    m_respaccept_fixeddelay, m_ParamMap)
```

returns *true* if the passed parameter map (`m_ParamMap`) contains a Boolean parameter named by the string "`parameterName`" where "`parameterName`" is the concatenation of "`myPrefix`" and "`paramName`". (Note that "`myPrefix`" is generally not used and set to "". If the parameter map does contain the parameter, the value of `m_respaccept_fixeddelay` is set to the value of that parameter.

The following is the complete code for the setModuleConfigurationMethod:

```
template<typename TdataCl>
void Master<TdataCl>::setModuleConfiguration(MapStringType&
passedMap){
    if (!(OCPParameters::getBoolOCPConfigValue("",
            "mrespaccept_fixeddelay",
            m_respaccept_fixeddelay, passedMap)) ){
        // Could not find the parameter so we keep the default value
#ifdef DEBUG
        cout << "Warning: paramter \"" << "mrespaccept_fixeddelay"
               << "\" was not found in the module parameter map."
               << endl;
        cout << "         setting missing parameter to "
             <<m_respaccept_fixeddelay<<"." << endl;
#endif
    }

    if (!(OCPParameters::getIntOCPConfigValue("",
            "mrespaccept_delay",
            m_respaccept_delay, passedMap)) ) {
        // Could not find the parameter so keep the default value
#ifdef DEBUG
        cout << "Warning: paramter \"" << "mrespaccept_delay"
               << "\" was not found in the module parameter map."
               << endl;
        cout << "         setting missing parameter to "
             <<m_respaccept_delay<<"." << endl;
#endif
    }
}
```

6.2.5 The end_of_elaboration() Method

The end_of_elaboration() method is called by SystemC after the model has been built and connected, but before the simulation begins. Sometime during the construction of the models, the master's `setModuleConfiguration` and `provideChannelConfiguration` functions should have been called with a parameter map of the master's parameters or a parameter map of the channels's configuration, respectively. During the end_of_elaboration`()` method, that master processes these parameter maps to set its own master parameters and to configure the connected channel.

At the end of elaboration point, the OCP channel must have already been connected to the core. The master takes advantage of this to configure the connected channel using the `setOCPMasterConfiguration` function of its port (ipP).

The following are some points regarding the code for the end_of_elaboration() method:

After the master has set all of its parameters, it configures the connected channel with the parameters it got through its provideChannelConfiguration() method using the configuration from cores mechanism:

```
ipP->setOCPMasterConfiguration(m_parameters.Map);
```

Also the master informs the channel (and thereby the connected slave) about the time it will start requests relative to the clock edge. This time is usually the same point of time at which the clock edge occurs, but as soon as SThreadBusy is used, the time requests get started depend on the time SThreadBusy is stable. So during end_of_elaboration() the masters provides this information to the channel using the timing distribution mechanism:

```
ipP->setOCPTL1MasterTiming(myTiming);
```

And additionally the master will also register at the channel as interested in the slaves timing as the master needs to know when SThreadBusy is stable to eventually update its own timing information:

```
ipP->registerTimingSensitiveOCPTL1Master(
                              (OCP_TL1_Slave_TimingIF*) this);
```

The following is code for the `end_of_elaboration` method.

```
// ----------------------------------------------------------------------
// SystemC Method Master::end_of_elaboration()
// ----------------------------------------------------------------------
//
//  At this point, everything has been built and connected.
//  We are now free to get our OCP parameters and to set up our
//  own variables that depend on them.
//
template<typename TdataCl>
void Master<TdataCl>::end_of_elaboration()
{
    // Call the System C version of this function first
    sc_module::end_of_elaboration();

    //---------------------------------------
    //  OCP Parameters
    //---------------------------------------

    // Get the number of threads
    m_threads=m_parameters.threads;

    // This Reference Master is single threaded.
    if (m_threads > 1) {
        cout << "ERROR: Single threaded Master \"" << name()
                 << "\" connected to OCP with " << m_threads
                 << " threads." << endl;
    }

    // is the MAddrSpace field part of the OCP channel?
```

```
    m_addrspace=m_parameters.addrspace;

    // is SThreadBusy part of the channel?
    m_sthreadbusy=m_parameters.sthreadbusy;

    // Is SThreadBusy compliance required?
    m_sthreadbusy_exact=m_parameters.sthreadbusy_exact;

    // is MThreadBusy part of the channel?
    m_mthreadbusy=m_parameters.mthreadbusy;

    // Is MThreadBusy compliance required?
    m_mthreadbusy_exact=m_parameters.mthreadbusy_exact;

    // is MRespAccept part of the channel?
    m_respaccept=m_parameters.respaccept;

    // Just a double check here
    if (m_mthreadbusy_exact && m_respaccept) {
        cout << "ERROR: Master \"" << name()
        << "\" connected to OCP with both MThreadBusy_Exact and
MRespAccept active which are exclusive." << endl;
    }

    // is Data Handshake part of the channel?
    m_datahandshake=m_parameters.datahandshake;
    // if so, quit. This core does not support it.
    assert(m_datahandshake == false);

    // is write response part of the channel?
    m_writeresp_enable=m_parameters.writeresp_enable;

    // is READ-EX part of the channel?
    m_readex_enable=m_parameters.readex_enable;

    // Are non-posted writes (write commands that receive responses)
    //part of the channel?
    m_writenonpost_enable=m_parameters.writenonpost_enable;

#ifdef DEBUG
    cout << "I am configuring a Master!" << endl;
    cout << "Here is my configuration map for Master >"
            << name() << "< that was passed to me." << endl;
    MapStringType::iterator map_it;
    for (map_it = m_parameters.Map.begin();
        map_it != m_parameters.Map.end(); ++map_it) {
        cout << "map[" << map_it->first << "] = "
             << map_it->second << endl;
    }
    cout << endl;
#endif
    ipP->setOCPMasterConfiguration(m_parameters.Map);
    //in case sthreadbusy is part of the channel,
    //this master will not be
    //default timing anymore and gets timing sensitive, too
    if (m_sthreadbusy){
      OCP_TL1_Master_TimingCl myTiming;
```

```
        //requests start after sthreadbusy is stable
        myTiming.RequestGrpStartTime=m_sthreadbusy_sample_time;
        ipP->registerTimingSensitiveOCPTL1Master(
                              (OCP_TL1_Slave_TimingIF*) this);
        ipP->setOCPTL1MasterTiming(myTiming);
      }
  }
```

6.2.6 The setOCPTL1SlaveTiming Method

This method is called whenevr the slave updates its timing information. If this happens the master will check if the provided timing contains some new information. If so, the master will update its internal timing information and provide new timing information to the channel in case the new slave timings affected the master's timings.

In this simple example, the master just checks if the slave decided to delay the assertion or deassertion of SThreadBusy. If this is the case, the master will simply remember to check SThreadBusy 1 ps after the slave sets/unsets it, to be sure its stable in this clock cycle and will inform the channel about the new start time of requests.

The following shows the code for that.

```
template<typename TdataCl>
void Master<TdataCl>::setOCPTL1SlaveTiming(OCP_TL1_Slave_TimingCl
slave_timing){
  if (slave_timing.SThreadBusyStartTime+sc_time(1,SC_PS)
       > m_sthreadbusy_sample_time){
    m_sthreadbusy_sample_time=
        slave_timing.SThreadBusyStartTime+sc_time(1,SC_PS);
    OCP_TL1_Master_TimingCl myTiming;
    myTiming.RequestGrpStartTime=m_sthreadbusy_sample_time;
    ipP->setOCPTL1MasterTiming(myTiming);
  }
}
```

6.2.7 SystemC Request Thread Process

For this master core example, the master request thread process works from a table of requests. The delays between the sending out of each request are also set in a table. For each table entry, the master sends the corresponding request then waits the corresponding time before moving on to the next table entry.

The `Commands` table is the table of commands to send out while the `NumWait` table contains the length of time to wait before sending out the next command. Each time is organized by row with each row being a "test" of up to four commands.

The following is an explanation of the code below:

Sets up the tables to be used by the process. The code then enters the infinite loop of the thread and waits for the first wait period before sending its first request.

After the wait is over, the code checks to see if the slave has set `threadbusy`. Therefore the thread will wait for SThreadBusy to become stable. Note that the parameter `m_sthreadbusy` was set by looking at the OCP channel's parameters during the `end_of_elaboration()` method. If SThreadBusy is part of the channel, and if that signal has been asserted, the request process will continue to wait until the slave releases threadbusy by driving it to zero.

Once the threadbusy hurdle has been cleared, the request process then tries to send a request. First it constructs the request by reading the next command from the table. If the command is incompatible with the channel that the master is connected to, the master changes the command to a simpler one that the channel can accept. If the command calls for data (that is, it is some sort of write command) new data is generated through a counter.

The data is sent with the OCP TL1 channel command:

```
ipP->startOCPRequestBlocking(req);
```

This command places the newly generated request on the channel. If there is already a request on the channel (for example, if the previous request has not yet been accepted), that command will block until the channel is free and the new command can be placed on the channel. The function returns once the request has started, but before it has been accepted by the slave. A blocking call like this one may only be used within a thread process. A SystemC method does not allow the context switching required by a blocking command.

After the blocking call returned the master will wait a cycle to make sure the next attempt to start a request takes place in the next cycle, thereby ensuring the correct value for SThreadBusy is used.

Finally, return to step 1, processing the table and setting up the wait time before the next command may be issued.

The following is the code for the Request Thread Process.

```
template<typename TdataCl>
void Master<TdataCl>::requestThreadProcess()
{
    Ta Addr[] = {0x1784, 0x20, 0x20, 0x40};

    // start time of requests
    int NumWait[NUM_TESTS][4] = {
            {100,   3, 0xF, 0xF},
            {7,   1,   3, 0xF},
            {6, 0xF, 0xF, 0xF},
            {10,  2,   1, 0xF},
            {7,   1,   3, 0xF},
            {6,   1,   1,   1},
            {7,   2, 0xF, 0xF},
            {8,   2,   1, 0xF},// no data handshake
            {7,   2,   2,   2}
        };

    // specifies the command to use
    OCPMCmdType Commands[NUM_TESTS][4]  = {
        {OCP_MCMD_WR, OCP_MCMD_RD, OCP_MCMD_IDLE, OCP_MCMD_IDLE},
        {OCP_MCMD_WR, OCP_MCMD_WR, OCP_MCMD_WR, OCP_MCMD_IDLE},
        {OCP_MCMD_RD, OCP_MCMD_IDLE, OCP_MCMD_IDLE, OCP_MCMD_IDLE},
        {OCP_MCMD_RD, OCP_MCMD_RD, OCP_MCMD_RD, OCP_MCMD_IDLE},
        {OCP_MCMD_RD, OCP_MCMD_RD, OCP_MCMD_RD, OCP_MCMD_IDLE},
        {OCP_MCMD_RD, OCP_MCMD_RD, OCP_MCMD_RD, OCP_MCMD_RD},
        {OCP_MCMD_RD, OCP_MCMD_RD, OCP_MCMD_IDLE, OCP_MCMD_IDLE},
        {OCP_MCMD_WR, OCP_MCMD_WR, OCP_MCMD_WR, OCP_MCMD_IDLE},
        {OCP_MCMD_RD, OCP_MCMD_RD, OCP_MCMD_RD, OCP_MCMD_RD}
    };
```

```
// number of specified transactions in a test
int NumTr[] = {2, 3, 1, 3, 3, 4, 2, 3, 4};

// ------------------------------------
// (1) processing and preparation step
// ------------------------------------

// initialize data
OCPRequestGrp<Td,Ta> req;
int              Count = 0;
int              Nr = 0;
sc_time          old_time;
sc_time          current_time;
bool             sthreadbusy;
unsigned int     my_data = 0;

// calculate the new waiting time
double  wait_for = NumWait[Nr][Count];

// Do requests contain data (or will it be sent separately)
// Always true as this core does not support data handshake
req.HasMData = true;

wait();

// main loop
while (true) {
    // wait for the time to send the current request

    if (m_debug_os_ptr) {
        (*m_debug_os_ptr) << "DB (" << name() << "): "
                    << "master wait_for = " << wait_for
                    << endl;
    }

    for (int i=0; i<wait_for; i++) wait();

    // remember the time
    old_time = sc_time_stamp();

    // ------------------------------------------------
    // (2) is SThreadBusy?
    // ------------------------------------------------

    // NOTE: we are single threaded so the thread busy signal
    // looks like a boolean (0 or 1).
    //   Abritration based on thread busy will be needed for a
    //   multi-threaded model.
    if (m_sthreadbusy_exact) {
        //wait until sthreadbusy is stable
        wait(m_sthreadbusy_sample_time);
        sthreadbusy = ipP->getSThreadBusy();
        while (sthreadbusy) {
            wait();
            //wait until sthreadbusy is stable
            wait(m_sthreadbusy_sample_time);
            sthreadbusy = ipP->getSThreadBusy();
```

```
    }
}

// ------------------------------------------------
// (3) send a request
// ------------------------------------------------

// NOTE: data handshake is not handled by
//       this simple example.

// Compute the next request
req.MCmd = Commands[Nr][Count];

// is this an extended command to be sent over a basic
// channel?
if ( (!m_readex_enable) && (req.MCmd == OCP_MCMD_RDEX) ) {
    // channel cannot handle READ-EX. Send simple READ.
    req.MCmd = OCP_MCMD_RD;
} else if ((!m_writenonpost_enable) &&
           (req.MCmd == OCP_MCMD_WRNP)){
    // channel cannout handle WRITE-NP. Send simple WRITE.
    req.MCmd = OCP_MCMD_WR;
}

// compute the address
req.MAddr = Addr[Count] + m_ID*0x40;
req.MByteEn = 0xf;
if (m_addrspace) {
    req.MAddrSpace = 0x1;
}
// compute the data
switch (req.MCmd) {
    case OCP_MCMD_WR:
    case OCP_MCMD_WRNP:
    case OCP_MCMD_WRC:
    case OCP_MCMD_BCST:
        // This is a write command - it has data
        my_data++;
        // put the data into the request
        req.MData = my_data + m_ID*0x40;
        break;
    case OCP_MCMD_RD:
    case OCP_MCMD_RDEX:
    case OCP_MCMD_RDL:
        // this is a read command - no data.
        req.MData = 0;
        break;
    default:
        cout << "ERROR: Master \"" << name()
             << "\" generates unknown command #"
                  << req.MCmd << endl;
}

if (m_debug_os_ptr) {
    (*m_debug_os_ptr) << "DB (" << name() << "): "
                      << "send request." << endl;
    (*m_debug_os_ptr) << "DB (" << name() << "): "
```

```
                        << "    t = " << sc_simulation_time()
                        << endl;
            (*m_debug_os_ptr) << "DB (" << name() << "): "
                              << "    MCmd: " << req.MCmd << endl;
            (*m_debug_os_ptr) << "DB (" << name() << "): "
                              << "    MData: " << req.MData << endl;
            (*m_debug_os_ptr) << "DB (" << name() << "): "
                             << "    MByteEn: " << req.MByteEn
                             << endl;
        }

        // send the request
        ipP->startOCPRequestBlocking(req);
        // -------------------------------
        // (1) processing and preparation step
        // -------------------------------

        wait(); //advance to next cycle to avoid
                //sending multiple request in same cycle

        // compute the next pointer
        if (++Count >= NumTr[Nr]) {
            Count = 0;
            if (++Nr >= NUM_TESTS) Nr = 1;
        }

        // calculate the new waiting time
        wait_for = NumWait[Nr][Count];
        current_time = sc_time_stamp();
        double delta_time =
              (current_time.value() - old_time.value()) / 1000;
        if (delta_time >= wait_for) {
            wait_for = 0;
        } else {
            wait_for = wait_for - delta_time;
        }
    }
}
```

6.2.8 SystemC Response Thread Process

The code for the master's response thread process is much simpler than that for the request. The code follows this pattern:

- The master receives a response.
- The master waits for a given amount of time.
- The master accepts the response.

The following is an explanation of the code below.

Once the process enters the infinite loop of the thread, it starts waits for a response to come from the slave. The command

```
ipP->getOCPResponseBlocking(resp);
```

gets the current response from the OCP channel that is connected to the `ipP` port. If there is no request waiting on the OCP channel, the command blocks until a new

request arrives. Because this is a blocking command, it may only be used in a thread process like this one. A SystemC method process does not allow for the context switching required by a blocking command.

Once the request has arrived, the response delay is calculated using the master parameters set from the passed parameter map.

The thread implements the delay based on the channel configuration. If the OCP channel has an MRespAccept signal, that signal is used to keep the slave from sending more responses. The following command is used to set MRespAccept to true to accept the response:

```
ipP->putMRespAccept();
```

If instead, the slave is `threadbusy_exact`, the MThreadBusy signal is used to pause the slave. The following command is used to set MThreadBusy to true:

```
ipP->putMThreadBusy(1);
```

The same command (with a different parameter) is used to unset MThreadBusy as well, that is:

```
ipP->putMThreadBusy(0);
```

In between the two calls to `putMThreadBusy()`, the response thread has to wait for `wait_for` OCP channel cycles before resuming.

It is very important to notice the wait() call before setting MThreadBusy. In this way MThreadBusy gets set at the next clock edge (viz. in the next cycle), because MThreadBusy (just like SThreadBusy) refers to the recent cycle (if not pipelined) and so it must not change during a cycle in which a response was received. If it would change immediately due to the response reception, there would be a combinatorial dependency between SResp and MThreadBusy which is explicitly forbidden by the OCP specification.

The following is the code for the master's response thread process.

```
template<typename TdataCl>
void Master<TdataCl>::responseThreadProcess()
{

    // initialization
    OCPResponseGrp<Td> resp;
    double         wait_for;

    wait();

    // main loop
    while (true) {
        // ------------------------------------------------
        // (1) wait for a response (blocking wait)
        // ------------------------------------------------

        // get the next response
        ipP->getOCPResponseBlocking(resp);

        // ------------------------
        // (2) process the response
        // ------------------------
```

```
            // compute the response acceptance time
            if (m_respaccept_fixeddelay) {
               wait_for = m_respaccept_delay;
            } else {
               // Go random up to max delay
               wait_for =
                (int)((m_respaccept_delay+1) * rand() /
                                  (RAND_MAX + 1.0));
            }

            // ------------------------------------------------
            // (3) generate a one-cycle-pulse MRespAccept signal
            // ------------------------------------------------

            if (m_respaccept) {
                if (wait_for == 0) {
                    // send an one-cycle-pulse MRespAccept signal
                    ipP->putMRespAccept();
                } else {
                    // wait for the acceptance pulse cycle
                    for (int i=0; i<wait_for; i++) wait();
                    // send an one-cycle-pulse MRespAccept signal
                    ipP->putMRespAccept();
                }
            }

            if (m_mthreadbusy_exact) {
               // use the MThreadBusy signal instead of resp accept
                if (wait_for > 0) {
                    // Set MThreadBusy
                    wait(); //wait until next cycle to set busy
                    ipP->putMThreadBusy(1);
                    // keep MThreadBusy on
                    for (int i=0; i<wait_for; i++) wait();
                    // now release it
                    ipP->putMThreadBusy(0);
                }
            }
        }
    }
```

6.2.9 SystemC Sideband Process

The code example shown in this section is a simple process that illustrates how the OCP TL1 API can be used to set sideband signals in the OCP channel.

The following is an explanation of the code below.

Before the start of the infinite loop of the thread, the sideband process checks the channel's parameters to determine which (if any) master sideband signals are available in the channel.

Once the code reaches the main loop, the process waits then sets all of the master sideband signals that are connected to it. It updates the values to be set next time and then repeats.

The following is the code for the master's sideband thread process.

```
template<typename TdataCl>
void Master<TdataCl>::exerciseSidebandThreadProcess(void)
{
    // Systematically send out sideband signals on
    // any signals that are attached to us.
    for (int i=0; i<10; i++) wait();
    int tweakCounter =0;
    bool hasMError=m_parameters.merror;
    bool nextMError = false;
    bool hasMFlag=m_parameters.mflag;
    int numMFlag=m_parameters.mflag_wdth;
    unsigned int nextMFlag = 0;
    unsigned int maxMFlag = (1 << numMFlag) -1;

    // main loop
    while (true) {
        // wait 10 cycles
        for (int i=0; i<10; i++) wait();

        // Now count through my sideband changes
        tweakCounter++;

        // Drive MError
        if (hasMError) {
            if (tweakCounter%2 == 0) {
                // Toggle MERROR
                nextMError = !nextMError;
                ipP->MputMError(nextMError);
            }
        }

        // Drive MFlags
        if (hasMFlag) {
            if (tweakCounter%1 == 0) {
                // go to next MFlag
                nextMFlag += 1;
                if (nextMFlag > maxMFlag) {
                    nextMFlag = 0;
                }
                ipP->MputMFlag(nextMFlag);
            }
        }
    }
}
```

6.2.10 Template Instantiation

The final line of the `master.cc` file makes sure that the compiler creates an instance of the Master template for the `OCP_TL1_SIGNAL_CL` type defined in the `globals.h` header file. The last line is

```
template class Master< OCP_TL1_SIGNAL_CL >;
```

6.3 A Configurable Slave Model

This section provides an example of a configurable slave model, which reacts like a target memory core and takes in or delays the acceptances of OCP requests based on parameterized settings. The slave model has a single-threaded slave OCP interface. This slave model not only has its own parameters but can also deal with different OCP parameter settings. For instance, the slave model can talk to an OCP channel with the following settings:

- cmdaccept == 1, sthreadbusy == 0 or 1, and sthreadbusy_exact == 0
- cmdaccept == 0, sthreadbusy == 1, and sthreadbusy_exact == 1
- respaccept == 0, mthreadbusy == 0, and mthreadbusy_exact == 0
- respaccept == 1, mthreadbusy == 0 or 1, and mthreadbusy_exact == 1
- respaccept == 0, mthreadbusy == 1, and mthreadbusy_exact == 1

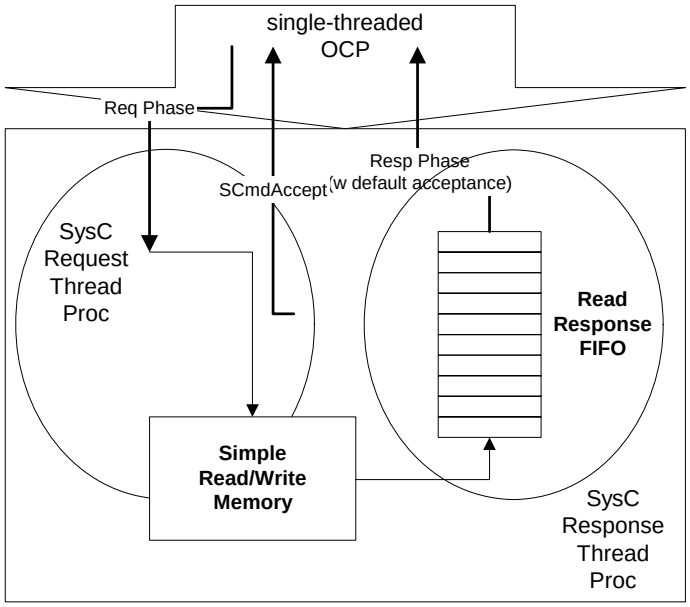
Parameters belonging to the slave model itself are:

`latencyX`
: This is the response latency for thread number X. There is a latency parameter for each thread in the channel. This parameter sets the minimum number of cycles between receiving the request and issuing the response.

`limitreq_max`
: The outstanding requests per thread are limited to `limitreq_max`

Figure 20 shows a diagram of the configurable slave model.

Figure 20 Slave Model



Since the supported configurations include sthreadbusy and mthreadbusy, the timing distribution mechanism as described in section 2.6 is also used by this slave.

6.3.1 Header File

The header file for the simple configurable slave calls the header files for the channel it is connected to and for the objects it uses. It then defines the template class that is the slave. The following are a few explanations regarding some of the highlights of the code. The full header file is provided below.

First, the slave includes the OCP TL1 channel header files:

```
// OCP-IP Channel header files
#include "globals.h"
#include "ocp_tl1_slave_port.h"
```

The file globals.h contains the definitions of the types used in the channel. This file also includes the header `ocp_tl1_data_cl.h` that defines the data class used by the OCP TL1 channel. The header `ocp_tl1_data_cl.h` in turn includes `ocp_globals.h`, which is used to define the structures used to pass requests and responses to the channel. If this core did not have an include file like `globals.h`, it would need to directly include `ocp_globals.h`.

The header ocp_tl1_slave_port.h is the slave port to the OCP TL1 channel. In addition to providing the slave interface to the channel, the port also provides event finders for all of the slave events and sideband events of the channel.

The header file then defines objects that are used by the slave. The file slave_response_queue.h defines a simple response queue that the slave uses to queue responses as they are waiting to go out on the channel. The file MemoryCl.h implements a simple memory.

Following the include statements, the slave header file defines the slave class. The slave is a template class and the parameter of the template is the data class that the slave will support over the OCP connection. A data class with 32-bit data width and a 32-bit address is specified as follows:

```
OCP_TL1_DataCl<OCPCHANNELBit32, OCPCHANNELBit32>
```

Where `OCPCHANNELBit32` is defined in the file `globals.h` as

```
typedef unsigned int OCPCHANNELBit32;
```

The simple configurable slave has a single port which connects to the OCP channel. The following code declares the slave port for the OCP channel:

```
// channel port
OCP_TL1_SlavePort<TdataCl> tpP;
```

The example slave does also posses a second port:

```
sc_in_clk clk;
```

This is the slave's clock port, through which it can access the system clock, which is provided by the environment.

Next the `slave` class declares functions that define SystemC thread or method processes used in your model. For example, in this slave core model, the following functions are defined:

```
// has SystemC processes
SC_HAS_PROCESS(Slave);
void requestThreadProcess();
void responseThreadProcess();
void exerciseSidebandThreadProcess();
```

The `SC_HAS_PROCESS(Slave)` macro tells SystemC that the slave core is a SystemC module with its own processes. In this case, the thread processes that follow. Each of these processes are explained in detail in below.

Lastly, the `Slave` class defines a SystemC `end_of_elaboration` function to be called automatically after all models are built and connected but just before the simulation is to start:

```
void end_of_elaboration();    // SystemC method
```

Following the declaration of the `end_of_elaboration` method, the `Slave` class defines a member of type OCPParameters within which the slave may store the configuration of the channel:

```
OCPParameters  m_parameters;
```

The rest of the data members of the `Slave` class hold the parameter and configuration values of the slave.

The following is the complete header file for the slave.

```
#ifndef _SIMPLE_SLAVE_H
#define _SIMPLE_SLAVE_H

#include <map>

// OCP-IP Channel header files
#include "globals.h"
#include "ocp_tl1_slave_port.h"
#include "ocp_tl_param_cl.h"

#include "slave_response_queue.h"

#include "MemoryCl.h"


// define the Slave class
template <typename TdataCl>
class Slave : public sc_module, public OCP_TL_Config_Listener,
public OCP_TL1_Master_TimingIF
{
  public:
    // -------------------------
    // public members and methods
    // -------------------------

    // type definitions
    typedef typename TdataCl::DataType Td;
    typedef typename TdataCl::AddrType Ta;
```

```
  typedef map< Ta, Td > MemMapType;

  // member definitions

  // channel port
  OCP_TL1_SlavePort<TdataCl> tpP;
  sc_in_clk clk;

  // Systemc macros

  // has SystemC processes
  SC_HAS_PROCESS(Slave);

  // constructor and destructor
  Slave(sc_module_name,
        int, Ta, ostream* debug_os_ptr = NULL);
  ~Slave();

  void requestThreadProcess();
  void responseThreadProcess();
  void exerciseSidebandThreadProcess();

  void set_configuration(OCPParameters& , std::string);
  void setOCPTL1MasterTiming(OCP_TL1_Master_TimingCl);
  void setModuleConfiguration(MapStringType&);

private:
  // ----------------------------
  // private members and methods
  // ----------------------------

  // SystemC methods
  void end_of_elaboration();

  // member definitions

  // slave identification
  int  m_ID;

  // ocp clock information
  sc_time m_clkPeriod;

  // number of memory bytes and the memory array
  Ta   m_MemoryByteSize;

  // model a per thread response queue
  ResponseQueue<TdataCl> m_ResponseQueue;

  MemoryCl<TdataCl> *m_Memory;

  ostream* m_debug_os_ptr;

  // current value of SThreadBusy as set by this Slave.
  int m_curSThreadBusy;

  // ------------------------------------------------------------
  //  Parameters of the connected OCP channel
```

```
    // ------------------------------------------------------------

    // Number of threads in the OCP channel
    int m_threads;

    // Does the channel use data handshaking?
    bool m_datahandshake;

    // Are writes with responses part of the OCP channel?
    bool m_writeresp_enable;

    // is SThreadBusy part of the OCP channel?
    bool m_sthreadbusy;

    // do we follow the rules of sthread_busy exact?
    bool m_sthreadbusy_exact;

    // is MThreadBusy part of the OCP channel?
    bool m_mthreadbusy;

    // is SCmdAccept part of the OCP channel?
    bool m_cmdaccept;

    // ------------------------------------------------------------
    //  Parameters of the Slave Model
    // ------------------------------------------------------------

    // maximum number of outstanding requests per thread
    // default = 4;
    int m_limitreq_max;

    // Response Latency
    int m_Latency;

    OCPParameters m_parameters;

    sc_time m_mthreadbusy_sample_time;

};

#endif // _SIMPLE_SLAVE_H
```

6.3.2 Constructor

In the slave model's constructor, the following items are implemented:

- The base `sc_module` class is initialized using the name parameter passed to the `Slave` class.
- The OCP slave interface port (`tpP`) is also initialized and named "`tpPort`".
- The clock port (clk) is also initialized and named "`clkPort`".
- The slave's configuration and parameters are given their initial default values. They will receive their parameter values at the end of elaboration.
- Functions for receiving requests, sending responses and for checking sideband signals on the channel are registered using the SystemC `SC_THREAD` macro.

- The time after which getMThreadBusy returns a stable value is guessed.

The following is the code for the constructor.

```
// ----------------------------------------------------------------------
// constructor
// ----------------------------------------------------------------------
template<typename TdataCl>
Slave<TdataCl>::Slave(
    sc_module_name n,
    int            id,
    Ta             memory_byte_size,
    ostream*       debug_os_ptr
) : sc_module(n),
    tpP("tpPort"),
    clk("clkPort"),
    m_ID(id),
    m_MemoryByteSize(memory_byte_size),
    m_Memory(NULL),
    m_debug_os_ptr(debug_os_ptr),
    m_curSThreadBusy(0),
    m_threads(1),
    m_datahandshake(false),
    m_writeresp_enable(false),
    m_sthreadbusy(false),
    m_sthreadbusy_exact(false),
    m_mthreadbusy(false),
    m_cmdaccept(true),
    m_limitreq_max(4),
    m_Latency(3)
{
    // Note: member variables that depend on values of
    // configuration parameters are constructed when those
    // values are known - at the end of elaboration.

    // setup a SystemC thread process, sensitive to the clock
    SC_THREAD(requestThreadProcess);
    sensitive<<clk.pos();

    // setup a SystemC thread process, sensitive to the clock
    SC_THREAD(responseThreadProcess);
    sensitive<<clk.pos();

    // setup a SystemC thread process to check and
    // set sideband signals
    SC_THREAD(exerciseSidebandThreadProcess);
    sensitive<<clk.pos();

    assert(m_Latency);

    //assuming default timing the slave will expect mthreadbusy to
    // be stable 1 ps after the clock edge
    m_mthreadbusy_sample_time=sc_time(1, SC_PS);
}
```

6.3.3 Destructor

The destructor cleans up the memory created in the `end_of_elaboration()` function.

The following is the code for the destructor.

```
template<typename TdataCl>
Slave<TdataCl>::~Slave()
{
    delete m_Memory;
}
```

6.3.4 The set_configuration() Method

After the slave has registered to the channel as a configuration listener, this method is called whenever the channel gets configured, either by the master or the environment.

Within this method the slave adjusts its internal configuration according to the passed channel configuration. Additionally the memory class object of the slave gets initialized. Note that this method could get triggered multiple times (if both master and environment configure the channel) so adequate precautions have to be taken.

The following shows the code of the set_configuration() method:

```
template<typename TdataCl>
void Slave<TdataCl>::set_configuration(OCPParameters& passedConfig,
std::string channelName){
    m_parameters=passedConfig;
    // Set the number of threads
    m_threads=passedConfig.threads;

    if (m_threads > 1) {
        cout << "Warning: Singled threaded reference Slave "
            << name() << " attached to multi-threaded OCP." << endl;
        cout << "Only commands sent on thread 0 will be processed."
            << endl;
    }

    // Does the channel use data handshaking?
    m_datahandshake=passedConfig.datahandshake;
    // Is so, quit as this Slave does not handle data handshake.
    assert(!m_datahandshake);

    // Do writes get reponses?
    m_writeresp_enable=passedConfig.writeresp_enable;

    // is SThreadBusy part of the channel?
    m_sthreadbusy=passedConfig.sthreadbusy;

    // is this slave expected to follow the threadbusy
    // exact protocol?
    m_sthreadbusy_exact=passedConfig.sthreadbusy_exact;

    // is MThreadBusy part of the channel?
    m_mthreadbusy=passedConfig.mthreadbusy;

    // is SCmdAccept part of the channel?
    m_cmdaccept=passedConfig.cmdaccept;
```

```
        // For Debugging
        if (m_debug_os_ptr) {
            (*m_debug_os_ptr) << "DB (" << name() << "): "
                << "Configuring Slave." << endl;
            (*m_debug_os_ptr) << "DB ("
                << name()
                   << "): was passed the following configuration map:"
                   << endl;
            MapStringType::iterator map_it;
            for (map_it = passedConfig.Map.begin();
                    map_it != passedConfig.Map.end(); ++map_it) {
                (*m_debug_os_ptr) << "map[" << map_it->first << "] = "
                        << map_it->second << endl;
            }
            cout << endl;
        }

        // Create the memory:

        if (m_Memory) {
            // Just in case we are called multiple times.
            delete m_Memory;
        }
        char id_buff[10];
        sprintf(id_buff,"%d",m_ID);
        string my_id(id_buff);
        m_Memory =
           new MemoryCl<TdataCl>(my_id,
                  passedConfig.addr_wdth,sizeof(Td));

    }
```

6.3.5 The setModuleConfiguration() Method

The setModuleconfiguration() method is called by the environment sometime after the construction of the slave module, but before the end_of_elaboration() method gets called.

Within this method the master uses functions in the `OCPParameters` class that extract integers and Booleans from string formatted parameter maps. For example, the complex looking function call

```
OCPParameters::getBoolOCPConfigValue(myPrefix, paramName,
    m_respaccept_fixeddelay, m_ParamMap)
```

returns *true* if the passed parameter map (`m_ParamMap`) contains a Boolean parameter named by the string "`parameterName`" where "`parameterName`" is the concatenation of "`myPrefix`" and "`paramName`". (Note that "`myPrefix`" is generally not used and set to "". If the parameter map does contain the parameter, the value of `m_respaccept_fixeddelay` is set to the value of that parameter.

The following is the complete code for the setModuleConfigurationMethod:

```
template<typename TdataCl>
void Slave<TdataCl>::setModuleConfiguration(MapStringType&
passedMap){
```

```
    if (!(OCPParameters::getIntOCPConfigValue("", "limitreq_max",
            m_limitreq_max, passedMap)) ) {
        // Could not find the parameter so we keep the default value
#ifdef DEBUG
        cout << "Warning: paramter \"" << "limitreq_max"
               << "\" was not found in the module parameter map."
               << endl;
        cout << "         setting missing parameter to "
             <<m_limitreq_max<<"." << endl;
#endif
    }

    if (!(OCPParameters::getIntOCPConfigValue("", "latency0",
            m_Latency, passedMap)) ) {
        // Could not find the parameter so we keep the default value
#ifdef DEBUG
        cout << "Warning: paramter \"" << "latency0"
               << "\" was not found in the module parameter map."
               << endl;
        cout << "         setting missing parameter to "
             <<m_Latency<<"." << endl;
#endif
    }
}
```

6.3.6 The end_of_elaboration() Method

This function is automatically called after the model has been built and connected but before the simulation begins. At the end of elaboration point, the OCP channel must have already been connected to the core.

The following are some points regarding the code for the end_of_elaboration() method:

After the slave has reset its response queue, it tries to get a pointer to the sc_clock that is connected to its clock port. If that fails, the simulation will terminate, since the slave needs a pointer to the clock to get information about the clock period. This information is needed by the slave to compute the response delays appropriately.

Also the slave informs the channel (and thereby the connected master) about the time it will set/unset SThreadBusy relative to the clock edge. This time is usually the same point of time at which the clock edge occurs, but as soon as MThreadBusy is used, the time SThreadBusy gets unset depends on the time MThreadBusy is stable. So during end_of_elaboration() the slave provides this information to the channel using the timing distribution mechanism:

```
tpP->setOCPTL1SlaveTiming(myTiming);
```

And additionally the slave will also register at the channel as interested in the masters timing as the slave needs to know when MThreadBusy is stable to eventually update its own timing information:

```
tpP->registerTimingSensitiveOCPTL1Slave(
                              (OCP_TL1_Master_TimingIF*) this);
```

The following is code for the `end_of_elaboration` method.

```
// ----------------------------------------------------------------
//  SystemC Method Slave::end_of_elaboration()
// ----------------------------------------------------------------
```

```
//
//  At this point, everything has been built and connected.
//  We are now free to get our OCP parameters and to set up our
//  own variables that depend on them.
//
template<typename TdataCl>
void Slave<TdataCl>::end_of_elaboration()
{
    sc_module::end_of_elaboration();
    // Clear the response queue
    m_ResponseQueue.resetState();

    //get clock period information
    if (dynamic_cast<sc_clock*>(clk.get_interface())){
      m_clkPeriod=
        (dynamic_cast<sc_clock*>(clk.get_interface()))->period();
    }
    else{
      cout<<"An sc_clock has to be connected to"
          <<" the clock port of "<<name()<<endl<<flush;
      assert(dynamic_cast<sc_clock*>(clk.get_interface()));
    }
    tpP->addOCPConfigurationListener(*this);

    if (m_mthreadbusy){
      OCP_TL1_Slave_TimingCl myTiming;
      //sthreadbusy gets deasserted after mthreadbusy got read
      //(see responseThreadProcess)
      myTiming.SThreadBusyStartTime=m_mthreadbusy_sample_time;
      tpP->registerTimingSensitiveOCPTL1Slave(
                        (OCP_TL1_Master_TimingIF*) this);
      tpP->setOCPTL1SlaveTiming(myTiming);
    }
}
```

6.3.7 The setOCPTL1MasterTiming Method

This method is called whenever the master updates its timing information. If this happens the slave will check if the provided timing contains some new information. If so, the slave will update its internal timing information and provide new timing information to the channel in case the new master timings affected the slave's timings.

In this simple example, the slave just checks if the master decided to delay the assertion or deassertion of MThreadBusy. If this is the case, the master will simply remember to check MThreadBusy 1 ps after the slave sets/unsets it, to be sure its stable in this clock cycle and will inform the channel about the new start time of SThreadBusy (which depends on MThreadBusy as will be seen in the ResponseThreadProcess).

The following shows the code for that.

```
template<typename TdataCl>
void Slave<TdataCl>::setOCPTL1MasterTiming(OCP_TL1_Master_TimingCl
master_timing){
  if (master_timing.MThreadBusyStartTime+sc_time(1, SC_PS) >
          m_mthreadbusy_sample_time){
    OCP_TL1_Slave_TimingCl myTiming;
```

```
        m_mthreadbusy_sample_time=
             master_timing.MThreadBusyStartTime+sc_time(1, SC_PS);
        myTiming.SThreadBusyStartTime=m_mthreadbusy_sample_time;
        tpP->setOCPTL1SlaveTiming(myTiming);
      }
    }
```

6.3.8 SystemC Request Thread Process

The request thread processes each new request as it arrives from the channel. This section explains some highlights of the code for the request thread process. The complete code for the request process is presented below.

The basis loop of the request thread process does the following: gets a new request, processes it, generates a response (if needed), then queues that response for the response thread to process. The request thread uses a blocking command to get the next request:

```
tpP->getOCPRequestBlocking(req, false);
```

This command gets the current request from the channel if there is one. If there is no request, the command blocks until a new request arrives. When a request is found, it is copied into the variable `req`. The second parameter to the command (`false`) indicates that the command should not automatically accept the request it receives. The thread then processes the command. Either it updates the memory (for a write command) or it extracts a value from the memory for a read command.

After receiving a request, the process then builds a response. In this slave model, all requests generate a response for the response queue. Some are actual responses such as the responses to a read request. These responses have SResp of type `OCP_SRESP_DV`A. Some of the responses are just placeholder responses. They are there to make sure that the timing for activities such as writes are accurate. Placeholder responses take up a spot in the response queue, but they have an SResp type of `OCP_SRESP_NULL` and are never sent on the OCP channel. Each item in the outgoing response queue consists of a response and a time stamp of the earliest time that the response may be sent (if it is an actual response) or cleared from the queue (if it is a place-holder response).

Note in the code (see comment 2 in the code below) how each element of the response structure is set by the slave. For example, the following line sets the response type of the out going response:

```
resp.SResp = OCP_SRESP_DVA;
```

If the outgoing response queue is full, the slave can no longer accept any new requests. Based on the configuration of the channel, the slave uses either SThreadBusy or a delay on accepting the request to keep the master from sending any new requests that cannot be processed due to the full queue (see comment 4 in the code below). Also note the wait() call before setting SThreadBusy. In this way SThreadBusy gets set at the next clock edge (viz. in the next cycle), because SThreadBusy (just like MThreadBusy) refers to the recent cycle (if not pipelined) and so it must not change during a cycle in which a request was received. If it would change immedialtely due to the reqeust reception, there would be a combinatorial dependency between MCmd and SThreadBusy which is explicitly forbidden by the OCP specification.

The following is the complete code for the slave's request thread process.

```
template<typename TdataCl>
void Slave<TdataCl>::requestThreadProcess()
```

```
{
    // The new request we have just received
    OCPRequestGrp<Td,Ta> req;

    // The response to the new request
    OCPResponseGrp<Td>   resp;

    // Time after which the response can be sent or this
    // request can be cleared from incoming queue.
    sc_time          send_time;

    // We are in the initialization call.
    // Wait for the first simulation cycle.
    wait();

    // main loop
    while (true) {
        // ------------------------------------------------------------
        // (1) Get the next request
        // ------------------------------------------------------------
        tpP->getOCPRequestBlocking(req,false);
        // ------------------------------------------------------------
        // (2) process the new request and generate a response.
        // ------------------------------------------------------------

        // compute the word address
        if (req.MAddr >= m_MemoryByteSize) {
            req.MAddr = req.MAddr - m_MemoryByteSize;
        }

        // send a response for writes if channel requires it.
        if ( m_writeresp_enable && (req.MCmd == OCP_MCMD_WR) ) {
            req.MCmd = OCP_MCMD_WRNP;
        }

        // write to or read from the memory
        switch (req.MCmd) {
            case OCP_MCMD_WR:
                // posted write to memory
                m_Memory->write(req.MAddr,req.MData,req.MByteEn);

                // note that posted writes do not have responses.
                // However, they do have a processing delay that can
                // contribute to a max request limit back up.
                // To solve this problem, requests that have no
                // response to generate a dummy respose with
                // SRESP=NULL which is defined as "No response".
                // Dummy responses are never sent out on the
                // channel.
                resp.SResp = OCP_SRESP_NULL;
                resp.SThreadID = req.MThreadID;
                break;

            case OCP_MCMD_RD:
            case OCP_MCMD_RDEX:
                // NOTE that for a single threaded slave,
                // Read-EX works just like Read
```

```
                    // read from memory
                    m_Memory->read(req.MAddr,resp.SData,req.MByteEn);
                    // setup a read response
                    resp.SResp = OCP_SRESP_DVA;
                    resp.SThreadID = req.MThreadID;
                    break;

                case OCP_MCMD_WRNP:
                    // Generate an acknowledgement response
                    resp.SResp = OCP_SRESP_DVA;
                    resp.SThreadID = req.MThreadID;
                    resp.SData = 0;
                    break;

                default:
                    cout << "MCmd #" << req.MCmd << " not supported
yet."
                         << endl;
                    sc_stop();
                    break;
            }

            // ------------------------------------------------------------
            // (3) generate a completion time stamp and add the response
            //     to the queue
            // ------------------------------------------------------------

            // compute pipelined response delay
            send_time = sc_time_stamp() + m_clkPeriod*m_Latency;

            // purge the queue of any posted write place holder
            // responses that have reached their send times
            m_ResponseQueue.purgePlaceholders();

            m_ResponseQueue.enqueueBlocking(resp, send_time);

            // ------------------------------------------------------------
            // (4) if our queue is full, generate back pressure halt
            //     the flow of requests. Otherwise, accept the request
            //     and move on.
            // ------------------------------------------------------------

            // Do we need to set SThreadBusy??
            if (m_sthreadbusy && (m_ResponseQueue.length() >=
                  m_limitreq_max)) {
                wait(); //wait until next cycle to set busy
                m_curSThreadBusy = 1;
                tpP->putSThreadBusy(m_curSThreadBusy);
            }

            // Should we accept this command?
            if ( m_cmdaccept ) {
                // if queue is full, delay accepting request
                while (m_ResponseQueue.length() >= m_limitreq_max) {
                    // Our queue is full. Wait for this to change.
                    wait();
                }
```

```
                // now it is okay to accept the request
                tpP->putSCmdAccept();
            }

        }
    }
```

6.3.9 SystemC Response Thread Process

The response thread process cycles through the response queues, and then places each response into the channel at the appropriate time. This section explains some highlights of the code for the response thread process. The complete code for the request process is presented below.

The basis loop of the response thread process does the following:

Clears and processes any writes that do not need a response, then it finds the next response to send out (if any)

Builds the response, makes sure the channel is free, then places the new response on the channel.

If no more responses are available to be sent, the process waits until responses arrive.

In case a read response is to be placed on the channel and MThreadBusy is part of the channel, the following loop checks to see if the master's MThreadbusy signal is true for our thread (thread zero). As long as the master keeps this signal high, the slave must wait before sending a new response on that thread.

```
if (m_mthreadbusy) {
  wait(m_mthreadbusy_sample_time);
  mthreadbusy = tpP->getMThreadBusy();
  while (mthreadbusy & 1) {
    wait();
    wait(m_mthreadbusy_sample_time);
    mthreadbusy = tpP->getMThreadBusy();
  }
}
```

Note that the slave always wait for MThreadBusy to be stable before reading it.

The following command will try to place the passed response unto the channel:

```
tpP->startOCPResponseBlocking(resp);
```

If the channel is busy (that is, there is already a response on the channel waiting to be accepted, the command will block until the response can be placed on the channel. Note that this command returns once the response has been placed on the channel, but before the response has been accepted by the master. Furthermore it is important to understand that if MThreadBusy was used the call will never block, because it will only be attempted when the master was not busy. As a consequence the call behaves as if it was non-blocking and so the code section that resets SThreadBusy will be reached immediately. That's why the SThreadBusy start time depends on the MThreadBusy sample time.

The following is the complete code for the Response Thread Process.

```
template<typename TdataCl>
void Slave<TdataCl>::responseThreadProcess()
{
    OCPResponseGrp<Td>   resp;
```

```
sc_time          send_time;
unsigned int     mthreadbusy;

wait();

// main loop
while (true) {

    // ------------------------------------------------
    // (1) Find a response to place on the channel
    // ------------------------------------------------

    // We are single threaded - always choose thread zero:
    int selectedThread = 0;

    // Get to next response (wait for one, if necessary).

    // First, clear any stale write latency waits
    m_ResponseQueue.purgePlaceholders();

    // Get the next request off of the queue
    m_ResponseQueue.dequeueBlocking(resp,send_time);
    resp.SThreadID = selectedThread;

    // check if we still need to wait
    while (send_time > sc_time_stamp()) {
        wait();
    }
    if (m_debug_os_ptr) {
        (*m_debug_os_ptr) << "DB (" << name() << "): "
            << "slave wait time = "
                << send_time.value() << endl;
    }

    // The response could be a place holder response
    // used to implement write latency. If this is the case,
    // skip the rest of the steps.

    if (resp.SResp == OCP_SRESP_NULL) {
        if (m_debug_os_ptr) {
            (*m_debug_os_ptr) << "DB (" << name() << "): "
                << "finished Write Latency waiting." << endl;
        }
    } else {

        // ----------------------------------
        // (2) is MThreadBusy?
        // ----------------------------------

        if (m_mthreadbusy) {
            wait(m_mthreadbusy_sample_time);
            mthreadbusy = tpP->getMThreadBusy();
            while (mthreadbusy & 1) {
                wait();
                wait(m_mthreadbusy_sample_time);
                mthreadbusy = tpP->getMThreadBusy();
            }
```

```
            }

            // ----------------------------------
            // (3) return a response
            // ----------------------------------

            if (m_debug_os_ptr) {
                (*m_debug_os_ptr) << "DB (" << name() << "): "
                    << "send response." << endl;
                (*m_debug_os_ptr) << "DB (" << name() << "): "
                    << "    t = " << sc_simulation_time() << endl;
                (*m_debug_os_ptr) << "DB (" << name() << "): "
                    << "    SResp: " << resp.SResp << endl;
                (*m_debug_os_ptr) << "DB (" << name() << "): "
                    << "    SData: " << resp.SData << endl;
            }

            // Send out the response
            tpP->startOCPResponseBlocking(resp);
        }

        // We must be able to clear ThreadBusy now as we just sent a
        // request (or cleared a write latency)
        if ( m_sthreadbusy && (m_curSThreadBusy==1) &&
                (m_ResponseQueue.length() < m_limitreq_max) ) {
            // Our queue has been shortened. Clear threadBusy.
            m_curSThreadBusy = 0;
            tpP->putSThreadBusy(m_curSThreadBusy);
        }

        // wait until next cycle to send out the next response
        // (if any)
        wait();

    }
}
```

6.3.10 The Sideband Thread Process

This slave process demonstrates how the sideband signals on the channel may be exercised. The code below reads the MError signal and then uses that to set the SError signal. This process also periodically changes the SInterrupt and SFlag signals as well.

The following is the complete code for the Sideband Thread Process.

```
// Exercises the sideband signals by setting them with
// a recurring pattern
// Also loops back error signal from the Master
// if both Master and Slave
// versions (MError and SError) are configured into the channel
template<typename TdataCl>
void Slave<TdataCl>::exerciseSidebandThreadProcess()
{
    // Systematically send out sideband signals on any signals
    // that are attached to us.
    for (int i=0; i<10;i++) wait();
    int tweakCounter =0;
    bool hasMError=m_parameters.merror;
```

```
        bool hasSError=m_parameters.serror;
        bool nextSError = false;
        bool hasSInterrupt=m_parameters.interrupt;
        bool nextSInterrupt = false;
        bool hasSFlag=m_parameters.sflag;
        int numSFlag=m_parameters.sflag_wdth;
        unsigned int nextSFlag = 0;
        unsigned int maxSFlag = (1 << numSFlag) -1;

        // main loop
        while (true) {
            // wait 10 cycles
            for (int i=0; i<10;i++) wait();

            // Now count through my sideband changes
            tweakCounter++;

            // Drive SError every time we are called
            if (hasSError) {
                if (hasMError) {
                    // loop MError back through SError
                    nextSError=tpP->SgetMError();
                    tpP->SputSError(nextSError);
                } else {
                    // Toggle SError
                    nextSError = !nextSError;
                    tpP->SputSError(nextSError);
                }
            }

            // Drive SInterrupt
            if (hasSInterrupt) {
                // Drive every other time we are called
                if (tweakCounter%2 == 0) {
                    // Toggle SInterrupt
                    nextSInterrupt = !nextSInterrupt;
                    tpP->SputSInterrupt(nextSInterrupt);
                }
            }

            // Drive SFlag
            if (hasSFlag) {
                // Drive every fourth time we are called
                if (tweakCounter%4 == 0) {
                    nextSFlag += 1;
                    if (nextSFlag > maxSFlag) {
                        nextSFlag = 0;
                    }
                    tpP->SputSFlag(nextSFlag);
                }
            }
        } // end while
    }
```

6.3.11 Template Instantiation

The final line of the `slave.cc` file makes sure that the compiler creates an instance of the Slave template for the `OCP_TL1_SIGNAL_CL` type defined in the `globals.h` header file. The final line is as follows:

```
// ------------------------------------------------
// explicit instantiation of the Slave template class
// ------------------------------------------------
template class Slave< OCP_TL1_SIGNAL_CL >;
```

6.4 The Main Program

The `main.cc` program processes its command line options with the `process_command_line()` function, then reads in the configuration parameters for the channel, master, and slave. The configuration files are converted into the STL maps in the `readMapFromFile()` function. The `main.cc` program then creates a channel and uses the new channel configuration map to configure it. The program then does the same for the master and slave. Finally, it connects the master to the channel and the slave to the channel.

Once the model has been build, the `main.cc` program calls the SystemC function:

```
sc_start(simulation_end_time,SC_NS);
```

that runs the simulation for `simulation_end_time` nano-seconds. After the simulation has completed, some minimal reporting is done.

The following is the complete code of the `main.cc` program.

```
////////////////////////////
//
// Simple Main to read in Map data from files
// and then use that to configure and connect
// a master and slave.
//
////////////////////////////

#include <map>
#include <set>
#include <string>
#include <algorithm>
#include <stdio.h>
#include <stdlib.h>
#include <iostream>

#include "systemc.h"
#include "channel_types.h"

#include "simpleMaster.h"
#include "simpleSlave.h"
#include "ocp_tl1_data_cl.h"
#include "ocp_tl_param_cl.h"
#include "ocp_tl1_channel_clocked.h"

#define OCP_CLOCK_PERIOD 1
#define OCP_CLOCK_TIME_UNIT SC_NS
```

```
void process_command_line(int   argc,
                          char* argv[],
                          string& ocp_params_file_name,
                          string& module_params_file_name,
                          double& simulation_end_time,
                          bool& debug_dump,
                          string& debug_file_name)
{
    // get the ocp parameters file name
    ocp_params_file_name = "";
    if (argc > 1) {
        string file_name(argv[1]);
        ocp_params_file_name = file_name;
    }

    // get the module parameters file name
    module_params_file_name = "";
    if (argc > 2) {
        string file_name(argv[2]);
        module_params_file_name = file_name;
    }

    // get the simulation end time
    simulation_end_time = 1000;
    if (argc > 3) {
        simulation_end_time = (double) atoll(argv[3]);
    }

    // do we dump out a log file?
    debug_dump= false;
    debug_file_name = "";
    if (argc > 4) {
        string file_name(argv[4]);
        debug_file_name = file_name;
        debug_dump = true;
    }
}

void readMapFromFile(const string &myFileName, MapStringType
&myParamMap)
{
    // read pairs of data from the passed file
    string leftside;
    string rightside;

    // (1) open the file
    ifstream inputfile(myFileName.c_str());
    assert( inputfile );

    // set the formatting
    inputfile.setf(std::ios::skipws);

    // Now read through all the pairs of values and add them to the
passed map
    while ( inputfile ) {
        inputfile >> leftside;
        inputfile >> rightside;
```

```
        myParamMap.insert(std::make_pair(leftside,rightside));
    }

    // All done, close up
    inputfile.close();
}

int
sc_main(int argc, char* argv[])
{
    OCP_TL1_Channel_Clocked< OCP_TL1_DataCl<OCPCHANNELBit32,
OCPCHANNELBit32> >* pOCP;
    Master< OCP_TL1_DataCl<OCPCHANNELBit32, OCPCHANNELBit32> >*
pMaster;
    Slave< OCP_TL1_DataCl<OCPCHANNELBit32, OCPCHANNELBit32> >*
pSlave;
    MapStringType  ocpParamMap;
    MapStringType  moduleParamMap;

    double         simulation_end_time;
    bool           debug_dump;
    string         ocpParamFileName;
    string         moduleParamFileName;
    string         dump_file_name;
    ofstream       debugFile;

    // --------------------------------
    // (1) process command line options
    //     and read my parameters
    // --------------------------------

process_command_line(argc,argv,ocpParamFileName,moduleParamFileName,
           simulation_end_time,debug_dump,dump_file_name);

    if ( ! ocpParamFileName.empty() ) {
        readMapFromFile(ocpParamFileName, ocpParamMap);
    }

    if ( ! moduleParamFileName.empty() ) {
        readMapFromFile(moduleParamFileName, moduleParamMap);
    }

    // open a trace file
    if (debug_dump) {
        cout << "Debug dumpfilename: " << dump_file_name << endl;
        debugFile.open(dump_file_name.c_str());
    }

    // ------------------------------------------------------------
    // (2) Create the clocked OCP Channel
    // ------------------------------------------------------------
    sc_clock clk("clk", OCP_CLOCK_PERIOD,OCP_CLOCK_TIME_UNIT) ;

    pOCP = new OCP_TL1_Channel_Clocked<
OCP_TL1_DataCl<OCPCHANNELBit32, OCPCHANNELBit32> > ("ocp0");

    // ------------------------------------------------------------
```

```
    // (3) Create the Master and Slave
    // ------------------------------------------------------------

    pMaster =
      new Master< OCP_TL1_DataCl<OCPCHANNELBit32, OCPCHANNELBit32> >
            ("master", 0, &debugFile);
    pSlave =
      new Slave< OCP_TL1_DataCl<OCPCHANNELBit32, OCPCHANNELBit32> >
            ("slave", 0, 0x3FF, &debugFile);

    pMaster->setModuleConfiguration(moduleParamMap);
    pSlave->setModuleConfiguration(moduleParamMap);

    // ------------------------------------------------------------
    // (4) connect channel, master, slave, and clock
    // ------------------------------------------------------------
    pMaster->ipP(*pOCP);
    pSlave->tpP(*pOCP);
    pMaster->clk(clk);
    pSlave->clk(clk);
    pOCP->p_clk(clk);

    // ------------------------
    // (5) start the simulation
    // ------------------------
    pMaster->provideChannelConfiguration(ocpParamMap);
    sc_start(simulation_end_time,SC_NS);

    // --------------------
    // (6) post processing
    // --------------------

    cout << "main program finished at "
         << sc_time_stamp().to_double() << endl;

    sc_simcontext* sc_curr_simcontext = sc_get_curr_simcontext();
    cout << "delta_count: " << dec << sc_curr_simcontext-
>delta_count()
            << endl;
    cout << "next_proc_id: " << dec << sc_curr_simcontext-
>next_proc_id()
         << endl;

    return (0);
}
```

7 Examples Using OCP TL2 Channel and API

The examples described in this section demonstrate the use of the OCP TL2 channel. The first example illustrates a single-threaded OCP communication between an OCP master and an OCP slave. Both are using the TL2 API to model the protocol.

The second example shows a more complex example in which a multi-threaded master communicates with a multi-threaded slave via the original OCP TL2 channel.

All the concerned files for these examples are located in '`tl_sc/examples/ocp_tl2_1`'. A `README` file details how to compile and run the code.

7.1 Example # 1

In this example, a simple TL2 master communicates with a simple TL2 slave. The OCP parameters describing the channel are stored in the `'ocpParams'` file. The master uses an OCP TL2 master port to connect the channel, and the slave uses an OCP TL2 slave port. These ports allow modules to perform access to all the TL2 API functions and events available.

The master and the slave use an `'OCPRequestGrp'` structure to pass/get all the request signals to the channel, and an `'OCPResponseGrp'` structure to store/send the response signals.

Both master and slave are non-pipelined modules, which use one single thread to handle requests and responses.

The communication between the master and the slave is composed of the following sequences:

7.1.1 Master Sequence

Master sends a 10-length WRITE burst to the slave using `sendOCPRequestBlocking`(). Only one chunk is used (i.e. transaction is atomic).

Master sends a 10-length READ burst to the slave using `sendOCPRequestBlocking`(). Only one chunk is used (i.e. transaction is atomic).

Master waits and get the corresponding response using two successive `getOCPResponseBlocking`() calls catching 5-length chunks.

Master performs a complete 20-length WRITE transaction using the serialized method 'OCPWriteTransfer()'. This call includes the following phases:

- request send
- request acknowledge

Master performs a complete 20-length READ transaction using the serialized method 'OCPReadTransfer()'. This call includes the following phases:

- request send
- request acknowledge
- response reception
- response acknowledge

7.1.2 Slave sequence

Slave receives a 10-length WRITE burst from the master, and stores the received data in an internal array.

Slave receives a 10-length READ burst from the master, and sends the response using two consecutive response chunks (5-length each) with a different 'SRespInfo' signal value.

Slave receives a 20-length WRITE burst from the master, and stores the received data in an internal array.

Slave receives a 20-length READ burst from the master, and sends the response using one response call.

7.2 Example #2

In this example, a multi-threaded TL2 master communicates with a multi-threaded TL2 slave. The OCP parameters describing the channel are stored in the `'ocpParams_complex'` file.

7.2.1 Slave Description

The TL2 slave emulates a '3 threads' OCP slave. It uses two SystemC threads, one for requests and one for responses. The request SC_THREAD catches every request, computes the response and stores it in one of the three response queues, depending on the ThreadID of the request. Then, the response SC_THREAD issues responses to the master. The slave acts as a memory: a write request updates an internal memory array, and a read request reads a cell of this array.

The slave accepts some parameters, described in the 'slaveParams' files:

- latencyX
- limitreq_enable
- limitreq_max

These parameters are described in section 6.1.3 of the OCP API documentation. Note that for TL2, delays are not expressed in terms of clock cycles but as absolute timings (unit is SC_NS in the slave).

7.2.2 Master Description

The TL2 master emulates a '3 threads' master. It sends requests labelled with a MThread ID varying from 0 to 2. Depending on the current thread, each request targets a different location in the target memory space (no overlap between thread operations). The master uses two SystemC threads, one for the requests and one for the responses.

The master accepts some parameters, described in the 'masterParams' file:

- mrespaccept_delay
- mrespaccept_fixeddelay
- command_cycles

The first two parameters are described in section 6.1.3. Note that for TL2, delays are not expressed in terms of clock cycles but as absolute timings (unit is SC_NS in the

master). 'Command_cycles' specifies the number of times the predefined TL2 requests sequence is sent.

8 Debugging Your Model Using SOCCREATOR® Tools

The OCP TL1, TL2 and TL3 channels all implement monitor interfaces. The user is able to create monitors which can be bound to the channels and used to obtain debug and analysis data from SystemC simulations. Some monitors have been implemented by OCP-IP.

For the OCP TL1 and TL2 channels, there is a trace monitor available. The TL1 trace monitor prints out the state of the OCP interface at the end of every OCP clock cycle.

The resulting OCP Monitor file can be processed with “ocpdis,” a tool that is available separately from the channel, which reformats the data for easy reading. The tool “ocpcheck,” also available separately, processes the OCP Monitor data and checks that the OCP channel followed the OCP protocol.

9 Debugging Your Model Using OCP Performance Monitor

The OCP TL1, TL2 and TL3 channels all implement monitor interfaces. The user is able to create monitors which can be bound to the channels and used to obtain debug and analysis data from SystemC simulations. Some monitors have been implemented by OCP-IP.

For all three channels there is a performance monitor available. These performance monitors enable intuitive performance analysis by means of fast transaction level recording. The analysis instrumentation is based on the SystemC Verification (SCV) standard. The monitor is available to the OCP-IP members in a separate release package together with the old OCPMon monitor class. For use, see the documentation included in the release package, which is available at www.ocpip.org.

10 Sideband Signals (OCP TL1)

The access methods for sending and receiving sideband signals are shared by both the base generic class API and the OCP TL1 API. The commands described in this section may be used with either API.

10.1 MError Signal

This section describes the methods for the MError signal.

void MputMError(bool nextValue)

Caller: Master

Purpose: Changes the next value of the MError signal. If the OCP channel is asynchronous, the change is immediate. If the channel is synchronous, the change occurs at the next update.

bool SgetMError( ) const

Caller: Slave

Purpose: Returns the current value of the MError signal in the channel.

const sc_event& SidebandMErrorEvent() const

Caller: Slave

Purpose: Returns the event associated with the MError signal. This event is triggered whenever the MError signal changes to a new value. Note that a call to `setMError()` or `resetMError()` will not always result in the event `SidebandMErrorEvent` occurring. For example, if the current value of MError is true and the function `setMError()` is called, the event `SidebandMErrorEvent` will not be triggered because the current value (true) and the next value (true) are the same. This method is called by the slave.

10.2 MFlag Signal

This section describes the methods for the MFlag signal.

void MputMFlag(int nextValue)

Caller: Master

Purpose: Changes the next value of the MFlag signal. If the OCP channel is asynchronous, the change is immediate. If the channel is synchronous, the change occurs at the next update.

void MputMFlag(int nextValue, unsigned int mask)

Caller: Master

Purpose: Changes the next value of the MFlag signal. Only nextValue & mask bits are written. If the OCP channel is asynchronous, the change is immediate. If the channel is synchronous, the change occurs at the next update.

int SgetMFlag( ) const

Caller: Slave

Purpose: Returns the current value of the MFlag signal in the channel.

const sc_event& SidebandMFlagEvent() const

Caller: Slave

Purpose: Returns the event associated with the MFlag signal. This event is triggered whenever the MFlag signal changes to a new value.

10.3 SError Signal

This section describes the methods for the SError signal.

void SputSError(bool nextValue)

Caller: Slave

Purpose: Changes the next value of the SError signal. If the OCP channel is asynchronous, change is immediate. If the channel is synchronous, the change occurs at the next update.

bool MgetSError( ) const

Caller: Master

Purpose: Returns the current value of the SError signal in the channel.

const sc_event& SidebandSErrorEvent() const

Caller: Master

Purpose: Returns the event associated with the SError signal. This event is triggered whenever the SError signal changes to a new value. Note that a call to `setSError()` or `resetSError()` will not always result in the event `SidebandSErrorEvent` occurring. For example, if the current value of SError is true and the function `setSError()` is called, the event `SidebandSErrorEvent` will not be triggered because the current value (true) and the next value (true) are the same.

10.4 SFlag Signal

This section describes the methods for the SFlag signal.

void SputSFlag(int nextValue)

Caller: Slave

Purpose: Changes the next value of the SFlag signal. If the OCP channel is asynchronous, the change is immediate. If the channel is synchronous, the change occurs at the next update.

void SputSFlag(int nextValue, unsigned int mask)

Caller: Slave

Purpose: Changes the next value of the SFlag signal. Only nextValue&mask bits are written. If the OCP channel is asynchronous, the change is immediate. If the channel is synchronous, the change occurs at the next update.

int MgetSFlag( ) const

Caller: Master

Purpose: Returns the current value of the SFlag signal in the channel.

const sc_event& SidebandSFlagEvent() const

Caller: Master

Purpose: Returns the event associated with the SFlag signal. This event is triggered whenever the SFlag signal changes to a new value.

10.5 SInterrupt Signal

This section describes the methods for the SInterrupt signal.

void SputSInterrupt(bool nextValue)

Caller: Slave

Purpose: Changes the next value of the SInterrupt signal. If the OCP channel is asynchronous, the change is immediate. If the channel is synchronous, the change occurs at the next update.

bool MgetSInterrupt() const

Caller: Master

Purpose: Returns the current value of the SInterrupt signal in the channel.

const sc_event& SidebandSInterruptEvent() const

Caller: Master

Purpose: Returns the event associated with the SInterrupt signal. This event is triggered whenever the SInterrupt signal changes to a new value. Note that a call to `setSInterrupt()` or `resetSInterrupt()` will not always result in the event `SidebandSInterruptEvent` occurring. For example, if the current value of SInterrupt is true and the function `setSInterrupt()` is called, the event `SidebandSInterruptEvent` will

not be triggered since the current value (true) and the next value (true) are the same.

10.6 Control Signal

This section describes the methods for the Control signal.

bool SysputControl(int nextValue)

Caller: System side

Purpose: If ControlBusy is false, this function changes the next value of the Control sideband signal. If the ControlBusy signal is part of the OCP channel configuration, and the current value of ControlBusy is true, the next value of the Control sideband signal will not be changed and the `setControl()` method will return false. Otherwise, the method will return true and will set the next value of the Control signal. If the OCP channel is asynchronous, the change to the Control signal is immediate. If the channel is synchronous, the change occurs at the next update.

int CgetControl() const

Caller: Core side

Purpose: Returns the current value of the Control signal in the channel.

const sc_event& SidebandControlEvent() const

Caller: Core side

Purpose: Returns the event associated with the Control signal. This event is triggered whenever the Control signal changes to a new value.

10.7 ControlWr Signal

This section describes the methods for the ControlWr signal.

void SysputControlWr(bool nextValue)

Caller: System side

Purpose: Changes the next value of the ControlWr signal. If the OCP channel is asynchronous, the change is immediate. If the channel is synchronous, the change occurs at the next update.

bool CgetControlWr( ) const

Caller: Core side

Purpose: Returns the current value of the ControlWr signal in the channel.

const sc_event& SidebandControlWrEvent() const

Caller: Core side

Purpose: Returns the event associated with the ControlWr signal. This event is triggered whenever the ControlWr signal changes to a new value.

10.8 ControlBusy Signal

This section describes the methods for the ControlBusy signal.

void CputControlBusy(bool nextValue)

Caller: Core side

Purpose: Changes the next value of the ControlBusy signal. If the OCP channel is asynchronous, the change is immediate. If the channel is synchronous, the change occurs at the next update.

bool SysgetControlBusy( ) const

Caller: Core side

Purpose: Returns the current value of the ControlBusy signal in the channel.

const sc_event& SidebandControlBusyEvent() const

Caller: System side

Purpose: Returns the event associated with the ControlBusy signal. This event is triggered whenever the ControlBusy signal changes to a new value. Note that a call to `setControlBusy()` or `resetControlBusy()` will not always result in the event `SidebandControlBusyEvent` occurring. For example, if the current value of ControlBusy is true and the function `setControlBusy()` is called, the event `SidebandControlBusyEvent` will not be triggered because the current value (true) and the next value (true) are the same.

10.9 Status Signal

This section describes the methods for the Status Signal.

void CputStatus(int nextValue)

Caller: Core side

Purpose: This function changes the next value of the Status sideband signal. If the OCP channel is asynchronous, the change to the Status signal is immediate. If the channel is synchronous, the change occurs at the next update.

int SysgetStatus( ) const

Caller: System side

Purpose: Returns the current value of the Status signal in the channel.

bool readStatus(int& currentValue) const

Caller: System side

Purpose: If the channel signal StatusBusy is false, then this function sets the passed parameter `currentValue` to the current value of the Status signal in the channel. Then the event `SidebandStatusRdEvent` is triggered and the function returns true. If the channel signal StatusBusy is true, the read is not performed, the event `SidebandStatusRdEvent` is not triggered, and the function returns false.

const sc_event& SidebandStatusEvent() const

Caller: System side

Purpose: Returns the event associated with the Status signal. This event is triggered whenever the Control signal changes to a new value.

10.10 StatusRd Signal

This section describes the methods for the StatusRd Signal.

void SysputStatusRd(bool nextValue)

Caller: System side

Purpose: Changes the next value of the StatusRd signal. If the OCP channel is asynchronous, the change is immediate. If the channel is synchronous, the change occurs at the next update.

bool CgetStatusRd( ) const

Caller: Core side

Purpose: Returns the current value of the StatusRd signal in the channel.

const sc_event& SidebandStatusRdEvent() const

Caller: Core side

Purpose: Returns the event associated with the StatusRd signal. This event is triggered whenever the ControlWr signal changes to a new value.

10.11 StatusBusy Signal

This section describes the methods for the StatusBusy signal.

void CputStatusBusy(bool nextValue)

Caller: Core side

Purpose: Changes the next value of the StatusBusy signal. If the OCP channel is asynchronous, the change is immediate. If the channel is synchronous, the change occurs at the next update.

bool SysgetStatusBusy( ) const

Caller: System side

Purpose: Returns the current value of the StatusBusy signal in the channel.

const sc_event& SidebandStatusBusyEvent() const

Caller: System side

Purpose: Returns the event associated with the StatusBusy signal. This event is triggered whenever the StatusBusy signal changes to a new value. Note that a call to `setStatusBusy()` or `resetStatusBusy()` will not always result in the event `SidebandStausBusyEvent` occurring. For example, if the current value of StatusBusy is true and the function `setStatusBusy()` is called, the event `SidebandStatusBusyEvent` will not be triggered because the current value (true) and the next value (true) are the same.

11 Sideband signals (OCP TL2)

The OCP TL2 channel has full sideband signal support.

Sideband API Function	Description
Called by Master	
bool MgetSError(void)	Returns the value of SError.
unsigned long long int MgetSFlag(void)	Returns the value of SFlag.
bool MgetSInterrupt(void)	Returns the value of SInterrupt.
void MputMError(bool nextValue)	Set the value of MError. Triggers SidebandMasterEvent.
void SputMFlag(unsigned long long int nextValue)	Set the value of MFlag. Triggers SidebandMasterEvent.
Called by Slave	
bool SgetMError(void)	Returns the value of MError.
unsigned long long int SgetMFlag(void)	Returns the value of MFlag.
void SputSError(bool nextValue)	Set the value of SError. Triggers SidebandSlaveEvent.
void SputSFlag(unsigned long long int nextValue)	Set the value of SFlag. Triggers SidebandSlaveEvent.
void SputSInterrupt(bool nextValue)	Set the value of SInterrupt. Triggers SidebandSlaveEvent.
Called by "System" side	
void SysputControl(unsigned int nextValue)	Set the value of Control. Triggers the SidebandSystemEvent.
bool SysgetControlBusy(void)	Gets the value of ControlBusy.
void SysputControlWr(bool nextValue)	Set the value of ControlWr. Triggers the SidebandSystemEvent.
unsigned int SysgetStatus(void)	Gets the value of Status.
bool SysgetStatusBusy(void)	Gets the value of StatusBusy.
void SysputStatusRd(bool nextValue)	Set the value of StatusRd. Triggers the SidebandSystemEvent.
Called by "Core" side	
unsigned int CgetControl(void)	Gets the value of Control.
void CputControlBusy(bool nextValue)	Set the value of ControlBusy. Triggers the SidebandCoreEvent.

unsigned int CgetControlWr(void)	Gets the value of ControlWr.
void CputStatus(unsigned int nextValue)	Set the value of Status. Triggers the SidebandCoreEvent.
void CputStatusBusy(unsigned int nextValue)	Set the value of StatusBusy. Triggers the SidebandCoreEvent.
bool CgetStatusRd(void)	Gets the value of StatusRd.

12 OCP TL1 Timing

Level-1 of the OCP TLM model is designed to allow cycle-accurate modelling of bus interfaces. Any OCP traffic pattern that is possible in hardware should also be possible to model at TL1, without modifications to the design hierarchy or topology, and in a fully modular manner. This means that the TL1 infrastructure needs to support, among other things:

- Modules with internal combinatorial paths from one OCP signal to another within a single OCP interface
- Modules with internal combinatorial paths from an OCP signal on one interface to OCP signals on another interface
- Cascading of modules with OCP interfaces to an arbitrary degree
- Modules that change the values of OCP signals at some time in the middle of a clock cycle rather than at the clock edges, for example scaled-synchronous clock bridges

As OCP is a synchronous clocked protocol, to model it at a cycle-accurate level means that at very least the OCP master must understand the location of the clock cycles in time. In fact it is usual that the OCP slave also needs an understanding of the OCP clock cycles, and when both master and slave have this information, it *must* be the same for both of them, otherwise the channel will not work correctly. Furthermore, the channel may be clocked and there may be one or more monitors attached to the channel, and these also need to be correctly synchronized with the OCP master.

The section below attempts to explain what is meant by *synchronization* in this context. This is followed by a section describing how the OCP-TL1 timing information distribution system can be used to support non-default cases.

12.1 OCP TL1 Synchronisation

In the OCP protocol time is divided into *clock cycles*. Clock cycles are generally of a constant duration, the *clock period*, but this is not obligatory. In hardware, each clock cycle begins with a rising edge of a single-wire clock signal. The clock signal returns to zero some time during the cycle and the cycle ends when the following cycle begins, with the next rising edge.

In SystemC it is usual to define clock cycles in the same way, using an sc_channel of type sc_signal<bool> or the convenient library module sc_clock. SystemC allows many other ways of defining clock cycles and most ways are tolerated by the OCP_TL1_Channel. However users are warned that exotic or unusual definitions of clock cycles will greatly reduce the chances of compatibility between modules.

The OCP_TL1_Channel understands only one way to define OCP clock cycles, and that is by using an sc_clock or sc_signal<bool>. If clock cycles are defined in any other way then the *untimed* version of the channel must be used instead of the timed version. The untimed version of the channel has reduced functionality.

There is a trace monitor available for the untimed channel. This monitor needs to understand the definition of the OCP clock cycles. It assumes that they have constant duration and start at the first delta cycle at time 0. This is one delta cycle different from the normal implementation and use of sc_clock, where the clock cycles start at the second delta cycle at time 0, because one delta cycle is consumed in the sc_clock's internal process. If the untimed trace monitor is used, then the OCP master and in

most cases the OCP slave need to understand the clock cycles in the same way that it does. That means they should not use sc_clock or sc_signal<bool> channels as an OCP clock. Rather they should be implemented with SC_THREAD() proceses containing wait(OCP_CLOCK_PERIOD)-type statements or SC_METHOD() processes contaning next_trigger(OCP_CLOCK_PERIOD)-type statements. On the other hand, a clocked OCP master/slave pair can use the untimed channel without trace monitor support.

For every OCP_TL1_Channel in a simulation, there are several other modules associated with it:

- Exactly one module with an OCP master port, the *master*
- Exactly one module with an OCP slave port, *the slave* (which is allowed to be the same module as the master)
- Optionally one or more monitors

The master and slave may contain processes that access the channel. If so, these processes must be synchronized with each other, so that they understand the same clock cycle boundaries, *down to delta-cycle-accuracy*.

If the channel or any monitor is clocked, it must be clocked with the same clock used in the master and slave for OCP clock cycle synchronisation.

There are several cases where the modules do not need to understand the clock cycles. For example:

- The channel has an untimed option, as discussed above
- An OCP slave can be fully event-driven. It can be implemented as a process which waits for the RequestStartEvent, then calls startOCPResponse() within the same clock cycle. This corresponds to a zero-latency (combinatorial) hardware module. Note that such a module is sensitive to the timing of the master and does not have default timing itself and as such it needs to use the timing information distribution system described below.
  In this case the master alone needs to understand the OCP clock cycle definition.
- A simple combinatorial bridge, for example a bridge to cut INCR bursts' lengths to some maximum value without introducing any latency, has both an OCP master port and an OCP slave port. It can be implemented as a pair of processes sensitive to RequestStartEvent and ResponseStartEvent, which modify slightly the OCPRequestGrp and OCPResponseGrp and forward them from one port to the other in the same cycle. Note that such a module is sensitive to the timing of the external OCP master and slave, and does not have default timing itself and as such it needs to use the timing information distribution system described below.
  In this case the external OCP master and possibly the external OCP slave need to understand the OCP clock cycle definition.

All modules that do need to understand the clock cycle definition need to understand it identically. Note that:

- All accesses from master or slave to the channel that change the channel's state do so with a delay of one delta cycle.
- Hence, at the boundary between two clock cycles, there is a single instant where a master (or slave) can read the OCP signals from the past cycle and write the OCP signals for the future cycle.
  A process arrives at this instant, in the case of the clocked channel, by executing a statement of the form wait(clock_port->posedge_event()) for an

SC_THREAD or next_trigger(clock_port->posedge_event()) for an SC_METHOD. Obviously static sensitivity works as well.

- This means that in many cases the master and slave modules can be implemented in a fully-synchronous style, having just a single process sensitive only to the clock's rising edge.
- Accesses to the channel at other times than at the instant between two clock cycles are fully within one clock cycle. This is true even if the accesses are at the same time as the cycle boundary but a different delta. At such times the master (or slave) can write the OCP signals only for the current cycle. Furthermore, it can reliably read the signals only from the current cycle. It needs to find out from the channel the timing of the slave (or master) on the other side of the channel and ensure that it does not attempt to read until these signals are stable (meaning they will not be changed again in the clock cycle).

12.2 Timing Information Distribution (OCP TL1)

There are certain cases where TL1 models are unable to use only the clock period boundaries as their timing reference. The underlying reason for this is that the TL1 methodology recommended for OCP does not permit the retraction of either an OCP request or command accept, or the equivalents for data-handshake and response phases.

These cases include:

- thread-busy-exact OCP interfaces, where the OCP protocol obliges the master (for sthreadbusy_exact) to choose its request only after having seen the SThreadBusy signals from the slave.
- a combinatorial request or response merger (arbiter), which needs to wait for a time long enough for all inputs to be stable before it chooses one of them. In particular where combinatorial OCP modules are cascaded some inputs may arrive later than others.
- the OCP TL1 channel after preemptive release has been set, which needs to wait sufficient time after a new request (or response/data-handshake) has been started, to allow the slave to de-assert the preemptive release.

To allow simple management of such cases, a mechanism is provided in the OCP TL1 channel which allows distribution of timing information at end-of-elaboration. Only OCP modules that are either "timing-sensitive" or "non-default-timing" need to use this mechanism. All other modules may ignore it completely.

12.2.1 Timing-sensitive Modules

A timing-sensitive module is a module which needs to know when inputs can safely be assumed to be stable, in order to work correctly. A non-timing-sensitive module might sample all inputs at the end of the OCP clock cycle, as a counter-example.

All OCP masters that are sthreadbusy-exact or sdatathreadbusy-exact are by definition timing-sensitive. All OCP slaves that are mthreadbusy-exact are by definition timing-sensitive.

Timing-sensitive modules register themselves with the OCP TL1 channel during end-of-elaboration. They do this by calling one of the channel methods:

- registerTimingSensitiveOCPTL1Master(this);

- registerTimingSensitiveOCPTL1Slave(this);

depending on whether they are a master or a slave. Here it is suggested that a pointer to the module itself be passed as parameter. This would mean the module is derived from OCP_TL1_Slave_TimingIF (for an OCP master) or OCP_TL1_Master_TimingIF (for an OCP slave). However this may be impractical in some cases, for example where a module has multiple OCP master ports. The alternative is that the OCP module contains one or more member variables of classes derived from OCP_TL1_Master_TimingIF or OCP_TL1_Slave_TimingIF as appropriate. Any class derived from OCP_TL1_Master_TimingIF is obliged to implement the method setOCPTL1MasterTiming() (and similar for the slave).

Once the module is registered with the channel as timing-sensitive, the channel will inform it of the timing parameters of the module on the other side of the channel. This may happen several times depending on the order of the end-of-elaboration calls in the SystemC simulation. The implementation of the method setOCPTL1MasterTiming() or setOCPTL1SlaveTiming() must allow it to be called multiple times during end-of-elaboration. The first time it is called might be before the registerTimingSensitive..() method returns.

If the other side of the OCP TL1 channel is a default-timing module, the channel will never call the callback.

12.2.2 Non-default-timing Modules

A non-default-timing module is a module whose outputs are not presented to the OCP TL1 channel immediately at the start of the OCP clock cycle. If a clock signal is used to synchronise the OCP master and OCP slave, this means that default-timing modules call all channel methods in the delta cycle after the clock rising edge.

Non-default timing modules must call the channel method setOCPTL1MasterTiming() or setOCPTL1SlaveTiming() (for masters and slaves respectively) during end-of-elaboration, providing their timing parameters.

A non-default timing module may not know its timing parameters when its own end-of-elaboration method is called. This is the case for example for a combinatorial module passing OCP requests from a slave port to a master port (an address translation bridge for example). A module like this is both timing-sensitive and non-default-timing. It must register itself as timing-sensitive on its OCP slave port and send its timing information to its OCP master port. It may occur that the module is provided several times with timing information from the OCP slave port, and every time that its setOCPTL1MasterTiming() method is called from the slave port channel, it should recalculate the timing parameters of its master port and call the setOCPTL1MasterTiming() method of the master port if they changed.

To avoid infinite loops at end-of-elaboration it is important that a non-default-timing module only call setOCPTL1XyyTiming() when necessary. It should not call this method if it has previously been called with the same parameters.

12.2.3 Start Times

Start times are sc_time variables. They indicate when a signal/group is given to the OCP_TL1_Channel by the OCP master or slave. The other side of the OCP interface can safely retrieve the signal/group from the OCP TL1 channel after waiting for the start-time and one delta cycle. It is then sure that the signal will not change again this clock cycle.

Start times give duration of simulated time after the start of an OCP clock cycle.

It is assumed that the OCP master and OCP slave are exactly synchronised.

- start_time = SC_ZERO_TIME

  This means that the signal/group is started immediately after the synchronisation event indicating the start of an OCP cycle. The other side of the OCP interface can sample safely after one delta.

- start_time > SC_ZERO_TIME

  This means that the signal/group is started after wait(start_time) after the synchronisation event indicating the start of an OCP cycle. Or before. It is not allowed that the signal/group be started some delta cycles after wait(start_time) (one delta = wait(SC_ZERO_TIME)). In this case the other side of the OCP interface must at least wait(start_time) AND wait(SC_ZERO_TIME) before sampling.

The most frequent example is a thread-busy-exact OCP. In the simplest case the slave produces SThreadBusy directly after the start of cycle. It has therefore default timing. The master must wait at least one delta before sampling SThreadBusy and starting an OCP request. Therefore the OCP request start time is +1 delta. This is impossible to represent as an sc_time, so the master must indicate a start-time strictly greater than 0.

It is recommended to use the function sc_get_time_resolution(), which returns an sc_time object, to create sample times as small as possible and as independent as possible from simulator configuration and clock frequency choices.

12.2.4 OCP TL1 Timing Example

In the distribution there is an example of how the TL1 timing distribution feature of the OCP TL1 channel can be used. It is a simulation of a multi-threaded non-blocking shared bus with zero-cycle minimum round-trip latency. In this design a request/response transfer can pass through up to 10 cascaded OCP_TL1_Channel() instances in the same clock cycle. For more details look in the source code and the readme.txt file, in the directory examples/supplementary/ocp_tl1_timing.