A SystemC™ OCP Transaction Level Communication Channel

V2.0.3 – October 20, 2004

Document version 1.5

DISCLAIMER

This OCP-IP document is provided "as is" with no warranties whatsoever, including any warranty of merchantability, noninfringement, fitness for any particular purpose, or any warranty otherwise arising out of any proposal, specification or sample. OCP-IP disclaims all liability for infringement of proprietary rights, relating to use of information in this document. No license, express or implied, by estoppel or otherwise, to any intellectual property rights is granted herein.

OCP International Partnership (OCP-IP) disclaims all warranties and liability for the use of this document and the information contained herein and assumes no responsibility for any errors that may appear in this document, nor does OCP-IP make a commitment to update the information contained herein.

Contact the OCP-IP office to obtain the latest revision of this document.

Questions regarding this document or membership in OCP-IP may be forwarded to:

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	10/20/04	New performance OCP TL2 channel model with timing points. New chapter on differences between TL1 and TL2 as well as the differences between the two channel models. Clean up of formatting.

Table of Contents

1 Introduction

This document describes the SystemC model of an Open Core Protocol (OCP) channel. This model is meant for the 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).

This document covers OCP specific versions of the SystemC channel: the OCP specific channels for Transaction Level One (TL1) and Transaction Level Two (TL2). A base generic model, which serves as the foundation of the OCP TL1 and TL2 channels, is described in *A SystemC™ Generic Transaction Level Communication Channel* specification (Refer to www.ocpip.org for more information). The OCP specific channel models were designed with the goals of OCP correctness and ease of use. These OCP specific models are useful for cores that require an accurate model of the OCP channel that is close to cycle accurate. As a group, the OCP specific commands are more powerful and mask some of the complexity of the channel. This version of the channel would be useful for all OCP cores except those legacy cores that require a Generic channel interface.

This document categorizes the communication abstraction levels 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 are as follows:

Transaction Level

Layer-3: Message Layer

Model untimed functionality

Point-point communication

Layer-2: Transaction Layer

Model/analyze SoC architecture

Start SW development

Estimate timing

Layer-1: Transfer Layer

Cycle true but faster than RTL

Detailed analysis, develop low-level SW

Pin Level

Layer-0: 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.

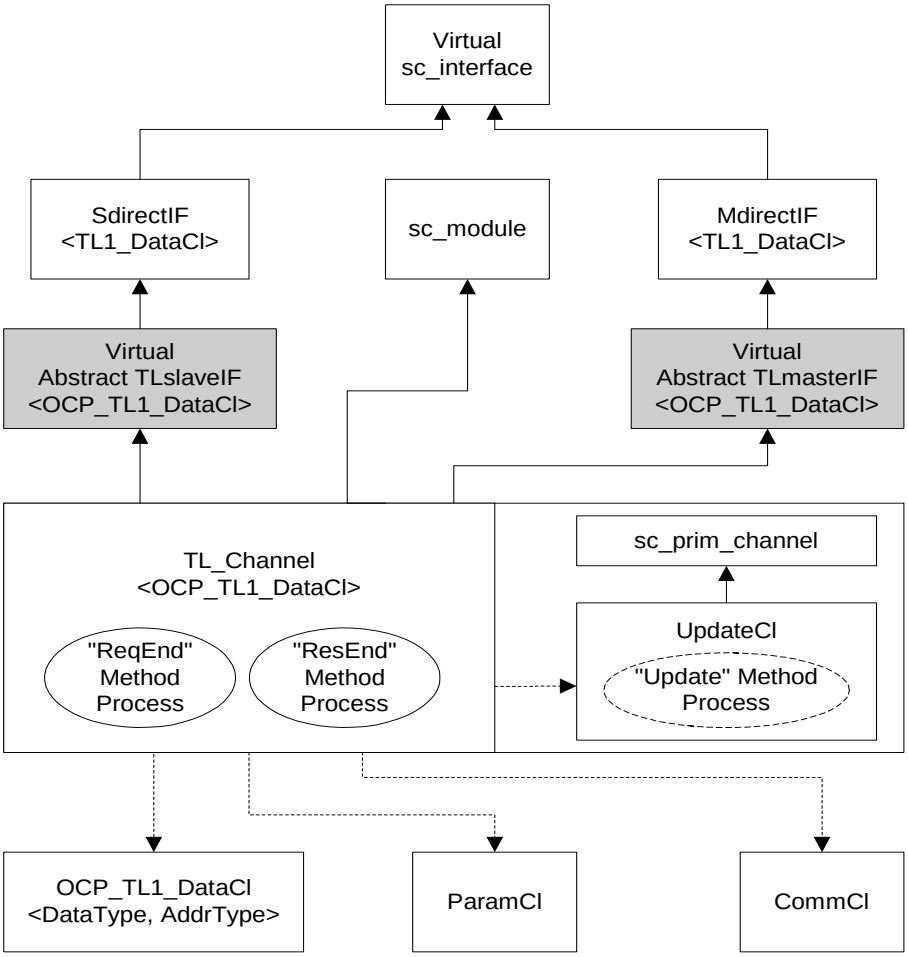
SystemC is a C++ modeling environment designed for both cycle based and higher level modeling of systems. This document assumes a basic understanding of the SystemC language. For more information on SystemC, go to www.systemc.org.

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.

2 Directory structure and Class Hierachy

The generic 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 internal class hierarchy for the generic channel. The generic model 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. Any type of data, even non-OCP data, can move through the generic base channel.

Figure 1 Generic Channel Class Hierarchy

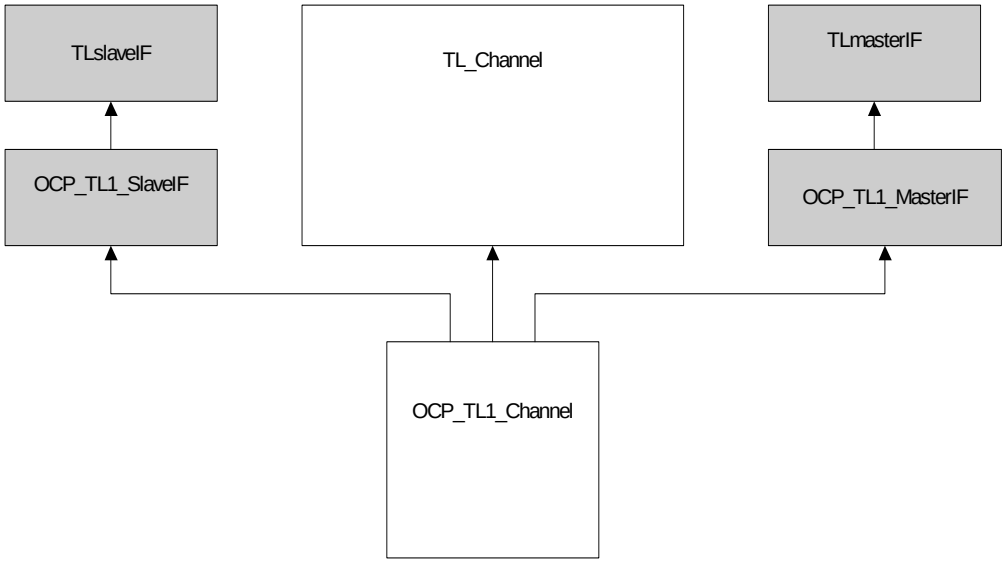


The OCP Specific channels are derived from the generic base channel model. The class hierarchy for the OCP_TL1_Channel is shown in Figure 2. The OCP_TL1_Channel adds OCP specific commands that process requests, responses, and data handshakes with single commands. In addition, the OCP TL1 channel is built 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 specific commands. OCP specific ports for the master and slave 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 TL1 Specific Channel Class Hierarchy (Inherited from TL Channel Class Hierarchy)

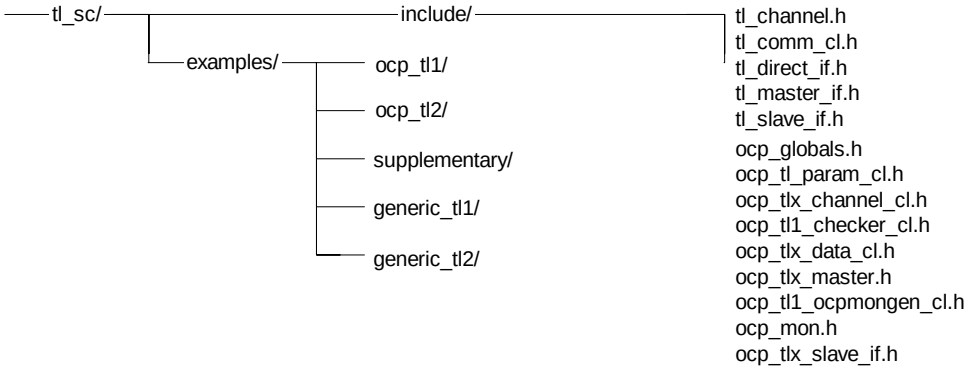


Like TL1, the original OCP Transaction Layer Two (TL2) channel is derived from the generic base channel model and provides OCP specific commands that process requests and responses with single commands. The interfaces `OCP_TL2_SlaveIF` and `OCP_TL2_MasterIF` provide port access to all of the OCP specific commands. OCP specific ports for the master and slave provide OCP specific event finders so that methods in the user's SystemC core model may be statically sensitive events in the channel.

The performance OCP TL2 channel is not layered upon the generic channel and thus not all of the generic commands are available with this channel. However, all of the OCP specific commands are fully supported. Whether the original or new TL2 channel is installed, the directory structure below remains the same. For more information on the new performance TL2 channel, please see the OCP TL2 chapter.

Figure 3 illustrates the directory structure for the OCP SystemC channel models.

Figure 3 OCP Channel Directory Tree



3 Overview of Transaction ChannelS

This document describes the OCP Specific channel with the following goal: the OCP Specific channel is designed to fully enforce the Open Core Protocol and to be close to cycle-accurate. As a result, the OCP Specific channel can maintain a notion of time and has additional restrictions on how and when its commands may be used.

3.1 OCP Specific Transaction Channel and Interfaces

While the base generic channel model is meant to support basic channel communication, the OCP specific channel models are built specifically to implement the OCP. The OCP specific 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.

3.2 Working with Different Channel Versions

This document covers all versions of the SystemC channel: the base generic channel as well as the OCP specific channels for Transaction Level One (TL1), Transaction Level Two (TL2), and TransactionLevel Three (TL3). [Alan K, This seems like a general introduction so I moved this paragraph to Section 1. – Alan M.]

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.

While it is possible to mix commands from one model with another, this is strongly discouraged because you must take great care to ensure that the model still behaves as expected. To ensure OCP correct behavior, you should not mix commands from the OCP-specific APIs with the generic channel commands. In particular, the generic channel's pointer access to the internals of the channel should be avoided[1]. If the core writer uses the base generic class data pointer to directly manipulate the OCP TL1 data, the channel may no longer be OCP correct.

1 Such generic channel pointer access to the channel's internal state is not supported in the performance version of the OCP TL2 channel.

4 OCP Specific TL1 Channel Model

The OCP TL1 specific channel has OCP specific commands for sending and accepting OCP requests, data, and responses. Because the channel model was designed specifically for OCP TL1 transactions, it is both easier to use and it ensures that the channel is OCP correct.

Since the OCP TL1 specific channel is built upon the base generic channel and the OCP TL1 data class, it is possible to use generic commands with the OCP TL1 channel. However, this is strongly discouraged as doing so may lead to unexpected behavior, which is out of the bounds of the OCP protocol.

4.1 OCP TL1 Channel Constructors

There are several 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 two timing modes can be used with blocking methods: Clocked and self-timed. (The first release of the OCP channel had only the self-timed mode.) The clocked mode simulates faster, but requires a pointer to an object providing the clock events (signal, port, or sc_clock). The clocked blocking methods are similar to using clock wait statements within master and slave threads. The self-timed mode is slower, but allows creating master and slave models, which don't have clock inputs at all. The clock period is given as a constructor parameter, and the channel will implement wait statements simulating the progression of bus cycles.

Notice that the self-timed mode is not compatible with the other timing modes: One cannot connect a non-clocked master, which uses blocking calls with self-timed channel to a clocked slave and expect cycle-accurate behavior. Also, the behavior of blocking methods is slightly different in clocked and self-timed modes: The clocked blocking methods always wait for clock edge before progressing, the self-timed ones progress immediately, and wait in the end.

Default Constructor

The default constructor can configure non-timed and self-timed channels. The default is non-timed (clock_period=0). 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 sync = true,
                bool use_event = true,
                bool use_default_event = true,
                sc_trace_file* vcd_tf = NULL,
                double clock_period = 0,
                sc_time_unit clock_time_unit = SC_NS,
                std::string monFileName = "",
                bool runtimeCheck = false)
```

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

Synch
: specifies whether the channel's internal state and events are updated synchronously (synch = true) or asynchronously (synch = false). Always set synch to true.

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.

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.

vcd_tf
: No Longer used. Always set to NULL.

clock_period
: The period of the OCP Channel cycle. If 0, non-timed mode is set, and blocking methods are not allowed.

clock_time_unit
: The time unit of the OCP channel's period.

MonFileName
: The name of the file to use to output the OCP monitor data. If this parameter is not set then no OCP Monitor data is recorded. If OCP Monitor package is not installed, the monitor file will contain only a warning message of this. The clock_period must be defined for the monitor to work.

RuntimeCheck
: Boolean to turn the run time checker on (true) or off (false). The run time checker provides basic debugging capability by monitoring the number of requests and responses and commands and command-accepts and ensuring that the counts match. Will only work as advertised if OCP Monitor package is installed.

  The simple constructors can configure self-timed or clocked channels. Notice that there is no parameter for turning off the OCP Checker. The checker can be turned off by defining preprocessor NDEBUG before including the channel header file. It is a good idea to keep the OCP Checker on (if installed) to ensure model behavior (pun intended).

The Simple Self-timed Constructor

```
OCP_TL1_Channel(std::string name,
                double clock_period,
                sc_time_unit clock_time_unit = SC_NS,
                std::string monFileName = "")
```

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

clock_period
: The period of the OCP Channel cycle. If 0, non-timed mode is set, and blocking methods are not allowed.

clock_time_unit
: The time unit of the OCP channel's period.

MonFileName
The name of the file to use to output the OCP Monitor data. If this parameter is not set then no OCP Monitor data is recorded. If OCP Monitor package is not installed, the monitor file will contain only a warning message of this. The clock_period must be defined for the monitor to work.

The Simple Clocked Constructors

```
OCP_TL1_Channel(std::string name,
                <clock_object> * clk,
                std::string monFileName = "")
```

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.

clock_time_unit
The time unit of the OCP channel’s period.

MonFileName
The name of the file to use to output the OCP Monitor data. If this parameter is not set then no OCP Monitor data is recorded. If OCP Monitor package is not installed, the monitor file will contain only a warning message of this.

4.2 OCP TL1 Specific Enum Types and Template Classes

The OCP TL1 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.

4.2.1 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

4.2.2 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

4.2.3 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_RESERVED	7	Reserved

4.2.4 OCPRequestGrp Template Class

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

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

4.2.4.1 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.

4.2.4.2 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. 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

Name	Data Type	Description
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
MConnId	unsigned int	Master connection identifier
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
MRefLast	bool	Last response in burst

4.2.4.3 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.

4.2.4.4 Assignment Operator (=)

OCPRequestGroup& operator=(const OCPRequestGroup& rhs)

The assignment operator assigns one OCPRequestGroup object to another.

4.2.4.5 copy

void copy(const OCPRequestGrp& src)

Copies one OCPRequestGrp object to another.

4.2.5 OCPResponseGrp Template Class

The OCPResponseGrp class is used to send and receive responses with the OCP TL1 specific 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

4.2.5.1 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.

4.2.5.2 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
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

4.2.5.3 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.

4.2.5.4 Assignment Operator (=)

OCPResponseGrp& operator=(const OCPResponseGrp& rhs)

The assignment operator assigns one OCPResponseGrp object to another.

4.2.5.5 copy

void copy(const OCPResponseGrp& src)

Copies one OCPResponseGrp object to another.

4.2.6 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

4.2.6.1 Data Type

The class template parameter Td indicates the data type of the MData signal. For instance, it can be int or unsigned long 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.

4.2.6.2 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
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.

4.2.6.3 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.

4.2.6.4 Assignment Operator (=)

OCPDataHSGrp& operator=(const OCPDataHSGrp& rhs)

The assignment operator assigns one OCPDataHSGrp object to another.

4.2.6.5 copy

void copy(const OCPDataHsGrp& src)

Copies one `OCPDataHSGrp` object to another.

4.3 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.

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.

Note

It is recommended that blocking TL1 calls be used only in test benches and somesuch, since their behavior depends on the channel timing configuration. Non-blocking calls are safe for modeling masters and slaves, since they require that masters and slaves be clocked.

4.3.1 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.

4.3.2 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 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 another request in progress or about to start or if the slave is busy.

Events: Default event and request start event. No event if return value is false.

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

Purpose: Self-timed behavior: Waits until the channel is free for a new request and then starts the passed request on the channel.

Clocked behavior: Repeat - Wait for a rising clock edge, try request - until successful.

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

Return: Returns false if there is already a blocking request waiting to be sent, or if the request could not be sent.

Events: Default event and request start event. No event if return value is false.

bool getSCmdAccept() const // Functionality changed

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.

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: None.

sc_event& RequestStartEvent()

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

Return: Request start event.

sc_event& RequestEndEvent()

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

Return: Request end event.

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.

4.3.3 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: None

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.

Return: Returns false if there is already another `getResponseBlocking` command in progress or if a response cannot be read.

Event: None

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: None

void putMRespAccept(bool accept)

Purpose: Sets or unsets the MRespAccept signal in the OCP channel. Can be called any time. One called, the accept state is persistent. See MreleasePE() of the Generic channel.

Event: None

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

void putNextMThreadBusy()

Purpose: Sets the value of the MThreadBusy signal at the beginning of the next clock cycle. The thread busy value passed in here will be placed on the channel at the very beginning of the next clock cycle, before any thread or method processes start. This function ensures that at the next cycle, the slave will have this value of the MThreadBusy signal in order to decide which response (if any) to send. Note that if this command is called more than once in the same cycle, the value passed in the last call will be used.

Return: None.

Event: None

sc_event& ResponseStartEvent()

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

Return: Response start event.

sc_event& ResponseEndEvent()

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

Return: Response end event.

4.3.4 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 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 another data handshake in progress or about to start or if the slave is busy.

Events: Default event and data handshake start event. No event when return value is false.

bool startOCPDataHSBlocking(
const OCPDataHSGrp<Td>& newData)

Purpose: Self-timed behavior: Wait until the channel is free for new data, start the passed data and set the OCP signal MDataValid to true.

Clocked behavior: Repeat - Wait for a rising clock edge, try request - until successful.

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

Return: Returns false if there is already a blocking data handshake waiting to be sent or if the data could not be sent.

Events: Default event and data handshake start event. No event when 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.

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: None.

sc_event& DataHSStartEvent()

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

Return: Data handshake start event.

sc_event& DataHSEndEvent()

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

Return: Data handshake end event.

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.

4.4 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.

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.

Note

It is recommended that blocking TL1 calls be used only in test benches and somesuch, since their behavior depends on the channel timing configuration. Non-blocking calls are safes for modeling masters and slaves, since they require that masters and slaves be clocked.

4.4.1 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.

4.4.2 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: None

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.

Return: Returns false if there is already another `getRequestBlocking` command in progress or if a request cannot be read.

Event: None.

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: None

Void putSCmdAccept(bool accept)

Purpose: Sets or unsets the SCmdAccept signal in the OCP. Can be called at any time during clock cycle. Persistent once called.

Event: None.

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.

void putNextSThreadBusy()

Purpose: Sets the value of the SThreadBusy signal at the beginning of the next clock cycle. The thread busy value passed in here will be placed on the channel at the very beginning of the next SystemC clock cycle, before any thread or method processes start. This function ensures that at the next cycle, the master will be have this value of the SThreadBusy signal in order to decide which request (if any) to send. Note that if this command is called more than once in the same cycle, the value passed in the last call will be used.

Return: None.

Event: None.

4.4.3 Response Phase

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

bool startOCPResponse(
const OCPResponseGrp<Td>& newResponse)

Purpose: Places the passed response onto the channel.

Return: Returns false if there is another response in progress or about to start or if the master is busy.

Event: Default event and response start event.

bool startOCPResponseBlocking(
const OCPResponseGrp<Td>& newResponse)

Purpose: Self-timed behavior: Wait until the channel is free for a new response, start response on the channel.

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

Note

The timing is different from clocked blocking request to allow single-cycle response. This method should never be called before a request is detected with some of the get request methods.

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

Return: Returns false if there is already a blocking response waiting to be sent or if the response could not be sent.

Event: Default event and response start event.

sc_event& RequestStartEvent()

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

Return: Request start event.

sc_event& RequestEndEvent()

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

Return: Request end event.

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.

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: None.

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: Response start event.

sc_event& ResponseEndEvent()

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

Return: Response end event.

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.

4.4.4 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 this function does not continue to wait until the data is accepted. Also 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

Return: Returns false if there is already another getDataBlocking command in progress or if the data cannot be read.

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: None.

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.

void putNextSDataThreadBusy()

Purpose: Sets the value of the SDataThreadBusy signal at the beginning of the next clock cycle. The thread busy value passed in here will be placed on the channel at the very beginning of the next SystemC clock cycle, before any thread or method processes start. This function ensures that at the next cycle, the master will be have this value of the SDataThreadBusy signal in order to decide which data

handshake (if any) to send. Note that if this command is called more than once in the same cycle, the value passed in the last call will be used.

Return: None.

Event: None.

5 Overview of the OCP TL2 Channel Models

The OCP Transaction Level Two channel model is designed for architectural evaluation and modelling. 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 through-put 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, when to use the TL2 channel, why there are a pair of SystemC models, and when to best use each model. The two chapters that follow cover the specifics of each of the OCP TL2 SystemC channel models.

5.1 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 are each covered below.

5.1.1 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 performance 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.

5.1.2 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 behaviour may be modelled 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 modelled directly in the OCP TL2 channel since they are always sent together. The performance 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.

5.1.3 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.

5.1.4 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 send the whole burst request at once. This eliminates much of the overhead and greatly improves the through-put of the channel.

5.1.5 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.

5.2 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.

5.2.1 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 not currently a 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.

Both versions of the OCP TL2 channel provide delayed accept functions to make this easier. In addition, the performance OCP TL2 channel also provides timing variables and helper functions that can automatically calculate the OCP timing of a request or response. More information on the timing variables may be found in the performance OCP TL2 channel chapter below.

5.2.2 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 master 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.

5.2.3 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 signalling 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 in the performance TL2, ChunkLen in the original TL2) as well as a flag to indicate whether or not this command is the last OCP TL2 command in the burst (LastOfBurst in the performance TL2, last_chunk_of_burst in the original TL2). 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.

5.2.4 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 need to dereference the pointers that passed with the OCP TL2 request or response.

DataLength is in the performance OCP TL2 channel only. In the original OCP TL2 channel, the fields ReqChunkLen and RespChunkLen are used instead of DataLength and are not part of the request and response groups – instead they are passed with the send request and send response commands.

5.2.5 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.

LastOfBurst is in the performance OCP TL2 channel only. In the original OCP TL2 channel, the field last_chunk_of_burst is used instead of LastOfBurst.

5.2.6 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.

5.2.7 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.

5.2.8 MBurstSingleReq

This field is not used by the performance OCP TL2 channel. Instead, it is assumed that if the OCP channel is configured to use Single Request / Multiple Data then the requests and responses are all sent in that format. Whether or not a burst is sent as SRSD (single data) or SRMD (multiple data) through the OCP channel only affects the timing the channel. The timing helper functions in the OCP TL2 performance channel take this into account.

Note that this field is used in the original OCP TL2 channel.

5.2.9 MAtomicLength

This field is not used in the performance OCP TL2 channel but is part of the original OCP TL2 channel.

5.2.10 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.

5.2.11 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.

5.3 Two OCP TL2 Channel Models

There are two different versions of the OCP TL2 channel model. The original OCP TL2 channel was written using a layered approach on top of the generic TL2 channel. As a result, the original OCP TL2 channel supports an OCP specific API (Application Interface) as well as the original generic API.

There is also a performance version of the OCP TL2 channel model. This version of the channel was re-written as a stand-alone model and does not contain the generic channel within it. The performance OCP TL2 channel has expanded functionality including timing points and helper timing functions for accurate timing calculations, and better burst support with integrated DataLength and LastOfBurst fields as well as (optional) pointers for ByteEn (byte enables) and DataInfo fields that may change with each data word in the burst. The performance TL2 channel has its own API to support the new functionality and is fully compatible with the original OCP TL2 channel API. However, the performance OCP TL2 channel does not support all generic commands.

The sections below further cover the differences between the two channels as well as how to convert models from the original OCP TL2 channel to the newer performance OCP TL2 channel.

5.3.1 When to Use the Performance OCP TL2 Channel

The performance OCP TL2 channel is the best choice when simulation performance is an issue as the performance channel has approximately 85% greater throughput than the original OCP TL2 channel. The performance channel is also the better choice if accurate timing is desired as the performance OCP TL2 channel provides timing points and timing helper functions that allow a master or slave core to calculate how long it took a request burst or response burst to cross the channel and to calculate when each of the individual data words of the burst was sent as well. Finally, the performance OCP TL2 channel is the better choice for if some of the data signals, such as ByteEn (byte enable) or DataInfo vary during the burst. The performance OCP TL2 channel provides (optional) pointers for passing these variable fields.

5.3.2 When to Use the Original OCP TL2 Channel

The original OCP TL2 channel is the best choice if your core uses the original generic commands to place requests and responses on the channel. The original OCP TL2 is built on top of the generic channel and supports the full set of generic commands while the performance OCP TL2 channel does not support all of them. The original OCP TL2 channel as better support for live reset, especially if the reset occurs in a blocking command called from an SC_THREAD. Finally, the original OCP TL2 channel supports the generic direct interface while the performance TL2 channel does not.

5.4 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.

5.4.1 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.

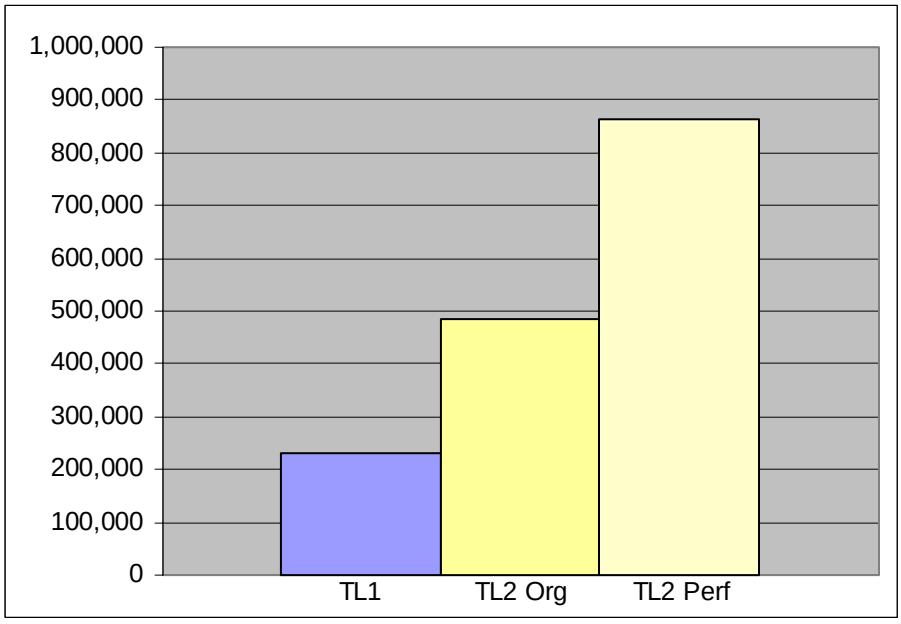
5.4.2 Single Data Word Writes and Reads

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 7 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



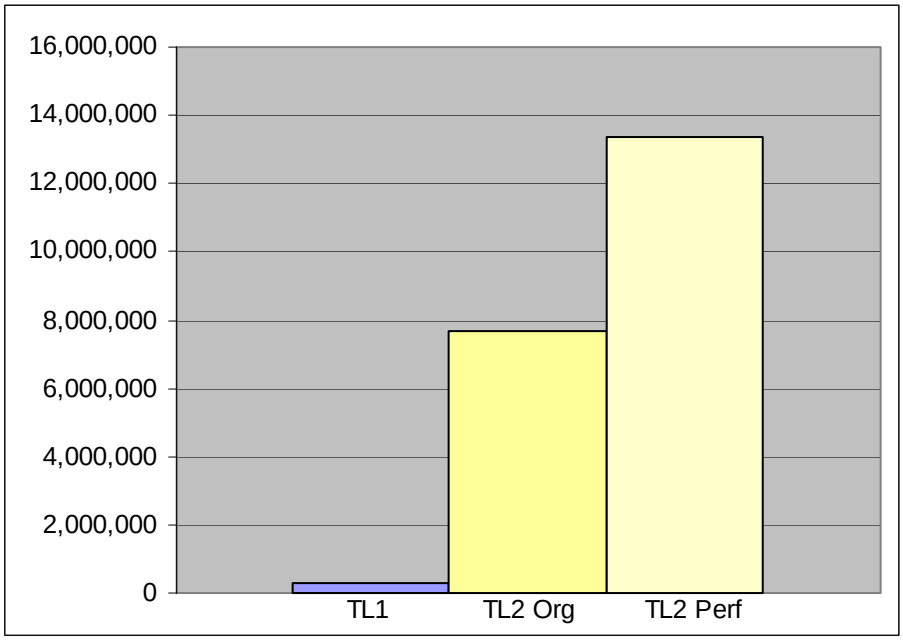
5.4.3 Burst Writes and Reads

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 8 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



5.5 Converting Cores Written for the Original OCP TL2 Channel

Cores written for the original OCP TL2 channel and the original OCP TL2 API will need a few minor tweaks in order to work with the new performance TL2 channel. This section lists what needs to be done.

5.5.1 Parameters

The template on the parameters has been changed since the performance channel does not use the generic templated data class. Thus the code that reads the channel parameters should be changed from the original TL2 form:

```
template <class TdataCl> *ParamCl;
ParamCl = ocpTL2Port->GetParamCl();
```

to the new template-free performance TL2 form:

```
OCPParameters *ParamCl;
ParamCl = ocpTL2Port->GetParamCl();
```

5.5.2 Default Events

The performance OCP TL2 channel does not define a default event for the channel. Thus, processes that are sensitive to an event on the channel should be specific about which event. For example, a process in the master that waits for new responses should be sensitive to ResponseStartEvent (a new response has been placed on the channel. If

the method was using the default event, its sensitivity should be changed from the original OCP TL2 channel code:

```
sensitive<<MasterP;
```

To the performance OCP TL2 code:

```
sensitive<<MasterP.ResponseStartEvent();
```

5.5.3 Data Class

The performance TL2 channel does not use the data class so reference to the data class (if any) should be removed. Use the OCP API instead.

5.5.4 Communications Class

The performance TL2 channel does not use the communications class so references to the communications class (if any) should be removed. Use the parameters class instead.

5.5.5 Direct Interfaces

The performance TL2 channel does not support the generic direct interfaces.

6 The Performance OCP TL2 channel

The performance OCP TL2 channel uses new TL2 specific data structures and an OCP TL2 specific API. While it is possible to use the original TL2 data structures and API with the new performance channel, the new structures and API will give better performance and enable all of the new features of the channel.

6.1 Data Structures for the performance 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.

6.1.1 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.

6.1.1.1 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.

6.1.1.2 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 9 OCPTL2RequestGrp Members

Name	Data Type	Description
MCmd	OCPMCmdType	Master command
MAddr	AddrType	Address of first data word of the request.
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.
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.

Name	Data Type	Description
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
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.
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.

6.1.2 OCPTL2ResponseGrp Template Class

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

Template<class Td>
OCPTL2ResponseGrp

6.1.2.1 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.

6.1.2.2 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 10 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.
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
SDataInfo	unsigned long long int	Extra information about the response data.

Name	Type	Description
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.

6.1.3 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.

6.1.3.1 Master Timing Variables

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

Table 11 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.

6.1.3.2 Slave Timing Variables

Table 12 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.

6.2 Building the OCP TL2 Channel

6.2.1 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.

6.2.2 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.

6.2.3 Setting the Parameters

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

void setConfiguration(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.

6.3 Performance OCP TL2 Master Interface Methods (ocp_tl2_master_if.h)

The methods described in this section handle the performance 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.
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.

6.4 Performance OCP TL2 Slave Interface Methods (ocp_tl2_slave_h)

The methods described in this section handle the performance 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.

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. Returns.
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.

6.5 Performance OCP TL2 Channel Events

The methods described in this section handle give access to the events generated by the performance 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.

6.6 Reset

The performance 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.

6.7 Sideband signals

The performance 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.

6.8 Timing Model for the Performance OCP TL2 Channel

The timing model for the performance 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.

6.8.1 Time in the Performance 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.

6.8.2 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[2]. 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.

6.8.3 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.

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

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 underling 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 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.

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>

6.8.4 Write Requests

Figure 6 Timing information for MRMD posted Write Burst with datahandshake

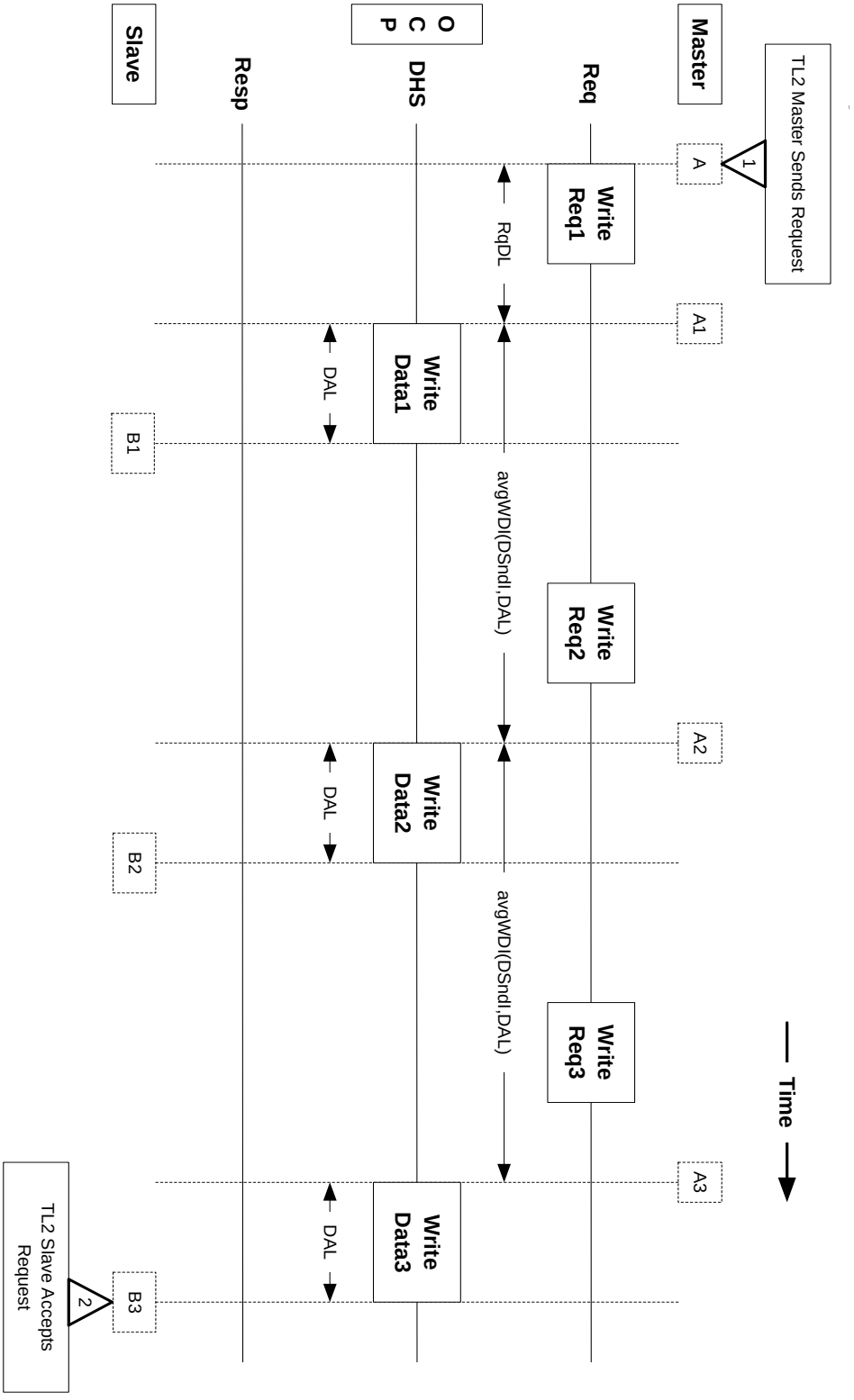
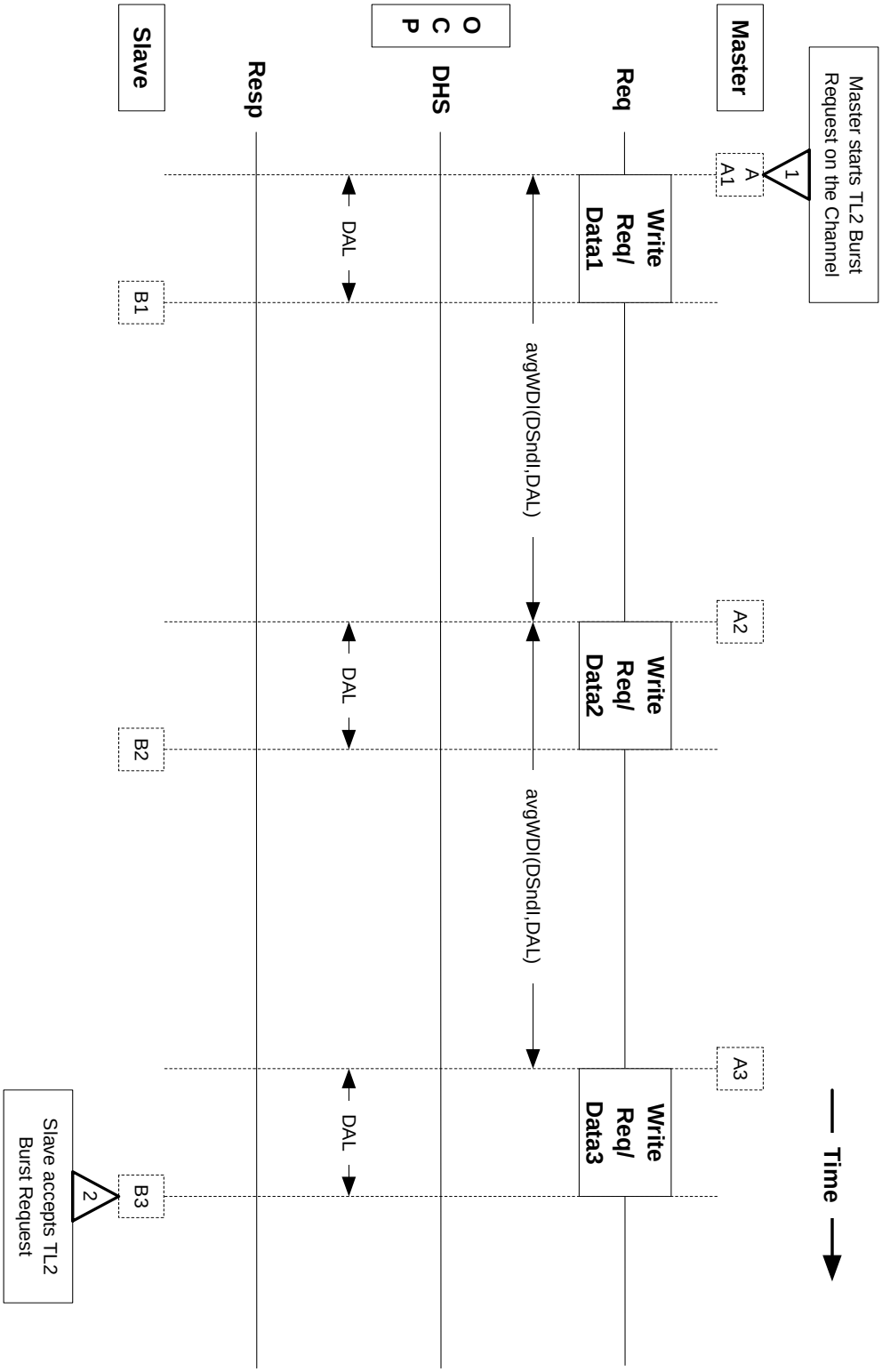
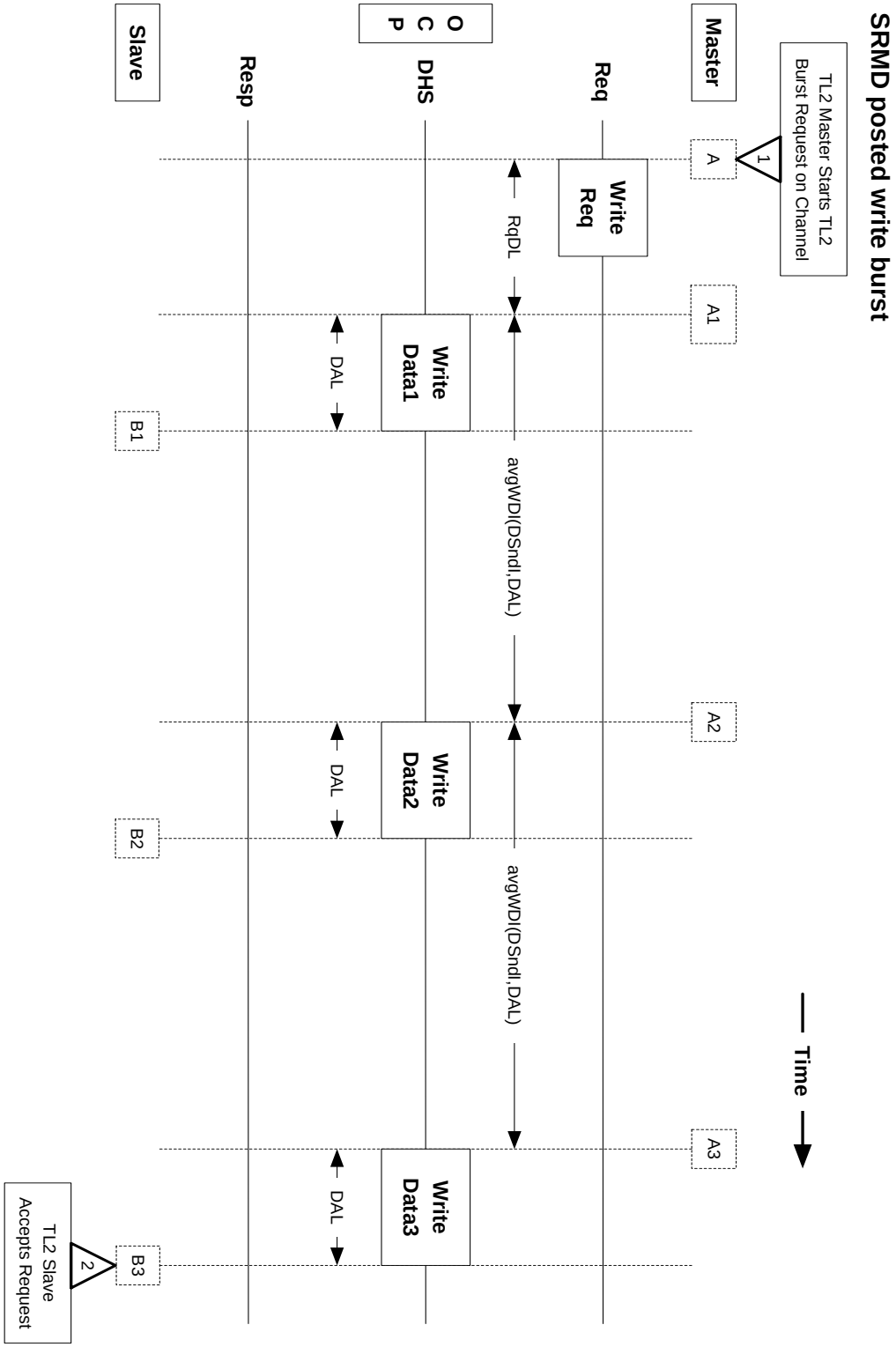


Figure 7 Timing information for MRMD posted Write Burst w/o datahandshake



6.8.5 OCP Posted Write Burst Timing

Figure 8 Timing information for SRMD posted Write Burst



The current OCP TL2 channel model 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.

6.8.5.1 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.

6.8.5.2 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:

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

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:

$$\text{avgWDI} = \max(\text{DSndI}, \text{DAL})$$

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).

6.8.5.3 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 performance 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.

6.8.5.4 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.

6.8.5.5 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.

6.8.6 Read Requests

Figure 9 Timing Information for MRMD Read Burst

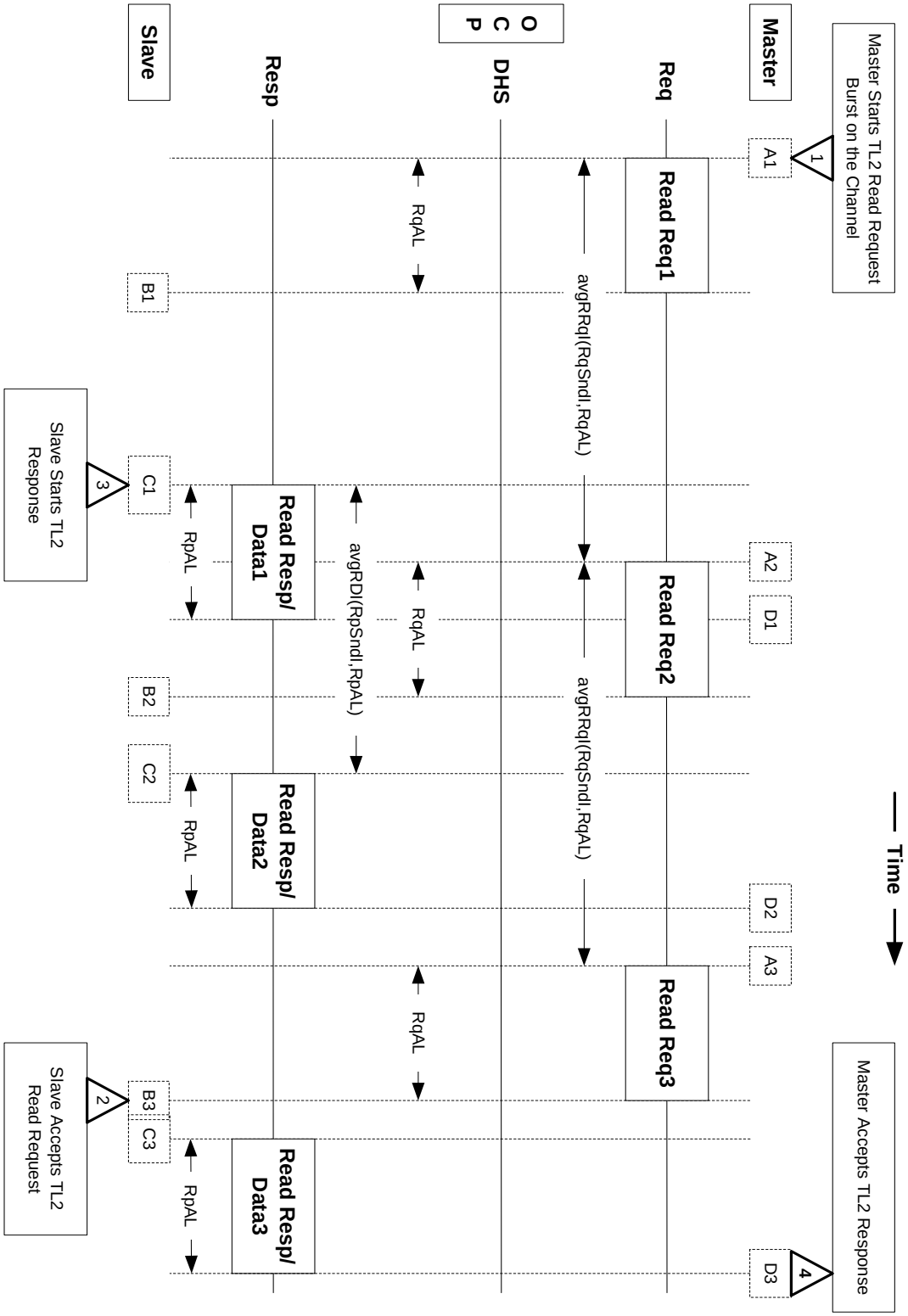
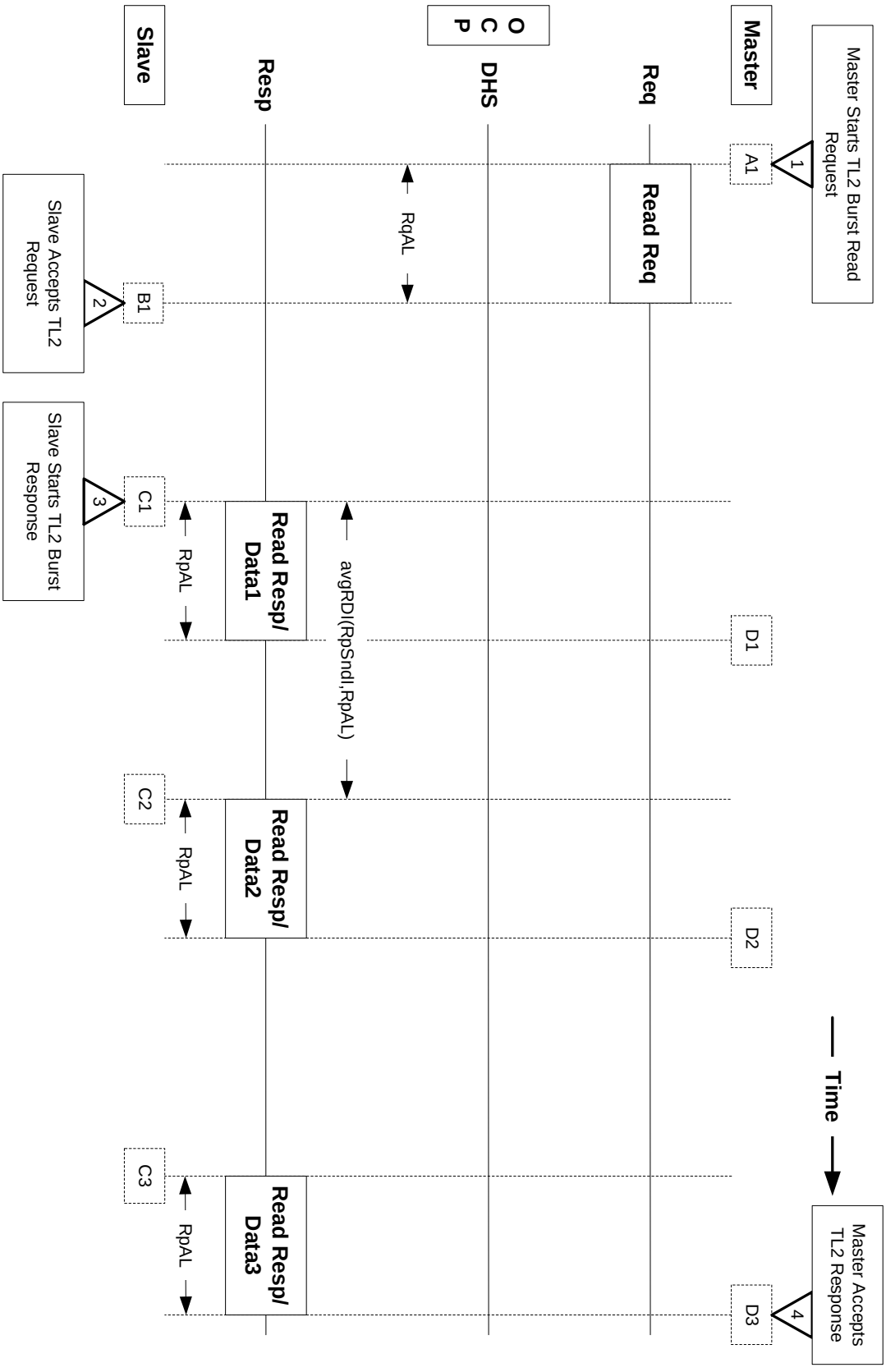


Figure 10 Timing information for SRMD Read Burst



6.8.7 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.

6.8.7.1 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. As shown in **Error! Reference source not found.**, we represent the read request interval value by the following function:

```
avgRRqI = max(RqSndI, 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).

6.8.7.2 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.

6.8.7.3 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".

6.8.7.4 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).

6.8.7.5 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.

6.8.8 Non-Posted Writes

Figure 11 MRMD Non-Posted Write Burst with data handshake

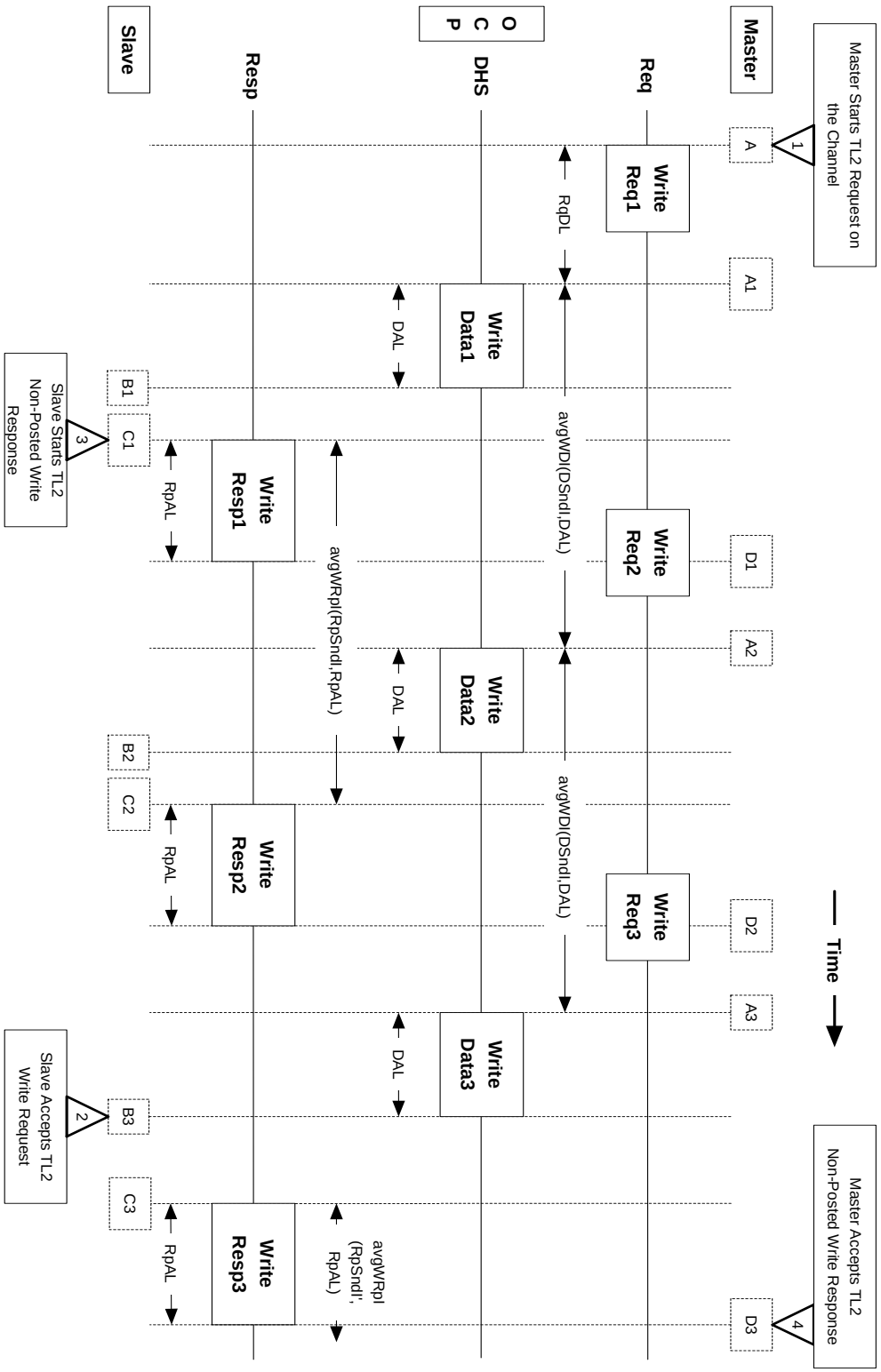


Figure 12 Timing information for MRMD non-posted Write Burst w/o datahandshake

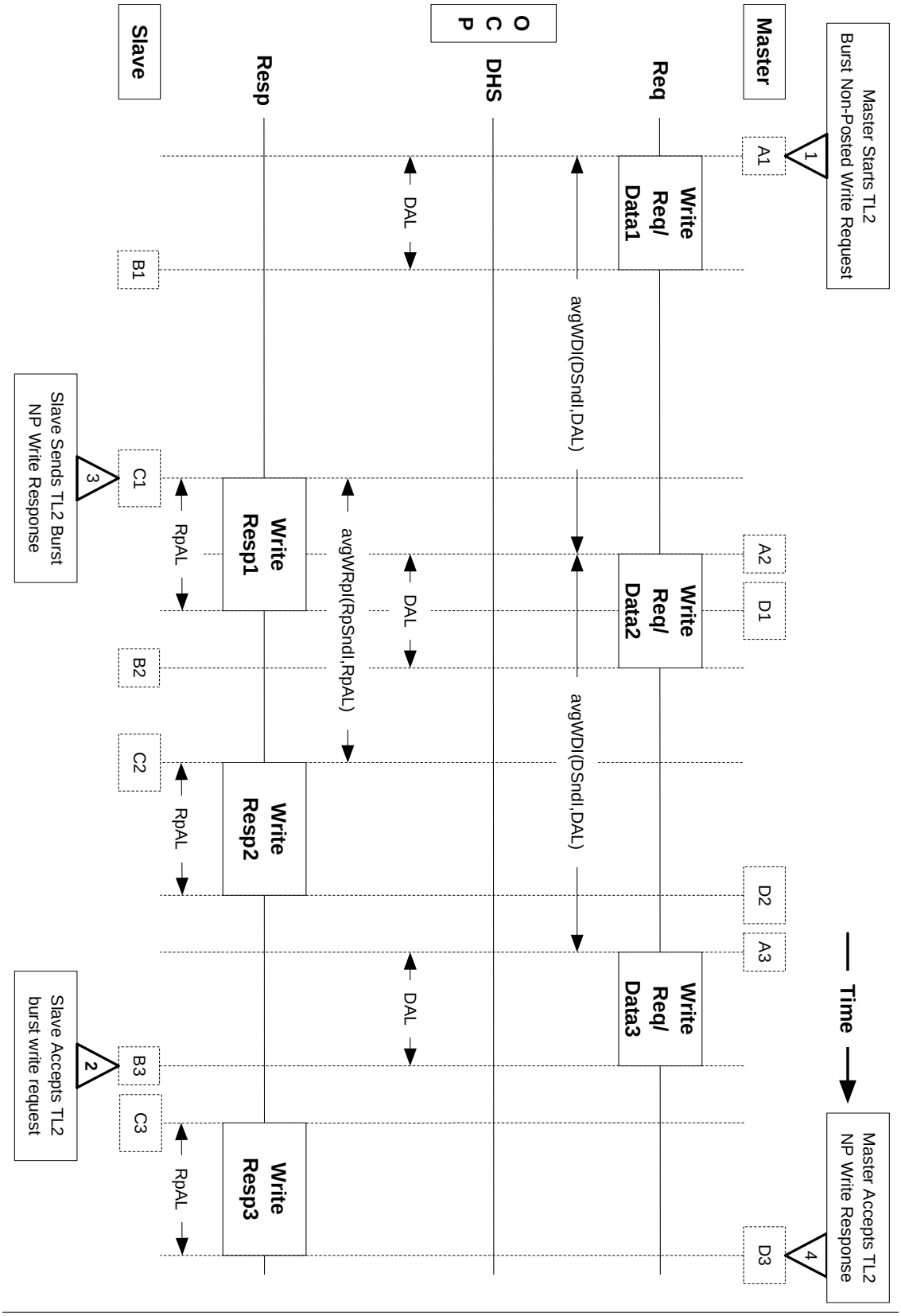
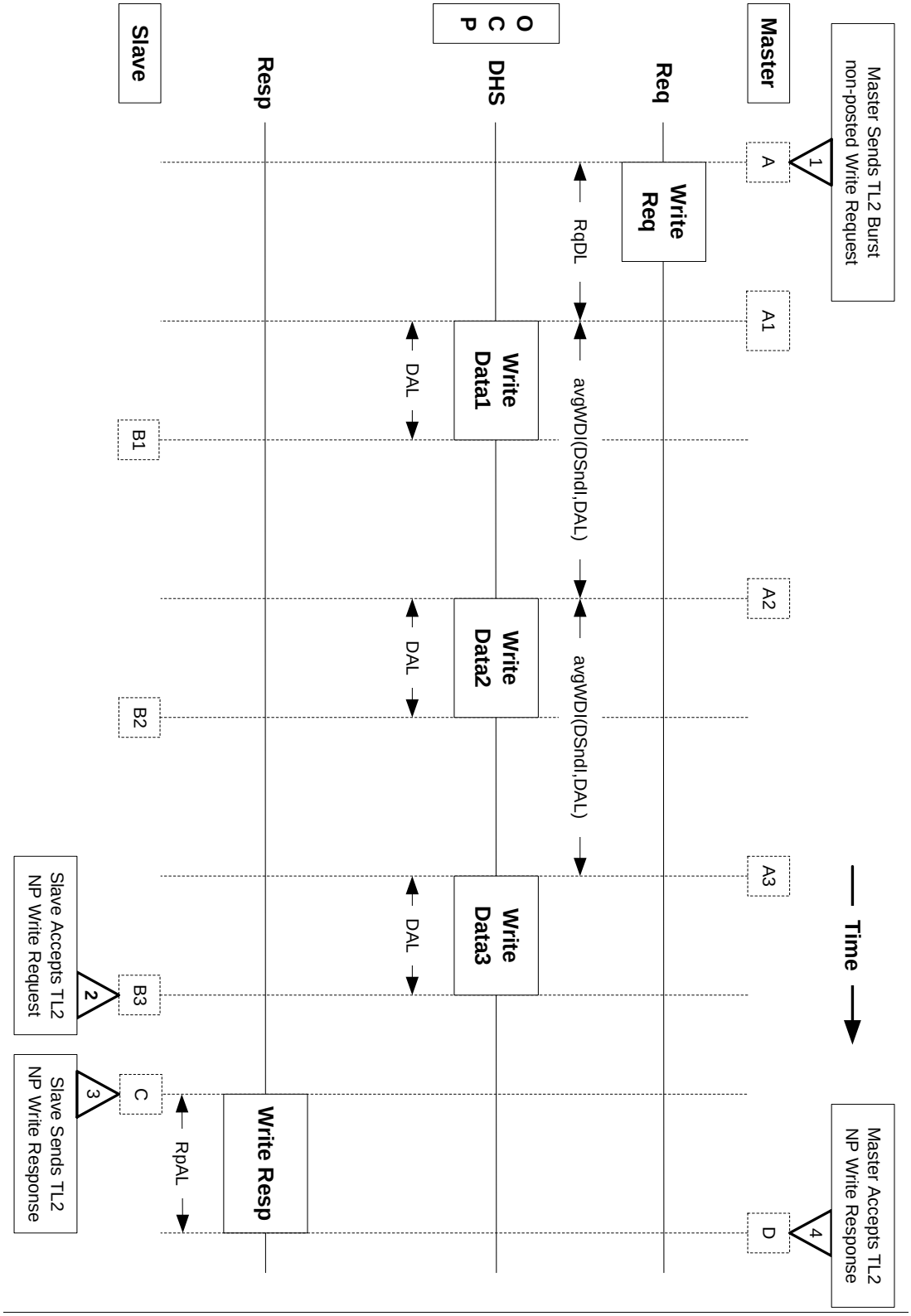


Figure 13 Timing information for SRMD non-posted Write Burst



6.8.9 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.

6.8.9.1 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.

6.8.9.2 Posted Write Burst with Responses

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

6.8.10 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 13 lists timing variables that are stored in the channel to help derive the timing points of the corresponding transfers.

Table 13 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.

6.8.11 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.

7 The Original OCP TL2 Channel

The original OCP TL2 specific channel has OCP specific commands for sending and accepting OCP requests and responses. Because the channel model was designed specifically for OCP TL2 transactions, it is both easier to use and it ensures that the channel is OCP correct.

Because the original OCP TL2 specific channel is built upon the base generic channel and the OCP TL2 data class, it is possible to use generic commands with the OCP TL2 channel. However, this is strongly discouraged. Doing so may lead to unexpected behavior that is out of the bounds of the OCP protocol.

7.1 Original OCP TL2 Channel Constructor

The OCP TL2 channel has the following constructor:

OCP_TL2_Channel(sc_module_name name)
name Name of the module (channel) instance.

7.2 Original OCP TL2 Specific Enum Types and Template Classes

The OCP TL2 commands can pass requests and responses through as single structures. This section describes those structures (actually template classes) as well as the Enum types used by elements of those structures. (See `tl_sc/include/ocp/ocp_tl_globals`.)

7.2.1 OCPMCmdType, OCPSRespType and OCPMBurstSeqType Enums

These enums are the same that are used for the OCP TL1 Specific channel. See section 4.2 “OCP TL1 Specific Enum Types and Template Classes.”

7.2.2 OCPRequestGrp Template Class

This class is the same as for the OCP TL1 Specific channel (See section 4.2.4 “OCPRequestGrp Template Class.” From the user’s point of view, the only difference is with the `MData` member of the structure, which should be ignored in TL2. Users should instead use the TL2 specific member `MDataPtr` to set or get the pointer on the master data array. Note that the assignment operator (=) and the `copy()` method copy the value of the pointer from one instance to another and do not copy the array itself.

7.2.3 OCPResponsetGrp Template Class

This class is the same as for the OCP TL1 Specific channel. (See section 4.2.5 “OCPResponseGrp Template Class.”) From the user’s point of view, the only difference between the classes is the `SData` member of the structure, which should be ignored in TL2. Users should instead use the TL2 specific member `SDataPtr` to set or get the pointer on the slave data array. Note that the assignment operator (=) and the `copy()` method copy the value of the pointer from one instance to another and do not copy the array itself.

7.3 Original TL2 Master Interface Methods (ocp_tl2_master_if.h)

The methods described in this section handle the original OCP TL2 master's transaction request phase and response phase.

7.3.1 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 Reset()

Purpose: Calls MResetAssert() and MResetDeassert()

Return: None.

Events: All start and end events fire (to release all waits in the system) immediately, and ResetEndEvent fires after a delta cycle.

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 after a delta cycle.

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.

7.3.2 Request Phase

bool getSBusy ()

Purpose: Status of the slave-busy semaphore. Indicates whether the slave has released the previous request.

Return: Immediately returns true if slave has not responded to the last request event, and false if it has.

bool getSCmdAccept()

Purpose: Returns the current value of the SCmdAccept signal.

Return: Returns true if the current command was accepted. Returns false if the current command has not been accepted, or if there is no current command.

waitSCmdAccept()

Purpose: Waits until SCmdAccept is asserted by the slave.

bool getSThreadBusyBit(unsigned int ThreadID = 0)

Return: Returns the right bit of the SThreadBusy signal corresponding to the ThreadID.

bool sendOCPRequest(OCPRequestGrp<Tdata,Taddr>& req,
int ReqChunkLen = 1,
bool last_chunk_of_a_burst = true)

Purpose: Places the passed request on the channel. The ReqChunkLen parameter specifies the length of the request chunk. Note that the data array pointed by the MdataPtr member of the request must have its size equal to ReqChunkLen in case of a WRITE request. The last_chunk_of_a_burst parameter indicates whether this request chunk is the last one of a complete request burst.

Return: Returns false in the following cases:

- Another request in progress
- The channel is configured with sthreadbusy_exact set to 1, SThreadBusy is tested (relatively to the MThreadID field of the request), and the method returns false if the slave thread is busy.
- The channel is in a reset state

bool startOCPRequest(OCPRequestGrp<Tdata,Taddr>& req,
int ReqChunkLen = 1,
bool last_chunk_of_a_burst = true)

Purpose: This function has exactly the same behaviour as 'sendOCPRequest' and can be considered as an alias. Its semantic is equivalent to the TL1 API corresponding function.

bool startOCPRequestBlocking(
OCPRequestGrp<Tdata,Taddr>& req,
int ReqChunkLen = 1,
bool last_chunk_of_a_burst = true)

Purpose: Waits until the channel is free for a new request then starts the passed request on the channel. This call returns once the request has started but before the slave has accepted the request. The parameters have the same meaning as for sendOCPRequest().The semantic of this function is equivalent to the TL1 API corresponding function.

Return: Returns false in the following cases:

- There is already a (send/start)OCPRequestBlocking waiting to be sent
- There is a (send/start)OCPRequest call in progress
- If the channel is configured with sthreadbusy_exact set to 1, SThreadBusy is tested (relatively to the MThreadID field of the request), and the method returns false if the slave thread is busy. Note that the

SThreadBusy test occurs at the beginning of the call before testing if the request channel is free.

- The channel is in a reset state

bool sendOCPRequestBlocking(
OCPRequestGrp<Tdata,Taddr>& req,
int ReqChunkLen = 1,
bool last_chunk_of_a_burst = true)

Purpose: Waits until the channel is free for a new request then starts the passed request on the channel. This call returns once the slave has accepted the request. The parameters have the same meaning as for sendOCPRequest().

Return: Returns false in the following cases:

- There is already a (send/start)OCPRequestBlocking waiting to be sent
- There is a (send/start)OCPRequest call in progress
- If the channel is configured with sthreadbusy_exact set to 1, SThreadBusy is tested (relatively to the MThreadID field of the request), and the method returns false if the slave thread is busy. Note that the SThreadBusy test occurs at the beginning of the call before testing if the request channel is free.
- The channel is in a reset state

sc_event& RequestStartEvent()

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

Return: SystemC event.

sc_event& RequestEndEvent()

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

Return: SystemC event.

sc_event& SThreadBusyEvent()

Purpose: This event is triggered when the SThreadBusy signal changes.

Return: SystemC event.

7.3.3 Response Phase

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. Note that after the response has been accepted, the OCP channel signal SResp is then automatically reset to “OCP_SRESP_NULL”.

bool putMRespAccept(sc_time after)

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

Return: Returns false if there is no response to accept. Note that after the response has been accepted, the OCP channel signal SResp is then automatically reset to “OCP_SRESP_NULL”.

void putMThreadBusyBit(bool nextBitValue,
unsigned int
ThreadID = 0)

Purpose: Sets the right bit of the MThreadBusy signal corresponding to the ThreadID in the OCP channel.

bool getOCPResponse(OCPResponseGrp<Tdata>& resp,
bool accept,
unsigned int& RespChunkLen,
bool& last_chunk_of_a_burst)

Purpose: If there is a new, unread response is available on the channel, the response is read and returned as “resp” , and if accept 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 accept parameter is ignored. The RespChunkLen parameter specifies the length of the response chunk. The Last_chunk_of_a_burst parameter indicates if this response chunk is the last one of a complete response burst.

Return: Returns false in the following cases:

- No response is available
- A getOCPResponse or a getOCPResponseBlocking has already read the response.
- A getOCPResponseBlocking command is already in progress.
- The channel is in a reset state

bool getOCPResponseBlocking(OCPResponseGrp<Tdata>& resp,
bool accept,
unsigned int& RespChunkLen,
bool& last_chunk_of_a_burst)

Purpose: Waits for a new, unread response to become available on the channel. The response is then read and eventually accepted (depending on the accept parameter) and returned as “resp”. Parameters have the same meaning as for getOCPResponse().

Return: Returns false if there is already another getResponseBlocking command in progress, or if the channel is in a reset state.

sc_event& ResponseStartEvent()

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

Return: SystemC event.

sc_event& ResponseEndEvent()

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

Return: SystemC event.

7.3.4 Serialized Methods

These methods could be used to write testbenches at the TL2 level. Serialized methods take charge of both request and response phases of a complete OCP transaction, making testbenches more compact and easier to code.

bool OCPReadTransfer(OCPRequestGrp<Tdata,Taddr>& req,
OCPResponseGrp<Tdata>& resp,
int TransferLen = 1)

Purpose: Issues a blocking request call to pass `req` on the channel, waits for the slave to release the request, then issues a blocking response call to retrieve the response, stores it in `resp`, and releases the response. The `TransferLen` parameter specifies the size of the data array pointed to by `req.MDataPtr`.

Return: Returns false in the following cases:

- Request Phase:
  - The `MCmd` request field is not equal to `OCP_MCMD_RD`
  - A (`send/start`)`OCPRequestBlocking` is already waiting to be sent
  - A (`send/start`)`OCPRequest` call in progress
  - If the channel is configured with `sthreadbusy_exact` is set to 1, `SThreadBusy` is tested (relatively to the `MThreadID` field of the request), and the method returns false if the slave thread is busy.
  - The channel is in a reset state
- Response Phase:
  - Another getOCPResponseBlocking command in progress is already in progress.
  - The SRespChunkLen response field is different from TransferLen, or the SRespChunkLast field is not equal to true. This can happen when the slave truncates the response into several response chunks. In this case, the user should use sendOCPRequest()/getOCPResponse() blocking calls instead.
  - The channel is in a reset state

Note:
Use of this function should be avoided when the OCP channel is configured to support several threads. Because this function gets the first response following the request without testing the SThreadID, there is no guarantee that the response corresponds to the ThreadID of the request.

bool OCPWriteTransfer(OCPRequestGrp<Tdata,Taddr>& req,
int TransferLen = 1)

Purpose: Issues a WRITE request to the slave, and waits for the slave to release the request. `TransferLen` specifies the size of the data array pointed to by `req.MDataPtr`. The transfer is atomic; that is,. the `MReqChunkLast` parameter is set to 1.

Return: Returns false in the following cases:

- The `MCmd` request field is not equal to `OCP_MCMD_WR`.

- A (send/start)OCPRequestBlocking is already waiting to be sent.
- A (send/start)OCPRequest call in progress.
- If the channel is configured with sthreadbusy_exact set to 1, SThreadBusy is tested (relatively to the MThreadID field of the request), and the method returns false if the slave thread is busy.
- The channel is in a reset state

7.4 Original TL2 Slave Interface Methods (ocp_tl2_master_if.h)

The methods described in this section handle the OCP TL2 slave's transaction request phase and response phase.

7.4.1 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 Reset()

Purpose: Calls SResetAssert() and SResetDeassert()

Return: None.

Events: All start and end events fire (to release all waits in the system) immediately, and ResetEndEvent fires after a delta cycle.

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 after a delta cycle.

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.

7.4.2 Request Phase

bool getOCPRequest(OCPRequestGrp<Tdata,Taddr>& req,
bool accept
int& ReqChunkLen,
bool& last_chunk_of_a_burst)

Purpose: If there is a new, unread request available on the channel, the request is read and returned as "req", and if accept 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 accept parameter is ignored. The ReqChunkLen parameter specifies the length of the request chunk. The Last_chunk_of_a_burst parameter indicates if this request chunk is the last one of a complete request burst.

Return: Returns false in the following cases:

- No request available
- A getOCPRequest or a getOCPRequestBlocking has already read the request.
- A getOCPRequestBlocking command is already in progress
- The channel is in a reset state

bool getOCPRequestBlocking(OCPRequestGrp<Tdata,Taddr>& req,
bool accept,
int& ReqChunkLen,
bool& last_chunk_of_a_burst)

Purpose: Waits for a new, unread request to become available on the channel. The request is then read, eventually accepted (depending on the accept parameter), and returned as "req". If not, the method returns false. The parameters have the same meaning as for getOCPRequest().

Return: Returns false if there is already another getOCPRequestBlocking command in progress, or if the channel is in a reset state.

void putSThreadBusyBit(bool nextBitValue,
unsigned int ThreadID = 0)

Purpose: Sets the right bit of the SThreadBusy signal corresponding to the ThreadID in the OCP channel.

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. Note that after the command has been accepted, the OCP channel signal MCmd is then automatically reset to "OCP_MCMD_IDLE".

bool putSCmdAccept(sc_time after)

Purpose: Sets the SCmdAccept signal in the OCP channel and releases the request after time delay.

Return: Returns false if there is no request to accept or if the current request has already been accepted. Note that after the command has been accepted, the OCP channel signal MCmd is then automatically reset to "OCP_MCMD_IDLE".

sc_event& RequestStartEvent()

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

Return: SystemC event.

sc_event& RequestEndEvent()

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

Return: SystemC event.

7.4.3 Response Phase

bool sendOCPResponse(OCPResponseGrp<Tdata>& resp,
int RespChunkLen = 1,
bool last_chunk_of_a_burst = true)

Purpose: Places the passed response onto the channel. The `ReqChunkLen` parameter specifies the length of the response chunk. Note that the data array pointed to by the `SdataPtr` member of the response must have its size equal to `ReqChunkLen`. The `last_chunk_of_a_burst` parameter indicates if this response chunk is the last one of a complete response burst.

Return: Returns false in the following cases:

- Another response in progress
- If the channel is configured with `mthreadbusy_exact` set to 1, `MThreadBusy` is tested, and the method returns false if the master is busy.
- The channel is in a reset state

bool startOCPResponse(OCPResponseGrp<Tdata>& resp,
int RespChunkLen = 1,
bool last_chunk_of_a_burst = true)

Purpose: This function has exactly the same behaviour as '`sendOCPResponse`' and can be considered as an alias. Its semantic is equivalent to the TL1 API corresponding function.

bool sendOCPResponseBlocking(
OCPResponseGrp<Tdata>& resp,
int RespChunkLen = 1,
bool last_chunk_of_a_burst = true)

Purpose: Waits until the channel is free for a new response and then starts the passed response on the channel. `sendOCPResponseBlocking()` returns when the master has accepted the response. The parameters have the same meaning as for `sendOCPResponse()`.

Return: Returns false in the following cases:

- A (`send/start`)`OCPResponseBlocking` is already waiting to be sent
- A (`send/start`)`OCPResponse` call in progress
- The channel is in a reset state

Note

If the channel is configured with `mthreadbusy_exact` set to 1, MThreadBusy is tested and the method returns false if the master is busy. Note that the MThreadBusy test occurs at the beginning of the call before testing if the response channel is free.

bool startOCPResponseBlocking(
OCPResponseGrp<Tdata>& resp,
int RespChunkLen = 1,
bool last_chunk_of_a_burst = true)

Purpose: Waits until the channel is free for a new response and then starts the passed response on the channel. This call returns once the response has started but **before the master has accepted the response**. The parameters have the same meaning as for `sendOCPResponse()`. The semantic of this function is equivalent to the TL1 API corresponding function.

Return: Returns false in the following cases:

- A (`send/start)OCPResponseBlocking` is already waiting to be sent
- A (`send/start)OCPResponse` call in progress
- The channel is in a reset state

Note

If the channel is configured with `mthreadbusy_exact` set to 1, MThreadBusy is tested and the method returns false if the master is busy. Note that the MThreadBusy test occurs at the beginning of the call before testing if the response channel is free.

bool getMBusy()

Purpose: Status of the master-busy semaphore. This method indicates whether the master has released the previous response.

Return: Immediately returns true if master has not responded to the last response event, and false if it has.

bool getMThreadBusyBit(unsigned int ThreadID = 0)

Purpose: Returns the right bit of the MThreadBusy signal corresponding to the `ThreadID`.

bool getMRespAccept()

Purpose: Checks whether the master has accepted the current response.

Return: Returns true if the current response has been accepted.

void waitMRespAccept()

Purpose: Waits until MRespAccept is asserted by the master

sc_event& ResponsetStartEvent()

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

Return: SystemC event.

sc_event& ResponseEndEvent()

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

Return: SystemC event.

sc_event& MThreadBusyEvent()

Purpose: This event is triggered when the MThreadBusy signal changes.

Return: SystemC event.

8 Example Using OCP Specific TL1 Channel and API

The example described in this section demonstrates the use of the OCP Specific 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 specific 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 specific TL1 API. The third part of the example is a configurable slave core that also uses the OCP specific TL1 API.

This example makes a heavy use of blocking TL1 methods, and timed wait statements. There are simpler examples included in the release package that use non-blocking methods and clocks.

8.1 Configuring the OCP Specific TL1 Channel

The OCP specific TL1 channel can be configured using any of the standard OCP configuration parameters. This section illustrates some of these parameters, but is by no means complete. For the complete list of OCP parameters, 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 the OCP specification document.

The channel should be configured anytime after it is created and before the simulation is started. To configure the channel, the channel's `setConfiguration()` function is called with a MAP object that contains all of the parameter settings, for example:

```
setConfiguration( 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 user's `main.cc` program, or it may be built by reading in parameter data from a file. Section 8.1.1 gives the details of the parameter MAP format.

8.1.1 Parameter Map Format

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.”

The following is an example that builds a simple parameter map and then uses it to configure the channel. OCP Parameters which are not set by the user are configured to their default value as specified in the OCP Specification.

```
// C++ STL include
# include <map>
// Create a parameter map:
map<string, string> myParamMap;
myParamMap.insert( make_pair( “cmdaccept”, “i:1” ) );
myParamMap.insert( make_pair( “addr_wdth”, “i:40” ) );
myParamMap.insert( make_pair( “endian”, “s:big” ) );
// etc…

// Send it to the channel
myOcpChannel->setConfiguration(myParamMap);
```

8.1.2 Building the Parameter Map from a File

You can also build the parameter map by using a file. 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 fil and stores them into an STL map. The map is then passed to the channel’s `setConfiguration` function.

8.1.3 Configurable Master and Slave

The same parameter map scheme described in section 8.1.1is used to configure the reference master and reference slave. The following table gives the parameters for the reference master.

Table 14 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>::setConfiguration( MapStringType& passedMap )
```

The following table gives the parameters for the reference slave:

Table 15 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
limitreq_enable	i	0 (false)	Should the slave limit how many requests it has outstanding?
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>::setConfiguration( MapStringType& passedMap )
```

8.1.4 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.

8.2 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

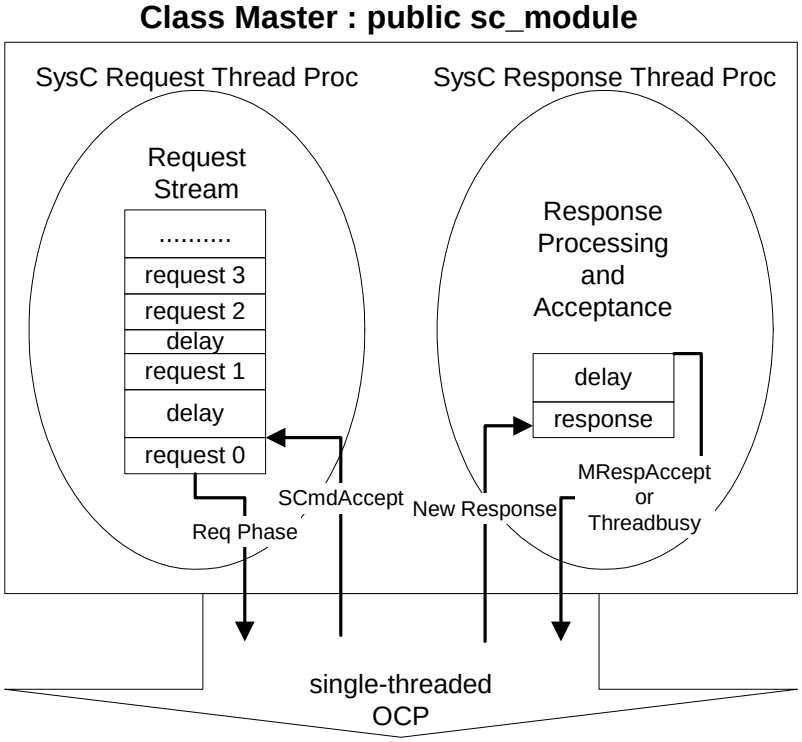
The address, the request type (WR or RD), and the write data of a request can also be specified.

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 14 Master Model



8.2.1 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 specific 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 specific 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 files ocp_tl1_data_cl.h and ocp_globals.h.

The header ocp_tl1_master_port.h contains the master port to the OCP specific 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 a 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 declaared the as follows, which would also work:

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

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 pointer that points to the OCP parameters of the OCP channel that is connected to the master core model's ipP port:

```
ParamCl<TdataCl>*  m_pOCPParam;  // pointer to OCP 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

#include <iostream>
#include "stdlib.h"
#include "globals.h"

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

// For multithreaded masters only
// #include "master_data_queue.h"

// define the Master transactor class
template <typename TdataCl>
class Master : public sc_module
{
  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;

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

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

    // methods
    void setConfiguration( MapStringType& passedMap );

    // 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;

    // ocp clock information
    double       m_ocpClkPeriod;
    sc_time_unit m_ocpClkTimeUnit;

    // model a per thread data queue
    // used for multi-threaded master
    // DataQueue<TdataCl> m_DataQueueThread0;

    //
    ostream* m_debug_os_ptr;

    // Parameters from the OCP Channel:

    // Class that holds all OCP parameters
    ParamCl<TdataCl>* m_OCPParamP;

    // 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
    MapStringType m_ParamMap;

};

#endif // _SIMPLE_MASTER_H
```

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

```
// ------------------------------------------------------------------
--
// constructor
// ------------------------------------------------------------------
--
template<typename TdataCl>
Master<TdataCl>::Master(
    sc_module_name name,
    double         ocp_clock_period,
    sc_time_unit   ocp_clock_time_unit,
    int            id,
    ostream*       debug_os_ptr
) : sc_module(name),
    ipP("ipPort"),
    m_ID(id),
    m_ocpClkPeriod(ocp_clock_period),
```

```
        m_ocpClkTimeUnit(ocp_clock_time_unit),
        m_debug_os_ptr(debug_os_ptr),
        m_OCPParamP(NULL),
        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_delay(0)
    {
        // setup a SystemC thread process, which uses dynamic sensitive
        SC_THREAD(requestThreadProcess);

        // setup a SystemC thread process, which uses dynamic sensitive
        SC_THREAD(responseThreadProcess);

        // setup a SystemC thread process to drive any connected
    sideband signals
        SC_THREAD(exerciseSidebandThreadProcess);
    }
```

8.2.3 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 `setConfiguration` function should have been called with a parameter map of the master's parameters. During the end_of_elaboration( ) method, that master processes this parameter map to set its own master parameters.

At the end of elaboration point, the OCP channel must have already been connected to the core. The master takes advantage of this to read the OCP parameters of the channel and then uses those parameters to configure itself to work with the channel it was connected to.

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

- The `GetParamCl()` method returns a pointer that points to the OCP channel's parameters. The master then uses this pointer to extract the channel's parameters and to use them to configure itself. For example,

  ```
  m_OCPParamP = ipP->GetParamCl();
  ```

- The master uses functions in the `ParamCl` class that extract integers and Booleans from string formatted parameter maps. For example, the complex looking function call

  ```
  ParamCl<TdataCl>::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 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
    //---------------------------------------

    // This Master adjusts to the OCP it is connected to.

    // Now get my OCP parameters from the port.
    m_OCPParamP = ipP->GetParamCl();

    // Get the number of threads
    m_threads = m_OCPParamP->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_OCPParamP->addrspace;

    // is SThreadBusy part of the channel?
    m_sthreadbusy = m_OCPParamP->sthreadbusy;

    // Is SThreadBusy compliance required?
    m_sthreadbusy_exact = m_OCPParamP->sthreadbusy_exact;

    // is MThreadBusy part of the channel?
    m_mthreadbusy = m_OCPParamP->mthreadbusy;

    // Is MThreadBusy compliance required?
    m_mthreadbusy_exact = m_OCPParamP->mthreadbusy_exact;

    // is MRespAccept part of the channel?
    m_respaccept = m_OCPParamP->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_OCPParamP->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_OCPParamP->writeresp_enable;

    // is READ-EX part of the channel?
    m_readex_enable = m_OCPParamP->readex_enable;

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

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

    // Retrieve any configuration parameters that were passed to
this block
    // in the setConfiguration command.

#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_ParamMap.begin(); map_it != m_ParamMap.end();
++map_it) {
        cout << "map[" << map_it->first << "] = " << map_it->second
<< endl;
    }
    cout << endl;
#endif

    string myPrefix = "";
    string paramName = "undefined";

    // MRespAccept delay in OCP cycles
    paramName = "mrespaccept_delay";
    if (!(ParamCl<TdataCl>::getIntOCPConfigValue(myPrefix,
paramName,
            m_respaccept_delay, m_ParamMap)) ) {
        // Could not find the parameter so we must set it to a
default
#ifdef DEBUG
        cout << "Warning: master paramter \"" << paramName
```

```
              << "\" for Master \"" << name()
               << "\" was not found in the parameter map." << endl;
        cout << "         setting missing parameter to 1." << endl;
#endif
        m_respaccept_delay = 1;
    }

    // MRespAccept Delay Style. 1=fixed delay : 0=random delay
    paramName = "mrespaccept_fixeddelay";
    if (!(ParamCl<TdataCl>::getBoolOCPConfigValue(myPrefix,
paramName,
            m_respaccept_fixeddelay, m_ParamMap)) ) {
        // Could not find the parameter so we must set it to a
default
#ifdef DEBUG
        cout << "Warning: master paramter \"" << paramName
              << "\" for Master \"" << name()
              << "\" was not found in the parameter map." << endl;
        cout << "         setting missing parameter to 1 (fixed
delay)."
                << endl;
#endif
        m_respaccept_fixeddelay = true;
    }
}
```

8.2.4 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`. 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 hurtle 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 specific 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.

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;

    ipP->ocpWait();

    // 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;
        }

        ipP->ocpWait(wait_for);

        // 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) {
            sthreadbusy = ipP->getSThreadBusy();
            while (sthreadbusy) {
                ipP->ocpWait();
                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
```

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

            // 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;
            }
        }
    }
```

8.2.5 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 following command causes the response thread to wait for wait_for OCP channel cycles before resuming:

```
ipP->ocpWait(wait_for);
```

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;

    ipP->ocpWait();

    // 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
                ipP->ocpWait(wait_for);
                //wait(ocpClkP->posedge_event());
```

```
                // 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
                ipP->putMThreadBusy(1);
                // keep MThreadBusy on
                ipP->ocpWait(wait_for);
                // now release it
                ipP->putMThreadBusy(0);
            }
        }
    }
}
```

8.2.6 SystemC Sideband Process

The code example shown in this section is a simple process that illustrates how the OCP specific 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.
    ipP->ocpWait(10);
    int tweakCounter =0;
    bool hasMError = m_OCPParamP->merror;
    bool nextMError = false;
    bool hasMFlag = m_OCPParamP->mflag;
    int numMFlag = m_OCPParamP->mflag_wdth;
    unsigned int nextMFlag = 0;
    unsigned int maxMFlag = (1 << numMFlag) -1;

    // main loop
    while (true) {
        // wait 10 cycles
        ipP->ocpWait(10);

        // 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);
                }
            }
        }
    }
```

8.2.7 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 >;
```

8.3 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

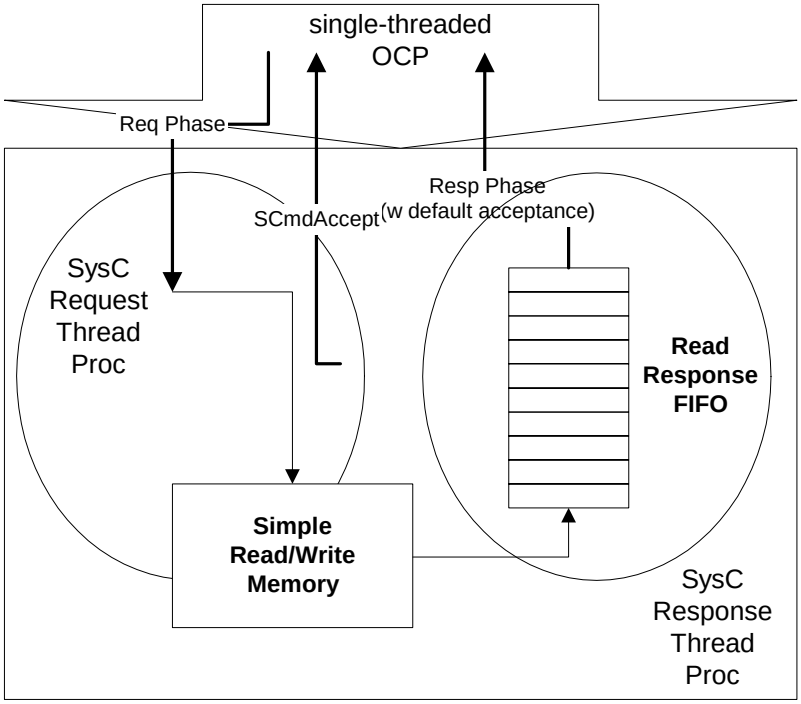
Parameters belonging to the slave model itself are:

`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_enable` and `limitreq_max`
: When the `limitreq_enable` parameter is set to 1, the outstanding requests per thread are limited to `limitreq_max`

Figure 15 shows a diagram of the configurable slave model.

Figure 15 Slave Model



Class Slave : public sc_module

8.3.1 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 specific TL1 channel header files:

```
// OCP-IP Channel header files
#include "globals.h"
#include "ocp_tl1_slave_port.h"
#include "ocp_tl_param_cl.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 specific 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_tl1_data_cl.h` and `ocp_globals.h`.

The header ocp_tl1_slave_port.h is the slave port to the OCP specific 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 ocp_tl_param_cl.h header file contains the definition of the parameter class. The configurable slave uses this class to read the channel's configuration and then uses that information to set up its own configuration to match the channel it is connected to.

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 a 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;
```

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 define 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_elaboartion method, the Slave class define a pointer that points to the OCP parameters of the OCP channel that is connected to the model's tpP port:

```
ParamCl<TdataCl>* m_OCPParamP;
```

Also, there is the following function for compatibility with the base generic channel class:

```
bool MputDirect(int, bool, Td*, Ta, int);
```

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

The following is the complete header file for the slave.

```
#ifndef _SIMPLE_SLAVE_H
#define _SIMPLE_SLAVE_H

#include <iostream>
#include <map>
#include "globals.h"

// OCP-IP Channel header files
#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:
    // --------------------------
    // 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;

    // Systemc macros

    // has SystemC processes
    SC_HAS_PROCESS(Slave);

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

    // methods
    void setConfiguration( MapStringType& passedMap );

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

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

  // SystemC methods
  void end_of_elaboration();

  // methods
  bool MputDirect(int, bool, Td*, Ta, int);

  // member definitions

  // slave identification
  int  m_ID;

  // ocp clock information
  double       m_ocpClkPeriod;
  sc_time_unit m_ocpClkTimeUnit;

  // 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
  // ------------------------------------------------------------

  ParamCl<TdataCl>* m_OCPParamP;

  // 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
    // ------------------------------------------------------------

    // should there be a limit to the number of outstanding requests
per
    // thread?
    // default = false;
    bool m_limitreq_enable;

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

    // Response Latency
    int m_Latency;

    MapStringType m_ParamMap;

};

#endif // _SIMPLE_SLAVE_H
```

8.3.2 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 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 following is the code for the constructor.

```
// ------------------------------------------------------------------
--
// constructor
// ------------------------------------------------------------------
--
template<typename TdataCl>
Slave<TdataCl>::Slave(
    sc_module_name n,
    double         ocp_clock_period,
    sc_time_unit   ocp_clock_time_unit,
    int            id,
    Ta             memory_byte_size,
    ostream*       debug_os_ptr
) : sc_module(n),
```

```
        tpP("tpPort"),
        m_ID(id),
        m_ocpClkPeriod(ocp_clock_period),
        m_ocpClkTimeUnit(ocp_clock_time_unit),
        m_MemoryByteSize(memory_byte_size),
        m_Memory(NULL),
        m_debug_os_ptr(debug_os_ptr),
        m_curSThreadBusy(0),
        m_OCPParamP(NULL),
        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_enable(1),
        m_limitreq_max(4),
        m_Latency(0)
    {
        // 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, which uses dynamic sensitive
        SC_THREAD(requestThreadProcess);

        // setup a SystemC thread process, which uses dynamic sensitive
        SC_THREAD(responseThreadProcess);

        // setup a SystemC thread process to check and
        // set sideband signals
        SC_THREAD(exerciseSidebandThreadProcess);
    }
```

8.3.3 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;
}
```

8.3.4 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 slave takes advantage of this to read the OCP parameters of the channel and then to use those parameters to configure itself to work with the channel it was connected to.

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

- The GetParamCl() method returns a pointer that points to the OCP channel's parameters. For example,

  ```
  m_OCPParamP = tpP->GetParamCl();
  ```

  The slave then uses this pointer to extract the channel's parameters and to use them to configure itself. Because the names of the channel parameters match the names in the OCP Specification document, the parameter look-up is one to one. The channel parameters are then stored locally in the core for convenience.

- Sometime before the end of elaboration, the setConfiguration() function was called and the slave's parameters were passed to it using a string to string parameter map. The read this map, the slave uses functions in the ParamCl class that extract integers and Booleans from string formatted parameter maps. The complex looking function call

  ```
  ParamCl<TdataCl>::getBoolOCPConfigValue(myPrefix, paramName,
     m_limitreq_enable, m_ParamMap)
  ```

  returns *true* if the passed parameter map (m_ParamMap) contains a Boolean parameter named by the string "paramName". If the parameter map does contain the parameter, the value of m_limitreq_enable is set to the value of that parameter. The parameter "myPrefix" is generally not used and can be set to "".

- Finally, the slave uses the values of its own parameters and the configuration of the channel to which it is connected to build the memory model that it will use during the simulation.

The following is the complete code for the slave's 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();

    ////////////
    //
    // Process OCP Parameters from the port
    //
    ////////////

    m_OCPParamP = tpP->GetParamCl();

    // Set the number of threads
    m_threads = m_OCPParamP->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 = m_OCPParamP->datahandshake;
        // Is so, quit as this Slave does not handle data handshake.
        assert(!m_OCPParamP->datahandshake);

        // Do writes get reponses?
        m_writeresp_enable = m_OCPParamP->writeresp_enable;

        // is SThreadBusy part of the channel?
        m_sthreadbusy = m_OCPParamP->sthreadbusy;

        // is this slave expected to follow the threadbusy exact
    protocol?
        m_sthreadbusy_exact = m_OCPParamP->sthreadbusy_exact;

        // is MThreadBusy part of the channel?
        m_mthreadbusy = m_OCPParamP->mthreadbusy;

        // is SCmdAccept part of the channel?
        m_cmdaccept = m_OCPParamP->cmdaccept;

        ////////////
        //
        // Process Slave Parameters
        //
        ////////////

        // 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 = m_ParamMap.begin();
                    map_it != m_ParamMap.end(); ++map_it) {
                (*m_debug_os_ptr) << "map[" << map_it->first << "] = "
                        << map_it->second << endl;
            }
            cout << endl;
        }

        // Here the prefix is not needed.
        // the future.
        string myPrefix = "";
        string paramName = "undefined";

        // latency(0), latency(1), ... , latency(n)
        paramName = "latency(0)";
        if (!(ParamCl<TdataCl>::getIntOCPConfigValue(myPrefix,
```

```
                                                    paramName,
                                                    m_Latency,
                                                    m_ParamMap)) ) {
        // Could not find the parameter so we must set it to a
default
#ifdef DEBUG
        cout << "Warning: paramter \"" << paramName
          << "\" for Slave \"" << name()
          << "\" was not found in the parameter map." << endl;
        cout << "         setting missing parameter to 3." << endl;
#endif
        m_Latency = 3;
    }

    // limitreq_enable
    paramName = "limitreq_enable";
    if (!(ParamCl<TdataCl>::getBoolOCPConfigValue(myPrefix,
                                                    paramName,
                                                    m_limitreq_enable,
                                                    m_ParamMap)) ) {
        // Could not find the parameter so we must set it to a
default
#ifdef DEBUG
        cout << "Warning: paramter \"" << paramName
              << "\" for Slave \"" << name()
               << "\" was not found in the parameter map." << endl;
        cout << "         setting missing parameter to false." <<
endl;
#endif
        m_limitreq_enable = false;
    }
    // limitreq_max
    paramName = "limitreq_max";
    if (!(ParamCl<TdataCl>::getIntOCPConfigValue(myPrefix,
                                                    paramName,
                                                    m_limitreq_max,
                                                    m_ParamMap)) ) {
        // Could not find the parameter so we must set it to a
default
#ifdef DEBUG
        cout << "Warning: paramter \"" << paramName
               << "\" for Slave \"" << name()
                 << "\" was not found in the parameter map." <<
endl;
        cout << "         setting missing parameter to 4." << endl;
#endif
        m_limitreq_max = 4;
    }

    ////////////
    //
    // Initialize the Slave with New Parameters
    //
    ////////////

    // Clear the response queue
    m_ResponseQueue.reset();
```

```
    // 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,m_OCPParamP-
>addr_wdth,sizeof(Td));

}
```

8.4 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_DVA`. Some of the responses are just place-holder responses. They are there to make sure that the timing for activities such as writes are accurate. Place-holder 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)

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.
    tpP->ocpWait();

    // 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() +
sc_time(m_Latency,m_ocpClkTimeUnit);

        // purge the queue of any posted write place holder
responses
        // that have reached their send times
        m_ResponseQueue.purgePlaceholders();

        m_ResponseQueue.enqueueBlocking(resp.SResp,resp.SData,
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)) {
            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.
                tpP->ocpWait();
            }
            // now it is okay to accept the request
            tpP->putSCmdAccept();
        }

    }
}
```

8.4.1 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.

The command following command changes the channel's SThreadBusy signal at the next delta cycle:

```
tpP->putSThreadBusy(m_curSThreadBusy);
```

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.

```
mthreadbusy = tpP->getMThreadBusy();
while (mthreadbusy & 1) {
    tpP->ocpWait();
    mthreadbusy = tpP->getMThreadBusy();
}
```

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.

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;
    sc_time          CurTime;
    unsigned int     mthreadbusy;

    tpP->ocpWait();

    // 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();


        // Can we free SThreadBusy??
        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);
        }

        // Get the next request off of the queue

m_ResponseQueue.dequeueBlocking(resp.SResp,resp.SData,send_time);
        resp.SThreadID = selectedThread;

        // check if we still need to wait
        CurTime = sc_time_stamp();
        if (send_time > CurTime) {
            tpP->ocpWait((send_time.value() -
CurTime.value())/1000);
        }

        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) {
                mthreadbusy = tpP->getMThreadBusy();
                while (mthreadbusy & 1) {
                    tpP->ocpWait();
                    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);
        } else {
            assert("Slave should have been able to clear
SThreadBusy");
        }

        // wait until next cycle to send out the next response (if
any)
        tpP->ocpWait();
    }
}
```

8.4.2 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.
    tpP->ocpWait(10);
    int tweakCounter =0;
    bool hasMError = m_OCPParamP->merror;
    bool hasSError = m_OCPParamP->serror;
    bool nextSError = false;
    bool hasSInterrupt = m_OCPParamP->interrupt;
    bool nextSInterrupt = false;
    bool hasSFlag = m_OCPParamP->sflag;
    int numSFlag = m_OCPParamP->sflag_wdth;
    unsigned int nextSFlag = 0;
    unsigned int maxSFlag = (1 << numSFlag) -1;

    // main loop
    while (true) {
        // wait 10 cycles
        tpP->ocpWait(10);

        // 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
}
```

8.4.3 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 >;
```

8.5 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 "master.h"
#include "slave.h"
#include "ocp_tl1_data_cl.h"
#include "ocp_tl_param_cl.h"
#include "ocp_tl1_channel.h"

#define OCP_CLOCK_PERIOD        1
#define OCP_CLOCK_TIME_UNIT     SC_NS

#define MASTER_CLOCK_PERIOD     1
#define MASTER_CLOCK_TIME_UNIT  SC_NS

#define SLAVE_CLOCK_PERIOD      1
#define SLAVE_CLOCK_TIME_UNIT   SC_NS


void process_command_line(int   argc,
                          char* argv[],
                          string& ocp_params_file_name,
                          string& master_params_file_name,
                          string& slave_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 master parameters file name
    master_params_file_name = "";
    if (argc > 2) {
        string file_name(argv[2]);
        master_params_file_name = file_name;
    }
    // get the slave parameters file name
    slave_params_file_name = "";
    if (argc > 3) {
        string file_name(argv[3]);
        slave_params_file_name = file_name;
    }

    // get the simulation end time
    simulation_end_time = 1000;
    if (argc > 4) {
```

```
        simulation_end_time = (double) atoll(argv[4]);
    }

    // do we dump out a log file?
    debug_dump= false;
    debug_file_name = "";
    if (argc > 5) {
        string file_name(argv[5]);
        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< OCP_TL1_DataCl<OCPCHANNELBit32,
OCPCHANNELBit32> >* pOCP;
    Master< OCP_TL1_DataCl<OCPCHANNELBit32, OCPCHANNELBit32> >*
pMaster;
    Slave< OCP_TL1_DataCl<OCPCHANNELBit32, OCPCHANNELBit32> >*
pSlave;
    MapStringType  ocpParamMap;
    MapStringType  masterParamMap;
    MapStringType  slaveParamMap;

    double         simulation_end_time;
    bool           debug_dump;
    string         ocpParamFileName;
    string         masterParamFileName;
    string         slaveParamFileName;
```

```
    string         dump_file_name;
    ofstream       debugFile;

    // --------------------------------
    // (1) process command line options
    //     and read my parameters
    // --------------------------------

process_command_line(argc,argv,ocpParamFileName,masterParamFileName,

slaveParamFileName,simulation_end_time,debug_dump,dump_file_name);

    if ( ! ocpParamFileName.empty() ) {
        readMapFromFile(ocpParamFileName, ocpParamMap);
    }

    if ( ! masterParamFileName.empty() ) {
        readMapFromFile(masterParamFileName, masterParamMap);
    }

    if ( ! slaveParamFileName.empty() ) {
        readMapFromFile(slaveParamFileName, slaveParamMap);
    }

    // open a trace file
    if (debug_dump) {
        cout << "Debug dumpfilename: " << dump_file_name << endl;
        debugFile.open(dump_file_name.c_str());
    }

    // ------------------------------------------------------------
    // (2) Create the self-timed OCP Channel
    // ------------------------------------------------------------

    pOCP = new OCP_TL1_Channel< OCP_TL1_DataCl<OCPCHANNELBit32,
               OCPCHANNELBit32> >

("ocp0",true,true,true,NULL,OCP_CLOCK_PERIOD,OCP_CLOCK_TIME_UNIT,"oc
p0.ocp");

    //   Alternatively, use a clocked channel
    //sc_clock clk("clk", OCP_CLOCK_PERIOD,OCP_CLOCK_TIME_UNIT);
    //pOCP = new OCP_TL1_Channel< OCP_TL1_DataCl<OCPCHANNELBit32,
    //           OCPCHANNELBit32> >
    //("ocp0", (sc_clock *)&clk, (std::string)"ocp0.ocp");

    pOCP->setConfiguration(ocpParamMap);


    // ------------------------------------------------------------
    // (3) Create the Master and Slave
    // ------------------------------------------------------------

    pMaster = new Master< OCP_TL1_DataCl<OCPCHANNELBit32,
OCPCHANNELBit32> >("master", MASTER_CLOCK_PERIOD,
MASTER_CLOCK_TIME_UNIT, 0, &debugFile );
```

```
    pSlave = new Slave< OCP_TL1_DataCl<OCPCHANNELBit32,
OCPCHANNELBit32> >("slave", SLAVE_CLOCK_PERIOD,
SLAVE_CLOCK_TIME_UNIT, 0, 0x3FF, &debugFile );

    // ------------------------------------------------------------
    // (4) connect channel, master, and slave, & clock
    // ------------------------------------------------------------
    pMaster->ipP(*pOCP);
    pSlave->tpP(*pOCP);

    // -------------------------
    // (5) start the simulation
    // -------------------------
    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);
}
```

9 Examples Using Original OCP Specific TL2 Channel and API

The examples described in this section demonstrate the use of the OCP specific TL2 channel. The first example illustrates a single-threaded OCP communication between an OCP master and an OCP slave. Both are using the TL2 specific 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`'. A `README` file details how to compile and run the code.

9.1 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 specific TL2 master port to connect the channel, and the slave uses an OCP specific 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:

9.1.1 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

9.1.2 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.

9.2 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.

9.2.1 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).

9.2.2 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.

10 Debugging Your Model Using SOCCREATOR® Tools

The main debugging tool available for the OCP channel model is the OCP monitor output. The OCP monitor is activated by passing a file name for the OCP monitor output when the channel is constructed. (See section 4.1 for more details about the channel constructor.) If the OCP monitor is used, the channel will print out its current state 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.

The OCP Monitor can also be instantiated as a separate component:

OCPMon< DataClass > m1("m1", &ch0, (std::string)"ocp0.ocp", &clk);

The OCP Monitor is available to OCP-IP members in a separate release package. The release package for the OCP channel 2.0.2 does not contain the monitor files.

11 Sideband Signals

The access methods for sending and receiving sideband signals are shared by both the base generic class API and the OCP TL1 specific API. The commands described in this section may be used with either API.

11.1 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.

11.2 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.

11.3 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.

11.4 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.

11.5 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.

11.6 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.

11.7 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.

11.8 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.

11.9 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.

11.10 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.

11.11 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.