OCP Layer Adapters

V2.0 –January 23, 2004

Stéphane Guntz, Prosilog

Yann Bajot, Prosilog

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

DISCLAIMER

Table of contents:

Table of figure :

1. INTRODUCTION

Layer adapters presented here are intended to illustrate the TL0-TL1 and TL1-TL2 layer adaptation processes for the OCP-based transaction level communication interface. They are based on the OCP2.0 specification and internally use the OCP-specific TL2/TL1 APIs, documented in the .pdf document 'OCP_TL_Channel_v2.0.pdf' included in the OCP channel package.

Instructions to compile and run the examples can be found in the 'README' file of the root directory.

1.1. TL0/TL1 adapters

These adapters are given as examples and implement only a subset of the whole OCP 2.0 dataflow interface. However, an effort was made here to address the main problems encountered during the layer adaptation process. The solutions proposed to solve these problems are explained and can be reused to extend the adapters to support more OCP features (see section 4). Note that we only consider the dataflow part of the OCP interface here (no sideband signals).

TL0-TL1 adapters are essential to connect TL1 systems to the TL0 (RTL) world: synthesizable SystemC, or VHDL/Verilog. Two kind of adapters are necessary to connect TL1 systems to TL0 slaves or TL0 masters.

Main issues concern how to manage signal glitches and propagation delays, which are very common for TL0 models. Signals coming from TL0 parts are not supposed to be clocked and therefore can change late in the clock cycle. It is also possible that some glitches appear on them, especially in the case of co-simulation with a HDL-simulated RTL Slave.

Once an interface method call has been done to a TL1 channel, it cannot be canceled anymore. Hence, adapters must issue only one TL1 call per cycle, at the right time when TL0 signals are supposed to be stable. Note that co-simulation with HDL-simulated modules is not addressed here, and requires the use of simulator-specific APIs such as FLI and PLI.

IMPORTANT NOTE : All the adapters presented here use events of the communication class to enable synchronization. As a result, TL1 channels connected to the adapters port MUST have their 'syncevent' constructor parameter set.

1.2. TL1/TL2 adapters

Layer adapters presented here are intended to illustrate the TL1-TL2 layer adaptation process for the OCP-based transaction level communication interface.
These adapters are given as examples and implement only a subset of the whole OCP 2.0 dataflow interface. Note that we only consider the dataflow part of the OCP interface here (no sideband signals).

2. TL0 SLAVE ADAPTERS

These adapters allow to connect a TL0 Slave to a TL1 system. In this section, we will present two examples of adapters that implement two different subsets of the OCP interface:

- Basic adapter: All the *Basic* signals (*Clock*, *MAddr*, *MCmd*, *MData*, *SCmdAccept*, *SData*, *SResp*) + *MRespAccept*
- Basic adapter with datahandshake: All the *Basic* signals + *MRespAccept* + *Datahandshake* signals (*MDataValid* + *SDataAccept*)

Four examples are provided (three for the basic adapter and one for the datahandshake adapter), which present the use of the slave adapters with different kinds of TL0 slaves and TL1 masters.

2.1. Basic TL0 Slave adapter

The code for this adapter can be found in files *layer_adapters/tl0_tl1/(include|src)/ocp_tl0_tl1_slave_adapter*(.h,cpp). During an OCP transaction, the adapter should behave as follows:

- For the request phase, the adapter catches each sent request on the TL1 Slave Port, extracts the TL1 data fields and drives the corresponding TL0 Request group signals. Then, it waits for '*SCmdAccept'* to be asserted by the slave and releases the request on TL1 channel.
- For the response phase, the adapter samples Response group signals driven by the TL0 Slave, test the '*SResp'* signal to identify a valid response and conditionally put it on the TL1 channel. Then, it waits for the TL1 Master to release the response and asserts the 'MrespAccept' signal during one cycle.

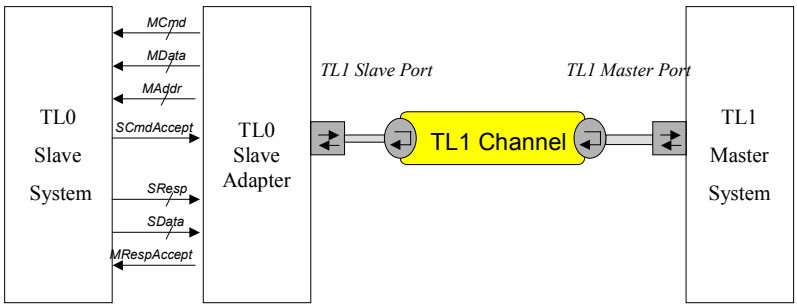


Figure 1: TL0 Slave Adapter

The adapter has to deal with potential late changes and glitches on its TL0 input signals. For a TL0 Slave, it concerns the following signals:

- *SCmdAccept*
- *SResp* and *SData* signals

The slave adapter issues some TL1 calls that can potentially change the state of the channel:

- *putSCmdAccept()* calls to release the request sent by the TL1 system
- *startOCPResponse()* calls to send response to the TL1 system

Once one of these calls has be done, the state channel has changed and there is no way to return to the previous state. Hence, it is essential to be sure that the TL0 signal values driving the TL1 calls are the good ones, i.e. they will not change until the end of the cycle.

The *'startOCPResponse()*' and associated API calls must happen only when TL0 associated signals *'SResp'* and *'SData'* are stable. Since the propagation delays for these signals are slave dependent, a *'Response Sample Delay'* parameter is used to determine the precise instant when these signals should be sampled during the cycle (cf. Figure 2). Once sampled, '*SResp*' is tested; if the response is valid (DVA value), a 'startOCPResponse()' call is done.

In the same manner; the *'putSCmdAccept()*' call must happen only when '*SCmdAccept*' is stable. Since the propagation delay for this signal is slave dependent, a '*CmdAcceptSampleDelay*' parameter is used to determine when '*SCmdAccept*' should be sampled during the cycle (cf. Figure 2).

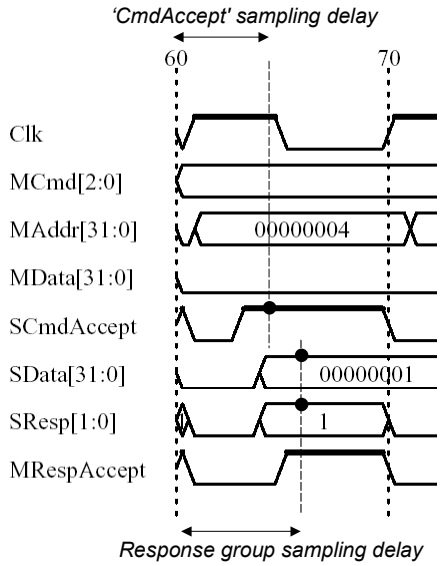


Figure 2: Slave Adapter sample delays

2.1.1. Default constructor

This constructor give defaults values to sampling delays. These values are compatible with the OCP Level 2 timing guidelines.

OCP_TL0_TL1_Slave_Adapter(sc_module_name name,
, int ID
, bool Combinational
, sc_time ClockCycle
, bool Check_setup_time = true
)

- *name*: Name of the module (slave adapter) instance.
- *ID*: Unique number identifying the Slave.
- *Combinational*: Boolean specifying if there is a combinational path between *Request* and *Response* signals. Sampling times will be set as follows:
  Combitional:
  * RequestSampleDelay = ClockCycle*99/100
  * ResponseSampleDelay = ClockCycle*60/100
  - Sequential:
  * RequestSampleDelay = ClockCycle*99/100
  * ResponseSampleDelay = ClockCycle*60/100
- *ClockCycle*: sc_time number specifying the OCP clock period
- *Check_setup_time*: boolean value specifying if '*check_setup_time()*' function is executed during simulation (see below).

'*Check_setup_time()*' function is proposed for debugging purposes, to ensure that TL0 slave output signals do not change after the sample delays, which would cause the adapter to fail. A good way to proceed is to perform the first simulation with checking function activated to determine the appropriate

sample delay value. Once TL0 behaviour is correct, checking function should be disabled to increase simulation speed.

2.1.2. Explicit timings constructor

This constructor can be used to explicitly specify sampling times, when default values do not match the timing requirements of the slave.

OCP_TL0_TL1_Slave_Adapter(sc_module_name name,
, int ID
, sc_time CmdAcceptSampleDelay
, sc_time ResponseSampleDelay
, bool Check_setup_time = true
)

- *name*: Name of the module (slave adapter) instance.
- *ID*: Unique number identifying the Slave.
- *CmdAcceptSampleDelay*: Time number specifying the delay between the clock rising edge and the actual sampling time for '*SCmdAccept*' signal.
- *ResponseSampleDelay*: Time number specifying the delay between the clock rising edge and the actual sampling time for response group signals.
- *Check_setup_time*: boolean value specifying if '*check_setup_time()*' function is executed during simulation (see below).

The smallest value for the sample delays is sc_get_time_resolution() ;it should be used in the case when TL0 master has pure 'zero-delay' signals. Users should always use a timed sample delay (and not a defined number of delta-cycles) since it is not possible to predict the number of delta-cycles required for signal stability.

2.2. Basic TL0 Slave adapter with Datahandshake

The code for this adapter can be found in files *layer_adapters/tl0_tl1/(include|src)/ocp_tl0_tl1_slave_adapter_hs*(.h,cpp). During an OCP transaction, the adapter should behave as follows:

- For the request phase, the adapter catches each sent request on the the TL1 Slave Port, extracts the TL1 data fields and drives the corresponding TL0 Request group signals. Then, it waits for 'SCmdAccept' to be asserted by the slave and releases the request on TL1 channel.
- For the datahandshake phase, the adapter catches each sent data on the the TL1 Slave Port, extracts the TL1 data fields and drives the corresponding TL0 Data group signals. Then, it waits for 'SDataAccept' to be asserted by the slave and releases the data on TL1 channel.
- For the response phase, the adapter samples Response group signals driven by the TL0 Slave, test the '*SResp'* signal to identify a valid response and conditionally put it on the TL1 channel. Then, it waits for the TL1 Master to release the response and asserts the 'MrespAccept' signal during one cycle.

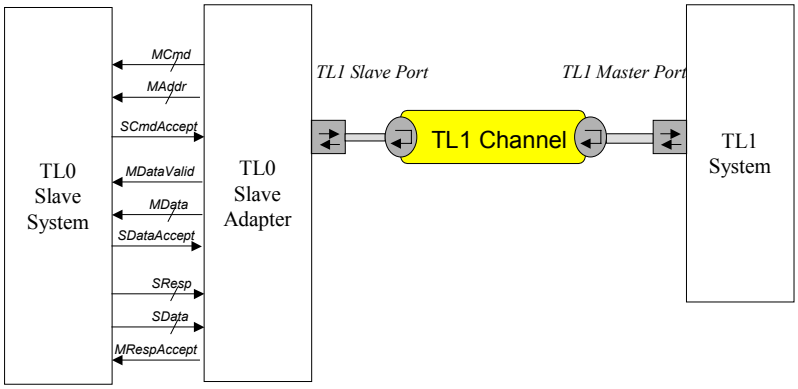


Figure 3: TL0 Slave Adapter with Datahandshake

The adapter has to deal with potential late changes and glitches on its TL0 input signals. For a TL0 Slave, it concerns the following signals:

- *SCmdAccept*
- *SDataAccept*
- *SResp* and *SData* signals

The slave adapter issues some TL1 calls that can potentially change the state of the channel:

- *putSCmdAccept()* calls to release the request sent by the TL1 system
- *putSDataAccept()* calls to release the data sent by the TL1 system
- *startOCPResponse()* calls to send response to the TL1 system

Once one of these calls has be done, the state channel has changed and there is no way to return to the previous state. Hence, it is essential to be sure that the TL0 signal values driving the TL1 calls are the good ones, i.e. they will not change until the end of the cycle.

The *'startOCPResponse()*' and associated API calls must happen only when TL0 associated signals *'SResp'* and *'SData'* are stable. Since the propagation delays for these signals are slave dependent, a *'Response Sample Delay'* parameter is used to determine the precise instant when these signals should be sampled during the cycle (cf. Figure 2). Once sampled, '*SResp*' is tested; if the response is valid (DVA value), a *'startOCPResponse*()' calls is done.

In the same manner; the *'putSCmdAccept()*' call must happen only when '*SCmdAccept*' is stable. Similarly, '*putSDataAccept*()' call must wait 'SDataAccept' to be stable. Since the propagation delay for these signals are slave dependent, a single '*AcceptSampleDelay*' parameter is used to determine when both '*SCmdAccept*' AND '*SDataAccept'* should be sampled during the cycle (cf Figure 4).

Note that we use a single sample delay for both signals for the following reasons:

- There is no combinational path between '*SCmdAccept'* and '*SDataAccept'*, i.e there is only one signal to be sampled at each cycle.
- The OCP combinational timings guidelines (see below 2.3.3) specify that the output delay constraints are identical for both signals.

If output delays are different for each signal, '*AcceptSampleDelay*' should be equal to the greatest value (see example 4, 2.3.4).

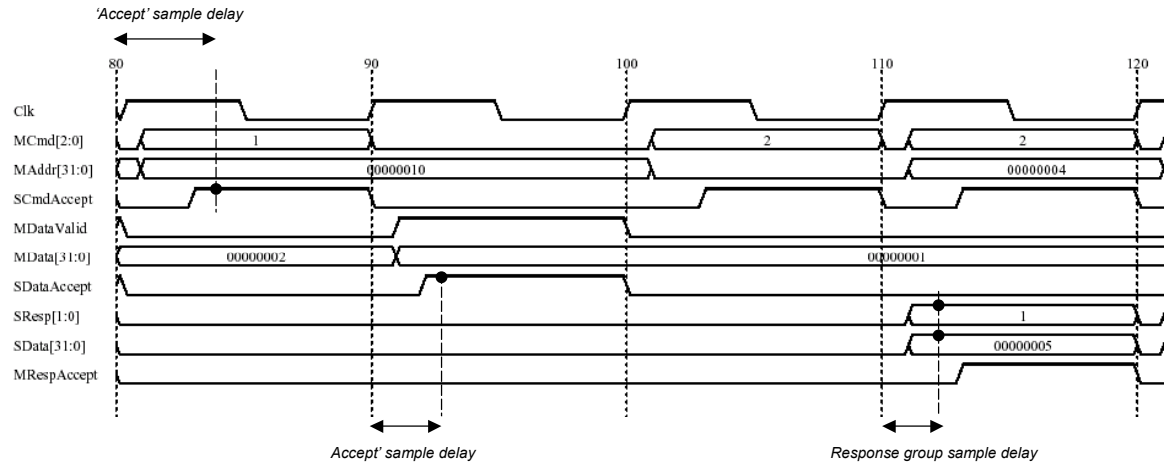


Figure 4: Slave Adapter sample delays (datahandshake)

2.2.1. Default constructor

This constructor give defaults values to sampling delays. These values are compatible with the OCP Level 2 timing guidelines.

OCP_TL0_TL1_Slave_Adapter_HS(sc_module_name name,
, int ID
, bool Combinational
, sc_time ClockCycle
, bool Check_setup_time = true
)

- *name*: Name of the module (slave adapter) instance.
- *ID*: Unique number identifying the Slave.
- *Combinational*: Boolean specifying if there is a combinational path between *DataHandshake* and *Response* signals. Sampling times will be set as follows:
  Combitional:
  * AcceptSampleDelay = ClockCycle*99/100
  * ResponseSampleDelay = ClockCycle*60/100
  - Sequential:
  * AcceptSampleDelay = ClockCycle*99/100
  * ResponseSampleDelay = ClockCycle*60/100
- *ClockCycle*: sc_time number specifying the OCP clock period
- *Check_setup_time*: boolean value specifying if '*check_setup_time()*' function is executed during simulation (see below).

'*Check_setup_time()*' function is proposed for debugging purposes, to ensure that TL0 slave output signals do not change after the sample delays, which would cause the adapter to fail. A good way to proceed is to perform the first simulation with checking function activated to determine the appropriate sample delay value. Once TL0 behaviour is correct, checking function should be disabled to increase simulation speed.

2.2.2. Explicit timings constructor

This constructor can be used to explicitly specify sampling times, when default values do not match the timing requirements of the slave.

```
OCP_TL0_TL1_Slave_Adapter_HS(sc_module_name name,
                , int ID
                , sc_time AcceptSampleDelay
                , sc_time ResponseSampleDelay
                , bool Check_setup_time = true
                )
```

- *name*: Name of the module (slave adapter) instance.
- *ID*: Unique number identifying the Slave.
- *AcceptSampleDelay*: Time number specifying the delay between the clock rising edge and the actual sampling time for '*SCmdAccept*' and '*SDataAccept'* signals.
- *ResponseSampleDelay*: Time number specifying the delay between the clock rising edge and the actual sampling time for response group signals.
- *Check_setup_time*: boolean value specifying if '*check_setup_time()*' function is executed during simulation (see below).

2.3. Examples

The following four examples (three for the basic adapter and one for the handshake adapter) present the use of the slave adapters in different system environments:

- Fully combinational transaction (Example 1).
- Synchronous split transaction with combinational '*Accept'* signals (Example 2).
- Synchronous split transaction with synchronous '*Accept'* signals (Example 3).
- Synchronous split transaction with combinational '*Accept'* signals and datahandshake management (Example 4).

For each example, we discuss about the correct values to give to the 'SampleDelay' parameters in order to match the TL0 slave output delays.

2.3.1. Fully combinational transaction (Example 1)

In this example, the Basic adapter connects a TL0 asynchronous slave to a TL1 asynchronous master. The top level C++ file is called top_slave_async.cpp. The example master and slave files are ocp_tl0_slave_async.cpp, ocp_tl1_master_async.cpp. Use the Makefile to build the example, and run 'run_slave_async' executable. Waveforms for OCP TL0 signals can be viewed using the generated VCD file 'top_slave_async.vcd'.

This example presents a fully combinational OCP connection, where requests and responses occur in the same clock cycle. The master issues 5 writes followed by 5 reads, and checks received data, assuming that the OCP Target acts as a RAM. Operation timings are described in the header of the top_slave_async.cpp file.

Setting 'CmdAcceptSampleDelay' parameter:
The '*SCmdAccept'* driving path is the following:

Clk (t=***0***) -> TL1_Master_MputRequest() (t=***T1***) -> TL0 ScmdAccept (t=***T1+T2***)

where :

- T1 specifies the delay (starting from the rising edge of the clock) used by the TL1 Master before issuing a request to the TL1 channel.
- T2 specifies the internal combinational delay of the TL0 slave between Request group input signals and the 'SCmdAccept' signal. This feature is slave-dependent

For this kind of combinational path on 'SCmdAccept', parameter '***CmdAcceptSampleDelay***' must be **greater than T1 + T2.**

Setting 'ResponseSampleDelay' parameter:
The response group signals driving path is the following:

Clk (t=***0***) -> TL1_Master_MputRequest() (t=***T1***) -> TL0 Response group (t=***T1+T3***)

where :

- T1 specifies the delay (starting from the rising edge of the clock) used by the TL1 Master before issuing a request to the TL1 channel.
- T3 specifies the internal combinational delay of the TL0 slave between Request group input signals and Response group output signals. This feature is slave-dependent.

For this kind of combinational path on response signals, parameter '***ResponseSampleDelay***' must be **greater than T1 +T3.**

2.3.2. Synchronous split transaction with combinational 'Accept' signals (Example 2)

In this example, the Basic adapter connects a TL0 synchronous slave to a TL1 master. The top level C++ file is called top_slave_sync.cpp. The example master and slave files are ocp_tl0_slave_sync.cpp, ocp_tl1_master_async.cpp. Use the Makefile to build the example, and run 'run_slave_sync' executable. Waveforms for OCP TL0 signals can be viewed using the generated VCD file 'top_slave_sync.vcd'.

In this example, responses occurs one cycle after requests, and *'Accept'* signals use combinational paths. The master issues 5 writes followed by 5 reads, and checks received data , assuming that the OCP Target acts as a RAM. Operation timings are described in the header of the top_slave_sync.cpp file.

Setting 'CmdAcceptSampleDelay' parameter:
The value is the same as for example 1: ***'CmdAcceptSampleDelay'*** must be **greater than T1 + T2**

Setting 'ResponseSampleDelay' parameter:
The response group signals driving path is the following:

Clk (t=***0***) -> Response group (t=***T3***)

where :

- T3 specifies the internal combinational delay of the TL0 slave between the rising edge of the clock and the response group assertion time. This feature is slave-dependent.

For this kind of combinational path on response signals, parameter '***ResponseSampleDelay***' must be **greater than T3.**

NOTE: OCP combinational timings guidelines
In this example, the TL1 master has its *'MCmd'* propagation delays equal to zero, and uses a split, synchronous request/response scheme. As a result, delays seen by the adapter on Response signals

and *'SCmdAccept'* are equal to the real output delays of the TL0 Slave. Hence, it becomes possible to check the compliance of the TL0 Slave to the OCP timings guideline (specified in the *OCP Specification Release 2.0*, Chapter 10).
By setting the sample delay values according to the Level-1 and Level-2 timing guidelines, we can verify that the TL0 Slave is only Level-2 compliant.

More generally, it is possible to check the compliance of a TL0 Slave/Master when the following conditions are fulfilled:

- TL1 system connected to the TL1 channel issues interface channel calls only at clock rising edges. In practical terms, the TL1 system must only use clocked process to issue TL1 calls and can not include any 'wait()' calls in the calling processes. In this way, the TL1 delays resulting from '*wait()*' statements do not have an impact on final TL0 output delays.
- There is no combinational path between the Master/Slave and the corresponding adapter. In this way, the 'Sample Delays' of the adapters do not have an impact on final TL0 output delays.

2.3.3. Synchronous split transaction with synchronous 'Accept' signals (Example 3)

In this example, the Basic adapter connects a TL0 synchronous slave to a TL1 master. The top level C++ file is called top_slave_sync2.cpp. The example master and slave files are ocp_tl0_slave_sync2.cpp, ocp_tl1_master_async.cpp. Use the Makefile to build the example, and run 'run_slave_sync2' executable. Waveforms for OCP TL0 signals can be viewed using the generated VCD file 'top_slave_sync2.vcd'.

In this example, responses occurs one cycle after requests, and 'Accept' signals assertion take one cycle (+ some propagation delays). The master issues 5 writes followed by 5 reads, and checks received data , assuming that the OCP Target acts as a RAM. Operation timings are described in the header of the top_slave_sync2.cpp file.

Setting 'CmdAcceptSampleDelay' parameter:
The 'SCmdAccept' driving path is the following:

Clk (t=***0***) -> TL0 ScmdAccept (t=***T2***)

where :

- T2 specifies the internal combinational delay of the TL0 slave between the rising edge of the clock and the 'SCmdAccept' assertion time. This feature is slave-dependent.

For this kind of combinational path on response signals, parameter '***ResponseSampleDelay***' must be **greater than T2.**

Setting 'ResponseSampleDelay' parameter:
The value is the same as for example 2: '***ResponseSampleDelay***' **greater than T3**

2.3.4. Synchronous split transaction with combinational 'Accept' signals and datahandshake management (Example 4)

In this example, we use the second slave adapter (Basic signals + datahandshake) to connect a TL0 synchronous slave with datahandshake support to a TL1 master. The top level C++ file is called top_slave_sync_hs.cpp. The example master and slave files are ocp_tl0_slave_sync_hs.cpp, ocp_tl1_master_async_hs.cpp. Use the Makefile to build the example, and run 'run_slave_sync_hs' executable. Waveforms for OCP TL0 signals can be viewed using the generated VCD file 'top_slave_sync_hs.vcd'.

In this example, responses occurs one cycle after requests, data are transmitted one cycle after corresponding requests (datahandshake), and both 'SCmdAccept' and 'SdataAccept' signals use

combinational paths. The master issues 5 writes followed by 5 reads, and checks received data , assuming that the OCP Target acts as a RAM. Operation timings are described in the header of the top_slave_sync_hs.cpp file.

Setting 'AcceptSampleDelay' parameter:
The '*SCmdAccept'* driving path is the following:

Clk (t=***0***) -> TL1_Master_MputRequest() (t=***T1***) -> TL0 ScmdAccept (t=***T1+T2***)

where :

- T1 specifies the delay (starting from the rising edge of the clock) used by the TL1 Master before issuing a request to the TL1 channel.
- T2 specifies the internal combinational delay of the TL0 slave between Request group input signals and the 'SCmdAccept' signal. This feature is slave-dependent

The '*SDataAccept'* driving path is the following:

Clk (t=***0***) -> TL1_Master_MputDataRequest() (t=***T3***) -> TL0 ScmdAccept (t=***T3+T4***)

where :

- T3 specifies the delay (starting from the rising edge of the clock) used by the TL1 Master before issuing a data request to the TL1 channel.
- T4 specifies the internal combinational delay of the TL0 slave between Data Request group input signals and the 'SDataAccept' signal. This feature is slave-dependent

Since the same delay is used to sample both 'SCmdAccept' and 'SdataAccept' signals, the ***'AcceptSampleDealy'*** has to be set to the maximum of these two values: ***max(T1+T2, T3+T4)***.

Setting 'ResponseSampleDelay' parameter:
The value is the same as for example 2: '***ResponseSampleDelay***' **greater than T5'**
where :

- T5 specifies the internal combinational delay of the TL0 slave between the rising edge of the clock and the response group assertion time. This feature is slave-dependent.

3. TL0 MASTER ADAPTER

These adapters enable to connect a TL0 Master to a TL1 system. In this section, we will present two examples of adapters that implement two different subsets of the OCP interface:

- Basic adapter: All the *Basic* signals (*Clock*, *MAddr*, *MCmd*, *Mdata*, *ScmdAccept*, *Sdata*, *Sresp*) + *MRespAccept*
- Basic adapter with datahandshake: All the *Basic* signals + *MRespAccept* + *Datahandshake* signals (*MdataVaild* + *SDataAccept*)

Four examples are provided (three for the basic adapter and one for the datahandshake adapter), which present the use of the master adapters with different kinds of TL0 masters and TL1 slaves.

3.1. Basic TL0 Master adapter

The code for this adapter can be found in files *layer_adapters/tl0_tl1/(include|src)/ocp_tl0_tl1_master_adapter*(.h,cpp). During an OCP transaction, the adapter should behave as follows:

- For the request phase, the adapter samples all the TL0 request group signals driven by the TL0 master, test the '*MCmd'* signal to identify a valid request and conditionally put a request on the

TL1 channel. Then, it waits for the TL1 Slave to release the request and asserts the '*SCmdAccept'* signal during one cycle.

- For the response phase, the adapter catches each sent response on its TL1 Master Port, extracts the TL1 data fields and drives the corresponding TL0 response group signals. Then, it waits for '*MRespAccept'* to be asserted by the master and releases the response on TL1 channel.

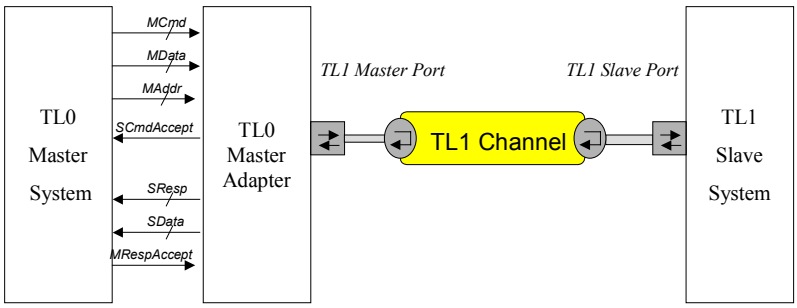


Figure 5: TL0 Master Adapter

The adapter has to deal with potential late changes and glitches on its TL0 input signals. For a TL0 master, it concerns the following signals:

- *MCmd, MData* and *MAddr*
- *MRespAccept*

The master adapter issues some TL1 calls that can potentially change the state of the channel:

- *startOCPRequest()* calls to send a request to the TL1 system
- *putMRespAccept()* calls to release responses sent by the TL1 system

Once one of these calls has been done, the state channel has changed and there is no way to return to the previous state. Hence, it is essential to be sure that the TL0 signal values driving the TL1 calls are the good ones, i.e. they will not change until the end of the cycle.

The *'startOCPRequest()*' and associated data class calls must only happen when TL0 associated signals *'MCmd'*, *'MData'* and *'MAddr'* are stable. Since the propagation delays for these signals are master dependent, a *'RequestSampleDelay'* parameter is used to determine when these signals should be sampled during the cycle. Once sampled, *'MCmd'* is tested and corresponding *'startOCPRequest'* call is issued to the channel.

The *'putMRespAccept ()*' call must happen only when *'MRespAccept'* is stable. Since the propagation delay for this signal is master dependent, a *'RespAcceptSampleDelay'* parameter is used to determine when *'MRespAccept'* should be sampled during the cycle (cf. Figure 6).

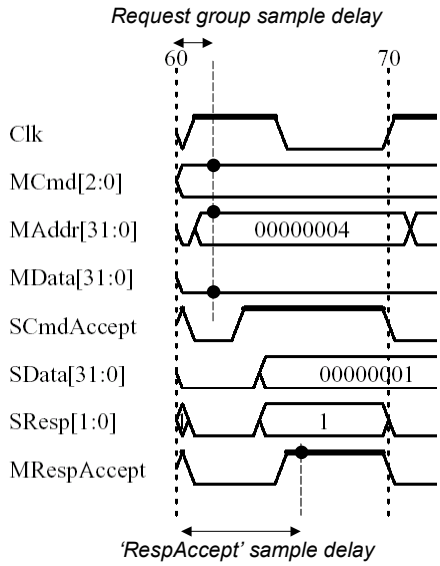


Figure 6: Master Adapter sample delays

3.1.1. Default constructor

This constructor give defaults values to sampling delays. These values are compatible with the OCP Level 2 timing guidelines.

OCP_TL0_TL1_Master_Adapter(sc_module_name name,
, int ID
, int Priority
, bool Combinational
, sc_time ClockCycle
, bool Check_setup_time = true
)

- *name*: Name of the module (slave adapter) instance.
- *ID*: Unique number identifying the Master.
- *Priority*: Positive number specifying the priority relative to the other masters for bus access. Higher numbers means higher priority. Masters can have the same priority.
- *Combinational*: Boolean specifying if there is a combinational path between *Request* and *Response* signals. Sampling times will be set as follows:
  Combitional:
  * RequestSampleDelay = ClockCycle*60/100
  * ResponseSampleDelay = ClockCycle*75/100
  - Sequential:
  * RequestSampleDelay = ClockCycle*60/100
  * ResponseSampleDelay = ClockCycle*75/100
- *ClockCycle*: sc_time number specifying the OCP clock period
- *Check_setup_time*: boolean value specifying if '*check_setup_time()*' function is executed during simulation (see below).

'*Check_setup_time()*' function is proposed for debugging purposes, to ensure that TL0 master output signals don't change after the sample delays, which would cause the adapter to fail. A good way to proceed is to perform the first simulation with checking function activated to determine the appropriate sample delay value. Once TL0 behaviour is correct, checking function should be disabled to increase simulation speed.

3.1.2. Explicit timings constructor

This constructor can be used to explicitly specify sampling times, when default values do not match the timing requirements of the master.

```
OCP_TL0_TL1_Master_Adapter(sc_module_name name,
                , int ID
                , int Priority
                , sc_time RequestSampleDelay
                , sc_time RespAcceptSampleDelay
                , bool Check_setup_time = true
                )
```

- *name*: Name of the module (slave adapter) instance.
- *ID*: Unique number identifying the Master.
- *Priority*: Positive number specifying the priority relative to the other masters for bus access. Higher numbers means higher priority. Masters can have the same priority.
- *RequestSampleDelay*: Time number specifying the delay between the clock rising edge and the actual sampling time for TL0 request group signals.
- *RespAcceptSampleDelay*: Time number specifying the delay between the clock rising edge and the actual sampling time for '*MrespAccept'* signal.
- *Check_setup_time*: boolean value specifying if '*check_setup_time()*' function is executed during simulation (see below).

The smallest value for the sample delays is sc_get_time_resolution() ; it should be used in the case when TL0 master has pure 'zero-delay' signals. Users should always use a timed sample delay (and not a defined number of delta-cycles) since it is not possible to predict the number of delta-cycles required for signal stability.

3.2. Basic TL0 Master adapter with Datahandshake

The code for this adapter can be found in files *layer_adapters/tl0_tl1/(include|src)/ocp_tl0_tl1_master_adapter_hs*(.h,cpp). During an OCP transaction, the adapter should behave as follows:

- For the request phase, the adapter samples Request group signals driven by the TL0 master, tests the '*MCmd'* signal to identify a valid request and conditionally puts it on the TL1 channel. Then, it waits for the TL1 Slave to release the request and asserts the 'SCmdAccept' signal during one cycle.
- For the datahandshake phase, the adapter samples Data Request group signals driven by the TL0 master, tests the '*MDataValid'* signal to identify a valid data request and conditionally puts it on the TL1 channel. Then, it waits for the TL1 Slave to release the data request and asserts the '*SDataAccept'* signal during one cycle.

- For the response phase, the adapter catches each sent request on its TL1 Master Port, extracts the TL1 data fields and drives the corresponding TL0 Response group signals. Then, it waits for '*MRespAccept'* to be asserted by the master and releases the response on TL1 channel.

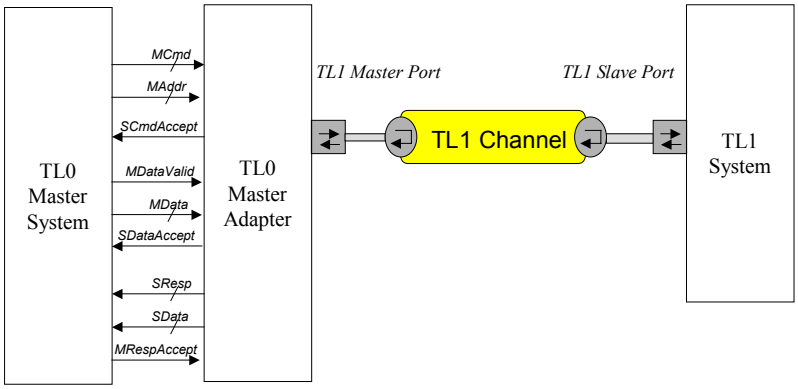


Figure 7: TL0 Master Adapter with Datahandshake

The adapter has to deal with potential late changes and glitches on its TL0 input signals. For a TL0 Master, it concerns the following signals:

- *MCmd, Mdata*
- *MDataValid, MData*
- *MRespAccept*

The master adapter issues some TL1 calls that can potentially change the state of the channel:

- *startOCPRequest()* calls to send a request to the TL1 system
- *startOCPDataHS()* calls to send data to the TL1 system
- *putMRespAccept()* calls to release responses sent by the TL1 system

Once one of these calls has been done, the state channel has changed and there is no way to return to the previous state. Hence, it is essential to be sure that the TL0 signal values driving the TL1 calls are the good ones, i.e. they will not change until the end of the cycle.

The *'startOCPRequest()*' and associated API calls must happen only when TL0 associated signals *'MCmd'* and *'MAddr'* are stable. Similarly, *'startOCPDataHS()'* call must wait '*MDataValid'* and '*MData'* to be stable. Since the propagation delays for these signals are master dependent, we use a *'RequestSampleDelay'* parameter to determine when these signals should be sampled during the cycle.

In the same way; the *'putMRespAccept()*' call must happen only when *'MRespAccept'* is stable. Since the propagation delay for this signal is master dependent, a *'RespAcceptSampleDelay'* parameter is used to determine when *'MRespAccept'* should be sampled during the cycle (cf Figure 8).

Note that we use a single sample delay for request and datahandshake for the following reasons:

- There is no combinational path between request group and datahandshake group signals, i.e there is only one group to be sampled at each cycle.
- The OCP combinational timings guidelines (see 2.3.3) specifies that the output delay constraints are identical for both groups.

In the case where output delays are different for each group, '*AcceptSampleDelay*' should be equal to the greatest value (see example 4, 3.3.4).

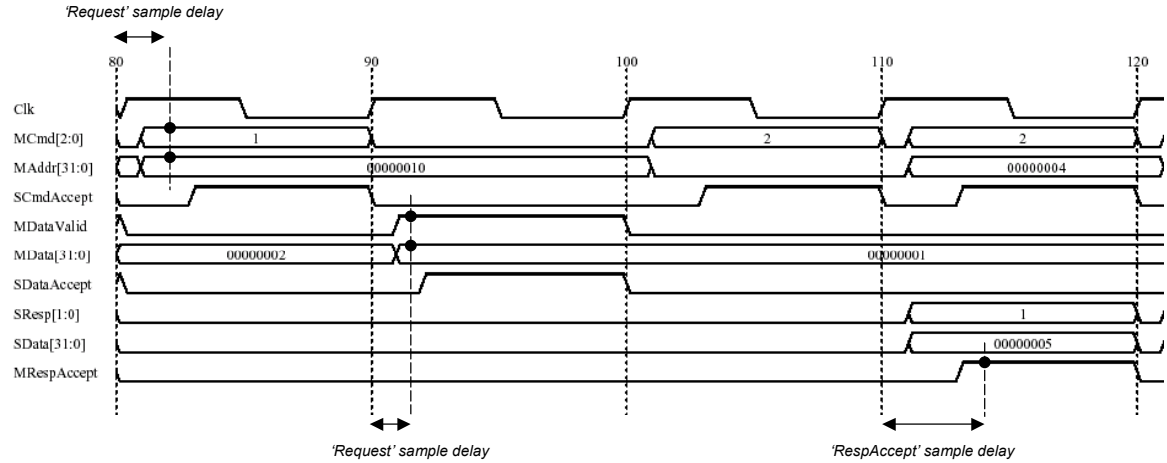


Figure 8: Master Adapter sample delays (datahandshake)

3.2.1. Default constructor

This constructor give defaults values to sampling delays. These values are compatible with the OCP Level 2 timing guidelines.

OCP_TL0_TL1_Master_Adapter_HS(sc_module_name name,
, int ID
, int Priority
, bool Combinational
, sc_time ClockCycle
, bool Check_setup_time = true
)

- *name*: Name of the module (slave adapter) instance.
- *ID*: Unique number identifying the Master.
- *Priority*: Positive number specifying the priority relative to the other masters for bus access. Higher numbers means higher priority. Masters can have the same priority.
- *Combinational*: Boolean specifying if there is a combinational path between *DataHandshake* and *Response* signals. Sampling times will be set as follows:
  Combitional:
  * RequestSampleDelay = ClockCycle*60/100
  * RespAcceptSampleDelay = ClockCycle*75/100
  - Sequential:
  * RequestSampleDelay = ClockCycle*60/100
  * RespAcceptSampleDelay = ClockCycle*75/100
- *ClockCycle*: sc_time number specifying the OCP clock period
- *Check_setup_time*: boolean value specifying if '*check_setup_time()*' function is executed during simulation (see below).

'*Check_setup_time()*' function is proposed for debugging purposes, to ensure that TL0 master output signals don't change after the sample delays, which would cause the adapter to fail. A good way to proceed is to perform the first simulation with checking function activated to determine the appropriate

sample delay value. Once TL0 behaviour is correct, checking function should be disabled to increase simulation speed.

3.2.2. Explicit timings constructor

This constructor can be used to explicitly specify sampling times, when default values do not match the timing requirements of the master.

```
OCP_TL0_TL1_Master_Adapter_HS(sc_module_name name,
                , int ID
                , int Priority
                , sc_time RequestSampleDelay
                , sc_time RespAcceptSampleDelay
                , bool Check_setup_time = true
                )
```

- *name*: Name of the module (slave adapter) instance.
- *ID*: Unique number identifying the Master.
- *Priority*: Positive number specifying the priority relative to the other masters for bus access. Higher numbers means higher priority. Masters can have the same priority.
- *RequestSampleDelay*: Time number specifying the delay between the clock rising edge and the actual sampling time for TL0 request AND datahandshake group signals.
- *RespAcceptSampleDelay*: Time number specifying the delay between the clock rising edge and the actual sampling time for '*MrespAccept'* signal.
- *Check_setup_time*: boolean value specifying if '*check_setup_time()*' function is executed during simulation (see below).

The smallest value for the sample delays is sc_get_time_resolution() ; it should be used in the case when TL0 master has pure 'zero-delay' signals. Users should always use a timed sample delay (and not a defined number of delta-cycles) since it is not possible to predict the number of delta-cycles required for signal stability.

3.3. Examples

The following four examples (three for the basic adapter and one for the handshake adapter) present the use of the master adapters in different system environments:

- Fully combinational transaction (Example 1).
- Synchronous split transaction with combinational 'Accept' signals (Example 2).
- Synchronous split transaction with synchronous 'Accept' signals (Example 3).
- Synchronous split transaction with combinational 'Accept' signals and datahandshake management (Example 4).

For each example, we discuss about the correct values to give to the 'SampleDelay' parameters in order to match the TL0 master output delays.

3.3.1. Fully combinational transaction (Example 1)

In this example, the Basic adapter connects a TL0 asynchronous master to a TL1 asynchronous slave. The top level C++ file is called top_master_async.cpp. The example master and slave files are

ocp_tl0_master_async.cpp, ocp_tl1_slave_async.cpp. Use the Makefile to build the example, and run 'run_master_async' executable. Waveforms for OCP TL0 signals can be viewed using the generated VCD file 'top_master_async.vcd'.

This example presents a fully combinational OCP connection, where requests and responses occur in the same clock cycle. The master issues 5 writes followed by 5 reads, and checks received data. Operation timings are described in the header of the top_master_async.cpp file.

Setting 'RequestSampleDelay' parameter:
The driving path for the request group signals is the following:

Clk (t=***0***) -> TL0 Request Group (t=***T1***)

where :

- T1 specifies the delay (starting from the rising edge of the clock) used by the TL0 Master before issuing a request.

For this kind of combinational path on request group signals, parameter '***RequestSampleDelay***' must be **greater than T1.**

Setting 'RespAcceptSampleDelay' parameter:
The driving path for the response group signals is the following:

Clk (t=***0***) -> TL0 Request Group (t=***T1***) -> TL0 Adapter Request sampling (t=***T5***) -> TL1 Slave Response (t=***T5+T3***) -> TL0 Master '*MrespAccept*' (t=***T5+T3+T4***)

where :

- T1 specifies the delay (starting from the rising edge of the clock) used by the TL0 Master before issuing a request.
- T5 specifies the '*RequestSampleDelay*' parameter.
- T3 specifies the delay (starting from the request reception) used by the TL1 Slave before issuing a response.
- T4 specifies the delay (starting from the response reception) used by the TL0 Master before issuing a response acknowledge.

For this kind of combinational path on '*MRespAccept*' response signals, parameter '***MRespAcceptSampleDelay***' must be **greater than '*RequestSampleDelay*' + T4+T3.**

3.3.2. Synchronous split transaction with combinational 'Accept' signals (Example 2)

In this example, the Basic adapter connects a TL0 synchronous master to a TL1 slave. The top level C++ file is called top_master_sync.cpp. The example master and slave files are ocp_tl0_master_async.cpp, ocp_tl1_slave_sync.cpp. Use the Makefile to build the example, and run 'run_master_sync' executable. Waveforms for OCP TL0 signals can be viewed using the generated VCD file 'top_master_sync.vcd'.

In this example, responses occurs one cycle after requests, and 'Accept' signals use combinational paths. The master issues 5 writes followed by 5 reads, and checks received data. Operation timings are described in the header of the top_master_sync.cpp file.

Setting 'RequestSampleDelay' parameter:
The value is the same as for example 1: ***'RequestSampleDelay'*** must be **greater than T1**

Setting 'RespAcceptSampleDelay' parameter:
The driving path for the response group signals is the following:

Clk (t=***0***) -> TL1 Slave Response (t=***T3***) -> TL0 Master '*MRespAccept*' (t=***T3+T4***)

where :

- T3 specifies the delay (starting from the rising edge of the clock following a request reception) used by the TL1 Slave before issuing a response.
- T4 specifies the delay (starting from the response reception) used by the TL0 Master before issuing a response acknowledge.

For this kind of combinational path on '*MRespAccept*' response signals, parameter '***MRespAcceptSampleDelay***' must be **greater than T3+T4.**

3.3.3. Synchronous split transaction with synchronous 'Accept' signals (Example 3)

In this example, the Basic adapter connects a TL0 synchronous master to a TL1 slave. The top level C++ file is called top_master_sync2.cpp. The example master and slave files are ocp_tl0_master_async.cpp, ocp_tl1_slave_sync.cpp. Use the Makefile to build the example, and run 'run_master_sync2' executable. Waveforms for OCP TL0 signals can be viewed using the generated VCD file 'top_master_sync2.vcd'.

In this example, responses occur one cycle after requests, and 'Accept' signals assertion take one cycle (+ some propagation delays). The master issues 5 writes followed by 5 reads, and checks received data. Operation timings are described in the header of the top_master_sync2.cpp file.

Sample delays for this example are the same as for example 2.

3.3.4. Synchronous split transaction with combinational 'Accept' signals and datahandshake management (Example 4)

In this example, we use the second slave adapter (Basic signals + datahandshake) to connect a TL0 synchronous master with datahandshake support to a TL1 slave. The top level C++ file is called top_master_sync_hs.cpp. The example master and slave files are ocp_tl0_master_sync_hs.cpp, ocp_tl1_slave_sync_hs.cpp. Use the Makefile to build the example, and run 'run_master_sync_hs' executable. Waveforms for OCP TL0 signals can be viewed using the generated VCD file 'top_master_sync_hs.vcd'.

In this example, responses occurs one cycle after requests, data are transmitted one cycle after corresponding requests (datahandshake), and both 'SCmdAccept' and 'SDataAccept' signals use combinational paths. The master issues 5 writes followed by 5 reads, and checks received data. Operation timings are described in the header of the top_master_sync_hs.cpp file.

Setting 'RequestSampleDelay' parameter:
The driving path for the *request group* signals is the following:

Clk (t=***0***) -> TL0 Request Group (t=***T1***)

where :

- T1 specifies the delay (starting from the rising edge of the clock) used by the TL0 Master before issuing a request.

The driving path for the *datahandshake group* signals is the following:

Clk (t=***0***) -> TL0 Request Group (t=***T2***)

where :

- T3 specifies the delay (starting from the rising edge of the clock) used by the TL0 Master before issuing a data.

Since the same delay is used to sample both *Request* and *Datahandshake* signals, the ***'RequestSampleDealy'*** has to be set to the maximum of these two values: ***max(T1, T2)***.

Setting 'RespAcceptSampleDelay' parameter:
The value is the same as for example 2.

4. OCP EXTENSION GUIDELINES

The presented examples use a subset of the OCP 2.0 dataflow interface, including :

- Basic signals
- '*MRespAccept*' signal
- Datahandshake signals

Extension of the adapters to support the full OCP 2.0 dataflow interface require some changes to the current adapters. We give here some guidelines to implement these extensions.

4.1. Additional Request group signals

The following request group signals are missing: *MAddrSpace*, *MBurst*, *MbyteEn, MConnID* and *MthreadID*. Since these signals belong to the same group as the basic signals, their implementation is straightforward and just require to:

- Add the corresponding TL0 signals on the adapter interface
- For the Master adapter, add the corresponding Request Group assignments (TL1_Request.signal = …) at the same place in the code than those used for basic signals.
- For the Slave adapter, add the corresponding Request Group TL0 signal write calls at the same place in the code than those used for basic signals.

4.2. Additional Datahandshake group signals

The only missing signals are '*MDataThreadID', 'MdataByteEn' and 'MdataInfo'*. As for request group signals, extension is simple and just require to:

- Add the TL0 signal on the adapter interface.
- For the Master adapter, add the corresponding Data Handshake assignments (TL1_dataHS.signal = …) at the same place as for MData signal.
- For the Slave adapter, add the corresponding Data Handshake read operations (variable = TL1_dataHS.signal) at the same place as for MData signal.

4.3. Threadbusy signals

In the OCP 1.0 release, *Threadbusy* signals are just a hint for the receiving agents. It means that there is no specific time for the receiving agent to take into account the threadbusy signals.

The easiest way to implement threadbusy in the adapters is just to add an SC_METHOD which is sensitive on the rising edge of the clock, tests the TL0 threadbusy signal and issue the corresponding TL1 call to the channel. This way, there is no need for an additional sample delay, but it is no possible to model exact threadbusy behaviours as described in the release OCP 2.0, due to the extra latency cycle introduced by the SC_METHOD.

To model the exact threadbusy behaviour, the only solution is to add a third sample delay triggering the threadbusy reading process.

5. TL1-TL2 MASTER ADAPTER

This adapter enables the connection of a TL1 master with a TL2 slave, through a TL1 channel on one side, and a TL2 channel on the other side.

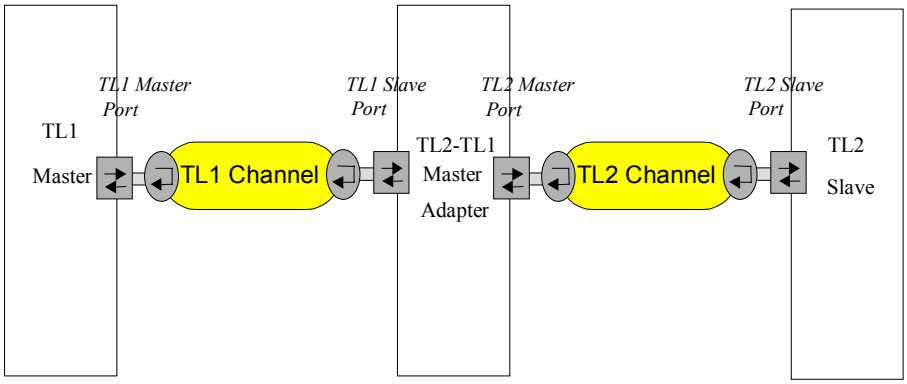


Figure 9: Connection of a TL1-TL2 Master Adapter with a TL1 master and a TL2 slave

The adapter's code can be found in the following files: *ocp_tl1_tl2_master_adapter.cpp* and *ocp_tl1_tl2_master_adapter.h.*

5.1. Template parameters

The TL1-TL2 Master Adapter is templatized over 2 parameters:

- the TL1 data class
- the TL2 data class

The two data classes must be the same as the ones used respectively by the TL1 master and channel, and the TL2 slave and channel. For the examples, it should be:

- OCP_TL1_DataCl<int , int> for the TL1 data class,
- OCP_TL2_DataCl<int , int> for the TL2 data class.

5.2. Constructor

The adapter's constructor takes 4 parameters:

OCP_TL1_TL2_Master_Adapter(sc_module_name name,
int max_chunk_length,
int adapter_depth,
sc_time timeout)

- *name :* name of the module instance

- *max_chunk_length:* maximum size of a TL2 request chunk
- *adapter_depth*: number of TL2 requests/TL2 responses that the adapter stores internally. It indicates how many successive request/response can be stored by the adapter, while processing the current request/response. When the adapter has stored *adapter_depth* requests or responses that need to be processed, it waits for the end of the current request/response before releasing the request/response channel.
- *timeout:* a warning is emitted if during a *timeout* period, no TL1 requests from the master and no TL2 responses from the slave have been received by the adapter

5.3. Interface

The TL1-TL2 master adapter has 3 ports:

- a TL1 slave port
  - name : *SlaveP*
  - type : *OCP_TL1_SlavePort<TdataCl_tl1* > (*TdataCl_tl1* is the template parameter of the module corresponding to the TL1 data class)
- a TL2 master port
  - name : *MasterP*
  - type : *OCP_TL2_MasterPort<Td, Ta* > (Td and Ta are the size of the data and address types for the TL2 data class)
- a clock port (for the TL1 side of the adapter, it is the same clock as the one used by the TL1 master)
  - name : *clk*
  - type : *sc_in_clk*

The TL1 slave port must be connected to the TL1 channel, the TL2 master port to the TL2 channel, and the clock port to the corresponding clock signal.

5.4. OCP Channel parameters

The adapter only supports the following OCP channel configurations (for both TL1 and TL2):

- threads=1
- mthreadbusy_exact=0
- sthreadbusy_exact=0
- respaccept=1
- cmdaccept=1
- datahandshake=0
- burstsinglereq=0

5.5. Functionality

The TL1-TL2 master adapter is totally pipelined, i.e. the request and response threads are totally independent.

Note: to print debug messages on the screen, uncomment the following line in the *ocp_tl1_tl2_master_adapter.h* file*:*
#define DEBUG_MASTER_ADAPTER_TL1

5.5.1. Burst support

The master adapter supports two modes:
- burst support (precise and imprecise bursts),
- no burst support.

The parameters of the channels define which mode is set. They are retrieved by the adapter in the *end_of_elaboration()* method.

Burst support is defined by setting burstseq and burstlength to 1. No burst support is defined by setting one of these fields to 0.

When burst is supported, two kinds of burst can be used:
- Precise (field burstprecise is set to1): the size of the burst is indicated by the MBurstLength field, which stays constant during the burst
- Imprecise (field burstprecise is set to 0): if MBurstLength signal is used by the master core, the end of the burst is indicated by MBurstLength=1. If not, the end of the burst is indicated by the optional signal MReqLast=true.

For the address sequence, only incrementing and streaming bursts can be handled by the adapter. Custom bursts will be separated and sent as single TL2 requests to the slave (see *OCP 2.0 specification* for more details about the type of bursts and the address sequence).

5.5.2. Variation of the request fields during a burst

The following request fields are supposed to remain constant during a whole burst:

- MCmd
- MAddressSpace
- MConnId
- MReqInfo
- MAtomicLength

If the burst mode is set: MBurstPrecise and MBurstSeq stay the same during the burst. If the burst is precise, MBurstLength must not change.

A variation of MByteEn and MDataInfo is allowed during a burst and indicates a new TL2 chunk of the burst.

5.5.3. Unsupported features

The following is unsupported:

- Single request, multiple data
- Data handshake
- Multiple threads
- SThreadBusy/MThreadBusy compliance

5.5.4. Master adapter's behaviour

Basically, the TL1-TL2 master adapter behaves as follows: the master adapter receives some TL1 requests from the master, which are stored internally. The adapter checks their parameters (request fields, address increment…) and packs them into TL2 chunks accordingly, which are sent to the slave. On the response side, the master adapter receives the TL2 responses from the slave, and sends corresponding TL1 requests to the master.

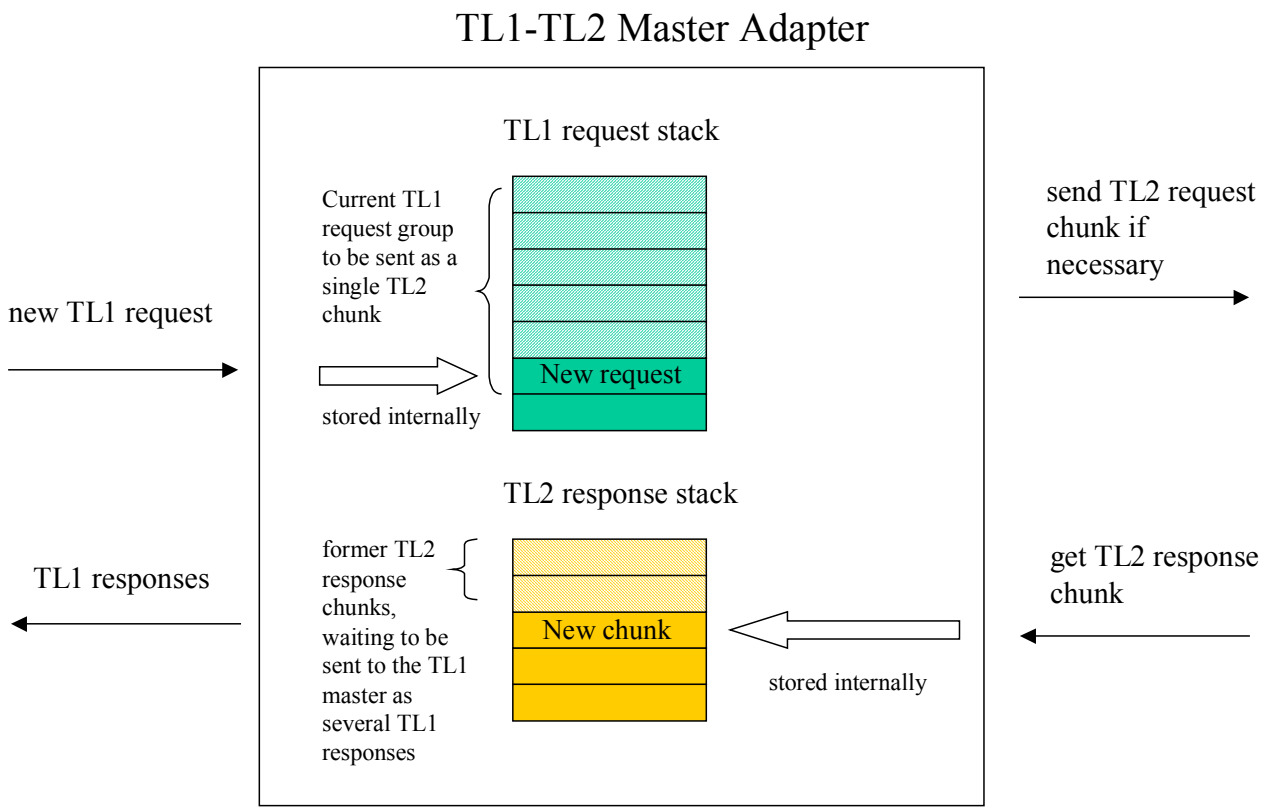


Figure 10: Functioning of the TL1-TL2 Master Adapter

More precisely, during an OCP transaction:

For the request thread:

The TL1 master:

sends some TL1 requests, which are part (or not, in the case of a single request) of a burst. These requests have certain characteristics, defined by the request fields (*MAtomicLength, MReqInfo, MAddr, MAddrSpace, MByteEn, MCmd, MByteEn, MDataInfo*…).

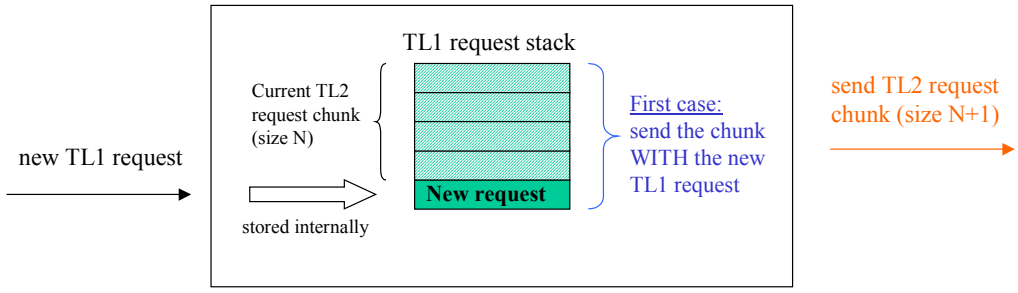
The TL1-TL2 Master Adapter:

- catches the TL1 request, stores its characteristics (*MAddr, MAddrSpace, MBurst, MCmd*…+data field),
- compares these characteristics with the characteristics of the TL2 chunk currently stored, if any chunk is being stored. If not, the adapter just stores the new request parameters.

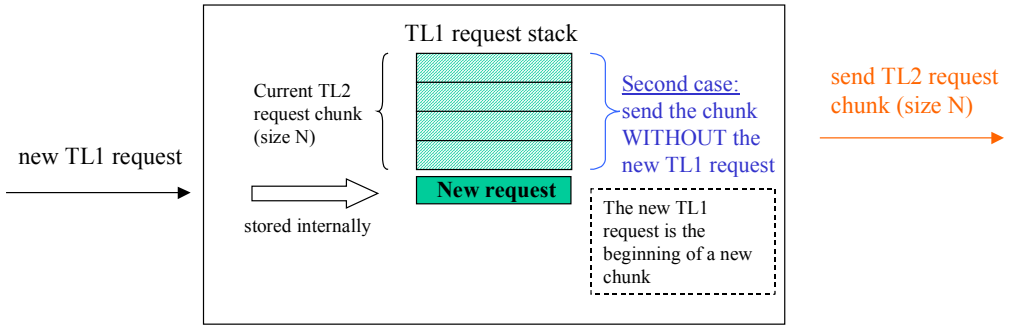
More precisely, when a current TL2 chunk is being stored in the adapter:

- Case 1: the adapter sends the current chunk WITH the new TL1 request:
  - if the request fields have not changed,
  - if the increment of address is right (see below for more details),

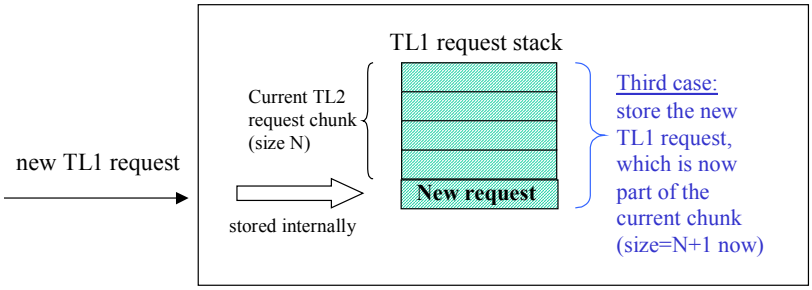
- and if it is the last request of the TL1 burst, or if the completed TL2 chunk has reached *max_chunk_length*.



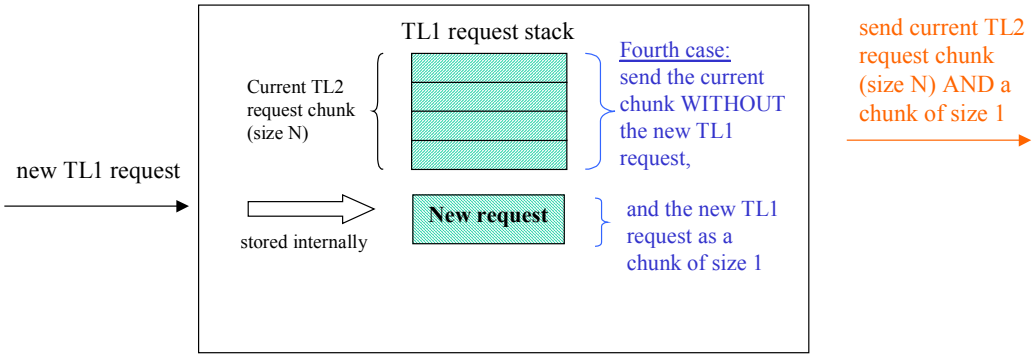
- Case 2: the adapter sends the current TL2 chunk WITHOUT the new TL1 request, which is stored internally as the start of a new TL2 chunk:
  - if the request fields have changed,
  - or if the increment of address does not correspond to the current chunk address increment.



- Case 3: the adapter stores the new TL1 request internally, which is added to the current TL2 chunk,
  - if the request fields have not changed
  - if the increment of address corresponds to the current chunk address increment.



- Case 4: the adapter sends the current TL2 chunk WITHOUT the new TL1 request, and sends the last TL1 request as a single TL2 request (chunk length = 1 and last_of_a_burst = true):
  - if the request fields have changed
  - or if the increment of address does not correspond to the current chunk address increment.
  - and if this new TL1 request is the last cell of the TL1 burst.



The increment of address should be:

- for an incrementing burst: the increment corresponding to the OCP data width,
- for a streaming burst: 0.

When the adapter has to send a TL2 request to the slave, corresponding to the TL1 requests that it has received, the adapter:

- specifies the TL2 request parameters,
- increments the number of pending TL2 requests that need to be sent to the slave,
- releases the TL1 request channel, if the request buffer is not full, so that the master can send other TL1 requests while the TL2 request is being processed. If the internal TL2 request buffer is full, the adapter has to wait for a TL2 request to be processed by the slave, in order to store a new TL2 request (made from the new TL1 requests coming from the TL1 master),
- makes some blocking calls on the TL2 master port (*sendOCPRequestBlocking),*
- decrements the number of pending TL2 requests, when the blocking call returns (meaning that the slave has accepted the TL2 request),
- if it was not done before, releases the TL1 request channel when the blocking call returns, meaning that the adapter can build a new TL2 request.

If there is a pending TL2 request (or several pending requests) that needs to be sent to the slave, the adapter makes other blocking *sendOCPRequestBlocking* calls corresponding to these requests, as soon as the current TL2 request has been accepted by the slave.

For the response thread:

The TL1-TL2 Master Adapter issues a blocking *getOCPResponseBlocking()* call, and waits for a new TL2 response sent by the slave.

When a new TL2 response arrives, the TL1 master adapter:

- transfers the response fields from the TL2 channel to the adapter,
- copies the response data fields internally,
- increments the number of groups of TL1 responses that need to be sent to the master (one TL2 response corresponds to one group of TL1 responses),

- releases the TL2 response channel, if the response buffer is not full, so that the slave can send other TL2 responses while the group of TL1 response is being processed. If the response buffer is full, the adapter has to wait that one group of TL1 response has been processed by the master, in order to store new responses coming from the TL2 slave.

Then the master adapter issues the corresponding TL1 responses to the TL1 master. It:

- copies the response fields into the TL1 channel
- sends the group of TL1 responses at each rising edge of the clock
- releases the Slave TL2 response thread when all the responses have been sent, if it was not done before,
- decrements the number of group of TL1 responses that needs to be sent to the master.

If there are some pending groups of TL1 responses that need to be sent to the master, the master adapter send them when the current group of TL1 responses has been sent. Otherwise, the adapter waits for another group of TL1 responses to send.

5.6. Example

The example demonstrates the connection of a TL1 master to a TL2 slave, through a TL1-TL2 master adapter and two channels (TL1 and TL2 channel).

The master and slave can be found in the following files:

- *ocp_tl1_master.cpp* and *ocp_tl1_master.h*,
- *ocp_tl2_slave.cpp* and *ocp_tl2_slave.h*,

The top-level can be found in the *top_tl1_tl2_master_adapter.cpp* file. Use the *Makefile* to build the example (specify the position of the channel directory in the TL_SC variable), and run *run_master_adapter* executable. The output messages can be found in the *output_tl1_tl2_master_adapter.txt* file.

The master adapter is templatized over the two data classes OCP_TL1_DataCl<int , int> and OCP_TL2_DataCl<int , int>.

The parameters of the adapter are the following:

- name of the instance: "master_adapter"
- maximum burst length: 8
- adapter depth: 6
- timeout: 2 nanoseconds

Master's behaviour

The TL1 master issues a READ precise burst composed of 15 atomic requests (start address=0, address increment=4, MBurstLength=15).
For the three first TL1 requests, MReqInfo=0. At the fourth request (read address=12), MReqInfo changes to 1.

On the response side, the master releases the response channel after two clock cycles.

Slave's behaviour

The TL2 slave stores incremental data:

- 0 at address 0,
- 1 at address 4,
- 2 at address 8…

It is defined with no acceptance delay for the request and response (4 last constructor parameters are 0 cycle).

Master adapter's behaviour

The adapter separates this READ burst into four request chunks:

- The first chunk has a length of 3 (start address=0, end address=8). It is sent when the fourth TL1 has been received by the adapter: it has detected that MReqInfo has changed but the other fields have remained the same. A new chunk is beginning with that request.
- The second chunk has a length of 5 (start address=12, end address=28). The adapter has checked that all the request fields are coherent. It sends the chunk when it has reached the *maximum chunk length.*
- The third chunk has also a length of 5 (maximum chunk length, start address=32, end address=48). The adapter has checked that all the request fields are coherent. It sends the chunk when it has reached the *maximum chunk length.*
- The fourth and last chunk has a length of 2 (start address=52, end address=56). The chunk is sent when the adapter detects that is it the last of the burst: 15 (=MBurstLength) TL1 requests have been received.

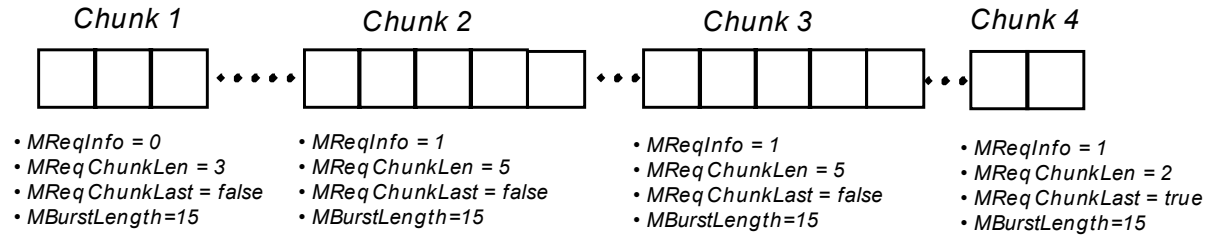


Figure 11: TL2 request sequence sent by the adapter to the TL2 slave

On the response side, the adapter retrieves 4 TL2 response chunks :

- First chunk: read data are 0,1,2,
- Second chunk: read data are 3,4,5,6,7,
- Third chunk: read data are 8,9,10,11,12,
- Fourth chunk: read data are 13,14.

15 corresponding TL1 responses are sent to the master. Three clock cycles separate each TL1 response, since the master releases the response channel after two clock cycles.

6. TL1-TL2 SLAVE ADAPTER

The TL1-TL2 slave adapter enables the connection of a TL2 master with a TL1 slave, through a TL2 channel on one side, and a TL1 channel on the other side.

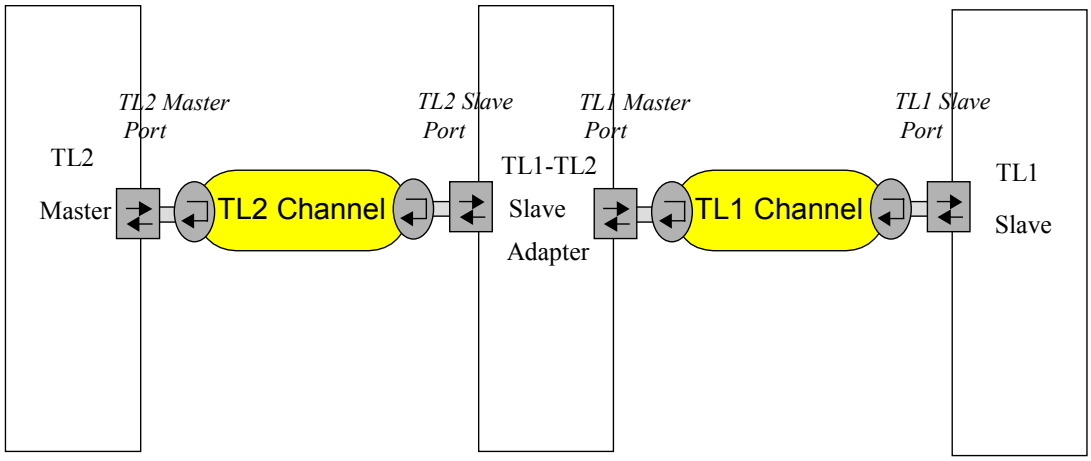


Figure 12: Connection of a TL1-TL2 Slave Adapter with a TL2 master and a TL1 slave

The adapter's code can be found in the following files: *ocp_tl1_tl2_slave_adapter.cpp* and *ocp_tl1_tl2_slave_adapter.h.*

6.1. Template parameters

The TL1-TL2 Slave Adapter is templatized over 2 parameters:

- the TL1 data class
- the TL2 data class

The two data classes must be the same as the ones used respectively by the TL2 master and channel, and the TL1 slave and channel. It can be the default data classes:

- OCP_TL1_DataCl<int , int> for the TL1 data class,
- OCP_TL2_DataCl<int , int> for the TL2 data class.

6.2. Constructor

The adapter's constructor takes 4 parameters:

OCP_TL1_TL2_Slave_Adapter(sc_module_name name,
int max_chunk_length,
int max_burst_length,
int adapter_depth)

- *name :* name of the module instance
- *max_chunk_length:* maximum size of a TL2 response chunk sent to the master
- *max_burst_length*: maximum size of a burst. If the master sends a burst which has a length greater than this parameter, a warning is emitted and the burst is truncated to the maximum size
- *adapter_depth*: number of TL2 requests/TL2 responses that the adapter stores internally. It indicates how many successive request/response can be stored by the adapter, while processing the current request/response. When the adapter has stored *adapter_depth* requests or responses that need to be processed, it waits for the end of the current request/response before releasing the request/response channel.

6.3. Interface

The TL1-TL2 slave adapter has 3 ports:

- a TL2 slave port
  - name : *SlaveP*
  - type : *OCP_TL2_SlavePort< Td, Ta >* (Td and Ta are the size of the data and address types for the TL2 data class)
- a TL1 master port
  - name : *MasterP*
  - type : *OCP_TL1_MasterPort< TdataCl_tl1 >* (*TdataCl_tl1* is the template parameter of the module corresponding to the TL1 data class)
- a clock port (for the TL1 side of the adapter, it is the same clock as the one used by the TL1 slave)
  - name : *clk*
  - type : *sc_in_clk*

The TL2 slave port must be connected to the TL2 channel, the TL1 master port to the TL1 channel, and the clock port to the corresponding clock signal.

6.4. Channel parameters

The following parameters of the TL1 and TL2 channel must be set to these values:

- threads=1
- mthreadbusy_exact=0
- sthreadbusy_exact=0
- respaccept=1
- cmdaccept=1
- datahandshake=0
- burstsinglereq=0

If the TL1 slave sends response to WRITE requests, writeresp_enable must be set to 1 in the TL1 channel.

6.5. Functionality

The adapter is totally pipelined, i.e. the request and response threads are totally independent.

Note: to print debug messages on the screen, uncomment the following line in the *opc_tl1_tl2_slave_adapter.h* file*:*
#define DEBUG_SLAVE_ADAPTER_TL1

6.5.1. Burst support

On the TL2 request side, the end of the request burst is detected by the *last_chunk_of_the_burst* parameter of the request chunk, retrieved by the adapter.

For the address sequence, only incrementing and streaming bursts can be handled by the adapter. If the ‘MBurstSeq’ field of the TL2 request is different from ‘INCR’ or ‘STRM’, an error message will be displayed and the simulation is stopped.

Note: The slave adapter does not take into account the MBurstLength, MBurstPrecise, or MReqLast fields. The size of the TL1 burst to be sent over the TL1 side is equal to the sum of the TL2 chunk lengths that compose the TL2 burst..

On the TL1 request side, the slave adapter generates precise TL1 bursts (MBurstSeq=INCR or STRM, MBurstPrecise=1, MBurstLength stays constant during the burst and is equal to the total number of TL1 requests).

6.5.2. Variation of the response fields during a burst

The SRespInfo field must not change during a burst. If the adapter retrieves some TL1 responses, which are part of the same TL1 burst, but have a different SRespInfo field, the burst is separated as different TL2 response chunks and a warning is emitted.
If during the TL1 response burst, the SResp field changes, the burst is separated in several TL2 response chunks. For each chunk, the SResp and SRespInfo do not change.

6.5.3. Unsupported features

The following is unsupported:

- Single request, multiple data
- Data handshake
- Multiple threads
- SThreadBusy/MThreadBusy compliance

6.5.4. Slave adapter's behaviour

The slave adapter is totally pipelined, i.e. the request and response threads are totally independent.

The slave adapter and the master adapter have a symmetrical behaviour: the response threads in one adapter correspond to the request threads in the other adapter. What is different is that the slave adapter has to know how many TL1 responses correspond to the same TL2 chunk, to know how to pack them as a whole TL2 response burst. We assume that the TL1 responses are safely handled to the slave adapter, and that no responses are lost during the process.

Basically, the slave adapter's purpose is to get a TL2 request, separates it in corresponding TL1 requests and sends them to the TL1 slave. (The slave adapter generates precise TL1 request bursts). On the response side, it gets some TL1 responses, packs them in one or several TL2 response chunks and sends them to the TL2 master. The slave adapter expects as many TL1 responses as the number of sent TL1 requests to the slave.

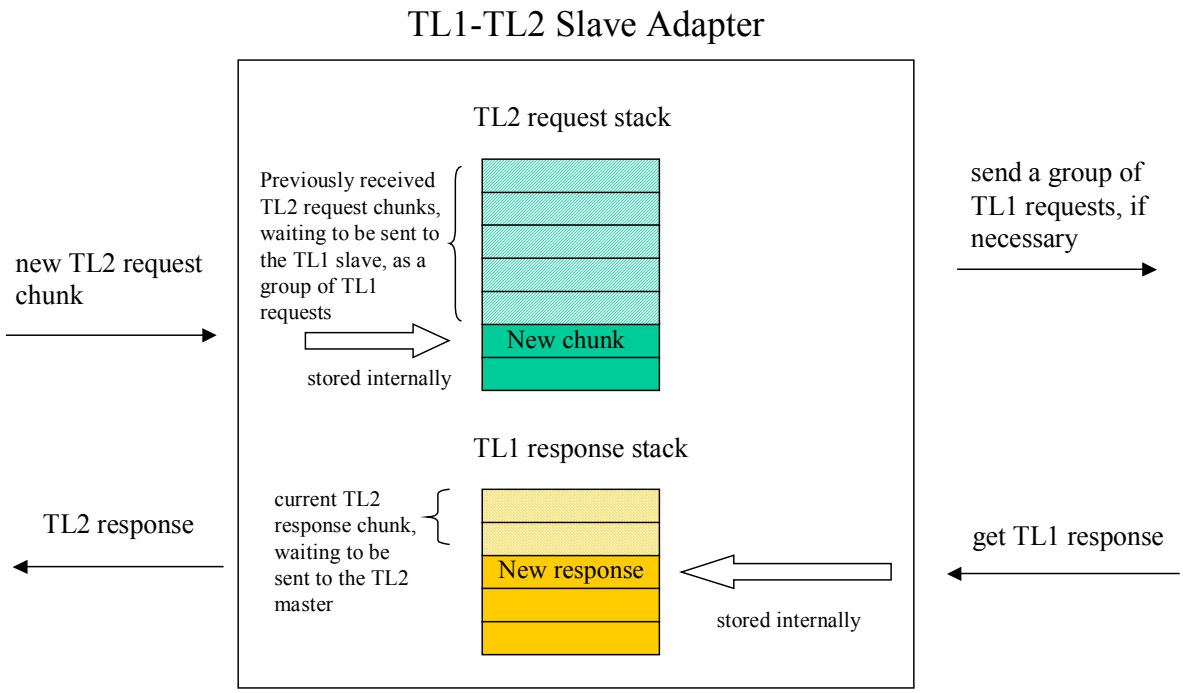


Figure 13: Functioning of the TL1-TL2 Slave Adapter

More precisely, during an OCP transaction, the adapter behaves as follows:

For the request thread:

The TL2-TL1 Slave Adapter issues a blocking *getOCPRequestBlocking()* call, and waits for a new TL2 request chunk sent by the master.

When a new TL2 request arrives, the TL2-TL1 slave adapter:

- retrieves the size of the chunk and if it is the last of the burst,
- copies the request fields from the TL2 channel to the adapter,
- copies the different data,
- checks if the size of the chunk is not greater than the maximum burst length. On the contrary, it issues a message and truncates the end of the request burst.
- computes the size of the whole response burst that it is expecting from the slave (i.e. as many responses as there were requests). The end of the request burst is detected by checking the last_of_a_burst TL2 field.
- increments the number of groups of TL1 requests that need to be sent to the master (one TL2 request corresponds to one group of TL1 requests),
- releases the TL2 channel, if the request buffer is not full, so that the master can send other TL2 requests while the group of TL1 requests is being processed. If the request buffer is full, the adapter has to wait that one group of TL1 requests has been processed by the TL1 slave, in order to store new requests coming from the TL2 master,

Then the slave adapter issues the corresponding TL1 requests to the TL1 slave. It:

- copies the requests fields into the TL1 channel, sets MBurstPrecise to 1, and the MBurstLength field as the size of the burst (since the adapter generates precise TL1 request bursts),
- sends one of the TL1 requests at each edge of the clock (using *startOCPRequest* function), when the TL1 request channel is free,
- releases the TL2 request thread when all the TL1 requests have been sent, if it was not done before (meaning that it can store other requests from the TL2 master),
- decrements the number of groups of TL1 requests that need to be sent to the slave.

If there are some pending groups of TL1 requests that need to be sent to the slave, the slave adapter sends them when the current group of TL1 requests has been sent. Otherwise, the slave adapter waits for another group of TL1 requests to send.

For the response thread:

The TL1 slave:

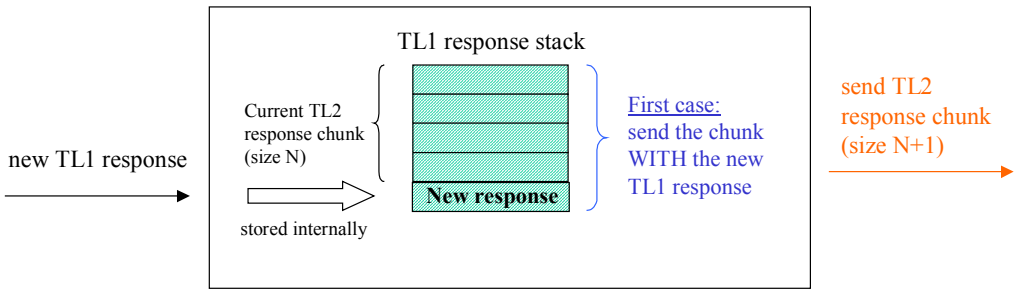
- sends some TL1 responses, which are part (or not, in the case of a single response) of an OCP burst. These responses have certain characteristics, defined by the response fields (*SResp*, *SRespInfo*). The number of responses which compose the burst is known by the slave adapter (from the TL2 requests it has received before)

The TL2-TL1 Slave Adapter:

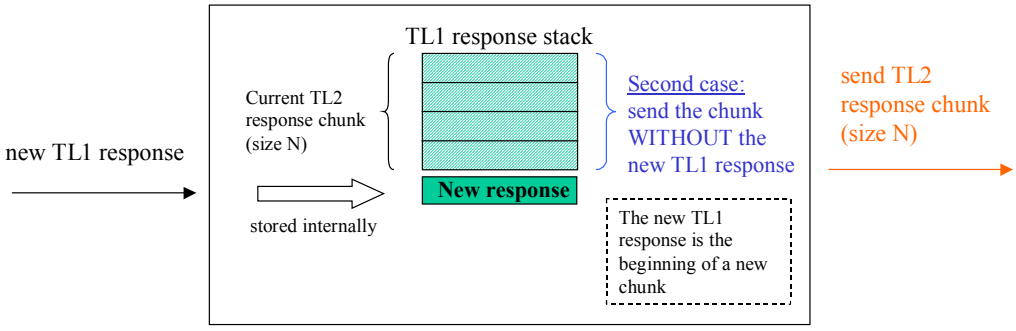
- catches the TL1 response, stores its characteristics (*SResp*, *SRespInfo, SThreadID* +data field),
- does the same with the other responses until the last response of the burst has arrived.

If a current TL2 response chunk is being stored in the adapter, the slave adapter can behave in different ways:

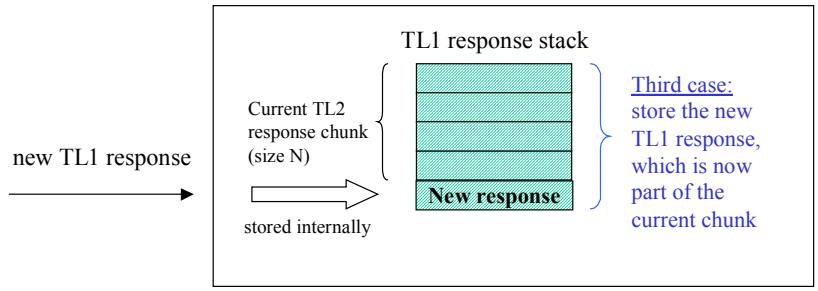
- Case 1: the adapter sends the current TL2 response chunk with the new TL1 response:
  - if the response fields have not changed,
  - and the new response is the last of the OCP burst or if the stored TL2 chunk has reached *max_chunk_length*,



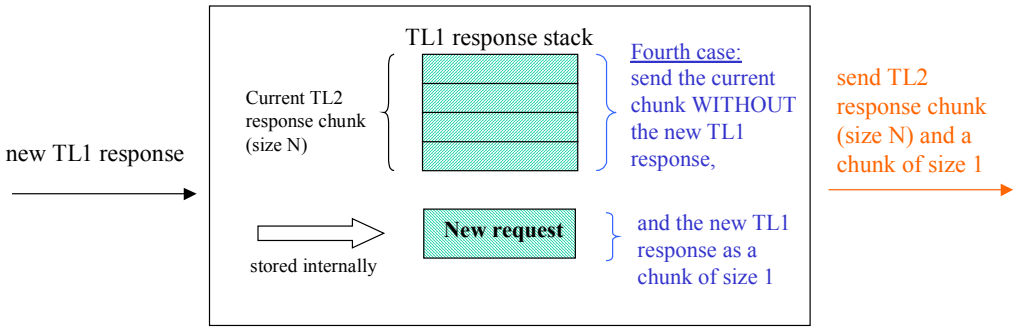
- Case 2: the adapter sends the current TL2 response chunk without the new TL1 response (if the response fields have changed),



- Case 3: the adapter stores the new TL1 response internally, which becomes a part of the current TL2 response chunk (if all the response fields have not changed),



- Case 4: the adapter sends the current stored TL2 response chunk and a TL2 response chunk of size 1 (when the last received TL1 response has different response info and is the last of the response burst)



When the slave adapter has to send a TL2 response to the master, corresponding to the group of TL1 responses that it has received, the adapter:

- specifies the TL2 response parameters,
- increments the number of pending TL2 responses that need to be sent to the master,
- releases the TL1 channel, if the response buffer is not full, so that the slave can send other TL1 responses while the TL2 response is being processed. If the response buffer is full, the adapter has to wait for a TL2 response to be retrieved by the master, in order to store new responses coming from the TL1 slave,
- makes some blocking calls on the TL2 slave port (using *sendOCPResponseBlocking()*),
- decrements the number of pending TL2 responses, when the blocking call returns (meaning that the master has accepted the response),
- if it was not done before, releases the TL1 response channel when the blocking call returns.

If there is a pending TL2 response (or several pending responses) that needs to be sent to the master, the adapter makes other blocking S*putResponseBlocking()* calls corresponding to these responses, as soon as the current TL2 response has been accepted by the master.

6.6. Example

The example demonstrates the connection of a TL2 master to a TL1 slave, through a TL1 slave adapter and two channels (TL1 and TL2 channel).

The master and slave can be found in the following files:

- *simple_ocp_tl2_master.cpp* and *simple_ocp_tl2_master.h*,
- *ocp_tl1_slave_sync.cpp* and *ocp_tl1_slave_sync.h*,

The top-level can be found in the *top_tl1_tl2_slave_adapter.cpp* file. Use the *Makefile* to build the example (specify the position of the channel directory in the TL_SC variable), and run *run_slave_adapter* executable. The output messages can be found in the *output_tl1_tl2_slave_adapter.txt* file.

The slave adapter is templatized over the two data classes OCP_TL1_DataCl<int , int> and OCP_TL2_DataCl<int , int>.

The parameters of the adapter are:

- name of the instance: "slave_adapter"
- maximum chunk length: 5
- maximum burst length: 25
- adapter depth: 6

Master's behaviour

The TL2 master successively issues the following requests:

- A WRITE burst composed of 20 requests in a single TL2 chunk (written data are 1, 2, 3…). The address increment is 4. The base address is 0.
- A READ burst composed of 10 requests in a single TL2 chunk. The address increment is 4, and the base address is 0.
- 2 nanoseconds after the second TL2 request chunk has been accepted, the master sends a READ burst composed of 8 requests, using two chunks of size 4 (the address increment is always 4 and the base address is 0). The second chunk is sent 3 nanoseconds after the first chunk
- 3 nanoseconds after the previous chunk has been accepted, the master finishes with a 15-length WRITE burst, using a single chunk. The base address is 0 and the address increment is 4.

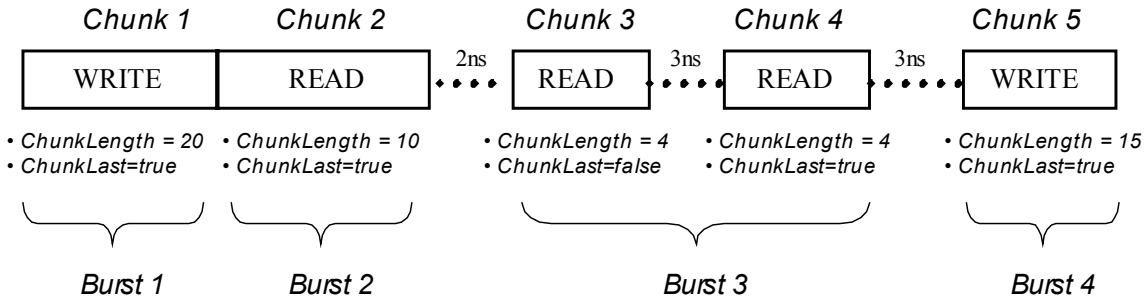


Figure 14: TL2 request sequence sent by the master to the TL1-TL2 slave adapter

Slave's adapter behaviour

The adapter separates the first TL2 WRITE burst in 20 atomic TL1 requests, which are sent to the slave. Written data are 0 to 19.

Immediately after sending the last TL1 WRITE request, the slave adapter sends the first TL1 READ request (address=0), since the READ burst is pending. It follows with the other 9 TL1 READ requests (end address=36). The slave adapter has sent 10 TL1 READ requests, and so it is expecting 10 TL1 responses from the slave.

After each TL1 READ request, the slave adapter gets a TL1 response from the slave (the slave has been configured to immediately send a response to a READ request).

The *maximum chunk length* parameter of the adapter is set to 5 responses. Hence, when the slave adapter has received 5 TL1 responses from the slave, the adapter packs these TL1 responses in a TL2 chunk (which is not the last of the burst, since there are 5 responses left) and sends this chunk to the master. Read data are 0,1,2,3,4.
The same is done with the other 5 TL1 responses, which are packed in a second TL2 chunk (last one of the burst, read data are 5,6,7,8,9).

Another TL2 request burst is pending and waits to be sent to the slave as a group of TL1 requests. After the last TL1 READ request of the previous burst has been sent, the slave adapter begins to send the 8-length READ burst. Similarly, when it has received 5 TL1 responses, it packs them in a TL2 chunk and sends it to the master (not the last of the burst, read data are 0,1,2,3,0).

The slave adapter packs the three other TL1 responses in a second TL2 chunk (last one of the burst, read data are 1,2,3).

The simulation ends with 15 TL1 WRITE requests sent by the slave adapter to the slave (written data are 0 to 14).

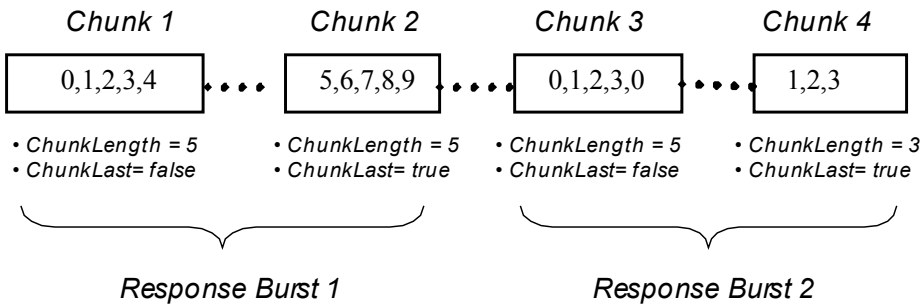


Figure 15: TL2 response sequence sent by the TL1-TL2 slave adapter to the master

To see the effect of the *adapter_depth* parameter: you can change it to 1, meaning that the slave adapter can only store one TL2 request/response at a time. The adapter releases the TL2 request channel when it has finished sending the 20 TL1 WRITE requests to the slave. When *adapter_depth* is equal or superior to 2, the TL2 request channel is released immediately after the first TL2 request chunk has been stored by the adapter: the TL2 master can immediately proceed with the next TL2 request chunk.