OCP Layer Adapters

V2.3: February 2, 2006

Stéphane Guntz, Prosilog

Yann Bajot, Prosilog

Hervé Alexanian, Sonics, Inc

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

Contents

1 Introduction

This document explains the usage and the operating principles of the OCP layer adapters. OCP-IP provides adapters between the TL0 (signal) and TL1 (cycle accurate transfer level) modeling layers, and between the TL1 and TL2 (timed transaction level) modeling layers. The adapters are specific to their OCP interface type, master or slave. Hence the adapters described here are:

- TL0/TL1 Master Adapter
- TL0/TL1 Slave Adapter
- TL1/TL2 Master Adapter
- TL1/TL2 Slave Adapter

The adapters are delivered to members in source form and we will give indications to incorporate them into a design.

2 TL0-TL1 Adapters

2.1 Definitions

TL1 and TL2 are common terminology in OCP transaction modeling but we need to briefly define the TL0 layer. A TL0 interface in the OCP sense is a collection of input and output ports for a given OCP configuration with a signal-level Systemc data type. With logic simulators offering native co-simulation flows between SystemC and other hardware description languages such as Verilog, the choice of signal-level data type is one that can be bound directly to a hardware signal, such as a Verilog wire. Examples of these are sc_bv<int N> and sc_lv<int N>. The adapters presented here currently use sc_bv for multi-bit ports, and bool for single bit ports.

2.2 Design Principles

2.2.1 Phase Assembly and Disassembly

The OCP TL1 channel operates at the transfer phase level. Phases in OCP are request, datahandshake and response. The OCP protocol specification gives precise rules for constructing phases from signals, as well as rules defining the precise start and end of each phase associated with signal transitions. Naturally, the TL0/TL1 adapters operate by following these rules to assemble signals into phases, and conversely, disassemble the TL1 channel phase operations into signals. Figure 1 shows the detection of a request phase start based on the transition of the MCmd signal from an IDLE state. When this event is detected, the adapter captures the value of all signals composing the request group (MAddr, MByteEn, MData if no data handshake) and creates a TL1 request structure: OCPRequestGrp<Td, Ta> where Td is the MData/SData type and Ta is the MAddr type of the TL1 master interface. With this request group assembled, the TL1 side of the adapter can effectively start the phase by calling the startOCPRequest() method of the TL1 master interface.

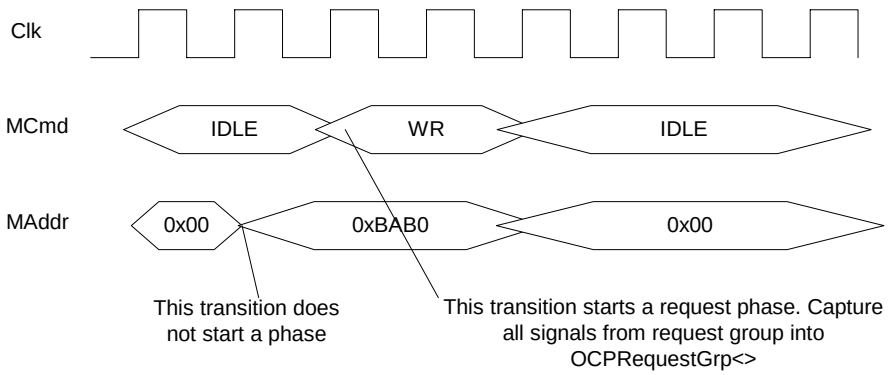

Figure 1: Request Phase Start

2.2.2 TL0 Sampling

The previous paragraph relies on the ability to "detect" a signal transition. Although on the TL0 side, adapters deal with sc_ports to sc_signals (sc_in/sc_out) with built-in value change events, we cannot simply rely on these events. On the TL0 side which may be connected to HDL signals, we need to be mindful of glitches and transient values. If we were to rely on MCmd switching value to a non idle state to start a TL1 request, we would be exposed to that signal switching back to idle, or a different command later during the same clock cycle. The TL1 interfaces do not allow retraction (once a request is sent it may not be taken back). Furthermore, even in the absence of glitches, we need to capture the value of several TL0 signals to start/end a TL1 phase and we could be exposed to ordering uncertainty should some of these signals transition at different times (delta cycles) during the clock period (see Figure 2) Therefore, the only possible approach is to sample the signals at a given fraction of the clock period and assume that they will not change anymore until the next clock edge. We will give details of the sample delays in effect in the TL0/TL1 adapters.

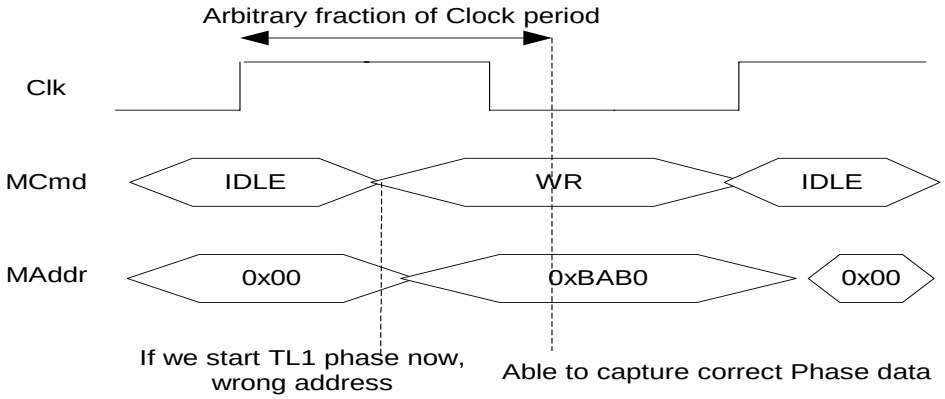

Figure 2: TL0 Signal Sampling

2.2.3 Configurable TL0 port set

Obviously, the configuration of an OCP connection determines which signals are present and how wide they are. This means that the TL0 interface of the adapters cannot be fixed. However, to ease integration with TL0/RTL flows, we have determined that the ability to express an adapter with standard sc_in and sc_out signal ports is very important. This

leads us to an approach where the core of the adapter is able to communicate with a port set defined by the user at compile time. Figure 3 shows a simplified object diagram illustrating how ports are defined and used by a TL0/TL1 adapter. The principle relies on the ability to present a single class, the proxy port base class, to the adapter for port access (actually one for input ports and one for output ports). The user module that needs to instantiate a TL0/TL1 adapter must also instantiate the actual ports, and this is done by using a proxy port class that inherits from both the required systemc port type for the OCP port, and the proxy port base class. This allows the adapter to be independent of the actual port types and the user module to bind it with explicit sc_port's of a signal type defined at compile time.

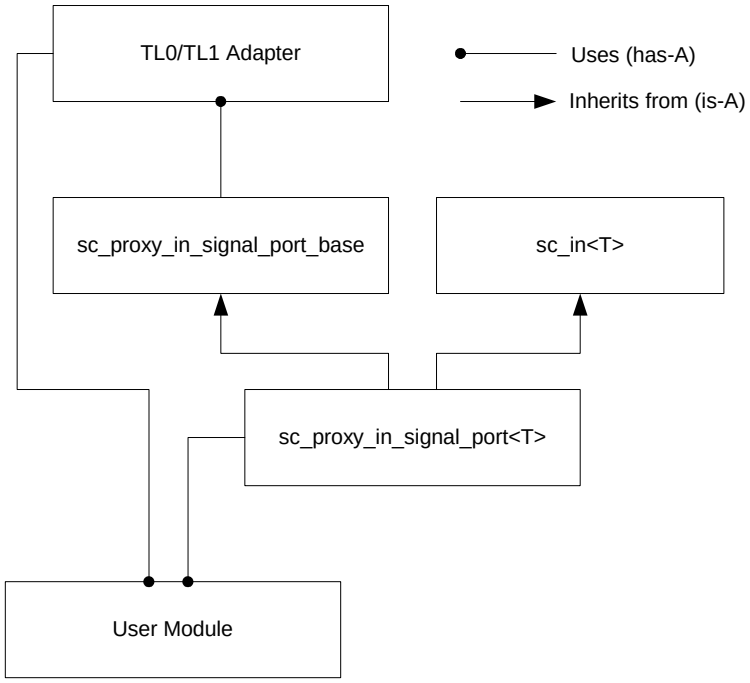

Figure 3: Port Proxies

2.3 TL0-TL1 Slave Adapter

This adapter enables the connection of a TL1 master with a TL0/RTL slave through a OCP signal set on one side and a TL1 channel on the other. The basic connectivity is shown on Figure 4. The TL0 master ports (M prefix) are outputs from the adapter, and the TL0 slave ports (S prefix) are inputs into the adapter.

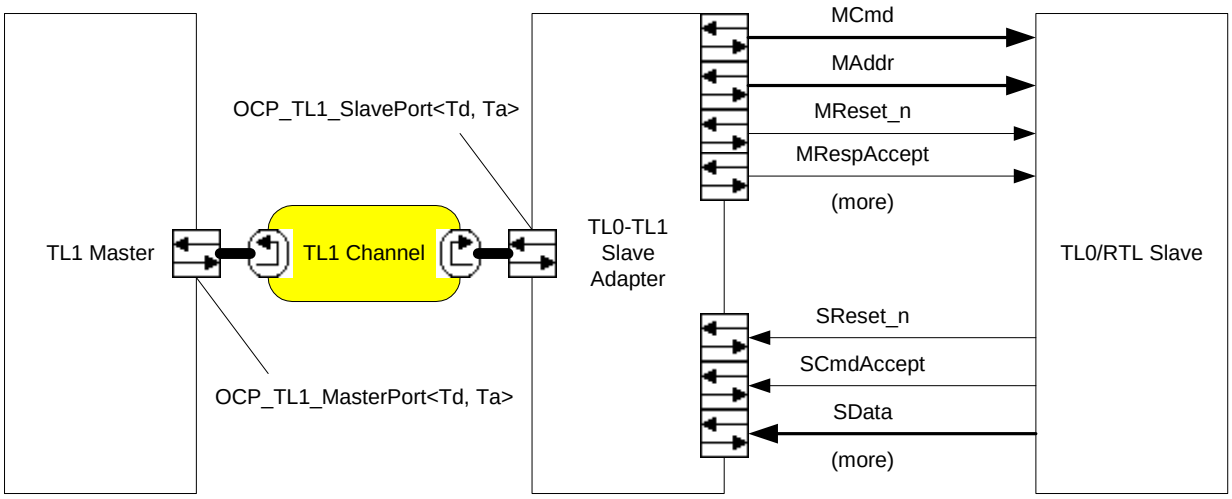

Figure 4: TL0/TL1 Slave Adapter Connectivity

2.3.1 OCP TL0 Master Port class

As can be seen on Figure 4, the TL0 side of the TL0/TL1 slave adapter, is a master OCP interface. It is a port set where the OCP master signals are outputs and the OCP slave signals are inputs. This set of ports with their direction is captured in the OCP2_TL0_MasterPorts class for the OCP 2.0 version of the protocol, and OCP_TL0_MasterPorts for the OCP 1.0 version of the protocol. The design of these classes is very simple and the only differences are in the port names. We will only focus on OCP2_TL0_MasterPorts for this illustration. All the required classes for expressing OCP port sets are in the layer adapter package under:

`include/tl0_tl1/ocp_tl0_port.h`

```
class OCP2_TL0_MasterPorts : public OCP_TL0_AdapterPorts {
    template <typename TdataCl> friend class OCP2_TL0_TL1_Slave_Adapter;
  public:
    template <typename DerivedT>
    struct Factory: public OCP_TL0_PortFactory<OCP2_TL0_MasterPorts> {
        virtual OCP2_TL0_MasterPorts* operator()() const {
            OCP2_TL0_MasterPorts* pTmp = new DerivedT();
            pTmp->bind();
            return pTmp;
        }
    };

    virtual void bind() = 0;

  protected:
    // ins
    sc_proxy_in_signal_port_base* SReset_n;
    sc_proxy_in_signal_port_base* SCmdAccept;
    sc_proxy_in_signal_port_base* SThreadBusy;
    sc_proxy_in_signal_port_base* SDataAccept;
    sc_proxy_in_signal_port_base* SDataThreadBusy;
    sc_proxy_in_signal_port_base* SResp;
    sc_proxy_in_signal_port_base* SRespInfo;
    sc_proxy_in_signal_port_base* SRespLast;
    sc_proxy_in_signal_port_base* SThreadID;
    sc_proxy_in_signal_port_base* SData;
```

```
    sc_proxy_in_signal_port_base* SDataInfo;
    sc_proxy_in_signal_port_base* SFlag;
    sc_proxy_in_signal_port_base* SInterrupt;
    sc_proxy_in_signal_port_base* SError;

    // outs
    sc_proxy_out_signal_port_base* MReset_n;
    sc_proxy_out_signal_port_base* MCmd;
    sc_proxy_out_signal_port_base* MConnID;
    sc_proxy_out_signal_port_base* MThreadID;
    sc_proxy_out_signal_port_base* MByteEn;
    sc_proxy_out_signal_port_base* MReqInfo;
    sc_proxy_out_signal_port_base* MReqLast;
    sc_proxy_out_signal_port_base* MAddr;
    sc_proxy_out_signal_port_base* MAddrSpace;
    sc_proxy_out_signal_port_base* MAtomicLength;
    sc_proxy_out_signal_port_base* MBurstLength;
    sc_proxy_out_signal_port_base* MBurstPrecise;
    sc_proxy_out_signal_port_base* MBurstSeq;
    sc_proxy_out_signal_port_base* MBurstSingleReq;
    sc_proxy_out_signal_port_base* MDataValid;
    sc_proxy_out_signal_port_base* MDataByteEn;
    sc_proxy_out_signal_port_base* MDataThreadID;
    sc_proxy_out_signal_port_base* MDataLast;
    sc_proxy_out_signal_port_base* MData;
    sc_proxy_out_signal_port_base* MDataInfo;
    sc_proxy_out_signal_port_base* MRespAccept;
    sc_proxy_out_signal_port_base* MThreadBusy;
    sc_proxy_out_signal_port_base* MFlag;
    sc_proxy_out_signal_port_base* MError;
};
```

To define the port set for an instance of the TL0/TL1 slave adapter, the user will derive a class from OCP2_TL0_MasterPorts and override the bind method to point the proxy pointers to the actual ports. For example:

```
class my_tl0_ports : public OCP2_TL0_MasterPorts
{
    public:
    // sc_port definitions
    sc_proxy_out_signal_port< bool > MReset_n;
    sc_proxy_out_signal_port< sc_bv< 3 > > MCmd;
    sc_proxy_out_signal_port< sc_bv< 16 > > MAddr;
    sc_proxy_out_signal_port< sc_bv< 3 > > MThreadID;
    sc_proxy_out_signal_port< sc_bv< 128 > > MData;
    sc_proxy_in_signal_port< sc_bv< 2 > > SResp;
    sc_proxy_in_signal_port< sc_bv< 3 > > SThreadID;
    sc_proxy_in_signal_port< sc_bv< 128 > > SData;

    // default constructor, names all ports
    my_tl0_ports():
    MReset_n( "MReset_n" ),
    MCmd( "MCmd" ),
    MAddr( "MAddr" ),
    MThreadID( "MThreadID" ),
    MData( "MData" ),
    SResp( "SResp" ),
    SThreadID( "SThreadID" ),
    SData( "SData" ),
    {}

    // bind ports to base interface class members
    void bind()
```

```
    {
        mapPort( MReset_n, OCP2_TL0_MasterPorts::MReset_n );
        mapPort( MCmd, OCP2_TL0_MasterPorts::MCmd );
        mapPort( MAddr, OCP2_TL0_MasterPorts::MAddr );
        mapPort( MThreadID, OCP2_TL0_MasterPorts::MThreadID );
        mapPort( MData, OCP2_TL0_MasterPorts::MData );
        mapPort( SResp, OCP2_TL0_MasterPorts::SResp );
        mapPort( SThreadID, OCP2_TL0_MasterPorts::SThreadID );
        mapPort( SData, OCP2_TL0_MasterPorts::SData );
    }
};
```

2.3.2 Interface and Class Details

The header file for the TL0/TL1 slave adapters in the layer adapter package are under:

`include/tl0_tl1/ocp2_tl0_tl1_slave_adapter.h (ocp 2.0)`

`include/tl0_tl1/ocp_tl0_tl1_slave_adapter.h (ocp 1.0)`

The TL0/TL1 slave adapter is an *sc_module* with the following signature:

```
template <typename TdataCl> class OCP2_TL0_TL1_Slave_Adapter
```

where TdataCl_tl1 is the OCP TL1 channel data class:

```
OCP_TL1_DataCl<Td, Ta>
```

So an example of a completely resolved instantiation of the adapter is:

```
OCP_TL0_TL1_Slave_Adapter<OCP_TL1_DataCl<unsigned, unsigned> >
```

The TL0/TL1 slave adapter is connected through these ports (*sc_port*)

- sc_in_clk clk: the clock for the OCP connection being adapted
- OCP_TL1_SlavePort<TdataCl_tl1> SlaveP the TL1 slave port
- OCP2_TL0_MasterPorts: the base class for the TL0 port set (see 2.3.1)

The constructor has the following arguments:

- sc_module_name name: the name of the module, can be a const char* string
- const OCP_TL0_PortFactory<OCP2_TL0_MasterPorts>&: a reference to a factory instance to create an instance of the users port set class (see following example)
- bool checkSetupTime: controls whether the adapter should check for late setup of the sampled TL0 signals. When true, signal transitions after the sample delay will be flagged as an error and simulation will be stopped. This argument is true by default.

```
OCP2_TL0_TL1_Slave_Adapter<unsigned int, unsigned int> my_adapter(
"tl0tl1_slave", OCP2_TL0_MasterPorts::Factory< my_tl0_ports >(), true );
```

where my_tl0_ports is a class as described in the example of 2.3.1.

2.4 TL0-TL1 Master Adapter

This adapter enables the connection of a TL0/RTL master with a TL1 slave through a OCP signal set on one side and a TL1 channel on the other. The basic connectivity is shown on Figure 5. The TL0 master ports (M prefix) are inputs into the adapter, and the TL0 slave ports (S prefix) are outputs from the adapter.

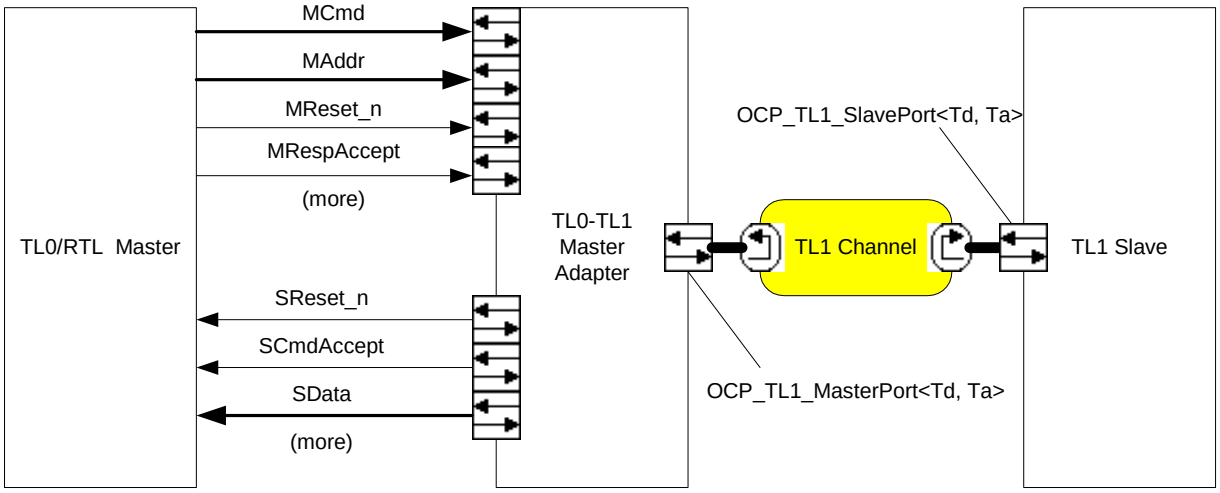

Figure 5: TL0/TL1 Master Adapter Connectivity

2.4.1 OCP TL0 Slave Port Class

As can be seen on Figure 5, the TL0 side of the TL0/TL1 master adapter, is a slave OCP interface. It is a port set where the OCP master signals are inputs and the OCP slave signals are outputs. This set of ports with their direction is captured in the OCP2_TL0_SlavePorts class for the OCP 2.0 version of the protocol, and OCP_TL0_SlavePorts for the OCP 1.0 version of the protocol. The design of these classes is very simple and the only differences are in the port names. We will only focus on OCP2_TL0_SlavePorts for this illustration. All the required classes for expressing OCP port sets are in the layer adapter package under:

`include/tl0_tl1/ocp_tl0_port.h`

```
class OCP2_TL0_SlavePorts : public OCP_TL0_AdapterPorts {
   template <typename TdataCl> friend class OCP2_TL0_TL1_Master_Adapter;
  public:
    template <typename DerivedT>
    struct Factory: public OCP_TL0_PortFactory<OCP2_TL0_SlavePorts> {
        virtual OCP2_TL0_SlavePorts* operator()() const {
            OCP2_TL0_SlavePorts* pTmp = new DerivedT();
            pTmp->bind();
            return pTmp;
        }
    };

    // supposed to be protected, but don't know how to let factory be a
friend
    virtual void bind() = 0;
```

```
  protected:
    // ins
    sc_proxy_in_signal_port_base* MReset_n;
    sc_proxy_in_signal_port_base* MCmd;
    sc_proxy_in_signal_port_base* MConnID;
    sc_proxy_in_signal_port_base* MThreadID;
    sc_proxy_in_signal_port_base* MByteEn;
    sc_proxy_in_signal_port_base* MReqInfo;
    sc_proxy_in_signal_port_base* MReqLast;
    sc_proxy_in_signal_port_base* MAddr;
    sc_proxy_in_signal_port_base* MAddrSpace;
    sc_proxy_in_signal_port_base* MAtomicLength;
    sc_proxy_in_signal_port_base* MBurstLength;
    sc_proxy_in_signal_port_base* MBurstPrecise;
    sc_proxy_in_signal_port_base* MBurstSeq;
    sc_proxy_in_signal_port_base* MBurstSingleReq;
    sc_proxy_in_signal_port_base* MDataValid;
    sc_proxy_in_signal_port_base* MDataByteEn;
    sc_proxy_in_signal_port_base* MDataThreadID;
    sc_proxy_in_signal_port_base* MDataLast;
    sc_proxy_in_signal_port_base* MData;
    sc_proxy_in_signal_port_base* MDataInfo;
    sc_proxy_in_signal_port_base* MRespAccept;
    sc_proxy_in_signal_port_base* MThreadBusy;
    sc_proxy_in_signal_port_base* MFlag;
    sc_proxy_in_signal_port_base* MError;

    // outs
    sc_proxy_out_signal_port_base* SReset_n;
    sc_proxy_out_signal_port_base* SCmdAccept;
    sc_proxy_out_signal_port_base* SThreadBusy;
    sc_proxy_out_signal_port_base* SDataAccept;
    sc_proxy_out_signal_port_base* SDataThreadBusy;
    sc_proxy_out_signal_port_base* SResp;
    sc_proxy_out_signal_port_base* SRespInfo;
    sc_proxy_out_signal_port_base* SRespLast;
    sc_proxy_out_signal_port_base* SThreadID;
    sc_proxy_out_signal_port_base* SData;
    sc_proxy_out_signal_port_base* SDataInfo;
    sc_proxy_out_signal_port_base* SFlag;
    sc_proxy_out_signal_port_base* SInterrupt;
    sc_proxy_out_signal_port_base* SError;
};
```

To define the port set for an instance of the TL0/TL1 master adapter, the user will derive a class from OCP2_TL0_SlavePorts and override the bind method to point the proxy pointers to the actual ports. For example:

```
class my_tl0_ports : public OCP2_TL0_SlavePorts
{
    public:
    // sc_port definitions
    sc_proxy_in_signal_port< bool > MReset_n;
    sc_proxy_in_signal_port< sc_bv< 3 > > MCmd;
    sc_proxy_in_signal_port< sc_bv< 16 > > MAddr;
    sc_proxy_in_signal_port< sc_bv< 3 > > MThreadID;
    sc_proxy_in_signal_port< sc_bv< 128 > > MData;
    sc_proxy_out_signal_port< sc_bv< 2 > > SResp;
    sc_proxy_out_signal_port< sc_bv< 3 > > SThreadID;
    sc_proxy_out_signal_port< sc_bv< 128 > > SData;

    // default constructor, names all ports
```

```
    my_tl0_ports():
    MReset_n( "MReset_n" ),
    MCmd( "MCmd" ),
    MAddr( "MAddr" ),
    MThreadID( "MThreadID" ),
    MData( "MData" ),
    SResp( "SResp" ),
    SThreadID( "SThreadID" ),
    SData( "SData" ),
    {}


    virtual ~my_tl0_ports() {}
    // bind ports to base interface class members
    void bind()
    {
        mapPort( MReset_n, OCP2_TL0_SlavePorts::MReset_n );
        mapPort( MCmd, OCP2_TL0_SlavePorts::MCmd );
        mapPort( MAddr, OCP2_TL0_SlavePorts::MAddr );
        mapPort( MThreadID, OCP2_TL0_SlavePorts::MThreadID );
        mapPort( MData, OCP2_TL0_SlavePorts::MData );
        mapPort( SResp, OCP2_TL0_SlavePorts::SResp );
        mapPort( SThreadID, OCP2_TL0_SlavePorts::SThreadID );
        mapPort( SData, OCP2_TL0_SlavePorts::SData );
    }
};
```

2.4.2 Interface and Class Details

The header file for the TL0/TL1 master adapters in the layer adapter package are under:

`include/tl0_tl1/ocp2_tl0_tl1_master_adapter.h (ocp 2.0)`

`include/tl0_tl1/ocp_tl0_tl1_master_adapter.h (ocp 1.0)`

The TL0/TL1 master adapter is an *sc_module* with the following signature:

```
template <typename TdataCl> class
OCP2_TL0_TL1_Master_Adapter
```

where TdataCl_tl1 is the OCP TL1 channel data class:

`OCP_TL1_DataCl<Td, Ta>`

So an example of a completely resolved instantiation of the adapter is:

`OCP_TL0_TL1_Master_Adapter<OCP_TL1_DataCl<unsigned, unsigned> >`

The TL0/TL1 master adapter is connected through these ports (*sc_port*)

- sc_in_clk clk: the clock for the OCP connection being adapted
- OCP_TL1_MasterPort<TdataCl_tl1> MasterP the TL1 master port
- OCP2_TL0_SlavePorts: the base class for the TL0 port set (see 2.3.1)

The constructor has the following arguments:

- sc_module_name name: the name of the module, can be a const char* string
- const OCP_TL0_PortFactory<OCP2_TL0_SlavePorts>&: a reference to a factory instance to create an instance of the users port set class (see following example)

- bool checkSetupTime: controls whether the adapter should check for late setup of the sampled TL0 signals. When true, signal transitions after the sample delay will be flagged as an error and simulation will be stopped. This argument is true by default.

```
OCP2_TL0_TL1_Master_Adapter<unsigned int, unsigned int> my_adapter(
"tl0tl1_master", OCP2_TL0_SlavePorts::Factory< my_tl0_ports >(), true );
```

where my_tl0_ports is a class as described in the example of 2.4.1.

2.5 TL0 Sampling Times

Sampling times within the adapters refer to the fraction of a clock cycle when TL0 signals are evaluated (see 2.2.2). A sample time is by definition a fraction, therefore a smaller time than one single clock period.

In order to enable the valid combinatorial dependencies listed in the OCP 2.0 specification, the sampling times of the adapters are divided into the following.

For the TL0/TL1 slave adapter (TL0 signals are those of a master interface)

- phase accept signals (SCmdAccept, SDataAccept)
- response signals (SResp and rest of response group)
- threadbusy signals (SThreadBusy, SDataThreadBusy)

These sampling times can be set explicitly by using the TL0/TL1 slave adapter's setSampleDelays method:

```
void setSampleDelays( const sc_time& acceptDelay, const
sc_time& responseDelay, const sc_time& threadBusyDelay );
```

For the TL0/TL1 master adapter (TL0 signals are those of a slave interface)

- request signals (MCmd and rest of request group)
- response accept signal (MRespAccept)
- threadbusy signal (MThreadBusy)

These sampling times can be set explicitly by using the TL0/TL1 master adapter's setSampleDelays method:

```
void setSampleDelays( const sc_time& requestDelay, const
sc_time& respAcceptDelay, const sc_time& threadBusyDelay );
```

The setSampleDelays() methods can be called after construction of the adapters until end of elaboration. Alternatively, all delays can be set automatically by the adapters when calling the setClockPeriod() method on each adapter.

```
void setClockPeriod( const sc_time& period );
```

This setting must match the period of the clock signal bound to the adapter's Clk port. When setClockPeriod is called, the sampling times are set as follows, based on the OCP specification's Level 2 timing guidelines.

For the TL0/TL1 slave adapter:

- phase accept signals: 75% of clock period
- response signals: 60% of clock period
- threadbusy signals: 10% of clock period

For the TL0/TL1 master adapter:

- request signals: 50% of clock period
- phase accept signals: 75% of clock period
- threadbusy signals: 10% of clock period

2.6 Usage

As mentioned in 2.3.2 and 2.4.2, the TL0/TL1 adapter classes are templated. This requires some precaution when instantiating the adapters in a design.

The implementation of the TL0/TL1 adapters in the layer adapter package is under:

- `src/tl0_tl1/ocp_tl0_tl1_slave_adapter.cpp`
- `src/tl0_tl1/ocp2_tl0_tl1_slave_adapter.cpp`
- `src/tl0_tl1/ocp_tl0_tl1_master_adapter.cpp`
- `src/tl0_tl1/ocp2_tl0_tl1_master_adapter.cpp`

When using gcc, the code for all templated functions and classes must be available when the code using the template is compiled. This means that the adapter's cpp file must be included when instantiating an adapter. We recommend the following approach.

- Refer to the adapter as a regular module in the design, simply including the corresponding header (.h) file.
- Create an extra C++ module named adapter AdapterInstances.cpp for example, containing the explicit instantiations of the required adapters. Make sure to link this module with your final program. This module should include the adapter's cpp file (not the header file) as shown in this example:

```
#include “oc2_tl0_tl1_slave_adapter.cpp”
template class
OCP2_TL0_TL1_Slave_Adapter<OCP_TL1_DataCl<unsigned,
unsigned> >;
```

3 TL1-TL2 Adapters

3.1 Design Principles

3.1.1 TL1 to TL2 Aggregation

The TL1/TL2 adapters need to aggregate the TL1 phases belonging to a burst into a corresponding TL2 transaction. A TL1 burst will be formed of multiple request and/or datahandshake phases and these need to be aggregated into a TL2 burst which can be expressed in a single TL1 request structure (OCPTL2RequestGrp<Td, Ta>). Conversely, TL1 responses to a burst need to be aggregated into a TL2 response structure (OCPTL2ResponseGrp<Td>). This pattern directs the creation of TL2 aggregator classes. An aggregator relies on a *next*() method that the adapter calls when the next TL1 phase in sequence is available. The aggregator class also provides a Boolean *finished*() method and a *get*() method returning the completed TL2 request or response.

Figure 6 illustrates the functioning of an aggregator with the example of aggregating TL1 requests. This would be the case in the TL1/TL2 master adapter. When the slave port of the adapter receives a TL1 request starting a new burst, a new aggregator is created for the burst with the new request information As the burst continues, new information is aggregated (byte enables, address for unknown sequences, data for writes) and when the adapter detects the end of the burst, the aggregator can finalize the TL2 request and the adapter is free to send it out through its master port. The aggregator can then be deleted.

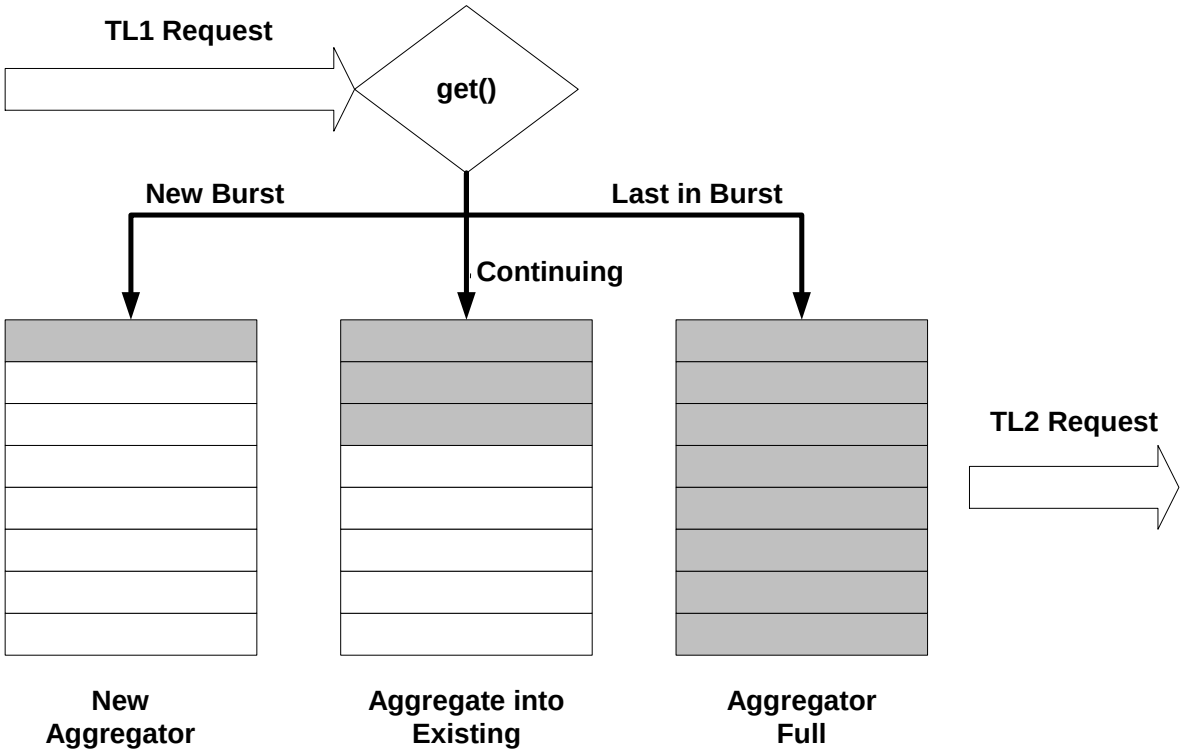

Figure 6: TL1-TL2 Aggregator Principle

3.1.2 Maximum Chunk Length

The previous example states that aggregators are used for entire bursts. This has an impact on the TL2 timing accuracy. For example, if a 16-long TL1 burst is sent to the TL1/TL2 master adapter, the corresponding TL2 request can not be issued until all 16 TL1 request phases, and datahandshake phases for a write are complete. This aggregation delay can be tuned with finer granularity by a construction time parameter setting the maximum chunk aggregation length. When an aggregator reaches its maximum chunk length, it is declared full and a new aggregator will be created when the next TL1 phase in the burst is received.

In Figure 7, a hypothetical TL1 burst of length 6 is shown with an adapter aggregating to a maximum length of 4 transfers per chunk. A new aggregator is created when the first request starts the burst until 4 requests are aggregated. A partial TL2 burst (LastOfBurst=false) of 4 can then be sent. A new aggregator is created again when the fifth request of the burst starts and it is used to aggregate the last 2 requests. At that point, the second aggregator is called full (burst complete) and the completing TL2 partial burst of 2 (LastOfBurst=true) can be sent.

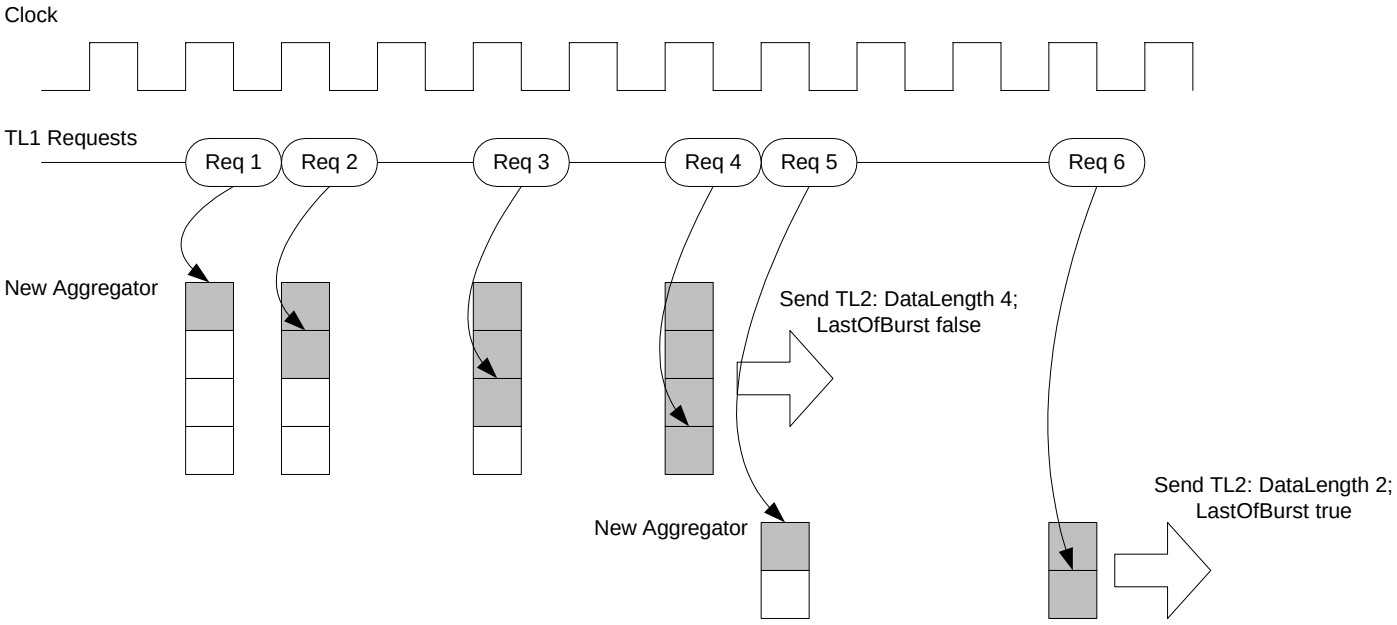

Figure 7: Maximum Chunk Aggregation

3.1.3 TL2 to TL1 Expansion

The dual dataflow of the adapter is to expand TL2 requests and responses into individual TL1 phases. A TL2 burst of length N needs to be expanded into 1 or N TL1 requests (depending on MBurstSingleReq) and N TL1 datahandshake phases if it is a write with datahandshake. Conversely, a TL2 response of length N needs to be expanded into 1 or N TL1 responses (1 for a single request write burst). This pattern directs the creation of TL1 expander classes. An expander class is constructed with a TL2 request or response and

provides a *next*() method that returns the following TL1 phase in sequence each time it is called. The *finished*() method informs the adapter that it no longer needs the expander.

Figure 8 illustrates the functioning of an expander with the example of expanding a TL2 burst request. This would be the case in the TL1/TL2 slave adapter. When the slave port of the adapter receives a new TL2 burst request, a new expander is created with the complete burst information. As the TL1 master port of the adapter gets ready to send a request, the adapter calls the expander's *next*() function to obtain a TL1 request to send out. When the expander has received the number of *next*() calls corresponding to the burst length, it is finished and the adapter can delete it.

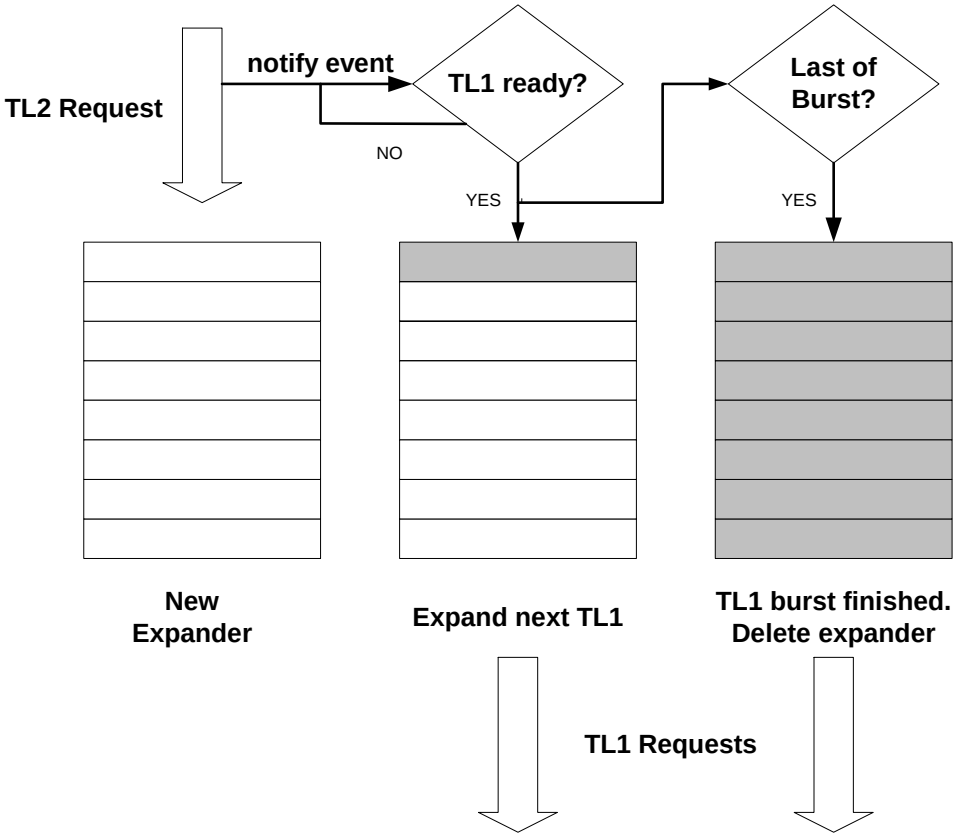

Figure 8: TL2-TL1 Expander Principle

3.2 TL1/TL2 Slave 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. The basic connectivity is shown in Figure 9.

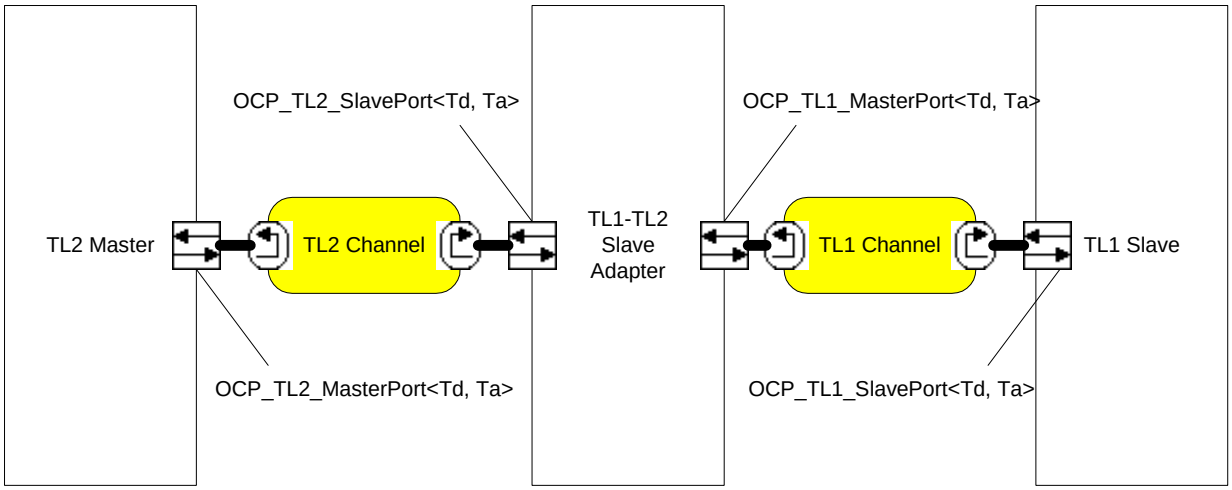

Figure 9: TL1/TL2 Slave Adapter Connectivity

3.2.1 Interface and Class Details

The header file for the TL1/TL2 slave adapter in the layer adapter package is under:

`include/tl1_tl2/ocp_tl1_tl2_slave_adapter.h`

The TL1/TL2 slave adapter is an *sc_module* with the following signature:

```
template <class TdataCl_tl1> class OCP_TL1_TL2_Slave_Adapter
```

where TdataCl_tl1 is the OCP TL1 channel data class:

```
OCP_TL1_DataCl<Td, Ta>
```

So an example of a completely resolved instantiation of the adapter is:

```
OCP_TL1_TL2_Slave_Adapter<OCP_TL1_DataCl<unsigned, unsigned> >
```

The TL1/TL2 slave adapter is connected through 3 ports (*sc_port*):

- `sc_in_clk clk`: the clock for the OCP connection being adapted
- `OCP_TL1_MasterPort< TdataCl_tl1> MasterP` : the TL1 master port
- `OCP_TL2_SlavePort<Td, Ta> SlaveP` : the TL2 slave port

The constructor has the following arguments:

- `sc_module_name name`: the name of the module, can be a const char* string
- `int max_chunk_length`: the maximum number of TL1 responses being aggregated into a TL2 response.
- Additionally, 2 integer arguments: `max_burst_length` and `adapter_depth` are supported for backwards compatibility but they have no effect and need not be specified since they have a default value.

3.2.2 Operation

The TL1/TL2 slave adapter makes use of an aggregator class for responses: `TL1ResponseAggregator<Td, Ta>` and two expander classes, one for requests and one for datahandshake: `TL2RequestExpander<Td, Ta>` and `TL2RequestExpanderDh<Td, Ta>.`

The adapter can handle queuing of incoming TL2 requests and TL1 responses regardless of the backpressure exercised by the other side of the adapter. Therefore it requires the aggregators and expanders to be queued. For simplicity in tracking bursts, the adapter stores aggregator and expander queues for each OCP thread. The corresponding data members are:

- ```
  typedef deque<Tl2RequestExpander<Td, Ta> >
  Tl2RequestExpanderQueue;
  vector<Tl2RequestExpanderQueue>  m_tl2RequestExpanderQueues;
  ```
- ```
  typedef deque<Tl2RequestExpanderDh<Td, Ta> >
  Tl2RequestExpanderDhQueue;
  vector<Tl2RequestExpanderDhQueue> m_tl2RequestExpanderDhQueues;
  ```
- ```
  typedef deque<Tl1ResponseAggregator<Td, Ta> >
  Tl1ResponseAggregatorQueue;
  vector<Tl1ResponseAggregatorQueue>m_tl1ResponseAggregatorQueues
  ```

Two important SC_METHOD's let the adapter react to incoming requests and responses and push expanders and aggregators to the proper thread queue.

- SlaveRequest captures the incoming TL2 requests from the slave port and creates new expanders for TL1 request and datahandshake phases on the corresponding OCP thread.
- MasterResponse captures the incoming TL1 responses from the master port and aggregates them into a TL2 response aggregator on the corresponding OCP thread.

The expander and aggregator queues are then processed and popped by SC_THREAD functions for each OCP thread.

- MasterRequest obtains the next TL1 request from the first queued expander and sends TL1 requests through the master port.
- MasterDataHs obtains the next TL1 datahandshake phase from the first queued expander and sends TL1 datahandshake phases through the master port.
- SlaveResponse queries the first queued TL2 response aggregator for completeness, and when complete sends the TL2 response through the slave port.

When the corresponding TL1 request and datahandshake expanders are both finished, the adapter accepts the originating TL2 request.
  ```

3.3 TL1-TL2 Master Adapter

This 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. The basic connectivity is shown in Figure 10.

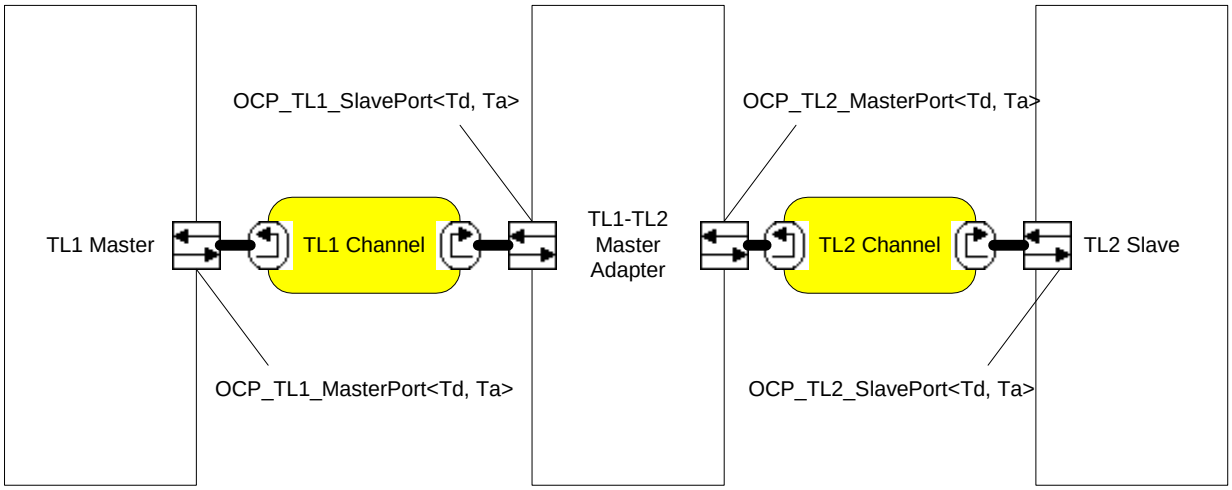

Figure 10: TL1/TL2 Master Adapter Connectivity

3.3.1 Interface and Class Details

The header file for the TL1/TL2 master adapter in the layer adapter package is under:

`include/tl1_tl2/ocp_tl1_tl2_master_adapter.h`

The TL1/TL2 master adapter is an *sc_module* with the following signature:

```
template <class TdataCl_tl1> class OCP_TL1_TL2_Master_Adapter
```

where TdataCl_tl1 is the OCP TL1 channel data class:

```
OCP_TL1_DataCl<Td, Ta>
```

So an example of a completely resolved instantiation of the adapter is:

```
OCP_TL1_TL2_Master_Adapter<OCP_TL1_DataCl<unsigned, unsigned> >
```

The TL1/TL2 master adapter is connected through 3 ports (*sc_port*):

- `sc_in_clk clk`: the clock for the OCP connection being adapted
- `OCP_TL2_MasterPort< TdataCl_tl1> MasterP` : the TL2 master port
- `OCP_TL1_SlavePort<Td, Ta> SlaveP` : the TL1 slave port

The constructor has the following arguments:

- `sc_module_name name`: the name of the module, can be a const char* string

- `int max_chunk_length`: the maximum number of TL1 requests being aggregated into a TL2 request.
- Additionally, 2 integer arguments: `adapter_depth` and `time_out` are supported for backwards compatibility but they have no effect and need not be specified since they have a default value.

3.3.2 Operation

The TL1/TL2 master adapter makes use of an aggregator class for requests: `TL1RequestAggregator<Td, Ta>` and an expander class for responses: `TL2ResponseExpander<Td>.`

The adapter can handle queuing of incoming TL1 requests and TL2 responses regardless of the backpressure exercised by the other side of the adapter. Therefore it requires the aggregators and expanders to be queued. For simplicity in tracking bursts, the adapter stores aggregator and expander queues for each OCP thread. The corresponding data members are:

- ```
  typedef deque<Tl1RequestAggregator<Td, Ta> >
  Tl1RequestAggregatorQueue;
  vector<Tl1RequestAggregatorQueue> m_tl1RequestAggregatorQueues;
  ```
- ```
  typedef deque<Tl2ResponseExpander<Td> >
  Tl2ResponseExpanderQueue;
  vector<Tl2ResponseExpanderQueue> m_tl2ResponseExpanderQueues;
  ```

Two important SC_METHOD's let the adapter react to incoming requests and responses and push expanders and aggregators to the proper thread queue.

- SlaveRequest captures the incoming TL1 requests from the slave port and aggregates them into a TL2 request aggregator on the corresponding OCP thread.
- MasterResponse captures the incoming TL2 responses from the master port and creates new expanders for TL1 responses on the corresponding OCP thread.

The expander and aggregator queues are then processed and popped by SC_THREAD functions for each OCP thread.

- MasterRequest queries the first queued TL2 request aggregator for completeness, and when complete sends the TL2 request through the master port.
- SlaveResponse obtains the next TL1 response from the first queued expander and sends TL1 responses through the slave port.

TL1 requests are accepted based on the statistical RqAL timing parameter of the TL2 channel, as this is meant to correspond to a single request acceptance and is more suitable than waiting for the TL2 slave to accept the complete transaction.

3.4 TL2 data allocation

The Tl2 request class OCPTL2RequestGrp<Td, Ta> requires allocated arrays to vehicle the MData, MDataInfo and MByteEn fields of all the transfers within the transaction.

Conversely, the TL2 response class OCPTL2ResponseGrp<Td> requires allocated arrays to vehicle SData and SDataInfo fields of all the responses within the transaction. For ease of use, the adapter manages the allocation of these arrays by allocating a large fixed array for each one of these uses. The size of the array can be configured at compile-time.

3.5 Sideband, ThreadBusy and Reset signals

Since both TL1 and TL2 channels offer an event for changes in each of the sideband group, the threadbusy signals and reset signals, the adapters are simply sensitive to the events on these signals or groups, and propagate the corresponding signal to the outgoing interface.

- The TL1/TL2 slave adapter propagates:
    - MFlag/MError from the slave TL2 port to the master TL1 port
    - SInterrupt from the master TL1 port to the slave TL2 port
    - MReset_n or SReset_n from the slave TL2 port to the master TL1 port
    - MReset_n or SReset_n from the master TL1 port to the slave TL2 port
    - SThreadBusy from the slave TL2 port to the master TL1 port
    - MThreadBusy from the master TL1 port to the slave TL2 port
- The TL1/TL2 master adapter propagates:
    - MFlag/MError from the slave TL1 port to the master TL2 port
    - SFlag/SError/SInterrupt from the master TL2 port to the slave TL1 port
    - MReset_n or SReset_n from the slave TL1 port to the master TL2 port
    - MReset_n or SReset_n from the master TL2 port to the slave TL1 port
    - SThreadBusy from the slave TL1 port to the master TL2 port
    - MThreadBusy from the master TL2 port to the slave TL1 port

3.6 Usage

3.6.1 Templating

As mentioned in 3.2.1 and 3.3.1, the TL1/TL2 adapter classes are templated. This requires some precautions when instantiating the adapters in a design.

The implementation of theTL1/TL2 adapters in the layer adapter package is under:

- `src/tl1_tl2/ocp_tl1_tl2_slave_adapter.cpp`
- `src/tl1_tl2/ocp_tl1_tl2_master_adapter.cpp`

When using gcc, the code for all templated functions and classes must be available when the code using the template is compiled. This means that the adapter's cpp file must be included when instantiating an adapter. We recommend the following approach.

- Refer to the adapter as a regular module in the design, simply including the corresponding header (.h) file.
- Create an extra C++ module named AdapterInstances.cpp for example containing the explicit instantiations of the required adapters. Make sure to link this module with your final program. This module should include the adapter's cpp file (not the header file) as show in this example:

```
#include "ocp_tl1_tl2_slave_adapter.cpp"
#include "ocp_tl1_tl2_master_adapter.cpp"
template class OCP_TL1_TL2_Slave_Adapter
<OCP_TL1_DataCl<unsigned, unsigned> >;
template class OCP_TL1_TL2_Slave_Adapter
<OCP_TL1_DataCl<sc_biguint<128>, unsigned> >;
template class OCP_TL1_TL2_Master_Adapter
<OCP_TL1_DataCl<unsigned, unsigned> >;
template class OCP_TL1_TL2_Master_Adapter
<OCP_TL1_DataCl<sc_biguint<128>, unsigned> >;
```

3.6.2 Additional Customization

The cpp implementation for each adapter contains 2 preprocessor definitions:

```
#define THREADS 16
```

As mentioned, the adapters create SC_THREAD functions for each OCP thread of the channel. Since the adapter does not know the configuration of its OCP interfaces until elaboration time, the choice was made to create a fixed number of SC_THREAD functions at construction time, and let the threads matching an out-of-range OCP thread number terminate upon first invocation. Increase this definition if you have more than 16 OCP threads in your configuration.

```
#define POOL_SIZE 1024
```

As mentioned in 3.4 the adapters maintain an array for the necessary TL2 allocated fields. Modify this number if more than 1024 words are necessary at a given time.