OCP Modelling Kit
User Manual

Robert Günzel (GreenSocs)
Herve Alexanian (Sonics)

April 16, 2009

DISCLAIMER

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

Supported OCP Versions

- 2.2

Contents

Chapter 1

Introduction

The OCP Modelling Kit provides a full interoperability standard for SystemC models of SOC components with OCP interfaces. The Kit is built on top of OSCI's TLM 2.0 technology, adding support for OCP protocol features and providing a wealth of support for code development and testing. All use cases for TLM modelling are supported, including verification, architecture exploration and software development.

The combination of a standard TLM interface for the OCP protocol, and the support code provided within the Kit, permits a major saving in development costs. It reduces the critical time interval between SOC specification availability and TLM model delivery. Models can be developed faster and better, reused more effectively, or sourced from external suppliers with confidence.

The Kit is a replacement for previous technology available from OCP-IP. This previous technology is now deprecated by OCP-IP. The motivation for replacing it with the OCP Modelling Kit is to provide compatibility with OSCI's TLM 2.0 technology. Using the OCP Modelling Kit, modules can be created that are fully interoperable with the OSCI TLM 2.0 *Base Protocol*, provided the OCP configuration allows this. This direct binding is only available at TL3. The Kit also includes adapters to enable binding between models using the legacy OCP-IP technology and models using this new kit.

Key features of the OCP Modelling Kit

- OCP Protocol Support
  - Versions 2.0, 2.1, 2.2 and 2.2.1 of OCP-IP supported in full
  - All OCP protocol features implemented using OSCI TLM 2.0 Generic Payload extensions
  - All OCP flow control options supported
  - OCP configuration management
    - May be hard-coded or supplied to a generic component model at run-time
    - Run-time resolution of master and slave OCP configurations
- Levels Of Abstraction Supported
  - Combined TL3 and TL4: inter-burst or no timing, equivalent to OSCI's Base Protocol
  - TL2: intra-burst timing
  - TL1: fully cycle-accurate, including support for clock cycle synchronization and combinatorial paths
- Content of Kit
  - Documentation
  - Examples
  - Performance and trace monitors
  - OCP TLM interoperability interface, including
    - TLM 2.0 extensions
    - Run-time OCP configuration resolution function
  - OCP master and slave sockets, providing
    - Memory management for extensions and payload objects
    - Payload event queues for timing annotation support or clock cycle synchronization

    - Convenience API for user code
    - Direct bind to OSCI TLM 2.0 sockets where functionally possible
  - Legacy adapters
  - RTL adapters

- Open Issues in the Kit
  - Polarity of `nonposted` extension is under review
  - TL2 implementation is not included in current release
  - API for use of extensions without a socket is not documented
  - Restrictions on OCP write response model for TL3 are under review
  - MReqInfo, MRespInfo and MDataInfo are not yet supported
  - Binding rules for multi-tagged and multi-threaded OCP interfaces are under review
  - OCP Reset not yet supported
  - Rules for support of streaming bursts at TL3 under review
  - This document is expected to grow significantly, including more details of use of the raw interoperability interface, deep dives into examples for each of TL1, TL2 and TL3, and so on.

Chapter 2

Basic Concepts of OCP TLM2

This chapter explains the basic concepts of the OCP Modelling Kit. It explains how to use the TLM-2.0 core interfaces to simulate OCP communication. It is strongly recommended to read the OSCI TLM-2.0 User Manual [1], and the Open Core Protocol Specification [2].

In general the rules and guide lines defined in [1] as the *base protocol* (BP) apply, but there are some restrictions and additions depending on the TL of the simulated OCP. The reasons for those additions are founded on the fact that the original OSCI TLM-2.0 kit's BP aims at TL3/4.

2.1 Simulating TL1 Communication

The rules and restrictions for TL1 are

1. OCP TL1 can use 4 different writes (with respect to phases)

   - Request phase with data
   - Request phase with data, and response phase
   - Request phase, and data handshake phase
   - Request phase, data handshake phase and response phase

   while the BP knows only a two phase write with a request and a response phase. Hence, within OCP TL1 exist the phases BEGIN_DATA and END_DATA when data handshake is used.

2. For a single OCP TL1 transaction there can be multiple phases of the same kind, while the BP only allows a single phase of a kind per transaction. The masters and slaves are obliged to emit/expect the number of phases (i.e. a BEGIN_X and a corresponding END_X timing point) per beat of the simulated burst as defined in [2]. See chapter 5 how to extract the OCP burst length from a transaction.

3. The BP is strictly sequential (for a single transaction), i.e. it does not allow phases to overlap, while different TL1 phases of a single transaction may overlap as defined in [2].

4. The BP allows to shortcut the protocol via TLM_COMPLETED while the TL1 protocol disallows the use of TLM_COMPLETED.

5. The BP allows to skip timing points, e.g. a BEGIN_RESP in return to BEGIN_REQ implies END_REQ. OCP TL1 does not allow that. It enforces the explicit use END_X, when a BEGIN_X has been received (the end may be sent either through the return or the fw/bw path).

6. The use of TLM_UPDATED is restricted to returning END_X to BEGIN_X. You may not return BEGIN_Y or END_Y to BEGIN_X.

7. The only allowed return to END_X is TLM_ACCEPTED.

8. For every BEGIN_X there has to be an END_X even if an OCP flow control is used that does not use XAccept signals, like thread_busy_exact or no flow control at all. Furthermore, when no accept flow control is use, the END_X must be returned immediately in response to a BEGIN_X via TLM_UPDATED.

9. In the presence of data handshake phases the data pointer in a write transaction may be uninitialized or NULL until the first BEGIN_DATA timing point.

10. In the presence of data handshake phases and with byteen=0 and mdatabyten=1 the byte enable pointer in a write transaction may be uninitialized or NULL until the first BEGIN_DATA timing point.

11. The data array can contain less bytes than the simulated transaction, i.e. the data length of the transaction may be less than ocp_burst_length*bus_width_in_bytes. See chapter 5 for more details.

12. The data array must be fully pre-allocated before the transaction starts, but it does not have to be fully pre-filled. The data array must contain at least the number of bytes so that the current beat can be successfully extracted from the array (but it may contain more). In other words the content of the data array is allowed to grow during the lifetime of the transaction. However, once a beat has been emitted the data associated with that beat may not change later on. See chapter 5 how to calculate the bytes of the data array that belong to a certain beat on a given point-to-point link.

13. The allocation and filling rules for the data array (see rule 12) do also apply to the byte enable array.

14. Synchronization takes place based on clock boundaries and/or TLM-2.0 interface method calls. If clocked cycle based synchronization is necessary communicating modules need to have exactly the same understanding of what a clock cycle is (see section 3.8 for more details).

15. Synchronization is achieved by time delays. Two modules A and B that are operating with the same real clock, may still be driven by different simulated clocks that may even appear in different delta cycles of the same simulated point of time. In other words, in the absence of additional timing information one must wait at least sc_time_resolution() to be sure to have received all transport calls for the current cycle. See section 3.8 for more details.

16. To signal thread busy changes a target may act as an initiator and emit a transaction allocated by the target (socket). A special phase THREAD_BUSY_CHANGE is used both by initiators and targets to transmit the thread busy change information. See section 4.7.3 for more information.

17. For imprecise bursts, the data length of the transaction is meaningless. A burst length extension has to be used to signal the end of the burst as defined in [2]. However, the rules 12 and 13 still apply. That means that before starting the burst the master has to pre allocate both a data and a byte enable array large enough to hold the complete burst. The assumption here is that there is always a known upper bound for the number of beats of an imprecise burst.

18. All dataflow signals of the OCP (see [2]) have a mapping on TLM-2.0 extensions or phases as defined in chapter 4.

2.2 Simulating TL2 Communication

2.3 Simulating TL3 Communication

The rules and restrictions for TL3 are

1. The OCP TL3 is designed to have maximum interoperability with OSCI BP. To this end all return codes, phase skipping etc. are supported.

2. OCP TL3 can use 2 different writes (with respect to phases)

   - Request phase
   - Request phase, and response phase

   while the BP always uses writes with responses, OCP TL3 will only do so if configured with writeresp=1.

3. Without any mandatory extensions, with write responses, and with accept flow control for both request and response OCP TL3 **matches the OSCI BP**.

4. For every BEGIN_X there has to be an END_X even if the OCP is configured not to use accept flow control. Note that thread busy flow control is not supported for TL3.

5. The data array organization depends on the used burst sequence as described in section 5. Note that the sequences INCR, WRAP, and XOR are indistinguishable in OCP TL3. The appropriate extension (cmp. chapter 4) may only be used as a hint to adapters.

6. Imprecise bursts are of no meaning to TL3, as a transaction is finished in one shot, hence must have a well known length at the very beginning. The appropriate extension (cmp. chapter 4) may only be used as a hint to adapters.

7. SRMD bursts are of no meaning to TL3, as a transaction is finished in one shot, hence has only a single request phase anyway. The appropriate extension (cmp. chapter 4) may only be used as a hint to adapters.

8. Further information about other extension can be found in chapter 4.

9. Synchronization is only achieved via TLM-2.0 interface method calls. Two connected modules do not need to have the same understanding of clock cycles or advance of time at all, as long as they obey the timing rules for the TLM-2.0 standard.

10. There are no data handshake phases in TL3.

Chapter 3

Using the Sockets

3.1 Directory Structure

The OCP Modelling Kit release is a header only release. Different releases of the kit can be installed in parallel. Assuming that releases 2.2.0 and 2.2.1 are installed the directory structure will look like shown in figure 3.1.

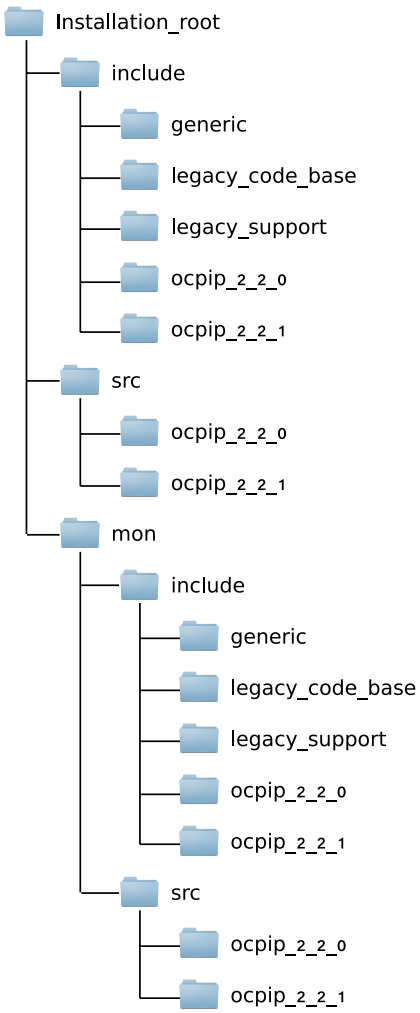

Figure 3.1: Sample directory structure of the OCP Modelling Kit

The include directory directly contains the files that shall be included by the user: ocpip.h, ocpip_2_2_0.h, and ocpip_2_2_1.h (see section 3.2 on how to use them). The subdirectory generic contains files that are shared by all releases, the subdirectory legacy_code_base contains the sources of OCP-IP SLD r2.2.1, and the subdirectory legacy_support contains the inclusion wrappers needed to compile legacy IP (see chapter 6 for details). The

release version tagged directories ocpip_X_Y_Z contain all the header files for the OCP Modelling Kit release X.Y.Z. The headers in those directories shall never be included directly, only include the files mentioned in section 3.2.

The src directory contains (in release tagged subdirectories) the tpp and hpp files for the corresponding releases. They are included from the main include files and shall never be included directly.

Finally, the mon directory[1] contains the monitor package for the various releases. Its subdirectory structure matches the one of the installation root package as described above. Note that the main include files for the OCP Modelling Kit releases will include their monitor packages if available. There is no need to include any file of the mon directory manually.

Note that each release of the OCP Modelling Kit is shipped with the OCP-IP SLD r2.2.1 kit in it. This kit is considered stable and frozen, so there is no need to version tag it. Given the unlikely case that a bug is discovered in the OCP-IP SLD r2.2.1 kit, the installation process assumes that the most recent OCP TLM-2.0 release contains the best available legacy code base, hence installing an older release will *not* replace the legacy code base. Only the installation of a newer release of the OCP Modelling Kit will overwrite the legacy code base (this applies both to the socket package and the monitor package).

3.2 Inclusion and namespaces

To use the OCP Modelling Kit in general one files has to be included:

- ocpip.h

Afterwards the OCP sockets and TLM-2.0 extensions are available in the namespace ocpip. Caution is required when different versions of the OCP Modelling Kit are installed. The files above and the namespace ocpip always point to the most recent release of the kit (given that all installed version are installed in the same location[2]). To include a specific version of the kit, the numbered versions of the include files must be used (ocpip_X_Y_Z.h). If the numbered versions are used, the sockets and extensions of this version are then available in the namespace ocpip_X_Y_Z, assuming the use of version X.Y.Z.

Each release contains a sub-namespace infr that encapsulates the infrastructure code (e.g. basic TLM-2.0 sockets) the OCP code is built upon. In general the user of the OCP kit will only be facing this namespace if he or she is using advanced features of the kit (like defining custom TLM extensions, or replacing the underlying infrastructure).

Every class or function mentioned in this document resides in namespace ocpip_X_Y_Z (or ocpip), which will not be explicitly mentioned. However, it will be explicitly stated whenever the sub-namespace infr has to be used.

Example The user installed releases 2.2.0 and 2.2.2 into location /foo/bar. Then the include path /foo/bar/include must be provided to the compiler. Afterwards the user can include

ocpip_2_2_2.h

which will make the release 2.2.2 available in namespace ocpip_2_2_2. He may also include

ocpip_2_2_0.h

which will make the release 2.2.0 available in namespace ocpip_3_0_0.

Afterwards he may use sockets of both releases by using the sockets of the according namespaces.

Basically the user could also include

ocpip_.h

which will make the release 2.2.2 available in namespace ocpip. This is just a namespace remap, so ocpip=ocpip_2_2_2 applies. However, it is strictly not recommended to mix numbered include file versions with the remapped ones. The non-versionized include files should only be used if no other versionized files are included. The normal case should be to only use the non-versionized include files to make your IP always use the latest OCP Modelling Kit.

Note that even though different versions can be included within one simulation, you cannot directly connect a socket of version X.Y.Z to a socket of version A.B.C, because both versions have their own infrastructure code. However, you may use the infrastructure code of one release with the OCP standard code of another release, given that the release notes allow that. Otherwise, it just allows to have modules with sockets of different versions within one simulation and to attach sockets of those versions to it. Also note that in this case a manual deep copy of the transaction is required when going from a socket to one of a different version.

[1] Only available if the monitor package is installed

[2] This is the recommendation. Otherwise support for multiple version of the kit within one simulation is not ensured.

3.3 Constructing the Sockets

There are socket versions for each TL. To simplify means we will use tlx as a placeholder for tl1, tl2 and tl3. If some things only apply to a specific TL it will be explicitly mentioned.

3.3.1 Master Socket

The master socket is defined as
template <**unsigned int** BUSWIDTH=32, **unsigned int** NUM_BINDS=1> **class** ocp_master_socket_tlx .
The first template argument defines the bus width in bits, and two OCP sockets can only be bound if their BUSWIDTH parameters match. The second template argument specifies the number of bindings allowed for the given socket. Note that NUM_BINDS=0 means an unlimited number of bindings is allowed for this socket. There is one constructor available for all TLs:

---

ocp_master_socket_tlx (**const char*** name,
ocp_master_socket_tlx :: allocation_scheme_type scheme=ocp_master_socket_tlx::mm_txn_only())

name The name of the socket.

scheme The allocation scheme used by the transaction pool inside the socket (see section 3.7).

Additionally there is a special constructor for TL1 sockets:

ocp_master_socket_tl1 (**const char*** name, MODULE* owner, **void** (MODULE::*timing_cb)(ocp_tl1_slave_timing),
ocp_master_socket_tl1 :: allocation_scheme_type scheme=ocp_master_socket_tl1:: mm_txn_only())

name The name of the socket.

owner A pointer to an object that owns a member function with the signature **void** fn(ocp_tl1_slave_timing) .

timing_cb A member function pointer to a member function of the object pointed to by owner with the given signature[3].

scheme The allocation scheme used by the transaction pool inside the socket (see section 3.7).

3.3.2 Slave Socket

The slave socket is defined as
template <**unsigned int** BUSWIDTH=32, **unsigned int** NUM_BINDS=1> **class** ocp_slave_socket_tlx .
The first template argument defines the bus width in bits, and two OCP sockets can only be bound if their BUSWIDTH parameters match. The second template argument specifies the number of bindings allowed for the given socket. Note that NUM_BINDS=0 means an unlimited number of bindings is allowed for this socket. There is one constructor available for all TLs:

---

ocp_slave_socket_tlx (**const char*** name)

name The name of the socket.

Additionally there is a special constructor for TL1 sockets:

ocp_slave_socket_tl1 (**const char*** name, MODULE* owner, **void** (MODULE::*timing_cb)(ocp_tl1_master_timing),

name The name of the socket.

owner A pointer to an object that owns a member function with the signature **void** fn(ocp_tl1_master_timing) .

timing_cb A member function pointer to a member function of the object pointed to by owner with the given signature[3].

---

[3]For more information about this callback and its use, first see section 3.5.3, then section 3.8

3.3.3 Binding two Sockets

To connect an OCP master with an OCP slave, the master socket of the former has to be bound to the slave socket of the latter. It is not allowed to bind the slave socket to the master socket. Additionally, the sockets can be bound hierarchically, so that sockets of submodules can be 'forwarded' to the boundaries of the owning modules. Examples are:

- **Master socket to slave socket binding**

  Assuming the master is called mst, its socket is called sock, the slave is called slv, and its socket is also called sock, binding the two is achieved by: mst.sock(slv.sock);

- **Master socket hierarchical binding**

  Assuming there is master module class called top_master that owns a submodule instance called sub_master. The top_master can forward the socket sub_sock of the sub_master to its boundaries, by binding it to its own socket top_sock. That will effectively make sub_sock and top_sock the *same* socket:

```
1 //ctor
2 top_master(sc_core::sc_module_name name) : ..., top_sock("sock"), ...
3 {
4   //forward sub_sock of sub_master to my own top_sock
5   sub_master.sub_sock(this->top_sock);
6 }
```

- **Slave socket hierarchical binding**

  Assuming there is slave module class called top_slave that owns a submodule instance called sub_slave. The top_slave can forward the socket top_sock from it boundaries to the socket sub_sock of the sub_slave, by binding it to its own socket top_sock to the sub_sock. That will effectively make sub_sock and top_sock the *same* socket[4] :

```
1 //ctor
2 top_slave(sc_core::sc_module_name name) : ..., top_sock("sock"), ...
3 {
4   //forward my own top_sock to sub_sock of sub_slave
5   this->top_sock(sub_slave.sub_sock);
6 }
```

3.4 Communication Interface

The communication interface is the TLM-2.0 interface. More precisely, for TL1,2 and 3 the non-blocking (nb_transport_fw/bw) interface shall be used, while for TL4 the blocking interface (b_transport) shall be used. The direct memory interface and the debug interface are of course part of the OCP sockets, but lie fully within user responsibility. In other words, the provided kit does not offer any further support for DMI and debug interfaces apart from allowing to register callbacks and call the appropriate TLM-2.0 functions.

It is very important to note that the signatures of the callbacks differ depending wether the used socket may be bound to exactly one socket (NUM_BINDS==1, hereinafter *a single socket*), or two more than one socket (NUM_BINDS!=1, hereinafter *a multi socket*). See below for more details.

3.4.1 Master Socket

A module owning an OCP Modelling Kit socket can both call TLM-2.0 functions on its socket and receive calls from its socket. Calling TLM-2.0 functions is exactly the same a described in [1]. In other words, use operatpor->() when performing interface method calls on a single socket, and operator[](unsigned int index) when performing interface method calls on a multi socket. In the latter case, index determines which bound socket will receive the interface method call (cmp. multiply bound sc_port).

The API to register callbacks is defined as listed below. Each API function is listed twice, once with the signature it has with a single socket, and once with the signature it has with a multi socket.

Single socket: `template<typename MODULE>`

```
void register_nb_transport_bw ( MODULE* mod,
  tlm::tlm_sync_enum (MODULE::*cb)(tlm::tlm_generic_payload&, tlm::tlm_phase&, sc_core::sc_time&));
```

[4]Note that the hierarchical bindings for master and slave sockets work inversely. As a hint: The master socket's hierarchical binding works like hierarchical bindings of sc_port, while the hierarchical binding of slave sockets work like hierarchical bindings of sc_export

Multi socket: `template<typename MODULE>`

```
void register_nb_transport_bw ( MODULE* mod,
  tlm::tlm_sync_enum (MODULE::*cb)(unsigned int,
    tlm::tlm_generic_payload&, tlm::tlm_phase&, sc_core::sc_time&));
```

mod An object offering the member function that is passed as the second argument.

cb For single sockets: A member function that matches the signature for nb_transport as defined in [1].

For multi sockets: A member function that matches the signature for nb_transport as defined in [1], extended by an additional leading unsigned int, that identifies from which rank of a multi socket the call was received.

Semantic Register a callback that will be called whenever the socket gets a call to nb_transport_bw from the slave. If called on a ocp_master_socket_tl1 a PEQ will be automatically inserted, that ensures all callbacks happen in sync with the simulation time, hence the sc_time argument will always be SC_ZERO_TIME. For all other TLs no PEQ will be inserted.

Single sockets: `template<typename MODULE>`

```
void register_nb_transport_bw ( MODULE* mod,
  tlm::tlm_sync_enum (MODULE::*cb)(tlm::tlm_generic_payload&, tlm::tlm_phase&, sc_core::sc_time&),
  bool use_peq);
```

Multi sockets: `template<typename MODULE>`

```
void register_nb_transport_bw ( MODULE* mod,
  tlm::tlm_sync_enum (MODULE::*cb)(unsigned int,
    tlm::tlm_generic_payload&, tlm::tlm_phase&, sc_core::sc_time&),
  bool use_peq);
```

mod An object offering the member function that is passed as the second argument.

cb For single sockets: A member function that matches the signature for nb_transport as defined in [1].

For multi sockets: A member function that matches the signature for nb_transport as defined in [1], extended by an additional leading unsigned int, that identifies from which rank of a multi socket the call was received.

use_peq If true a PEQ will be inserted that syncs the callback with the simulation time, hence the sc_time argument will always be SC_ZERO_TIME.

Semantic Register a callback that will be called whenever the socket gets a call to nb_transport_bw from the slave.

Single socket: `template<typename MODULE>`

```
void register_invalidate_direct_mem_ptr (MODULE* mod,
  void (MODULE::*cb)(sc_dt::uint64, sc_dt::uint64));
```

Multi socket: `template<typename MODULE>`

```
void register_invalidate_direct_mem_ptr (MODULE* mod,
  void (MODULE::*cb)(unsigned int, sc_dt::uint64, sc_dt::uint64));
```

mod An object offering the member function that is passed as the second argument.

cb For single sockets: A member function that matches the signature for invalidate_direct_mem_ptr as defined in [1].

For multi sockets: A member function that matches the signature for invalidate_direct_mem_ptr as defined in [1], extended by an additional leading unsigned int, that identifies from which rank of a multi socket the call was received.

Semantic Register a callback that will be called whenever the socket gets a call to invalidate_direct_mem_ptr from the slave.

Note that a runtime error will occur if the slave calls a function for which the master has not registered a callback.

3.4.2 Slave Socket

A module owning an OCP Modelling Kit socket can both call TLM-2.0 functions on its socket and receive calls from its socket. Calling TLM-2.0 functions is exactly the same a described in [1]. In other words, use operatpor->() when performing interface method calls on a single socket, and operator[](unsigned int index) when performing interface method calls on a multi socket. In the latter case, index determines which bound socket will receive the interface method call (cmp. multiply bound sc_port).

The API to register callbacks is defined as listed below. Each API function is listed twice, once with the signature it has with a single socket, and once with the signature it has with a multi socket.

Single socket: `template<typename MODULE>`

```
void register_nb_transport_fw ( MODULE* mod,
  tlm::tlm_sync_enum (MODULE::*cb)(tlm::tlm_generic_payload&, tlm::tlm_phase&, sc_core::sc_time&));
```

Multi socket: `template<typename MODULE>`

```
void register_nb_transport_fw ( MODULE* mod,
  tlm::tlm_sync_enum (MODULE::*cb)(unsigned int,
    tlm::tlm_generic_payload&, tlm::tlm_phase&, sc_core::sc_time&));
```

mod An object offering the member function that is passed as the second argument.

cb For single sockets: A member function that matches the signature for nb_transport as defined in [1].

For multi sockets: A member function that matches the signature for nb_transport as defined in [1], extended by an additional leading unsigned int, that identifies from which rank of a multi socket the call was received.

Semantic Register a callback that will be called whenever the socket gets a call to nb_transport_fw from the master. If called on a ocp_master_socket_tl1 a PEQ will be automatically inserted, that ensures all callbacks happen in sync with the simulation time, hence the sc_time argument will always be SC_ZERO_TIME.

Single socket: `template<typename MODULE>`

```
void register_nb_transport_fw ( MODULE* mod,
  tlm::tlm_sync_enum (MODULE::*cb)(tlm::tlm_generic_payload&, tlm::tlm_phase&, sc_core::sc_time&),
  bool use_peq);
```

Multi socket: `template<typename MODULE>`

```
void register_nb_transport_fw ( MODULE* mod,
  tlm::tlm_sync_enum (MODULE::*cb)(unsigned int,
    tlm::tlm_generic_payload&, tlm::tlm_phase&, sc_core::sc_time&),
  bool use_peq);
```

mod An object offering the member function that is passed as the second argument.

cb For single sockets: A member function that matches the signature for nb_transport as defined in [1].

For multi sockets: A member function that matches the signature for nb_transport as defined in [1], extended by an additional leading unsigned int, that identifies from which rank of a multi socket the call was received.

use_peq If true a PEQ will be inserted that syncs the callback with the simulation time, hence the sc_time argument will always be SC_ZERO_TIME.

Semantic Register a callback that will be called whenever the socket gets a call to nb_transport_fw from the master.

Single socket: `template<typename MODULE>`

```
void register_b_transport (MODULE* mod,
  void (MODULE::*cb)(transaction_type&, sc_core::sc_time&));
```

Multi socket: template<typename MODULE>

void register_b_transport (MODULE* mod,

void (MODULE::*cb)(unsigned int, transaction_type&, sc_core::sc_time&));

mod An object offering the member function that is passed as the second argument.

cb For single sockets: A member function that matches the signature for b_transport as defined in [1].

For multi sockets: A member function that matches the signature for b_transport as defined in [1], extended by an additional leading unsigned int, that identifies from which rank of a multi socket the call was received.

Semantic Register a callback that will be called whenever the socket gets a call to b_transport from the master.

Single socket: template<typename MODULE>

void register_transport_dbg (MODULE* mod,

unsigned int (MODULE::*cb)(transaction_type& txn));

Multi socket: template<typename MODULE>

void register_transport_dbg (MODULE* mod,

unsigned int (MODULE::*cb)(unsigned int, transaction_type& txn));

mod An object offering the member function that is passed as the second argument.

cb For single sockets: A member function that matches the signature for transport_dbg as defined in [1].

For multi sockets: A member function that matches the signature for transport_dbg as defined in [1], extended by an additional leading unsigned int, that identifies from which rank of a multi socket the call was received.

Semantic Register a callback that will be called whenever the socket gets a call to transport_dbg from the master.

Single socket: template<typename MODULE>

void register_get_direct_mem_ptr (MODULE* mod,

bool (MODULE::*cb)(transaction_type& txn, tlm::tlm_dmi& dmi));

Multi socket: template<typename MODULE>

void register_get_direct_mem_ptr (MODULE* mod,

bool (MODULE::*cb)(unsigned int, transaction_type& txn, tlm::tlm_dmi& dmi));

mod An object offering the member function that is passed as the second argument.

cb For single sockets: A member function that matches the signature for get_direct_mem_ptr as defined in [1].

For multi sockets: A member function that matches the signature for get_direct_mem_ptr as defined in [1], extended by an additional leading unsigned int, that identifies from which rank of a multi socket the call was received.

Semantic Register a callback that will be called whenever the socket gets a call to get_direct_mem_ptr from the master.

Note that a runtime error will occur if the slave calls a function for which the master has not registered a callback.

3.5 Configuring the Sockets

The sockets can be configured with respect to its OCP configuration, its PEQ utilization and its TL1 timing sensitivity. The former has to be done for each socket, the latter two are optional.

3.5.1 OCP Configuration

An OCP socket must have a valid OCP configuration at the end of construction. In general that is done by assigning an instance of the OCP parameters set to the socket that shall be configured. The OCP parameters are evaluated by two connected sockets when they get bound. If the OCP parameter sets are incompatible, runtime errors will appear. Hence prior to the actual configuration API the OCP parameters shall be described.

OCP Parameters Class

The configuration of a socket is captured in the ocp_parameters class. The ocp_parameters class is defined as

```
1
2  typedef std::map<std::string, std::string> map_string_type;
3
4  class ocp_parameters
5  {
6  public:
7    // Constructor
8    ocp_parameters();
9
10   // The parameter myParamValue is set only if the parameter is found in the
11   // configuration map and if it is of type 'i' for integer.
12   static bool
13   getBoolOCPConfigValue(
14     const std::string& myPrefix,
15     const std::string& myParamName,
16     bool &myParamValue,
17     map_string_type& Map,
18     std::string my_name="");
19
20   // The parameter myParamValue is set only if the parameter is found in the
21   // configuration map and if it is of type 'i' for integer.
22   static bool
23   getIntOCPConfigValue(
24     const std::string& myPrefix,
25     const std::string& myParamName,
26     int &myParamValue,
27     map_string_type& Map,
28     std::string my_name="");
29
30   // The parameter myParamValue is set only if the parameter is found in the
31   // configuration map and if it is of type 's' for std::string.
32   static bool
33   getStringOCPConfigValue(
34     const std::string& myPrefix,
35     const std::string& myParamName,
36     std::string& myParamValue,
37     map_string_type& Map,
38     std::string my_name="");
39
40   void set_ocp_configuration(std::string ocp_name, map_string_type& passedMap);
41
42   //this function dumps the whole parameter set into a string
43   std::string to_string ()const;
44
45   //this function compares ocp parameter set provided as the function argument to
46   // the ocp parameters set on which the function is called
47   // It returns true if there is a difference
48   bool diff(const ocp_parameters& other);
49
50   // OCP parameters
51   float ocp_version;
52   std::string name;
53   //an entry per OCP configuration parameter as defined in the OCP Specification
54   // the names are exact matches of those in the OCP Specification
55   ...
56 };
```

To create an ocp_parameters class it must be instantiated, and afterwards the members of the instance can be set, since they are all public members. For example if the instance was called my_params, then my_params.byteen=true, would set the configuration parameter that enables byte enables. Note that it is advised to set the member name of the parameters class, since it is used when there occur any problems with the parameter set, e.g. when binding to another socket.

Building the ocp_parameters class from a Parameter Map

The ocp_parameters class may also be created using a map object that contains all of the parameter settings. Use the function in line 40 to assign such a map to an instance of the ocp_parameters class.

The map object is a C++ Standard Template Library (STL) object that is an associative array. In this case, the map is string-to-string with the key string being the name of the parameter and the value string being the parameter value. This parameter map may be automatically generated by a configuration tool. It may be hand

coded in the source code for the master or slave, or in the main.cc program, or it may be built by reading in parameter data from a file.

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

The value side of the parameter map has the following format: type_char:value Where type_char is a single character is one of the following:

- "i" specifies an integer or Boolean
- "f" specifies a floating point value
- "s" specifies a string.

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

- "i:1" An integer value 1 or the Boolean value TRUE.
- "f:3.14159" The floating point value for PI.
- "s:little" The string value "little".

Building the Parameter Map from a File

The ocp_parameters class may also be configured by using a text file. Additionally this can be useful because the file name may be passed to the main program that builds the simulation. Also, the file name may be changed on the command line so the parameters are changed without having to recompile the model.

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

```
1 *sof config.txt*
2 cmdaccept i:1
3 addr_wdth i:40
4 endian s:both
5 *eof config.txt*
```

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

The Configuration API of the Sockets

The functions available are:

`void set_ocp_config (const ocp_parameters& config);`

config The OCP parameters class that contains the configuration for the socket.

Semantic Assign the provided set of parameters to the socket.

`ocp_parameters get_ocp_config () const;`

return value The OCP parameters class that contains the configuration that was provided to the socket via set_ocp_config .

Semantic Get the set of OCP parameters from a socket that was originally assigned to the socket.

`ocp_parameters get_resolved_ocp_config (unsigned int index=0) const;`

index The rank of the binding for which the resolved configuration shall be returned. Since the rank of a single socket is always zero, zero is provided as the default to allow single socket users to simply omit the index.

return value The OCP parameters class that contains the resolved configuration for the (given rank of the) socket. This may be different from what has originally been assigned to the socket, because during binding some tie offs can be performed.

Semantic Get the current set of OCP parameters from a socket.

---

void make_generic();

Semantic Tell the socket to accept every set of OCP parameters when being bound, and to adopt that set of parameters after binding. This call can only be used before the binding is complete, that means at construction or before_end_of_elaboration, but not later.

---

bool is_generic (**unsigned int**& index);

index A reference to an integer. If the function returns true, it will be set to the smallest index of a multi socket that is generic.

Semantic Ask the socket if it is (still) generic. After binding that will always return false, because the socket will adopt a configuration during bind, thereby seizing to be generic.

Single socket: **template** <**typename** MODULE>

void register_configuration_listener_callback (MODULE* owner,

void (MODULE::*set_config_cb)(**const** ocp_parameters&, **const** std::string&));

Multi socket: **template** <**typename** MODULE>

void register_configuration_listener_callback (MODULE* owner,

void (MODULE::*set_config_cb)(**const** ocp_parameters&, **const** std::string&, **unsigned int**));

owner A module providing the member function that is passed as the second argument.

set_config_cb A callback of the given signature.

Semantic Register the given callback with the socket. As soon as the binding of the socket has successfully completed the callback will be called. It will provide the resolved (i.e. with tie offs applied) set of parameters and the name of the socket who has just been bound. This enables to register the same callback with different sockets of the same module. In case of multi sockets, additionally the rank that has just been bound will be provided as a third argument.

3.5.2 PEQ Configuration

When registering an nb_transport callback for TL1 the default registration functions (see sections 3.4.1 and 3.4.2) will insert PEQs. For all other TLs the use of PEQs must be explicitly activated.

When PEQs are used the sockets allow to switch them into delta cycle protection mode. That means they will delay every incoming nb_transport call for an additional time resolution unit. Thanks to that, every module can know that at the time its clocked process executes no nb_transport for the current cycle has arrived yet. See section 3.8 for more details on that topic.

The API for the PEQ configuration is:

---

void activate_delta_cycle_protection ()

Semantic Activate the delta cycle protection mode of the PEQ within the socket. Note that calling this function if no PEQ is used will lead to a warning.

3.5.3 TL1 Timing Configuration

TL1 sockets offer the possibility to announce to their connected sockets if they use default timing[5] or not. See section 3.8 for a detailed discussion about TL1 timing.

To do so, timing information classes are exchanged. Those will be explained before the actual API can be shown.

[5] that means the execute nb_transport calls at the same simulation time at which the clock edge occurs

Master Timing Class

```
class ocp_tl1_master_timing {
  public:
    sc_core::sc_time RequestGrpStartTime;
    sc_core::sc_time DataHSGrpStartTime;
    sc_core::sc_time MThreadBusyStartTime;

    // default constructor for sc_core::sc_time makes SC_ZERO_TIME - this is "default timing"

    // test for equality
    bool operator == (const ocp_tl1_master_timing& rhs) const;
    bool operator != (const ocp_tl1_master_timing& rhs) const;

    static const ocp_tl1_master_timing& get_default_timing();
};
```

The class contains members that indicate at which time (after the clock edge has been seen) the request group (i.e. nb_transport with phase BEGIN_REQ), the data group (i.e. nb_transport with phase BEGIN_DATA), and the master's thread busy signal change (i.e. nb_transport with phase THREAD_BUSY_UPDATE and the type being M_THREAD) starts.

Additionally, there are comparison two operators. The first will return false as soon as one of the times doesn't match the times in the class that is compared. The second operates inverse to the first. The static function shall be used to get a master timing class that reflects the default timing (it can be used to test if a group matches the default timing or not).

Slave Timing Class

```
class ocp_tl1_slave_timing {
  public:
    sc_core::sc_time ResponseGrpStartTime;
    sc_core::sc_time SThreadBusyStartTime;
    sc_core::sc_time SDataThreadBusyStartTime;

    // default constructor for sc_core::sc_time makes SC_ZERO_TIME - this is "default timing"

    // test for equality
    bool operator == (const ocp_tl1_slave_timing& rhs) const;
    bool operator != (const ocp_tl1_slave_timing& rhs) const;
    static const ocp_tl1_slave_timing& get_default_timing();
};
```

The class contains members that indicate at which time (after the clock edge has been seen) the response group (i.e. nb_transport with phase BEGIN_RESP), the slave's thread busy signal change (i.e. nb_transport with phase THREAD_BUSY_UPDATE and the type being S_THREAD), and the slave's data thread busy signal change (i.e. nb_transport with phase THREAD_BUSY_UPDATE and the type being S_DATA_THREAD) starts.

Additionally, there are comparison two operators. The first will return false as soon as one of the times doesn't match the times in the class that is compared. The second operates inverse to the first. The static function shall be used to get a master timing class that reflects the default timing (it can be used to test if a group matches the default timing or not).

Master Timing Configuration API

The functions available are:

`void set_master_timing(const ocp_tl1_master_timing& my_timing, unsigned int index);`

my_timing The timing information that the master wants to announce to the slave.

index The rank of the multi socket binding to which to announce the timing information.

Semantic Announce a non default timing to a single connected slave sockets (identified via the rank of the binding). The call can be performed already in the constructor of the master, the socket will transmit the information as soon as it is bound to a slave socket.

`void set_master_timing(const ocp_tl1_master_timing& my_timing);`

my_timing The timing information that the master wants to announce to the slave.

Semantic Announce a non default timing to all connected slave sockets. For single sockets that is equivalent to using the previous function with index=0. The call can be performed already in the constructor of the master, the socket will transmit the information as soon as slave sockets are bound to the master socket.

Constructor with timing callback registration (see section 3.3.1)

Semantic Register a callback with the constructed socket that is called when some non-default timing is announced by the slave (see section 3.8.2 for more information).

Slave Timing Configuration API

The functions available are:

```
void set_slave_timing (const ocp_tl1_slave_timing & my_timing, unsigned int index);
```

my_timing The timing information that the slave wants to announce to the master.

index The rank of the multi socket binding to which to announce the timing information.

Semantic Announce a non default timing to a single connected master socket (identified via the rank of the binding). The call can be performed already in the constructor of the slave, the socket will transmit the information as soon as it is bound to a master socket.

```
void set_slave_timing (const ocp_tl1_slave_timing & my_timing);
```

my_timing The timing information that the slave wants to announce to the master.

Semantic Announce a non default timing to all connected master sockets. For single sockets that is equivalent to using the previous function with index=0. The call can be performed already in the constructor of the slave, the socket will transmit the information as soon master sockets are bound to the slave socket.

Constructor with timing callback registration (see section 3.3.2)

Semantic Register a callback with the constructed socket that is called when some non-default timing is announced by the master (see section 3.8.2 for more information).

It is important to note that the timing callbacks do **NOT** inform the owner of the callback from which multi socket rank the timing was received. The reason for that is that the timing distribution is not performed using TLM-2.0 interfaces, hence bypasses the sockets and thereby blurs the rank information. Additionally, always waiting for the latest signal will ensure stable signals on all ranks, so the rank information is not of great significance. Moreover, multi-sockets are used when there is a reasonable degree of symettry between them, which is why we do not need separate timing for each one).

3.6 Accessing Extensions

Please refer to chapter 4 and especially section 4.4 for detailed information about extension and how to use them.

3.7 Using the Memory Management of the Sockets

The OCP sockets offer memory management facilities to the user, the use of the transaction pool within the master sockets is strongly recommended, although not a strict requirement to use the sockets. However, when not using the pools memory management of the transactions is fully within user responsibility. The memory management of the extensions is a given, due to the nature of the extensions and the provided API for extension accesses. The user is never confronted with the need to allocate or deallocate extensions.

3.7.1 Transaction Memory Management

The API provided by the master socket for transaction memory management is:

---

tlm :: tlm_generic_payload * get_transaction ();

Semantic Get a memory managed transaction from the pool of a master socket. The transaction is already acquired on behalf of the master; there is no need to manually acquire this transaction.

---

void release_transaction (tlm :: tlm_generic_payload * txn);

Semantic Release a transaction that was previously taken from a pool of the same socket. The master shall call this function when he is done with the transaction.

Note that all other modules (Slaves, and modules that possess master sockets, but that do not use their pools because the only forward transactions from slave to master sockets and vice versa) shall use the transaction member functions acquire and release as described in [1].

3.7.2 Data and Byte Enable Array Memory Management

The pool within the master sockets knows four different operation modes that are encoded in an enumerated type. This type is provided as a typedef within the used OCP socket (see line 5 in the listing below), to allow changes to the type under the hood of the OCP kit. A mode for the pool is set through constructor parameter scheme of the master sockets (see section 3.3.1). The possible values are provided through static member functions of the OCP sockets (see lines 6 through 9 of the listing below).

```
1  template<unsigned int BUSWIDTH, unsigned int NUM_BINDS>
2  class ocp_master_socket_tlx
3  {
4  public:
5    typedef ... allocation_scheme_type;
6    static allocation_scheme_type mm_txn_only();
7    static allocation_scheme_type mm_txn_with_data();
8    static allocation_scheme_type mm_txn_with_be();
9    static allocation_scheme_type mm_txn_with_be_and_data();
10 };
```

The different semantics of the modes are:

mm_txn_only() The pool will only pool transactions. Data and byte enable arrays must be provided/managed by the master.

mm_txn_with_data() The pool will pool transactions and a data array for each of the transactions. The data array memory management functions (see below) will be enabled. Byte enable arrays must be provided/managed by the master.

mm_txn_with_be() The pool will pool transactions and a byte enable array for each of the transactions. The byte enable array memory management functions (see below) will be enabled. Data arrays must be provided/managed by the master.

mm_txn_with_be_and_data() The pool will pool transactions, a byte enable array and a data array for each of the transactions. The byte enable array and the data array memory management functions (see below) will be enabled.

Depending on the modes above none, one or both of the following memory management become available:

---

void reserve_data_size (tlm :: tlm_generic_payload & txn, **unsigned int** size);

txn The transaction for which to reserve a data array.

size The number of byte to reserve for the data array.

Semantic The socket will make the data_ptr of the transaction point to the data array that was pooled for this transaction. If the size is larger than the current size of the pooled array, the array will be enlarged accordingly. If it is larger than or equal to the size no allocation/deallocation will be performed. Additionally, the socket will set the data_length attribute of the transaction to the value of size. If the operation mode of the socket is not mm_txn_with_data() or mm_txn_with_be_and_data(), assertions will be triggered. However, when compiling with -DNDEBUG runtime errors might appear if the function is called on a socket whose pool is not set to the correct operation mode.

---

unsigned int get_reserved_data_size (tlm :: tlm_generic_payload &);

txn The transaction for which to determine that currently allocated array size.

Return value The number of bytes that are currently allocated.

Semantic The socket will return the size of the data array that is pooled for this transaction. This function is mainly for debug, but may prove helpful in some occasions. If the operation mode of the socket is not mm_txn_with_data() or mm_txn_with_be_and_data(), assertions will be triggered. However, when compiling with -DNDEBUG runtime errors might appear if the function is called on a socket whose pool is not set to the correct operation mode.

---

void reserve_be_size (tlm :: tlm_generic_payload & txn, **unsigned int** size);

txn The transaction for which to reserve a byte enable array.

size The number of byte to reserve for the byte enable array.

Semantic The socket will make the byte_enable_ptr of the transaction point to the byte enable array that was pooled for this transaction. If the size is larger than the current size of the pooled array, the array will be enlarged accordingly. If it is larger than or equal to the size no allocation/deallocation will be performed. Additionally, the socket will set the byte_enable_length attribute of the transaction to the value of size. If the operation mode of the socket is not mm_txn_with_be() or mm_txn_with_be_and_data(), assertions will be triggered. However, when compiling with -DNDEBUG runtime errors might appear if the function is called on a socket whose pool is not set to the correct operation mode.

---

unsigned int get_reserved_be_size (tlm :: tlm_generic_payload &);

txn The transaction for which to determine that currently allocated array size.

Return value The number of bytes that are currently allocated.

Semantic The socket will return the size of the byte enable array that is pooled for this transaction. This function is mainly for debug, but may prove helpful in some occasions. If the operation mode of the socket is not mm_txn_with_be() or mm_txn_with_be_and_data(), assertions will be triggered. However, when compiling with -DNDEBUG runtime errors might appear if the function is called on a socket whose pool is not set to the correct operation mode.

3.8 TL1 Timing

Level-1 of the OCP TLM model is designed to allow cycle-accurate modelling of bus interfaces. Any OCP traffic pattern that is possible in hardware should also be possible to model at TL1, without modifications to the design hierarchy or topology, and in a fully modular manner. This means that the TL1 infrastructure needs to support, among other things:

- Modules with internal combinatorial paths from one OCP signal to another within a single OCP interface
- Modules with internal combinatorial paths from an OCP signal on one interface to OCP signals on another interface
- Cascading of modules with OCP interfaces to an arbitrary degree
- Modules that change the values of OCP signals at some time in the middle of a clock cycle rather than at the clock edges, for example scaled-synchronous clock bridges

As OCP is a synchronous clocked protocol, to model it at a cycle-accurate level means that at very least the OCP master must understand the location of the clock cycles in time. In fact it is usual that the OCP slave also needs an understanding of the OCP clock cycles, and when both master and slave have this information, it must be the same for both of them[6], otherwise the connection will not work correctly. Furthermore, there may be one or more monitors attached to the connection, and these also need to be correctly synchronized with the OCP master. The section below attempts to explain what is meant by synchronization in this context. This is followed by a section describing how the OCP-TL1 timing information distribution system can be used to support non-default cases.

[6]as mentioned in section 2.1, rule 15, the 'same' only applies to the point in time a clock edge occurs, not necessarily to delta cycles

3.8.1 OCP TL1 Synchronisation

In the OCP protocol time is divided into clock cycles. Clock cycles are generally of a constant duration, the clock period, but this is not obligatory. In hardware, each clock cycle begins with a rising edge of a single-wire clock signal. The clock signal returns to zero some time during the cycle and the cycle ends when the following cycle begins, with the next rising edge.

In SystemC it is usual to define clock cycles in the same way, using an sc_channel of type sc_signal<bool> or the convenient library module sc_clock. SystemC allows many other ways of defining clock cycles and most ways are tolerated by the OCP Modelling Kit. However users are warned that exotic or unusual definitions of clock cycles will greatly reduce the chances of compatibility between modules. For every OCP Modelling Kit connection in a simulation, there are several other modules associated with it:

- Exactly one module with an OCP master socket, the *master*
- Exactly one module with an OCP slave port, the *slave* (which is allowed to be the same module as the master)
- Optionally one or more monitors

The master and slave may contain processes that access the connection. If so, these processes must be synchronized with each other, so that they understand the same clock cycle boundaries. As mentioned before that means they only have to know the point in time at which the clock edge appears. The actual clock edge event of the clocks that drive master and slave may happen at different delta cycles of that point of time. If any monitor is clocked, it must be clocked with the a clock whose clock edge happens at the same point of time as the ones used in the master and slave for OCP clock cycle synchronization (Again: the delta cycles may differ).

There are several cases where the modules do not need to understand the clock cycles. For example:

- An OCP slave can be fully event-driven. It can be implemented as a process which waits for the events triggered from within its nb_transport_fw callback function, then calls nb_transport_bw within the same clock cycle. This corresponds to a zero-latency (combinatorial) hardware module. Note that such a module is sensitive to the timing of the master and does not have default timing itself and as such it needs to use the timing information distribution system described below. In this case the master alone needs to understand the OCP clock cycle definition.
- A simple combinatorial bridge, for example a bridge to cut INCR burstsÕ lengths to some maximum value without introducing any latency, has both an OCP master socket and an OCP slave socket. It can be implemented without any processes. Its nb_tranport_fw callback will just modify slightly the transaction and then call nb_transport_fw on its master socket in the same cycle. Its nb_tranport_bw callback will work similarly, and modify slightly the transaction and then call nb_transport_bw on its slave socket in the same cycle. Note that such a module is sensitive to the timing of the external OCP master and slave, and does not have default timing itself and as such it needs to use the timing information distribution system described below. In this case the external OCP master and possibly the external OCP slave need to understand the OCP clock cycle definition.

All modules that do need to understand the clock cycle definition need to understand it identically. Note that:

- In case of default timing
  - In the absence of a delta cycle protection PEQ, a module must expect calls to nb_transport at any delta cycle of the time of the clock edge. More precisely, it must tolerate that nb_transport executes before and after the execution of the module's clocked processes.
  - In the presence of a delta cycle protection PEQ, a module can rely on the fact that all calls to nb_transport arrive at least one time resolution unit after the time of the clock edge. Hence, the module can expect nb_transport to execute only after the module's clocked process.
- In case of non default timing
  - In the absence of a delta cycle protection PEQ, a module must expect calls to nb_transport at any delta cycle of the non default time.
  - In the presence of a delta cycle protection PEQ, a module can rely on the fact that all calls to nb_transport arrive at least one time resolution unit after the non default time.

Assuming that nb_transport calls update the state of some module internal variables, that default timing, and delta cycle protection PEQs are used, we can say:

- that in many cases the master and slave modules can be implemented in a fully-synchronous style, having just a single process sensitive only to the clockÕs rising edge.

- Accesses at the time of the clock edge to the variables that are changed by nb_transport return the values of the previous cycle.

- Accesses at one time resolution unit after the time of the clock edge to the variables that are changed by nb_transport are unsafe[7] (there is a race between the module's internal process and the delta cycle protection PEQ's process)

- Accesses at two time resolution units after the clock edge to the variables that are changed by nb_transport return the values of the current cycle.

- Calls to nb_transport at a times unequal to the time of the clock are only allowed if it was previously announced via the timing information distribution as described below.

With non-default timing, and delta cycle protection, we can say:

- Accesses at the time of the clock edge to the variables that are changed by nb_transport return the values of the previous cycle.

- Accesses at or before at the non default time to the variables that are changed by nb_transport are unsafe[7], because it is not clear at which exact time the non-default timing call will happen.

- Accesses on time resolution unit after the non default time to the variables that are changed by nb_transport are unsafe[7] (there is a race between the module's internal process and the delta cycle protection PEQ's process)

- Accesses at two time resolution units after after the non default time to the variables that are changed by nb_transport return the values of the current cycle.

- Calls to nb_transport at a times unequal to the time of the clock are only allowed if it was previously announced via the timing information distribution as described below.

With default timing, and no delta cycle protection, we can say:

- Accesses at the time of the clock edge to the variables that are changed by nb_transport return are unsafe[7] (there is a race between the module's internal process and the sending process)

- Accesses at one time resolution unit after the time of the clock edge to the variables that are changed by nb_transport return the values of the current cycle.

- Calls to nb_transport at a times unequal to the time of the clock are only allowed if it was previously announced via the timing information distribution as described below.

With non-default timing, and no delta cycle protection, we can say:

- Accesses before the non default time to the variables that are changed by nb_transport are unsafe [7], because it is not clear at which exact time the non-default timing call will happen.

- Accesses at the non-default time to the variables that are changed by nb_transport are unsafe[7] (there is a race between the module's internal process and the sending process)

- Accesses at one time resolution unit after the non-default time to the variables that are changed by nb_transport return the values of the current cycle.

- Calls to nb_transport at a times unequal to the time of the clock are only allowed if it was previously announced via the timing information distribution as described below.

[7]Note that this can be safe if the user added appropriate means to make it safe. But without such means it is unsafe.

3.8.2 Timing Information Distribution (OCP TL1)

There are certain cases where TL1 models are unable to use only the clock period boundaries as their timing reference. The underlying reason for this is that the TL1 methodology recommended for OCP does not permit the retraction of either an OCP request or command accept, or the equivalents for data-handshake and response phases. These cases include:

- thread-busy-exact OCP interfaces, where the OCP protocol obliges the master (for sthreadbusy_exact) to choose its request only after having seen the SThreadBusy signals from the slave.
- a combinatorial request or response merger (arbiter), which needs to wait for a time long enough for all inputs to be stable before it chooses one of them. In particular where combinatorial OCP modules are cascaded some inputs may arrive later than others.

To allow simple management of such cases, a mechanism is provided in the OCP TL1 sockets which allows distribution of timing information at end-of-elaboration. Only OCP modules that are either "timing-sensitive" or "non-default-timing" need to use this mechanism. All other modules may ignore it completely.

Timing-sensitive Modules

A timing-sensitive module is a module which needs to know when inputs can safely be assumed to be stable, in order to work correctly (that means it needs to know when nb_transport with a certain phase is known to have been called or not). A non-timing-sensitive module might always use the values of the previous cycle, as a counter-example.

All OCP masters that are sthreadbusy_exact or sdatathreadbusy_exact are by definition timing-sensitive (unless they are using thread-busy-pipelined). All OCP slaves that are mthreadbusy_exact are by definition timing-sensitive (unless they are using thread-busy-pipelined). Timing-sensitive modules register themselves with the OCP TL1 sockets during construction.

They do this by using special constructors of the sockets (this ensure that the callbacks are registered after construction time):

- ```
  template<typename MODULE>
  ocp_master_socket_tl1 (const char* name, MODULE* owner,
    void (MODULE::*timing_cb)(ocp_tl1_slave_timing),
    ocp_master_socket_tl1 :: allocation_scheme_type scheme=ocp_master_socket_tl1:: mm_txn_only())
  ```
- ```
  template<typename MODULE>
  ocp_slave_socket_tl1 (const char* name, MODULE* owner,
    void (MODULE::*timing_cb)(ocp_tl1_master_timing))
  ```

depending on whether they are a master or a slave. Here it is suggested that a pointer to the module itself be passed as parameter. This would mean the module implement a function void fn(ocp_tl1_slave_timing) (for an OCP master) or void fn(ocp_tl1_master_timing) (for an OCP slave). If a module has multiple sockets of the same kind (master or slave socket), it may either have member variables of classes that implement such functions and registers a different variable with each socket, or it has a distinct function per socket.

Once the module is registered with the socket as timing-sensitive, the socket will inform it of the timing parameters of the module on the other side. This may happen several times depending on the order of the end-of-elaboration calls in the SystemC simulation. The implementation of the registered callbacks must allow it to be called multiple times during end-of-elaboration.

If the other side of the OCP TL1 connection is a default-timing module, the socket will never call the callback.

Non-default-timing Modules

A non-default-timing module is a module whose nb_transport calls are not performed at the time of the clock edge. If a clock signal is used to synchronise the OCP master and OCP slave, this means that default-timing modules call nb_transport at the time of the clock rising edge. Non-default timing modules must call the socket methods void set_master_timing(const ocp_tl1_master_timing& my_timing); or void set_slave_timing(const ocp_tl1_slave_timing& my_timing); (for masters and slaves respectively) during end-of-elaboration, providing their timing parameters.

A non-default timing module may not know its timing parameters when its own end-of-elaboration method is called. This is the case for example for a combinatorial module passing OCP requests from a slave port to a

master port (an address translation bridge for example). A module like this is both timing-sensitive and non-default-timing. It must register itself as timing-sensitive on its OCP slave socket and send its timing information to its OCP master socket. It may occur that the module is provided several times with timing information from the OCP slave socket, and every time that its master timing callback is called from the slave socket, it should recalculate the timing parameters of its master socket and call the set_master_timing() method of the master port if they changed.

To avoid infinite loops at end-of-elaboration it is important that a non-default-timing module only call set_x_timing() when necessary. It should not call this method if it has previously been called with the same parameters.

Start Times

Start times are sc_time variables. They indicate when a nb_transport call with a certain phase is given to the socket by the OCP master or slave. The other side of the OCP interface can safely access variables that are changed by that nb_transport call after waiting for the start-time and one time resolution unit. It is then sure that nb_transport call with that phase has happened (if at all) in this cycle and will not happen again.

Start times give duration of simulated time after the start of an OCP clock cycle. It is assumed that the OCP master and OCP slave are exactly synchronised.

- start_time = SC_ZERO_TIME

  This means that the nb_transport call is started at the same sc_time_stamp as the synchronisation event indicating the start of an OCP cycle. The other side of the OCP interface can access the values changed by that call safely after waiting one time resolution unit.

- start_time > SC_ZERO_TIME

  This means that the nb_transport call is started after wait(start_time) after the synchronisation event indicating the start of an OCP cycle. Or before. It is not allowed that the nb_transport call starts some time after wait(start_time). In this case the other side of the OCP interface must at least wait(start_time+sc_get_time_resolution()) before accessing values changed by the nb_transport call.

The most frequent example is a thread-busy-exact OCP. In the simplest case the slave produces SThreadBusy directly after the start of cycle. It has therefore default timing. The master must wait at least one time resolution unit before accessing the member that was updated by nb_transport with phase THREAD_BUSY_UPDATE and starting an OCP request. Therefore the OCP request start time is +1 time resolution unit.

OCP TL1 Timing Example

In the distribution there is an example of how the TL1 timing distribution feature of the OCP TL1 sockets can be used. It is a simulation of a multi-threaded non-blocking shared bus with zero-cycle minimum round-trip latency. In this design a request/response transfer can pass through up to 10 cascaded OCP Modelling Kit TL1 connections in the same clock cycle. For more details look in the source code and the readme.txt file, in the directory examples/supplementary/ocp_tl1_timing.

3.9 Helper Functions

Some helper functions and classes are provided within the kit. This section will list them.

3.9.1 Data Class

Mainly for the support of the legacy monitors a data_class is provided (see chapter 7). It can be regarded as a type generator that generates the most appropriate types for given bus and address widths. It exists in two flavors: once with an unsigned data type, and once with a signed data type.

```
1  template <unsigned int BUSWIDTH, unsigned int ADDRWIDTH>
2  struct ocp_data_class_unsigned{
3    ...
4    typedef ...  DataType; //unsigned in this class
5    typedef ...  AddrType; //always unsigned
6    ...
7  };
8
9  template <unsigned int BUSWIDTH, unsigned int ADDRWIDTH>
10 struct ocp_data_class_signed{
11   ...
12   typedef ...  DataType; //signed in this class
13   typedef ...  AddrType; //always unsigned
14   ...
15 };
```

If users are unsure which unsigned type fits best for a certain bus width may using ocp_data_class_unsigned :: DataType for that is a good idea.

3.9.2 Burst Length Calculation Functions

As can be seen in chapter 5 extracting the OCP burst length out of a given transaction (in the absence of the burst length extension) is not totally trivial. Hence helpers are provided that aid the user in this task.

---

unsigned int calculate_ocp_address_offset (tlm :: tlm_generic_payload & txn, **const unsigned int** bus_byte_width);

txn The transaction from which to extract the address offset.

bus_byte_width The bus width in bytes of the socket/link.

Semantic Return the offset of the transaction address (see section 5.1, equations 5.2 and 5.3 for a definition).

---

template <**typename** Ta> **unsigned int** calculate_ocp_burst_length (tlm :: tlm_generic_payload & txn
, **const unsigned int** bus_byte_width
, **unsigned int** offset
, **bool** burst_aligned_INCR
, Ta& new_ocp_address
, burst_sequence* b_seq)

txn The transaction for which to calculate the burst length

bus_byte_width The bus width in bytes of the socket/link.

offset The address offset of the transaction address

burst_aligned_INCR An indicator if the transaction is a burst aligned INCR burst.

new_ocp_address A reference to a variable that can store an address. Will only be updated when burst_aligned_INCR is true.

b_seq The pointer to the burst length extension of the transaction. May be NULL if not available.

Semantic Calculate the effective OCP burst length of a given transaction. If the transaction is a burst aligned incrementing burst, the calculation of the address is also provided by that function and the result is places into new_ocp_address. Whenever there is a valid burst sequence extension in the transaction, provide it to the function so that it can use it to correctly calculate the burst length.

Example: Calculate the burst length for an INCR burst that appeared on a link with BUSWIDTH=32, that has burst_aligned=1 and that can handle INCR, STRM and WRAP bursts:

```
1
2  tlm::tlm_sync_enum nb_transport(tlm::tlm_generic_payload& gp, tlm::tlm_phase& ph, sc_core::sc_time& t){
3    switch(ph){
4      case tlm::BEGIN_REQ:{
5        //get potential address offset
6        unsigned int offset=calculate_ocp_address_offset(gp, 4); //4 byte bus width
7        //we guess this is a STRM burst and put the bus width aligned address into our class member
8        m_address=gp.get_address()-offset;
9        //try to get the burst seq extension
10       ocpip::burst_sequence b_seq;
11       if (socket.get_extension(b_seq, gp)) m_current_seq=b_seq->value.sequence; //if valid remember it
12       else b_seq=NULL; //if not valid we neglect the pointer we got
13       //store the burst length in a class member
14       // if this is a INCR now our address might get changed because of burst_aligned=true
15       m_b_length= calculate_ocp_burst_length<long>(gp, 4, offset, true, m_address, b_seq);
16       ...
17     }
18     break;
19     ...
20   }
21 }
```

3.10 Adding Req/Resp/MData/SData-Info extensions

The OCP kit does not (yet) provide pre-defined extensions to transmit req/resp/mdata/sdata-info fields. The recommendation is that a user should define functional extensions for that manually. Future releases will provide more sophisticated help for that, offering pre-defined ways how to map the functional extension on bit maps and vice versa, so that the info fields can e.g. appear in traces generated by the trace monitor.

To simplify the definition of extensions and to make them fit into the memory management policies of the OCP kit, a set of C++ base classes are provided that help defining extensions. The most important ones are

- **namespace** infr{ **template** <**typename** T> **struct** ocp_guard_only_extension;}

  Derive an empty class or struct from this struct in the following way:

  struct my_guard_ext : **public** infr :: ocp_guard_only_extension<my_guard_ext>{};

  This will create a ready to use guard extension (see section 4.4).

- **namespace** infr{ **template** <**typename** T, **typename** VAL> **struct** ocp_single_member_data;}

  Derive an empty class or struct from this struct in the following way:

  struct my_data_ext : **public** infr :: ocp_single_member_data<my_data_ext, **unsigned int**>{};

  This will create a ready to use data extension (see section 4.4), with a single member named value, whose type is VAL (which is unsigned int in the given example).

  Note that for monitoring purposes std::ostream& operator<<(std::ostream &, const member_type&) has to be defined for VAL.

- **namespace** infr{ **template** <**typename** T, **typename** VAL> **struct** ocp_single_member_guarded_data;}

  Derive an empty class or struct from this struct in the following way:

  struct my_guard_data_ext : **public** infr :: ocp_single_member_guarded_data<my_guard_data_ext, **int**>{};

  This will create a ready to use guarded data extension (see section 4.4), with a single member named value, whose type is VAL (which is int in the given example).

  Note that for monitoring purposes std::ostream& operator<<(std::ostream &, const member_type&) has to be defined for VAL.

There are other base classes that allow for a more fine grained definition of extension, which are documented in the source file ocp_infr_extensions.h that contains the macro definitions.

To illustrate the use of such a custom extension and how to use them to as e.g. a reqinfo extension, consider the following example: A master wants to decorate each read request with the consecutive number of the read transaction and the beat number of the request within this transaction. In this example this info will be transmitted as a std::string to illustrate that such info can be any abstract data type.

The extension is defined like this[8]:

```
1  //file req_info_extension.h
2  #ifndef __req_info_extension_h__
3  #define __req_info_extension_h__
4
5  #include "ocpip.h"
6
7
8  struct req_info_container{
9    std::vector<std::string> infos;
10 };
11
12 inline std::ostream& operator<< (std::ostream & os, const req_info_container & req_info_cont){
13   os<<"OStream serialization of req info container not supported";
14   return os;
15 }
16
17 struct my_req_info : public ocpip::infr::ocp_single_member_guarded_data<my_req_info, req_info_container>{};
18
19 #endif
```

The pseudo code for the extension 'generated' by this derivation is therefore:

```
1  struct my_req_info :
2    public ocp_tlm_guarded_data_extension
3  {
4    req_info_container value;
5  };
```

[8]To keep the example simple the stream operator << is not generating a reasonable output stream. Real life extensions should of course try to output information that aids debugging and monitoring.

Due to this structure you could now add more members to req_info_container, and still use the same simple base class ocp_single_member_guarded_data.

The master can now add reqinfo to requests like that:

```
if (req->get_command() == tlm::TLM_WRITE_COMMAND){
 //...
}
else{ //read
  //cnt is the overall count of emitted request beats
  unsigned int burstcnt=cnt&0x7;
  my_req_info* req_info;
  ipP.get_extension<my_req_info>(req_info, *req);
  std::stringstream s;
  s<<"This is the req info for beat "<<burstcnt
   <<" of read transaction no."<<((cnt>>4)+1)<<" from "<<sc_module::name();
  req_info->value.infos[burstcnt]=s.str();
}
```

And the slave can access the reqinfo like this:

```
  my_req_info* req_info;
  //reqcnt is the count of received requests for the current MRMD burst (starting at 1)
  if (tpP.get_extension<my_req_info>(req_info, *req)){
    //the req has an info field
    std::cout<<"Slave got info for a request!"<<std::endl
             <<"The info is:"<<std::endl
             <<"  "<<req_info->value.infos[reqcnt-1]<<std::endl;
  }
```

The whole example with all the surrounding code can be found in the ocp_tl1_imprecise_burst_profile_with_req_info directory of the examples shipped together with the OCP kit.

Chapter 4

TLM-2.0 extensions for OCP

This chapter will name all the TLM extension for OCP, and defines their mutabilities, bindabilities, phase associations, extension types and semantics. Prior to that mutability, bindability, phase association and extension types will be defined.

4.1 Phase association

A certain extension is always associated with a (set of) phase(s). It cannot be considered valid when receiving a phase it is not associated with.

4.2 Mutability

OCP TLM distinguishes between three different mutabilities: end-to-end invariant, point-to-point invariant, and point-to-point variant. The mutability of an extension determines if and when an extension may be changed as well as the validity.

1. End-to-end invariant (E2E)

   E2E extensions may be set once with the very first associated phase by an end-to-end module and may not be changed until the transaction's lifetime has finished. In other words, an E2E extension is temporally and spatially[1] invariant. It is valid with every subsequent associated phase of the transaction.

2. Point-to-point invariant (X2X)

   X2X extensions may be set/changed once with the very first associated phase by an end-to-end module or intermediate module and may not be changed later by the same module. That means, if an intermediate module receives a transaction for the first time with an associated phase it may change an X2X extension, but may not change it later on. Consequently, X2X extensions are only valid with the very first associated phase. The value cannot be considered valid in subsequent associated phases of the transaction. Hence, if a module needs access to an X2X extension in subsequent phases it has to make a local copy of it, e.g. it can put it into an instance specific extension. Note that this also applies to the original setter of the extension.

   More formally, an X2X extension is spatially variant, but temporally invariant on a given point-to-point link.

3. Point-to-point variant (P2P)

   P2P extensions may be set/changed with every associated phase by every module. They have to be evaluated/set with every reception/sending of an associated phase. They are valid only in the very call that delivers the transaction with an associated phase. Consequently, when delaying a transaction that carries a P2P extension, the value of the P2P extension must be saved within the function that delivers the transaction (the extension is valid only at this time), because the value inside the transaction may have changed during the delay. In case the transaction shall be passed on after the expiration of the delay with the P2P extension carrying the value it had at reception, the value must be copied back into the extension before it is passed on (we denote that as save-and-restore of an extension).

   More formally, a P2P extension is both temporally and spatially variant.

[1] The space of a transaction is the set of all point-to-point links it passes on its way from the master to the slave.

4.3 Bindability

OCP TLM distinguishes between three different *bindability levels* (BL): mandatory, optional and rejected. The BL of an extension is not defined for an extension alone. It can only be defined for an extension in conjunction with an OCP role[2]. Hence, for each BL we define *bindability rules* against other BLs.

The use of extensions and their respective BLs is extracted from the OCP configuration of a given socket.

1. Mandatory

   Meaning: The extension is vital for the correct operation of the module.

   Bindability rules: If the other module's BL for the extension is *rejected*, the bind attempt will fail. If the other module's BL for the extension is *optional* or *mandatory*, the bind attempt will succeed.

2. Optional

   Meaning: The module can operate correctly with or without the extension.

   The bind attempt will always succeed.

3. Rejected

   Meaning: The module is unable to handle the extension.

   If the other module's BL for the extension is *mandatory*, the bind attempt will fail. If the other module's BL for the extension is *optional* or *rejected*, the bind attempt will succeed.

4.4 Extension types

OCP TLM distinguishes between three (it's always three, isn't it?) extension types: guard extensions, data extension and guarded data extensions.

1. Guard extension

   A guard extension does not carry any value, the information carried by the extension is its mere extension in a transaction. Assuming there is a guard extension foo and a transaction txn, we call foo *validated* in txn if foo is part of txn. If it is not part of txn it is called *invalidated*.

   A transaction taken from a pool that is part of an OCP socket will always have all guard extensions invalidated.

   Since guard extensions do not carry values, there is no need to handle explicit objects of guard extension, only the type is of interest. The API provided by the OCP sockets for working with guard extensions is:

   (a) `template <typename EXT> bool validate_extension(tlm_generic_payload& txn);`
   This call will validate the guard extension type EXT in txn. The return value is true if there was a memory manager in txn, and false otherwise. In TL1 the return value can be ignored, since there has to be a memory manager inside a TL1 transaction. It might be of importance at higher abstraction layers that work with and without memory management. Depending on if there was a memory manager or not, the caller is responsible for invalidating the extension.
   Example:
   ```
   tlm_generic_payload txn;
   my_socket.validate_extension<foo>(txn);
   ```

   (b) `template <typename EXT> void invalidate_extension(tlm_generic_payload& txn);`
   This call will invalidate the guard extension type EXT in txn.
   Example:
   ```
   tlm_generic_payload txn;
   my_socket.validate_extension<foo>(txn);
   ```

   (c) `template <typename EXT> EXT* get_extension(tlm_generic_payload& txn);`
   This call will test if the guard extension type EXT is in txn. If so, it will return a non-NULL pointer, otherwise it will return the NULL pointer.
   Example:
   ```
   if (my_socket.get_extension<foo>(txn)) do_stuff_with_foo(); else do_stuff_without_foo();
   ```

[2] Master or Slave

2. Data extension

   A data extension carries a value whose validity is indeterminable. That means out of any given transaction the data can be extracted. The question whether this value can be trusted or not cannot be derived from the transaction itself, it must have been negotiated prior to the transaction exchange. E.g. if a protocol goo requires extension bar in any conceivable communication, bar can be a data extension if both communication partners agreed to talk goo only.

   The API provided by the OCP sockets for working with data extensions is:

   (a) **template** <**typename** EXT> EXT* get_extension(tlm_generic_payload& txn);
   This call always returns a valid pointer to the data extension of type EXT of txn. There is no way to determine whether someone else accessed the extension before. Note that any given transaction can be considered to carry all data extensions, i.e. the call will never return the NULL pointer. The extension is 'added' to the transaction by just calling this function and assigning a value to the extension.
   Example:
   my_socket.get_extension<foo>(txn)->value=3; //set the value of a data extension
   my_val=my_socket.get_extension<foo>(txn)->value; //get the value of a data extension

3. Guarded data extension

   A guarded data extension carries a value whose validity is determinable. That means out of any given transaction the data can be extracted. The question whether this value can be trusted or not can be derived from the transaction itself. A guarded data extension foo is basically two extensions: a data extension (denoted as the *data part of* foo) that carries the value and a guard extension (denoted as the *guard of* foo) that signals the validity, however the provided API will hide this distinction from the user.

   Assuming there is a guarded data extension foo and a transaction txn, we call foo *validated* in txn if the guard of foo is part of txn. If the guard is not part of txn it is called *invalidated*.

   A transaction taken from a pool that is part of an OCP socket will always have all guards of guarded data extensions invalidated.

   (a) **template** <**typename** EXT> **bool** validate_extension(tlm_generic_payload& txn);
   This call will validate the guard of guarded data extension type EXT in txn. The return value is true if there was a memory manager in txn, and false otherwise. In TL1 the return value can be ignored, since there has to be a memory manager inside a TL1 transaction. It might be of importance at higher abstraction layers that work with and without memory management. Depending on if there was a memory manager or not, the caller is responsible for invalidating the extension.
   Note that a guard of a guarded data extension may only be validated after the data part of the extension has been accessed before (not shown in the example below).
   Example:
   tlm_generic_payload txn;
   my_socket.validate_extension<foo>(txn);

   (b) **template** <**typename** EXT> **void** invalidate_extension(tlm_generic_payload& txn);
   This call will invalidate the guard of guarded data extension type EXT in txn.
   Example:
   tlm_generic_payload txn;
   my_socket.validate_extension<foo>(txn);

   (c) **template** <**typename** EXT> **bool** get_extension(EXT*& ext, tlm_generic_payload& txn);
   This call will test if the guard of guarded extension type EXT is in txn. If so, it will return true, otherwise it will return false. After the call ext will always have a valid pointer to the data part of guarded data extension type EXT. To 'add' a guarded data extension to a transaction just call this function (it should return false, otherwise it was already added and validated by someone else), assign a value to it and validate the extension.
   Example1 (setup):

```
foo* p_foo;
tlm_generic_payload txn;
//ignore return value, because we know it is false
//since txn is fresh and new
my_socket.get_extension<foo>(p_foo, txn);
p_foo->value=10;  //set the value
my_socket.validate_extension<foo>(txn); //mark the data as valid
```

Example2 (test):

```
foo* p_foo;
if (my_socket.get_extension<foo>(p_foo, txn))
  do_stuff_with_foo(p_foo->value);
else
  do_stuff_without_foo();
```

Please note that none of the types requires the user to allocate any extensions. Extensions can always be 'taken' from the transactions directly.

4.5 Extension List

OCP TLM uses several extensions. Using the definitions above, this section will now characterize all of them one by one. They appear in alphabetic order.

4.5.1 address_space

Extension type : guarded data

Definiton :

```
struct address_space :
  public ocp_tlm_guarded_data_extension
{
  unsigned int value;
};
```

Phase association : BEGIN_REQ

Mutability : X2X

Bindability :

OCP Abstraction layer	OCP Configuration parameter: addrspace	Bindability	
		Master	Slave
TL1	1	mandatory	optional
	0	optional	rejected
TL2	1	mandatory	optional
	0	optional	rejected
TL3	1	mandatory	optional
	0	optional	rejected

That means for TL1, TL2 and TL3 a master that uses the extension can only connect to a slave that understands the extension, while a slave that understands the extension can connect to both a master that does and a master that does not use the extension.

Semantics : The value of the extension has the same semantic as the OCP signal named MAddrSpace. Since the extension is a guarded data extension, the value may only be considered valid if the extension is validated.

4.5.2 atomic_length

Extension type : guarded data

Definiton :

```
struct atomic_length :
  public ocp_tlm_guarded_data_extension
{
  unsigned int value;
};
```

Phase association : BEGIN_REQ

Mutability : X2X

Bindability :

OCP Abstraction layer	OCP Configuration parameter: atomiclength	Bindability	
		Master	Slave
TL1	1	mandatory	optional
	0	optional	rejected
TL2	1	mandatory	optional
	0	optional	rejected
TL3	1	optional	optional
	0	optional	optional

That means for TL1, and TL2 a master that uses the extension can only connect to a slave that understands the extension, while a slave that understands the extension can connect to both a master that does and a master that does not use the extension.

For TL3 the atomic length is not of great value since TL3 keeps phases completely atomic. So both master and slave don't mind seeing this extension. It is only important if bursts are segmented at TL3, and in this case the receiver can check if there is atomic length information by testing the existince of the guarded data extension.

Semantics : The value of the extension has the same semantic as the OCP signal named MAtomicLength. Since the extension is a guarded data extension, the value may only be considered valid if the extension is validated.

4.5.3 broadcast

Extension type : guard

Definiton :

```
struct broadcast :
  public ocp_tlm_guard_extension
{
};
```

Phase association : BEGIN_REQ

Mutability : E2E

Bindability :

OCP Abstraction layer	OCP Configuration parameter: broadcast_enable	Bindability	
		Master	Slave
TL1	1	mandatory	optional
	0	optional	rejected
TL2	1	mandatory	optional
	0	optional	rejected
TL3	1	mandatory	optional
	0	optional	rejected

That means for TL1, TL2 and TL3 a master that uses the extension can only connect to a slave that understands the extension, while a slave that understands the extension can connect to both a master that does and a master that does not use the extension.

Semantics : If this extension is validated in conjunction with a write command, the command has to be interpreted like an BCST command (cmp. OCP Specification). It must not be validated in conjunction with read commands.

4.5.4 burst_length

Extension type : guarded data

Definiton :

```
struct burst_length :
  public ocp_tlm_guarded_data_extension
{
  unsigned int value;
};
```

Phase association : BEGIN_REQ

Mutability : Imprecise burst: P2P, Otherwise: X2X

Bindability :

OCP Abstraction layer	OCP Configuration parameter: burstlength	Bindability	
		Master	Slave
TL1	1	optional	optional
	0	optional	optional
TL2	1	optional	optional
	0	optional	optional
TL3	1	optional	optional
	0	optional	optional

That means for TL1, TL2 and TL3 the burst length extension is ignorable. Any master can connect to any slave (with respect to their use of the extension), because TLM 2.0 mandates the use of the data length field of the generic payload from which the burst length can be derived.

Semantics : The value of the extension has the same semantic as the OCP signal named MBurstLength. Since the extension is a guarded data extension, the value may only be considered valid if the extension is validated. For precise burst, if the extension is not used, but the data length exceeds the width of the connection, the function

unsigned int ocpip::calculate_ocp_burst_length(tlm::tlm_generic_payload& txn, unsigned int buswidth)

can be used to calculate the OCP burst length.

For imprecise burst the burst length cannot be computed, so in this case it has to be provided by the master. It is considered an error if the master does not provide the burst length for an imprecise burst.

Note that in the imprecise case the burst length is P2P, i.e. it changes with every beat on every hop, while for precise bursts it is X2X.

4.5.5 burst_sequence

Extension type : guarded data

Definiton :

```
enum burst_seqs{
  INCR  = OCP_MBURSTSEQ_INCR,
  DFLT1 = OCP_MBURSTSEQ_DFLT1,
  WRAP  = OCP_MBURSTSEQ_WRAP,
  DFLT2 = OCP_MBURSTSEQ_DFLT2,
  XOR   = OCP_MBURSTSEQ_XOR,
  STRM  = OCP_MBURSTSEQ_STRM,
  UNKN  = OCP_MBURSTSEQ_UNKN,
  BLCK  = OCP_MBURSTSEQ_BLCK
};

struct burst_seq_info{
  burst_seqs sequence;
  unsigned int block_height;
  unsigned int block_stride;
  unsigned int blck_row_length_in_bytes;

  sc_dt::uint64 xor_wrap_address;

  unsigned int unkn_dflt_bytes_per_address;
  bool unkn_dflt_addresses_valid;
  std::vector<sc_dt::uint64> unkn_dflt_addresses;

};

struct burst_sequence :
  public ocp_tlm_guarded_data_extension
{
  burst_seq_info value;
};
```

Phase association : BEGIN_REQ

Mutability :

sequence, block_height, block_stride, xor_wrap_address X2X

unkn_dflt_addresses, blck_row_length_in_bytes, unkn_dflt_bytes_per_address, unkn_dflt_bytes_per_address_valid E2E

Bindability :

OCP Abstraction layer	OCP Configuration parameter: burstseq	Bindability	
		Master	Slave
TL1	1*	mandatory	optional
	1**	optional	optional
	0	optional	rejected
TL2	1*	mandatory	optional
	1**	optional	optional
	0	optional	rejected
TL3	1***	mandatory	optional
	1****	optional	optional
	0	optional	rejected

*) (burstseq_dflt1_enable | burstseq_dflt2_enable | burstseq_unkn_enable | burstseq_wrap_enable | burstseq_xor_enable | burstseq_blck_enable)==1

**) (burstseq_dflt1_enable | burstseq_dflt2_enable | burstseq_unkn_enable | burstseq_wrap_enable | burstseq_xor_enable | burstseq_blck_enable)==0

***) (burstseq_dflt1_enable | burstseq_dflt2_enable | burstseq_unkn_enable | burstseq_blck_enable)==1

****) (burstseq_dflt1_enable | burstseq_dflt2_enable | burstseq_unkn_enable | burstseq_blck_enable)==0

That means for TL1 and TL2 a master that uses sequences other than INCR or STRM can only connect to a slave that understands the extension, while a slave that understands the extension can connect to both a master that does and a master that does not use the extension.

For TL3 a master that uses sequences other than INCR, WRAP, XOR or STRM can only connect to a slave that understands the extension, while a slave that understands the extension can connect to both a master that does and a master that does not use the extension. The reason for the difference to TL1 is that at TL3 INCR, WRAP, and XOR are indistinguishable, hence there is no need to check the actual burst sequence code if only those four are supported.

Semantics : The member sequence of the extension has the same semantic as the OCP signal named MBurstSeq, and the members block_height and block_stride have the same semantics as the OCP signals named MBlockHeight and MBlockStride. They are only valid if the sequence is BLCK.

The member xor_wrap_address is the 'entry' point into the data array, the vector unkn_dflt_addresses carries the un- or user-defined address sequences, and unkn_dflt_bytes_per_address_valid indicates if the vector is valid or not. The members unkn_dflt_bytes_per_address is required when accessing the data array of UNKN, DFLT1 or DFLT2 bursts, and member blck_row_length_in_bytes is required to access the data array of BLCK bursts. For a detailed description of the members see chapter 5.

Since the extension is a guarded data extension, all members may only be considered valid if the extension is validated. If the extension is invalid, a slave has to use the tlm_generic_payload members to distinguish between streaming or incrementing bursts. That's why a master that just uses INCR or STRM is free to decide whether or not to use the extension.

Note that a master is always obliged to set up the tlm_generic_payload members properly, even if it uses the extension to signal INCR or STRM.

4.5.6 conn_id

Extension type : guarded data

Definiton :

```
1 struct conn_id :
2   public ocp_tlm_guarded_data_extension
3 {
4   unsigned int value;
5 };
```

Phase association : BEGIN_REQ

Mutability : X2X

Bindability :

OCP Abstraction layer	OCP Configuration parameter: connid	Bindability	
		Master	Slave
TL1	1	mandatory	optional
	0	optional	rejected
TL2	1	mandatory	optional
	0	optional	rejected
TL3	1	mandatory	optional
	0	optional	rejected

That means for TL1, TL2 and TL3 a master that uses the extension can only connect to a slave that understands the extension, while a slave that understands the extension can connect to both a master that does and a master that does not use the extension.

Semantics : The value of the extension has the same semantic as the OCP signal named MConnID. Since the extension is a guarded data extension, the value may only be considered valid if the extension is validated.

4.5.7 imprecise

Extension type : guard

Definiton :

```
struct imprecise :
  public ocp_tlm_guarded_extension
{
};
```

Phase association : BEGIN_REQ

Mutability : X2X

Bindability :

OCP Abstraction layer	OCP Configuration parameter: burstprecise	Bindability	
		Master	Slave
TL1	1	mandatory	optional
	0	optional	rejected
TL2	1	mandatory	optional
	0	optional	rejected
TL3	1	optional	optional
	0	optional	optional

That means for TL1 and TL2 a master that uses the extension can only connect to a slave that understands the extension, while a slave that understands the extension can connect to both a master that does and a master that does not use the extension.

For TL3 a master or slave doesn't care about the extension as at TL3 there is always only one phase of a kind, hence there is no way to model imprecise burst. However, the existence of the extension won't hurt.

Semantics : If the extension is validated the given transaction is considered an imprecise burst, if invalidated it is considered precise.

4.5.8 lock

Extension type : guarded data

Definiton :

```
struct lock_object_base{
  virtual ~lock_object_base(){}
  virtual void atomic_txn_completed()=0;
  bool     lock_is_understood_by_slave;
  unsigned int number_of_txns;
};

struct lock :
  public ocp_tlm_guarded_data_extension
{
  lock_object_base* value;
};
```

Phase association : BEGIN_REQ

Mutability : Validity: X2X, Value E2E

Bindability :

OCP Abstraction layer	OCP Configuration parameter: readex_enable	Bindability	
		Master	Slave
TL1	1	mandatory	optional
	0	optional	rejected
TL2	1	optional	optional
	0	optional	optional
TL3	1	optional	optional
	0	optional	optional

That means for TL1 a master that uses the extension can only connect to a slave that understands the extension, while a slave that understands the extension can connect to both a master that does and a master that does not use the extension.

For TL2 and TL3 the extension is basically ignorable, because extension is designed so that it can cross lock-unaware interconnect and still have the correct effect at a lock-aware slave (see below). Hence the successful understanding of the lock will be checked at runtime by the master not at bind time by the sockets.

Semantics : The lock extension allows to lock an arbitrary group of transactions together, as long as one of them is a read transaction. The idea is that the initiator owns (a pool of) a lock_object derived from lock_object_base. To lock transactions together, the value of the lock extensions of all transactions will point to the same lock object. The bool lock_is_understood_by_slave has to be set to false, and number_of_txns shall identify the number of transactions that are locked together.

Later on, the transactions will arrive at the slave. If the slave knows about the lock extension it will set the bool lock_is_understood_by_slave to true when sending the response for a read. Additionally, whenever the slave finished a transaction that carried a valid lock extension with BEGIN_REQ, it shall count it as belonging to the locked group with lock ID value. When the slave has processed number_of_txns transactions with lock ID value it shall call atomic_txn_completed on the lock object pointed to by the value of the lock extension. At this time the lock is released at the slave.

By that, the master can identify if the lock succeeded when looking at the bool lock_is_understood_by_slave when getting the a read response (a transaction with a read response can be trusted to have arrived at the slave, while a posted write response cannot). If it is true the slave understood the lock (it may understand and reply with an error if it was unable to perform the lock, though), if it is false it did not and the master should take the appropriate actions (Note that the lock object will never get a call to atomic_txn_completed in this case).

When the lock object gets the call to atomic_txn_completed, it knows the it may now be reused (from a memory management perspective) even if there were posted writes or writes without responses in the locked group, because the virtual call will always short cut from the final slave to the initial master.

The lock extensions is designed so that at higher abstraction layers, interconnects need not to understand the lock, as long as the final slave does. The final slave can distinguish transactions that belong to the locked group from transactions that do not belong to the locked group by just looking at/comparing the lock object pointers in the lock extension. It knows when the lock can be released by counting the transactions of the locked group and comparing against the number of expected transactions that were transmitted in the extension.

For OCP, number_of_txns is always 2: One is the locking RDEX, the other is the unlocking write. If the extension is validated in conjunction with a read command the command has to be interpreted as a RDEX command (cmp. OCP Specification). If the extension is valid for a write is can be considered an unlocking write for a prior RDEX. After the unlocking write has finished the slave has to call atomic_txn_completed. If the slave supports RDEX it has to set bool lock_is_understood_by_slave to true when sending the response for the RDEX.

4.5.9 nonposted

Extension type : guard

Definiton :

```
struct nonposted :
  public ocp_tlm_guard_extension
{
};
```

Phase association : BEGIN_REQ

Mutability : X2X

Bindability :

OCP Abstraction layer	OCP Configuration parameter: writenonpost_enable	Bindability	
		Master	Slave
TL1	1	mandatory	optional
	0	optional	rejected
TL2	1	mandatory	optional
	0	optional	rejected
TL3	1	mandatory	optional
	0	optional	rejected

That means for TL1, TL2 and TL3 a master that uses the extension can only connect to a slave that understands the extension, while a slave that understands the extension can connect to both a master that does and a master that does not use the extension.

Semantics : If the extension is validated in conjunction with a write command the command has to be interpreted as a WRNP command (cmp. OCP Specification). It must not be validated in conjunction with read commands.

4.5.10 semaphore

Extension type : guarded data

Definiton :

```
struct semaphore :
  public ocp_tlm_guarded_data_extension
{
  bool value;
};
```

Phase association : Validity: BEGIN_REQ Value: BEGIN_RESP

Mutability : X2X

Bindability :

OCP Abstraction layer	OCP Configuration parameter: rdlwrc_enable	Bindability	
		Master	Slave
TL1	1	mandatory	optional
	0	optional	rejected
TL2	1	mandatory	optional
	0	optional	rejected
TL3	1	mandatory	optional
	0	optional	rejected

That means for TL1, TL2 and TL3 a master that uses the extension can only connect to a slave that understands the extension, while a slave that understands the extension can connect to both a master that does and a master that does not use the extension.

Semantics : If the extension is validated in conjunction with a read command the command has to be interpreted as a RDL command (cmp. OCP Specification). If the extension is validated in conjunction with a write command the command has to be interpreted as a WRC command (cmp. OCP Specification). The value of the extension is only of interest for the WRC case. It shall be initialized by the master to false,

and set by the slave to true in case of a failure of the WRC. In other words, the value does not need to be touched by the slave in case of successful WRC.

Note that the value is undefined in fresh and new transactions so the master always needs to initialize it. Also note that the validity of the extension is only related to BEGIN_REQ, because the validity is used to identify the command type, while the value is considered valid at BEGIN_RESP no matter if the extension is still validated.

4.5.11 srmd

Extension type : guard

Definiton :

```
struct srmd :
  public ocp_tlm_guard_extension
{
};
```

Phase association : BEGIN_REQ

Mutability : X2X

Bindability :

OCP Abstraction layer	OCP Configuration parameter: burstsinglereq	Bindability	
		Master	Slave
TL1	1	mandatory	optional
	0	optional	rejected
TL2	1	mandatory	optional
	0	optional	rejected
TL3	1	optional	optional
	0	optional	optional

That means for TL1 and TL2 a master that uses the extension can only connect to a slave that understands the extension, while a slave that understands the extension can connect to both a master that does and a master that does not use the extension.

For TL3 srmd is ignorable, because at TL3 MRDM and SRMD transfers are indistinguishable.

Semantics : If the extension is validated the transaction is considered to be an single request multiple data burst, otherwise it is considered to be a multiple request multiple data burst.

4.5.12 tag_id

Extension type : guarded data

Definiton :

```
struct srmd :
  public ocp_tlm_guarded_data_extension
{
  unsigned int value;
};
```

Phase association : BEGIN_REQ

Mutability : X2X

Bindability :

OCP Abstraction layer	OCP Configuration parameter: tags>1	Bindability	
		Master	Slave
TL1	1	optional	optional
	0	optional	rejected
TL2	1	optional	optional
	0	optional	rejected
TL3	1	optional	optional
	0	optional	rejected

That means for TL1, TL2 and TL3 a masters and slave can always connect, no matter how they treat the tag IDs.

Semantics : The validity of the extension inversely maps onto the OCP signal MTagInOrder. That means if the extension is validated MTagInOrder is considered invalid, and therefore the tag ID must be used. If the extension is not validated MTagInOrder is considered to be true, so that the tag ID is don't care.

The value of the extension represents the tag ID of a transaction. It is only valid with the first BEGIN_REQ of a transaction, hence can be considered a replacement for MTagID. To get MDataTagID and STagID a module shall place the value of the tag_id extension into an instance specific extension (or other appropriate storage) with the first BEGIN_REQ, so it can be extracted from there when data or response phases arrive.

4.5.13 thread_busy

Extension type : data

Definiton :

```
enum thread_busy_id{
  M_THREAD,
  S_THREAD,
  S_DATA_THREAD
};

struct thread_busy_update{
  thread_busy_id type;
  unsigned int mask;
};

struct thread_busy :
  public ocp_tlm_data_extension
{
 thread_busy_update value;
};
```

Phase association : THREAD_BUSY_CHANGE

Mutability : N/A (see semantics)

Bindability :

OCP Abstraction layer	OCP Configuration parameter	Bindability	
		Master	Slave
TL1	xthreadbusy_exact*	mandatory	mandatory
	!xthreadbusy_exact* & xthreadbusy*	optional	optional
	!xthreadbusy_exact* & ! xthreadbusy*	rejected	rejected
TL2	xthreadbusy_exact*	mandatory	mandatory
	!xthreadbusy_exact* & xthreadbusy*	optional	optional
	!xthreadbusy_exact* & ! xthreadbusy*	rejected	rejected
TL3	1	rejected	rejected
	0	rejected	rejected

*) x is sthread, sdatathread and/or mthread

That means for TL1 a module that depends on thread busy flow control (exact), can only bind to a module that supplies the thread busy information. Additionally, modules that provide thread busy information as a hint (not exact) can bind to both modules that require thread busy and those who don't.

TL3 modules reject the extension, hence will not bind to modules that use it, as at TL3 a module cannot react to thread busy correctly.

Semantics : Thread busy information is exchanged via dedicated thread busy transaction that are part of each OCP socket. Since those transaction are not allowed to pass more than one point-to-point link, mutability does not apply to the thread busy extension. It is a data only extension that is considered valid whenever the phase THREAD_BUSY_CHANGE is transferred.

The member type of the value identifies which thread busy signal is changed, the member mask carries the bit mask of busy threads as described in the OCP specification.

4.5.14 thread_id

Extension type : guarded data

Definiton :

```
struct srmd :
  public ocp_tlm_guarded_data_extension
{
  unsigned int value;
};
```

Phase association : BEGIN_REQ

Mutability : X2X

Bindability :

OCP Abstraction layer	OCP Configuration parameter: threads>1	Bindability	
		Master	Slave
TL1	1	mandatory	optional
	0	optional	rejected
TL3	1	mandatory	optional
	0	optional	rejected
TL3	1	mandatory	optional
	0	optional	rejected

That means for TL1, TL2 and TL3 a master that uses the extension can only connect to a slave that understands the extension, while a slave that understands the extension can connect to both a master that does and a master that does not use the extension.

Semantics : The value of the extension has the same semantic as the OCP signal named MThreadID. Since the extension is a guarded data extension, the value may only be considered valid if the extension is validated.

4.5.15 tl2_timing

Extension type : data

Definiton :

```
struct tl2_master_timing_group {
    unsigned int RqSndI; // Request Send Interval
    unsigned int DSndI;  // Data Send Interval
    unsigned int RpAL;   // Response Accept Latency
};

struct tl2_slave_timing_group {
    unsigned int RqAL;   // Request Accept Latency
    unsigned int DAL;    // Data Accept Latency
    unsigned int RpSndI; // Response Send Interval
};

enum tl2_timing_type {
  MASTER_TIMING,
  SLAVE_TIMING
};

struct tl2_timing_group {
    tl2_master_timing_group   master_timing;
    tl2_slave_timing_group    slave_timing;

    tl2_timing_type type;
};

struct thread_busy :
  public ocp_tlm_data_extension
{
 tl2_timing_group value;
};
```

Phase association : TL2_TIMING_CHANGE

Mutability : N/A (see semantics)

Bindability :

OCP Abstraction layer	Bindability	
	Master	Slave
TL1	rejected	rejected
TL2	mandatory	mandatory
TL3	reject	reject

That means a TL2 socket can only bind to another TL2 socket. Cross abstraction rules will defined later.

Semantics : This extension is for TL2 only. It transmits the TL2 timing information. It is exchanged via dedicated timing information transactions that are part of each TL2 OCP socket. Since those transaction are not allowed to pass more than one point-to-point link, mutability does not apply to the TL2 timing information extension. It is a data only extension that is considered valid whenever the phase TL2_TIMING_CHANGE is transferred.

The member type of the value identifies which timing group has been changed, the member master_timing carries the master timing information, while the member slave_timing carries the slave timing information.

4.5.16 word_count

Extension type : guarded data

Definiton :

```
1  struct tl2_burst_word_count {
2      unsigned int request_wc;
3      unsigned int data_wc;
4      unsigned int response_wc;
5  };
6
7  struct word_count :
8    public ocp_tlm_guarded_data_extension
9  {
10  tl2_burst_word_count value;
11 };
```

Phase association : BEGIN_REQ, BEGIN_DATA, BEGIN_RESP

Mutability : P2P

Bindability :

OCP Abstraction layer	Bindability	
	Master	Slave
TL1	rejected	rejected
TL2	mandatory	mandatory
TL3	rejected	rejected

That means a TL2 socket can only bind to another TL2 socket. Cross abstraction rules will be defined later.

Semantics : This extension is for TL2 modeling only. It transmits information about how many beats of a certain phase are transmitted in a single nb_transport call. When this extension is present, the nb_transport implementation must look up the matching phase word count member:

Phase	word_count member
BEGIN_REQ	request_wc
BEGIN_DATA	data_wc
BEGIN_RESP	response_wc

When the master initiates a BEGIN_REQ (resp. BEGIN_DATA) phase, it may extend the transaction with a word_count setting request_wc (resp. data_wc). This word_count is expressed in OCP data words, the same unit as the burst_length extension. The slave will interpret such a phase as a TL2 request (resp. data_handshake) phase containing request_wc (resp. data_wc) individual burst beats. Conversely, a slave may initiate a BEGIN_RESP phase with a word_count setting response_wc to represent that number of response burst beats. The word_count is cumulative across subsequent similar phases of the same transaction. The total word_count set across similar phases must equal the total burst length of the transaction. For example, if a master wishes to initiate a TL2 transaction representing a 12-word OCP burst, it may issue multiple BEGIN_REQ phases with a word_count extension such that the total of the request_wc values is exactly equal to 12. For example, the following:

```
1 BEGIN_REQ word_count.request_wc=3 (cumulative 3)
2 BEGIN_REQ word_count.request_wc=5 (cumulative 8)
3 BEGIN_REQ word_count.request_wc=4 (cumulative 12=burst_length)
```

would constitute a valid sequence of TL2 request phases for a 12-word OCP burst. For phases with associated data words (BEGIN_DATA or BEGIN_RESP for a READ transaction) the cumulative word_count must be available in the payload's data array. The receiver of the phase must keep track of the cumulative word_count in order to access the data array. Although the transaction's data buffer may be pre-allocated, the receiver should never access data beyond te current cumulative word_count for the phase in progress.

OCP's phase ordering rules must also be respected. This means that during any phase of the transaction, the cumulative response_wc may not be greater than the cumulative request_wc. For a WRITE with data handshake, the cumulative response_wc may not be greater than the cumulative data_wc and the cumulative data_wc may not be greater than the cumulative request_wc.

If a phase is presented with an invalid word_count extension, it is considered as the only phase of its type for the entire burst. As a corollary to the rule that the cumulative word_count must equal the OCP burst length, a phase of a burst following another similar phase where a word_count was valid must have a valid word_count extension as well.

4.6 Multi beat semantics of generic payload members

Since the TLM 2.0 standard, especially the base protocol, does not deal with multi beat transactions OCP TLM explicitly defines the multi beat semantics of the generic payload members.

4.6.1 address

Phase association : BEGIN_REQ

Mutability : P2P

Remarks : The address is totally mutable as defined by OSCI. For OCP X2X mutability would suffice. Note that for burst sequences that do not allow to calculate the address of a beat from the beat number and the first address, the address_vector extension has to be used.

4.6.2 command

Phase association : All

Mutability : E2E

Remarks : The command is always valid, just as defined by OSCI TLM 2.0.

4.6.3 data/byte enable pointer

Phase association :

Type of transaction	Phase association
Read	All
Write without data handshake	All
Write with data handshake	First BEGIN_DATA and all following

Mutability : E2E

Remarks : Once set the data/byte enable pointers cannot change anymore. Given that OSCI BP is only read or write without data handshake, OSCI BP compatible OCP transactions match the OSCI definitions for the data/byte enable pointers.

4.6.4 data/byte enable length

Phase association :

Type of transaction	Phase association
Read	All
Write without data handshake	All
Write with data handshake	First BEGIN_DATA and all following

Mutability : precise bursts: E2E, imprecise: special

Remarks : For precise bursts the data/byte enable length is fixed as soon as the data pointer is fixed, and since OSCI BP only supports precise burst, OSCI BP compatible OCP transactions match the OSCI definitions for the data/byte enable length. Imprecise bursts are very special. The master has to guess up front the maximum size of its imprecise transfer to be able to allocate a large enough buffer for the transaction data. It will then set the data length to the size of that buffer. Usually that buffer is too large, which is uncritical because the OCP modules will only use the burst length extension and not the data length. At the time the final beat of the imprecise burst is done, the master may now reduce the data length to the real length of the transfer.

If the master mis-guessed the buffer size (i.e. it is too small) it cannot do anything else but start a new transaction, because a reallocation would mean a change to the data pointer.

If each word of an imprecise burst has a unique byte enable mask, the mechanism described for the data length also applies to the byte enable length, otherwise (in case of a repeated byte enable mask) the byte enable length is fixed with the first data or request phase.

4.6.5 response status

Phase association : BEGIN_RESP

Mutability : P2P

Remarks : The response status could change for every response beat in a read or write burst, hence it can have different values on different hops in the system at a time, and it can have different values for different beats on a single point-to-point link. Consequently, it has to be considered P2P. However, for single beat transfers this degenerates to X2X, and assuming a response is not changed by interconnects this degenerates to E2E, thereby matching the OSCI BP definition, because OSCI BP is single beat only.

For OCP a TLM_OK_RESPONSE is to be treated as DVA or FAIL[3]. Every other TLM-2.0 response code maps on the OCP ERR response code.

4.6.6 streaming width

Phase association : BEGIN_REQ

Mutability : E2E

Remarks : The streaming width is always valid, just as defined by OSCI TLM 2.0. Since streaming OCP burst are not packing, and the structure of the TLM generic payload is a packet a width conversion for streaming bursts will require a deep copy, when going converting from wide to narrow. Narrow to wide conversion do not require a change to the streaming width and do not require a deep cop Hence, the streaming width can be E2E.

4.6.7 dmi hint

Phase association : All

Mutability : E2E

Remarks : OCP TLM does not differ in its use of the dmi hint from OSCI BP.

4.7 Extended phases

Finally OCP TLM has defined some additional phases. This section lists their names and semantics.

[3]depends on the state of the appropriate extension

4.7.1 BEGIN_DATA

This tlm_phase marks the begin of a data phase. When a BEGIN_DATA has crossed a given point to point link, we say there is an outstanding data phase on that link. The outstanding data phase is removed when a END_DATA crosses the same point to point link. There may only be one oustanding data phase on a given point to point link.

Assuming that a M byte wide TL1 target counts data phases, and this count is currently at N, then BEGIN_DATA indicates that now at least $(N+1)*M$ bytes are valid in the data array of the tlm_generic_payload.

Assuming that a M byte wide TL2 target sums up the word counts it has seen so far and that count is at N, then BEGIN_DATA with a data word count (data_wc) of L indicates that at least $(N+L)*M$ bytes are valid in the data array of the tlm_generic_payload.

4.7.2 END_DATA

This tlm_phase marks the end of a data phase. This phase may only occur if there is an outstanding data phase that will be finished by END_DATA.

4.7.3 THREAD_BUSY_CHANGE

This phase marks the change of a thread busy signal on the point to point link it crosses. It may only be used in conjunction with dedicated thread busy transactions that can be obtained from OCP sockets and which must not be forwarded from one point-to-point link to another. The phase must not be used with any other transaction.

4.7.4 TL2_TIMING_CHANGE

This phase marks the change of a TL2 timing group on the point to point link it crosses. It may only be used in conjunction with dedicated tl2 timing transactions that can be obtained from OCP sockets and which must not be forwarded from one point-to-point link to another. The phase must not be used with any other transaction.

Chapter 5

TLM transaction data interpretation within OCP

In general the data array is organized as defined by the TLM 2.0 standard. However, OCP knows much more sophisticated burst sequences than what is covered by the TLM 2.0 standard. This chapter will explain the data array organization for all burst sequences supported by OCP. For completeness reasons that will also cover sequences already covered by the TLM 2.0 standard.

5.1 Terminology

To fully understand the explanations in the following sections, reading the TLM 2.0 reference manual is strongly recommended. Some recurring terms are:

Data array :

This term refers to the unsigned char* m_data member of the tlm::tlm_generic_payload. Its elements will be denoted as D[0], D[1], D[2], ... throughout the section.

Data size :

This term refers to the unsigned int m_length member of the tlm::tlm_generic_payload. Its identifies last integer i with $i = data\_size - 1$ for which D[i] will access valid memory.

Transaction address :

This term refers to the sc_dt::uin64 m_address member of the tlm::tlm_generic_payload.

Bus width :

This term refers to the next largest power of two of the template argument BUSWIDTH of two connected sockets divided by 8. In other words it is the bus width of the connection in full bytes.

$$bus\_width = \lfloor \frac{BUSWIDTH + 7}{8} \rfloor \tag{5.1}$$

Address offset :

This term refers to the difference between an address and next smallest address aligned to the current bus width.

$$offset(address) = address - (address \ mod \ bus\_width) \tag{5.2}$$

To simplify some of the following equations, we define

$$Toffset = offset(transaction\_address) \tag{5.3}$$

Streaming width :

This term refers to the unsigned int m_streaming_width member of the tlm::tlm_generic_payload.

Beat number :

In TL1 the beat number is obtained by either counting the request phase (write without data phase), the data phases (write with data phase) or the response phases (read). In TL2 the current beat number is calculated by accumulating the word count of request phases (write without data phase), of data phases (write with data phase), or the response phases (read).

The first beat number of a burst is 1.

Burst sequence extension This term refers to the value of the burst_sequence extension (see. section 4.5.5. If the following sections mention setting a member of this extensions it always implies a subsequent validation of the extension. Additionally when getting a member of this extension is mentioned, validity of the extension is assumed.

5.2 Incrementing burst: INCR

The incrementing burst is the simplest conceivable burst. To mark a transaction as INCR, the streaming width has to be set to a value equal to or larger than the data size. The burst sequence extension may either be invalidated (or kept invalid) or the member sequence of the burst sequence extension may be set to INCR.

The transaction address identifies the address of byte D[0]. There are no obligations concerning the offset of the transaction address. The address of each succeeding byte of the data array can be calculated by

$$address(D[i]) = transaction\_address + i \tag{5.4}$$

In the absence of a valid burst_length extension the OCP burst length of a given INCR transaction can be calculated by

$$burst\_length = \lfloor \frac{data\_size + Toffset + bus\_width - 1}{bus\_width} \rfloor \tag{5.5}$$

Note that the burst_length extension always takes precedence over equation 5.5.

The bytes that form the word for beat number b are part of the set $Word(b)$

$$Word(b) = \left\{ \begin{array}{lll} b == 1 \mid & D[i]: & 0 \leq i < bus\_width - Toffset \\ & & \\ b > 1 \mid & D[i]: & (b-1) * bus\_width - Toffset \\ & & \leq i < \\ & & min(b * bus\_width - Toffset, data\_size) \end{array} \right\} \tag{5.6}$$

If for a given b equation 5.6 cannot yield any valid i (because each $i > data\_size$) the set $Word(b)$ is considered the empty set. Note that the definition of equation 5.6 explicitly allow the data array to end at an intermediate byte of any word of the burst. In other words, just like the first word, the word at which the data array ends can be transmitted partially without explicitly using the byte enable array.

Data setters[1] must ensure that at the time they emit data beat b[2] the data array bytes that form $Word(b)$ are properly filled. Data readers can rely on the validity of data bytes in $Word(b)$ when receiving data beat b.

To actually form a word out of such a set $Word(b)$ the data array bytes $D[i]$ have to be mapped on word bytes $W[j]$

$$j = address(D[i])\ mod\ bus\_width \tag{5.7}$$

If a set does not contain any or enough bytes to form a word, the missing bytes in that word are considered disabled[3], regardless if there is a byte enable array or not. Note that the order of the word bytes $W[j]$ depends on the host endianess.

5.2.1 Burst Aligned Incrementing Burst

The burst aligned incrementing burst is a very special restricted variant of the INCR burst. The burst length must be a power of two and the address must be aligned to the burst length and data width. However, those restrictions only apply to the simulated burst not to the transaction that simulates the burst. Hence, there must be rules how to extract the simulated burst length and address from the transaction for burst aligned incrementing bursts.

[1] For writes that is the master, for reads that is the slave

[2] For writes that is either a request or data phase, depending if there are data phases or not. For reads that is a response phase

[3] If the OCP configuration lacks byte enables, such implicit encoding of byte enables is not allowed

Just like for normal INCR bursts the address of byte D[0] is identified by the transaction address, and equation 5.4 applies. In the presence of burst length extension the following euqation must be fulfilled

$$burst\_length = 2^x \wedge x \in IN \tag{5.8}$$

To determine the burst length of the burst aligned INCR burst, we first define

$$words\_in\_txn = \lfloor \frac{data\_size + Toffset + bus\_width - 1}{bus\_width} \rfloor \tag{5.9}$$

$$min\_pow\_two = 2^{\lceil log_2(words\_in\_txn) \rceil} \tag{5.10}$$

$$aligned\_address(bytes) = \lfloor \frac{transaction\_address}{bytes} \rfloor * bytes \tag{5.11}$$

where *words_in_txn* is the minimal number of full bus words needed to transmit the bytes that are (will be) part of the transaction; *min_pow_two* is the next largest power of two compared to *words_in_txn*. *aligned_adress*(*bytes*) is a function that aligns the transaction address to the given number of bytes.

With the help of those definitions we can define a function to determine the shortest possible burst aligned INCR burst for a given transaction.

Listing 5.1: Algorithm to determine the minimal burst aligned INCR for a transaction on a given link

```
function determine_min_aligned_burst_length;
  integer burst_length=min_pow_two;
  integer bytes=burst_length*bus_width;
  while(aligned_address(bytes)+bytes<transaction_address+data_size);
  do
    burst_length=burst_length*2;
    bytes=burst_length*bus_width;
  done
  output(burst_length, aligned_address(bytes));
endfunction;
```

The basic idea is that the algorithm in listing 5.1 first aligns the transaction address to the minimal power of two burst length that is long enough to transmit all words in the transaction (min_pow_two). If after the alignment the final address of the transaction (transaction_address+data_size) lies not within that burst aligned burst, the algorithm will increase the burst length to the next power of two and test again. As soon as the loop is done, the algorithm will return both the burst length and the base address of the transaction[4].

As already shown in the plain INCR sequence a transaction can have empty trailing beats. That means one can receive a data beat, whose associated data array entries lie outside the data size. Then the data beat is considered to be transmitted with all bytes disabled (of course this is an error on links that do not use byte enables).

In contrast to that, the burst aligned INCR sequence can have empty leading beats. The reason is that the bytes of the data array can lie somewhere in the middle of the simulated transaction, whereas with a plain INCR the transaction address always lies within the first beat.

Assuming the burst length is known, either by the burst length extension or by using the algorithm in listing 5.1, the number of empty leading beats is given by

$$empty\_leading\_beats = \frac{transaction\_address - aligned\_address(burst\_length * bus\_width)}{bus\_width} \tag{5.12}$$

The bytes that form the word for beat number c is given by the set $Word(b)$ as defined in equation 5.6, and c maps on b via

$$b = \begin{cases} (c - empty\_leading\_beats) < 1 & | \; words\_in\_txn + 1 \\ (c - empty\_leading\_beats) \geq 1 & | \; c - empty\_leading\_beats \end{cases} \tag{5.13}$$

The equation 5.13 makes sure that an empty leading beat c is mapped on a beat b that completely lies outside the data array and whose set $Word(b)$ will therefore evaluate to the empty set.

[4] That is the address that would be transmitted for the first beat in hardware.

5.3 Wrapping incrementing burst: WRAP

The wrapping burst is a special variant of the INCR burst. To mark a transaction as WRAP, the streaming width has to be set to a value equal to or larger than the data size. The member sequence of the burst sequence extension has to be set to WRAP.

The addresses of the data array bytes are calculated following equation 5.4. Just like for INCR the OCP burst length is calculated by equation 5.5, and the burst_length extension takes precedence over the result of that calculation as well.

The difference from WRAP to INCR is that the data array is not filled starting at the lowest address of the data array, but somewhere in between, and that when the data filling hits the highest addressable byte of the burst, it continues at the lowest address.

To indicate at which address the WRAP burst starts the user must set the member xor_wrap_address of the burst sequence extension. Since the transaction address (the member of the generic payload) always points to the first byte of the data array, the xor_wrap_address must always be greater or equal to the transaction address.

If that address is unequal to the transaction address, it has to be aligned to the bus width of the current link, because it points to an intermediate word of a consecutive data array, hence the intermediate word cannot be partially in the data array.

The *base_address* of a WRAP sequence is given by

$$base\_address = transaction\_address - Toffset \tag{5.14}$$

Assuming the existence of a function $wrap\_address(c, burst\_length, xor\_wrap\_address)$ that calculates the address of beat c for a given WRAP burst, the bytes that form the word for beat number c are part of the set $Word(b)$, where the set $Word(b)$ is defined as in equation 5.6, and c maps on b via

$$b = \frac{wrap\_address(c, burst\_length, xor\_wrap\_address) - base\_address}{bus\_width} \tag{5.15}$$

5.4 Critical-word first cache line burst: XOR

The critical-word first cache line burst burst is a special variant of the INCR burst. To mark a transaction as XOR, the streaming width has to be set to a value equal to or larger than the data size. The member sequence of the burst sequence extension has to be set to XOR.

The addresses of the data array bytes are calculated following equation 5.4. Just like for INCR the OCP burst length is calculated by equation 5.5, and the burst_length extension takes precedence over the result of that calculation as well.

The difference from XOR to INCR is that the data array is not filled starting at the lowest address of the data array, but somewhere in between, and that it does not fill the bytes consecutively but it 'jumps' through the data array, following the rules for XOR addressing.

To indicate at which address the XOR burst starts the user must set the member xor_wrap_address of the burst sequence extension. Since the transaction address (the member of the generic payload) always points to the first byte of the data array, the xor_wrap_address must always be greater or equal to the transaction address.

If that address is unequal to the transaction address, it has to be aligned to the bus width of the current link, because it points to an intermediate word of a consecutive data array, hence the intermediate word cannot be partially in the data array.

The *base_address* of a XOR sequence is given by equation 5.14

Assuming the existence of a function $xor\_address(c, burst\_length, xor\_wrap\_address)$ that calculates the address of beat c for a given XOR burst, the bytes that form the word for beat number c are part of the set $Word(b)$, where the set $Word(b)$ is defined as in equation 5.6, and c maps on b via

$$b = \frac{xor\_address(c, burst\_length, xor\_wrap\_address) - base\_address}{bus\_width} \tag{5.16}$$

5.5 Streaming burst: STRM

A streaming burst is a burst that repeats a certain address sequence to feed its data to e.g. a fifo. The transaction address identifies the address of byte D[0]. There are no obligations concerning the offset of the transaction address. The address of each succeeding byte of the data array can be calculated by

$$address(D[i]) = transaction\_address + (i \bmod streaming\_width) \tag{5.17}$$

To mark a transaction as STRM, the streaming width has to be set to a value smaller than the data size. The burst sequence extension may either be invalidated (or kept invalid) or the member sequence of the burst sequence extension may be set to STRM. Note that it is considered an error to send a transaction over a link with the $streaming\_width + Toffset > bus\_width$.

In the absence of a valid burst_length extension the OCP burst length of a given INCR transaction can be calculated by

$$burst\_length = \lfloor \frac{data\_size + streaming\_width - 1}{streaming\_width} \rfloor \tag{5.18}$$

Note that the burst_length extension always takes precedence over equation 5.18.

The bytes that form the word for beat number b are part of the set $Word(b)$

$$Word(b) = \{D[i] | (b-1) * streaming\_width \leq i < min(b * streaming\_width, data\_size)\} \tag{5.19}$$

If for a given b equation 5.19 cannot yield any valid i (because each $i > data\_size$) the set $Word(b)$ is considered the empty set. Note that the definition of equation 5.19 explicitly allows the data array to end at an intermediate byte of any word of the burst. In other words, just like the first word, the word at which the data array ends can be transmitted partially without explicitly using the byte enable array.

Data setters[5] must ensure that at the time they emit data beat b[6] the data array bytes that form $Word(b)$ are properly filled. Data readers can rely on the validity of data bytes in $Word(b)$ when receiving data beat b.

To actually form a word out of such a set $Word(b)$ the data array bytes $D[i]$ have to be mapped on word bytes $W[j]$

$$j = address(D[i]) \ mod \ bus\_width \tag{5.20}$$

If a set does not contain any or enough bytes to form a word, the missing bytes in that word are considered disabled[7], regardless if there is a byte enable array or not. Note that the order of the word bytes $W[j]$ depends on the host endianess.

5.6 Two dimensional burst: BLCK

As defined in the OCP specification the BLCK burst is a set of chained BLCK bursts. To mark a transaction as BLCK, the streaming width has to be set to a value equal to or larger than the data size, and the member sequence of the burst sequence extension must be set to BLCK. The members block_height, block_stride and blck_row_length_in_bytes of the burst sequence extension must be set as well. The two former members have the same semantics as the OCP signals MBlockHeight and MBlockStride, the latter member specifies how many bytes of the data array belong to a row of the BLCK burst.

The transaction address identifies the address of byte D[0]. There are no obligations concerning the offset of the transaction address.

The row number a byte $D[i]$ of the data array belongs to can be calculated by

$$row(i) = \lfloor \frac{i}{blck\_row\_length\_in\_bytes} \rfloor \tag{5.21}$$

The address of each succeeding byte of the data array can be calculated by

$$address(D[i]) = transaction\_address + row(i) * block\_stride + (i \ mod \ blck\_row\_length\_in\_bytes) \tag{5.22}$$

In the absence of a valid burst_length extension the OCP burst length of a given BLCK transaction can be calculated by

$$burst\_length = \lfloor \frac{blck\_row\_length\_in\_bytes + Toffset + bus\_width - 1}{bus\_width} \rfloor \tag{5.23}$$

Note that the burst_length extension always takes precedence over equation 5.23.

The first byte of a row r is given by

$$rs(r) = r * blck\_row\_length\_in\_bytes \tag{5.24}$$

[5] For writes that is the master, for reads that is the slave

[6] For writes that is either a request or data phase, depending if there are data phases or not. For reads that is a response phase

[7] If the OCP configuration lacks byte enables, such implicit encoding of byte enables is not allowed

The row number rn, the beat number relative to a row rb, and the first byte relative to the first byte of a row fb for a given beat b are defined as

$$rn(b) = \lfloor \frac{b}{burst\_length} \rfloor \tag{5.25}$$

$$rb(b) = b - rn(b) * burst\_length \tag{5.26}$$

$$fb(b) = rb(b) * bus\_width - Toffset \tag{5.27}$$

The bytes that form the word for beat number b are part of the set $Word(b)$ (Note that blck_row_length_in_bytes is abbreviated with row_bytes)

$$Word(b) = \left\{ \begin{array}{rl|l} rb(b) == 1 & D[i] : & \begin{array}{l} rs(rn(b)) \\ \leq i < \\ min(rs(rn(b)) + buswidth - Toffset, data\_size) \end{array} \\ \hline 1 < rb(b) \wedge \\ fb(b) < row\_bytes & D[i] : & \begin{array}{l} rs(rn(b)) + (rb(b) - 1) * bus\_width - Toffset \\ \leq i < \\ min(rs(rn(b)) + rb(b) * buswidth - Toffset, data\_size) \end{array} \\ \hline otherwise & \emptyset & \end{array} \right\} \tag{5.28}$$

If for a given b equation 5.28 cannot yield any valid i (because each $i > data\_size$) the set $Word(b)$ is considered the empty set. Note that with $fn(b) = 0$ equation 5.28 degenerates to equation 5.6, which is correct as simple INCR is of course a subset of a set of INCR. Data setters[8] must ensure that at the time they emit data beat b[9] the data array bytes that form $Word(b)$ are properly filled. Data readers can rely on the validity of data bytes in $Word(b)$ when receiving data beat b.

To actually form a word out of such a set $Word(b)$ the data array bytes $D[i]$ have to be mapped on word bytes $W[j]$

$$j = address(D[i])\ mod\ bus\_width \tag{5.29}$$

If a set does not contain any or enough bytes to form a word, the missing bytes in that word are considered disabled[10], regardless if there is a byte enable array or not. Note that the order of the word bytes $W[j]$ depends on the host endianess.

5.7 Non-predefined burst: UNKN

The UNKN burst follows an address sequence that cannot be calculated or at least the receiving slave is unknowing of how to calculate it. This requires to explicitly associate an address with every word to be transmitted.

To mark a transaction as UNKN, the streaming width has to be set to a value equal to or larger than the data size, and the burst sequence extension the member sequence of the burst sequence extension must be set to UNKN. Additionally the member unkn_dflt_bytes_per_address must be set to the bus width of the initial master and the member unkn_dflt_addresses_valid must be set to true to indicate that the member unkn_dflt_addresses can be safely accessed. When setting up the UNKN burst, the vector has to be large enough to hold an address per beat the master wants to transmit, but its size does not need to reflect the exact number of beats. Whenever the initial master emits a request beat it must ensure that the associated entry of the address vector is filled.

The addresses in the vector must be aligned to unkn_dflt_bytes_per_address, offsets are not allowed. The transaction address must match the first entry of the address vector. There is no way to calculate the global address sequence, but local subsections are considered calculable: Assume $unkn\_dflt\_bytes\_per\_address = n$, $bus\_width = w$ and a beat b.

$$n \leq w \ \left| \ \begin{array}{l} \forall D[i], (b-1) * n \leq i < min(b * n, data\_size) : \\ address(D[i]) = address\_vector[b] + (i\ mod\ n) \end{array} \right. \tag{5.30}$$

$$\left. \begin{array}{r} n > w \wedge \\ \frac{n}{w} = f \wedge \\ f = 2^x, x \in IN \end{array} \right| \ \begin{array}{l} \forall D[i], (b-1) * w \leq i < min(b * w, data\_size) : \\ address(D[i]) = address\_vector[\lfloor \frac{b}{f} \rfloor] + (i\ mod\ n) \end{array} \tag{5.31}$$

[8] For writes that is the master, for reads that is the slave

[9] For writes that is either a request or data phase, depending if there are data phases or not. For reads that is a response phase

[10] If the OCP configuration lacks byte enables, such implicit encoding of byte enables is not allowed

As can be seen in equation 5.30 , when the bus width is larger than or equal to the bytes per UNKN address the burst is considered non packing, while with the bytes per address larger than the bus width (equation 5.31) it is considered packing.

Note that *unkn_dflt_bytes_per_address* > *bus_width* is only permitted if the fraction is a power of two. Hence, if a UNKN sequence is packing when going narrow to wide, if it is non-packing when going wide to narrow, or if the fraction of bytes per address and bus width is not a power of two a deep copy is required.

In the absence of a valid burst_length extension the OCP burst length of a given UNKN transaction can be calculated by

$$burst\_length = \lfloor \frac{data\_size + unkn\_dflt\_bytes\_per\_address - 1}{unkn\_dflt\_bytes\_per\_address} \rfloor * \lfloor \frac{unkn\_dflt\_bytes\_per\_address + bus\_width - 1}{bus\_width} \rfloor \tag{5.32}$$

Note that the burst_length extension always takes precedence over equation 5.32.

The set $Word(b)$ that contains the bytes for data beat b with $unkn\_dflt\_bytes\_per\_address = n$ and $bus\_width = w$ is given by

$$Word(b) = \left\{ \begin{array}{rl} n \leq w & D[i] : (b-1) * n \leq i < min(b * n, data\_size) \\ n > w \wedge & \\ \frac{n}{w} = f \wedge & D[i] : (b-1) * w \leq i < min(b * w, data\_size) \\ f = 2^x, x \in IN & \end{array} \right\} \tag{5.33}$$

To actually form a word out of such a set $Word(b)$ the data array bytes $D[i]$ have to be mapped on word bytes $W[j]$

$$j = address(D[i])\ mod\ bus\_width \tag{5.34}$$

If a set does not contain any or enough bytes to form a word, the missing bytes in that word are considered disabled[11], regardless if there is a byte enable array or not. Note that the order of the word bytes $W[j]$ depends on the host endianess.

5.8 User defined packing burst: DFLT1

The DFLT1 burst follows an address sequence that can be calculated in a user defined way. However, some modules may not be in possession of knowledge about how to calculate it. To this end, the sender may explicitly associate an address with every word to be transmitted, but he does not have to. If a functional module is not in possession of knowledge about how to calculate the sequence and receives a DFLT1 burst, it should raise an error as it cannot sensibly process the burst. If a non-functional module (like a monitor) is not in possession of knowledge about how to calculate the sequence and receives a DFLT1 burst it shall use the transaction address as the address of each beat and raise a warning that it did so.

To mark a transaction as DFLT1, the streaming width has to be set to a value equal to or larger than the data size, and the burst sequence extension the member sequence of the burst sequence extension must be set to DFLT1. Additionally the member unkn_dflt_bytes_per_address must be set to the bus width of the initial master and the member unkn_dflt_addresses_valid must indicate whether there is an explicit address for each word or not. When setting up the DFLT1 burst with explicit address information, the vector has to be large enough to hold an address per beat the master wants to transmit, but its size does not need to reflect the exact number of beats. Whenever the initial master emits a request beat it must ensure that the associated entry of the address vector is filled.

The addresses in the vector must be aligned to unkn_dflt_bytes_per_address, offsets are not allowed. The transaction address must match the first entry of the address vector. Without knowledge of the address sequence calculation there is no way to calculate the global address sequence, but local subsections are considered calculable when explicit address information isn available: Assume $unkn\_dflt\_bytes\_per\_address = n$, $bus\_width = w$ and a beat b.

$$\left. \begin{array}{r} n \leq w \wedge \\ \frac{w}{n} = f \wedge \\ f = 2^x, x \in IN \end{array} \right| \begin{array}{l} b = 1 : \forall D[i], 0 \leq i < min(w - Toffset, data\_size) : \\ \quad address(D[i]) = address\_vector[0] + i \\ b > 1 : \forall D[i], (b-1) * w - Toffset \leq i < min(b * w - Toffset, data\_size) : \\ \quad address(D[i]) = address\_vector[b * f - \frac{Toffset}{n}] + ((i + Toffset)\ mod\ w) \end{array} \tag{5.35}$$

[11] If the OCP configuration lacks byte enables, such implicit encoding of byte enables is not allowed

$$\left.\begin{array}{r} n > w \wedge \\ \frac{n}{w} = f \wedge \\ f = 2^x, x \in IN \end{array}\right| \begin{array}{l} \forall D[i], (b-1) * w \leq i < min(b * w, data\_size) : \\ \qquad address(D[i]) = address\_vector[\lfloor \frac{b}{f} \rfloor] + (i \ mod \ n) \end{array} \tag{5.36}$$

As can be seen in equations 5.35 and 5.36, the burst is always considered packing.

Note that *unkn_dflt_bytes_per_address* > *bus_widthorunkn_dflt_bytes_per_address* < *bus_width* is only permitted if the fraction is a power of two. Hence, if the fraction of bytes per address and bus width is not a power of two a deep copy is required.

In the absence of a valid burst_length extension the OCP burst length of a given DFLT1 transaction can be calculated by equation 5.5. Note that the burst_length extension always takes precedence over equation 5.5.

The set $Word(b)$ that contains the bytes for data beat b with $unkn\_dflt\_bytes\_per\_address = n$ and $bus\_width = w$ is given by

$$Word(b) = \left\{ \begin{array}{l|l} & b = 1 | D[i] : 0 \leq i < min(w - Toffset, data\_size) \\ \begin{array}{r} n \leq w \wedge \\ \frac{w}{n} = f \wedge \\ f = 2^x, x \in IN \end{array} & b > 1 \left| \begin{array}{ll} & (b-1) * w - Toffset \\ D[i] : & \leq i < \\ & min(b * w - Toffset, data\_size) \end{array} \right. \\ \\ \begin{array}{r} n > w \wedge \\ \frac{n}{w} = f \wedge \\ f = 2^x, x \in IN \end{array} & D[i] : (b-1) * w \leq i < min(b * w, data\_size) : \end{array} \right\} \tag{5.37}$$

To actually form a word out of such a set $Word(b)$ the data array bytes $D[i]$ have to be mapped on word bytes $W[j]$

$$j = address(D[i]) \ mod \ bus\_width \tag{5.38}$$

If a set does not contain any or enough bytes to form a word, the missing bytes in that word are considered disabled[12], regardless if there is a byte enable array or not. Note that the order of the word bytes $W[j]$ depends on the host endianess.

5.9 User defined non-packing burst: DFLT2

The DFLT2 burst follows an address sequence that can be calculated in a user defined way. However, some modules may not be in possession of knowledge about how to calculate it. To this end, the sender may explicitly associate an address with every word to be transmitted, but he does not have to. If a functional module is not in possession of knowledge about how to calculate the sequence and receives a DFLT2 burst, it should raise an error as it cannot sensibly process the burst. If a non-functional module (like a monitor) is not in possession of knowledge about how to calculate the sequence and receives a DFLT2 burst it shall use the transaction address as the address of each beat and raise a warning that it did so.

To mark a transaction as DFLT2, the streaming width has to be set to a value equal to or larger than the data size, and the burst sequence extension the member sequence of the burst sequence extension must be set to DFLT2. Additionally the member unkn_dflt_bytes_per_address must be set to the bus width of the initial master and the member unkn_dflt_addresses_valid must indicate whether there is an explicit address for each word or not. When setting up the DFLT2 burst with explicit address information, the vector has to be large enough to hold an address per beat the master wants to transmit, but its size does not need to reflect the exact number of beats. Whenever the initial master emits a request beat it must ensure that the associated entry of the address vector is filled.

The addresses in the vector must be aligned to unkn_dflt_bytes_per_address, offsets are not allowed. The transaction address must match the first entry of the address vector. Without knowledge of the address sequence calculation there is no way to calculate the global address sequence, but local subsections are considered calculable when explicit address information isn available: Assume $unkn\_dflt\_bytes\_per\_address = n$, $bus\_width = w$ and a beat b.

$$n \leq w \ \left| \begin{array}{l} \forall D[i], (b-1) * n \leq i < min(b * n, data\_size) : \\ \qquad address(D[i]) = address\_vector[b] + (i \ mod \ n) \end{array} \right. \tag{5.39}$$

[12] If the OCP configuration lacks byte enables, such implicit encoding of byte enables is not allowed

As can be seen in equations 5.39 the DFLT2 sequence is not packing. Moreover, it is an error to transmit a DFLT2 burst with $unkn\_dflt\_bytes\_per\_address > bus\_width$.

In the absence of a valid burst_length extension the OCP burst length of a given DFLT2 transaction can be calculated by

$$burst\_length = \lfloor \frac{data\_size + unkn\_dflt\_bytes\_per\_address - 1}{unkn\_dflt\_bytes\_per\_address} \rfloor \tag{5.40}$$

Note that the burst_length extension always takes precedence over equation 5.40.

The set $Word(b)$ that contains the bytes for data beat b with $unkn\_dflt\_bytes\_per\_address = n$ and $bus\_width = w$ is given by

$$Word(b) = \{n \leq w | D[i] : (b-1) * n \leq i < min(b * n, data\_size)\} \tag{5.41}$$

To actually form a word out of such a set $Word(b)$ the data array bytes $D[i]$ have to be mapped on word bytes $W[j]$

$$j = address(D[i]) \ mod \ bus\_width \tag{5.42}$$

If a set does not contain any or enough bytes to form a word, the missing bytes in that word are considered disabled[13], regardless if there is a byte enable array or not. Note that the order of the word bytes $W[j]$ depends on the host endianess.

5.10 Byte Enables

Independent of the burst sequence for every set $Word(b)$ applies that all bytes in the set are enabled if the byte_enable_ptr of the transaction is NULL. In the presence of a byte_enable_ptr the following equations determines the set of valid bytes of the bytes in set $Word(b)$

$$Valid(Word(b)) = \{D[i] | D[i] \in Word(b) \wedge byte\_enable\_ptr[i \ mod \ byte\_enable\_length] = 0xFF\} \tag{5.43}$$

[13] If the OCP configuration lacks byte enables, such implicit encoding of byte enables is not allowed

Chapter 6

Connecting legacy IP

The OCP Modelling Kit release is radically different from the previous OCP kits up to OCP-IP SLD r2.2.1. However, there is still a large number of IP models available that use the OCP TLM interfaces of OCP-IP SLD r2.2.1. To allow a seamless and smooth migration from OCP-IP SLD r2.2.1 to OCP Modelling Kit, the OCP Modelling Kit contains the whole OCP-IP SLD r2.2.1 kit and adapters that allow connecting OCP Modelling Kit IP to OCP-IP SLD r2.2.1 IP and vice versa.

6.1 Including the Legacy Support Classes

Normally the inclusion of the legacy support classes (the complete OCP-IP SLD r2.2.1kit and the legacy adapters) is deactivated to minimize the number of files that are included by ocpip.h. To activate the legacy support the compile time switch OCP_USE_LEGACY has to be set.

For example, if you are using gcc then the compile time switch has to be set via the command line argument -DOCP_USE_LEGACY. Afterwards, the complete OCP-IP SLD r2.2.1kit is available in namespace ocpip_legacy, and legacy channels, ports, classes, etc. can be used through this namespace.

If legacy IP shall be included in a system together with the OCP Modelling Kit the include path ocp_installation_path/include/legacy_support has to be provided to the compiler. By that all header files of the OCP-IP SLD r2.2.1kit will be available. To avoid changes to the legacy IP the namespace ocpip_legacy will be opened globally (via using namespace ocpip_legacy;).

Afterwards, both legacy IP as well as the adapters can be instantiated and used. There is no need to change the legacy code in any way. However, if the legacy code is written for a release version that is not compatible to 2.2.1 you are basically on your own...

6.2 Instantiating and Connecting Adapters

After setting OCP_USE_LEGACY and providing the include path include/legacy_support to the compiler as described above, legacy IP can be compiled with the OCP Modelling Kit and can be connected to OCP Modelling Kit IP via adapters.

There are mutliple adapters available, which will be explained in the following subsections.

6.2.1 TL1 master legacy adapter

This adapter has a OCP-IP SLD r2.2.1 TL1 slave port, and an OCP Modelling Kit TL1 master socket. Hence, you can connect an an OCP-IP SLD r2.2.1 TL1 master to this adapter via an OCP-IP SLD r2.2.1 TL1 channel, and you can connect an OCP Modelling Kit TL1 slave to the adapter's master socket.

The conversion it pretty straight forward, the only noticeable property of the adapter is that it has to bridge from the time unit based delta cycle protection of the OCP Modelling Kit to the delta cycle based delta cycle protection of the OCP-IP SLD r2.2.1 kit. As described in section 3.8, in the OCP Modelling Kit with default timing a call to nb_transport may happen at any delta cycle of the time of the clock cycle. However, this can be non-default in the OCP-IP SLD r2.2.1 kit (if the calls do not happen in the same delta as the event that marks the start of the OCP cycle). The same applies to non-default timing. In the OCP-IP SLD r2.2.1 kit, when a start time is set to X, then waiting for X and an additional delta cycle will suffice. In the OCP Modelling Kit, a start time being set to X means that the according nb_transport call can start at any delta (not the very first) of the given time.

Hence, the adapter will use a delta cycle protection PEQ, thereby ensuring that every call to nb_transport arrives at the very first delta cycle of the time that is one time resolution after the according start time. By that the adapter ensures:

- It is always non-default towards the legacy IP.
- It will always update the legacy channel at the very first delta of a non-default timing point.

Given that, the legacy reaction to that non-default timing (wait for the start time and a delta cycle) will work as expected by the legacy IP.

The class is defined as

```
template <typename DataCl, unsigned int BUSWIDTH=DataCl::SizeCalc::bit_size>
  class ocp_tl1_master_legacy_adapter {
    ...
    ocpip_legacy :: OCP_TL1_SlavePort<DataCl> slave_port;
    ocp_master_socket_tl1 <BUSWIDTH> master_socket;
    ...
  }
```

DataCl This template argument must be the same as for the OCP-IP SLD r2.2.1 TL1 channel that shall be connected to the adapter.

BUSWIDTH If the provided class does not provide static bit size calculation facilities (the OCP_TL1_DataCl from the legacy code base, and the data class as described in 3.9.1 both provide such facilities), or if the statically calculated size is not correct for the given use case[1], this parameter can be set manually.

slave_port The OCP-IP SLD r2.2.1 slave port to which to connect the OCP-IP SLD r2.2.1 TL1 channel.

master_socket The OCP Modelling Kit master socket to which to connect the OCP Modelling Kit slave.

The constructor is defined as

```
ocp_tl1_master_legacy_adapter (sc_core :: sc_module_name name, unsigned int max_impr_burst_length=64)
```

name The module name of the adapter.

max_impr_burst_length The maximum length of imprecise bursts that may pass this adapter. The information is required to enable the adapter to allocate large enough data buffers for imprecise bursts. If an imprecise burst passes the adapter that exceeds the maximum length the behavior is undefined.

Example: Connect a OCP-IP SLD r2.2.1 master to an OCP Modelling Kit slave.

```
1  int sc_main(int,char**){
2    //the clock
3    sc_core::sc_clock clk;
4
5    // Submodules
6    Master ms1("ms1"); //the SLD kit master
7    Slave sl1("sl1"); //the TLM-2.0 kit slave
8
9    // Set OCP configuration for slave
10   // the map was read from a file
11   ocpip::ocp_parameters params;
12   params.set_ocp_configuration("sl1", ocpParamMap);
13   sl1.ipP.set_ocp_config(params);
14
15
16   typedef  ocpip_legacy::OCP_TL1_DataCl<uint32_t, uint32_t > data_type;
17   typedef  ocpip_legacy::OCP_TL1_Channel_Clocked< data_type > channel_type;
18
19   //create the SLD kit TL1 channel
20   channel_type ch0("ocp0");
21   ch0.p_clk(clk); //connect the clock
22
23   //create the adapter
24   ocpip::ocp_tl1_master_legacy_adapter<data_type> adapter("adapter");
25
26   //connect SLD kit master to adapter via SLD kit TL1 channel
27   // the config from the TLM-2.0 slave socket will arrive at this
```

[1] E.g. the data class could use a 32 bit data type but the simulated bus width shall only be 16 bit

```
28    // channel through SLD kit config from cores, so there is no
29    // need to configure it manually
30    ms1.ipP(ch0);
31    adapter.slave_port(ch0);
32
33    //connect adapter to TLM-2.0 slave
34    adapter.master_socket(sl1.ipP);
35
36    //connect clock ports
37    ms1.clk(clk);
38    sl1.clk(clk);
39
40    //start
41    sc_core::sc_start();
42    return 0;
43  }
```

6.2.2 TL1 slave legacy adapter

This adapter has a OCP-IP SLD r2.2.1 TL1 master port, and an OCP Modelling Kit TL1 slave socket. Hence, you can connect an an OCP-IP SLD r2.2.1 TL1 slave to this adapter via an OCP-IP SLD r2.2.1 TL1 channel, and you can connect an OCP Modelling Kit TL1 master to the adapter's slave socket.

As for the master, the conversion is simple and straight forward, and the delta protection scheme adaption is performed in exactly the same way.w

---

The class is defined as

```
template <typename DataCl, unsigned int BUSWIDTH=DataCl::SizeCalc::bit_size>
    class ocp_tl1_slave_legacy_adapter {
        ...
        ocpip_legacy::OCP_TL1_MasterPort<DataCl> master_port;
        ocp_slave_socket_tl1 <BUSWIDTH> slave_socket;
        ...
    }
```

DataCl This template argument must be the same as for the OCP-IP SLD r2.2.1 TL1 channel that shall be connected to the adapter.

BUSWIDTH If the provided class does not provide static bit size calculation facilities (the OCP_TL1_DataCl from the legacy code base, and the data class as described in 3.9.1 both provide such facilities), or if the statically calculated size is not correct for the given use case[1], this parameter can be set manually.

master_port The OCP-IP SLD r2.2.1 master port to which to connect the OCP-IP SLD r2.2.1 TL1 channel.

slave_socket The OCP Modelling Kit slave socket to which to connect the OCP Modelling Kit master.

The constructor is defined as

```
ocp_tl1_slave_legacy_adapter (sc_core::sc_module_name name)
```

name The module name of the adapter.

Example: Connect a OCP-IP SLD r2.2.1 slave to an OCP Modelling Kit kit master.

```
1  int sc_main(int,char**){
2    //the clock
3    sc_core::sc_clock clk;
4
5    // Submodules
6    Master ms1("ms1"); //the TLM-2.0 kit master
7    Slave sl1("sl1"); //the SLD kit slave
8
9    // Set OCP configuration for master
10   // the map was read from a file
11   ocpip::ocp_parameters params;
12   params.set_ocp_configuration("ms1", ocpParamMap);
13   ms1.ipP.set_ocp_config(params);
14
15
16   typedef  ocpip_legacy::OCP_TL1_DataCl<uint32_t, uint32_t > data_type;
17   typedef  ocpip_legacy::OCP_TL1_Channel_Clocked< data_type > channel_type;
18
19   //create the SLD kit TL1 channel
```

```
20    channel_type ch0("ocp0");
21    ch0.p_clk(clk); //connect the clock
22
23    //create the adapter
24    ocpip::ocp_tl1_slave_legacy_adapter<data_type> adapter("adapter");
25
26    //connect TLM-2.0 master to adapter
27    ms1.ipP(adapter.slave_socket);
28
29    //connect adapter to SLD kit slave via SLD kit TL1 channel
30    // the config from the TLM-2.0 master socket will arrive at this
31    // channel through SLD kit config from cores, so there is no
32    // need to configure it manually
33    adapter.master_port(ch0);
34    sl1.ipP(ch0);
35
36    //connect clock ports
37    ms1.clk(clk);
38    sl1.clk(clk);
39
40    //start
41    sc_core::sc_start();
42    return 0;
43  }
```

Chapter 7

Monitoring Connections

Monitoring OCP Modelling Kit connection is currently done with the monitors of the OCP-IP SLD r2.2.1 kit. They are connected to the OCP Modelling Kit connections through monitor adapters. This section will explain how to instantiate and connect the monitor adapters and legacy monitors. Note that they are only available if you have installed the OCP Modelling Kit monitor package.

7.1 Connection Monitor

To enable monitoring the compile time switch USE_OCP_MONITOR has to be provided to the compiler[1] When using more than one release of the OCP Modelling Kit simultaneously that will enable the monitors in all releases. If one release is installed without monitors, setting USE_OCP_MONITOR will lead to problems (missing include files). In this case the lines conaitning ifdef USE_OCP_MONITOR in the appropriate ocpip_standard_X_Y_Z.h should be replaced with ifdef 0.

The OCP Modelling Kit provides a generic connection monitor that shall be used as a connector for OCP Modelling Kit sockets instead of connecting them directly.

The class is defined as

```
template <unsigned int BUSWIDTH> class ocp_connection_monitor;
```

BUSWIDTH The bus width in bits of the connection that shall be monitored. That template argument must match those of the master and slave socket that form the link that shall be monitored.

The constructor is defined as

```
ocp_connection_monitor(ocp_master_socket<BUSWIDTH>& msock, ocp_slave_socket<BUSWIDTH>& ssock)
```

msock The master socket of the link that shall be monitored.

ssock The slave socket of the link that shall be monitored.

Example: Connect two sockets with a connection monitor

```
1  int sc_main(int , char**){
2    //the clock
3    sc_core::sc_clock clk;
4
5    // Submodules
6    Master ms1("ms1"); //TLM-2.0 kit TL1 master
7    Slave sl1("sl1"); //TLM-2.0 kit TL1 slave
8
9    // Set OCP configuration for master
10   // the map was read from a file
11   ocpip::ocp_parameters params;
12   params.set_ocp_configuration("ms1", ocpParamMap);
13   ms1.ipP.set_ocp_config(params);
14
15   //connect the master and slave socket via connection monitor
16   ocpip::ocp_connection_monitor<32> ocp_monitor_t_piece(ms1.ipP, sl1.tpP);
17   //instead of
18   //ms1.ipP(sl1.tpP);
19
```

[1] USE_OCP_MONITOR enables the monitors in the kit with the highest version number, just as namespace ocpip refers to the highest version number. To enable monitors of a certain version you can use USE_OCP_MONITOR_ocpip_X_Y_Z. This will activate the monitors in release X.Y.Z..

```
20   //connect clock ports
21   ms1.clk(clk);
22   sl1.clk(clk);
23
24   //start
25   sc_core::sc_start();
26   return 0;
27 }
```

7.2 TL1 Monitors

To connect the TL1 monitors of the OCP-IP SLD r2.2.1, an adapter is provided that offers the OCP_TL1_MonitorIF to its environment. An arbitrary number of legacy monitors can be connected (e.g. trace or performance monitors).

The adapter class is defined as

```
template <unsigned int BUSWIDTH, unsigned int ADDRWIDTH>
  class ocp_tl1_monitor_adapter
    : public ocpip_legacy::OCP_TL1_MonitorIF<ocp_data_class_unsigned<BUSWIDTH,ADDRWIDTH> >
  {
    ...
    typedef ocp_data_class_unsigned<BUSWIDTH,ADDRWIDTH> data_class_type;
    ...
  };
```

BUSWIDTH This template argument specifies the width of the connection that is to be monitored. It shall match the appropriate template argument of the used connection monitor.

ADDRWIDTH This template argument specifies the address width of the connection that is to be monitored. It is used to determine the appropriate data type to hold the address information within the adapter.

OCP_TL1_MonitorIF As mentioned above the adapter provides the OCP-IP SLD r2.2.1 TL1 monitor interface. Note that it is fixed to use the ocp_data_class_unsigned as described in section 3.9.1. So the attached legacy monitors must use the same.

data_class_type Since the legacy monitors must use the same data class as the OCP_TL1_MonitorIF of the adapter, the adapter provides a typedef to get the correct data class type directly from the used adapter.

The constructor is define as

```
ocp_tl1_monitor_adapter(infr::monitor<BUSWIDTH, tlm::tlm_base_protocol_types>& mon);
```

mon The reference to the connection monitor to which to connect the adapter. As can be seen, the provided type is from namespace infr. However, the user does not have to use this namespace explicitly when using monitors, as the class infr::monitor is a base for the class ocp_connection_monitor.

Example: Monitor a TL1 connection with both a legacy performance and trace monitor.

```
1  int sc_main(int, char**){
2    //the clock
3    sc_core::sc_clock clk;
4
5    // OCP TLM-2.0 TL1 modules
6    Slave sl1("sl1");
7    Master ms1("ms1");
8
9    // Set OCP configuration
10   //  The map has been read from a file
11   ocpip::ocp_parameters params;
12   params.set_ocp_configuration("sl1", ocpParamMap);
13   sl1.ipP.set_ocp_config(params);
14
15   params.set_ocp_configuration("ms1", ocpParamMap);
16   ms1.ipP.set_ocp_config(params);
17
18   // netlist
19   // connect the two socket using a connection monitor
20   ocpip::ocp_connection_monitor<32> ocp_monitor_t_piece(ms1.ipP, sl1.ipP);
```

```
21
22    // attach the adapter to the connection monitor
23    ocpip::ocp_tl1_monitor_adapter<32,32> mon_adapt(ocp_monitor_t_piece);
24
25    //the clocks of the TL1 modules
26    ms1.clk(clk);
27    sl1.clk(clk);
28
29    //the monitors need to use the same type as the adapter
30    typedef ocpip::ocp_tl1_monitor_adapter<32,32>::data_class_type data_class_type;
31
32    // transaction recording perf monitor
33    scv_tr_text_init();
34    scv_tr_db db("ocp_db");
35    scv_tr_db::set_default_db(&db);
36    bool ChannelRecording = true;
37    bool SystemRecording = false;
38    ocpip_legacy::OCP_TL1_Perf_Monitor<data_class_type> pmon0("pmon0",ChannelRecording,SystemRecording);
39    pmon0.p_mon(mon_adapt);
40    pmon0.p_clk(clk);
41
42    // trace monitor
43    ocpip_legacy::OCP_TL1_Trace_Monitor_Clocked<data_class_type> tracer("Tracer", "trace.ocp");
44    tracer.p_mon(mon_adapt);
45    tracer.p_clk(clk);
46
47    // start the simulation
48    sc_core::sc_start(70, sc_core::SC_NS);
49    return 0;
50  }
```

7.3 TL2 Monitors

7.4 TL3 Monitor

For TL3 there is a simple text file logger called ocp_tl3_imc_logger. The constructor is defined as

```
template<unsigned int BUSWIDTH>
    ocp_tl3_imc_logger( infr::monitor<BUSWIDTH, tlm::tlm_base_protocol_types>& mon
        , const char* filename
        , unsigned int trace_extensions=0
        , unsigned int trace_data=0
        , unsigned int trace_be=0
        , bool check_release=false);
```

mon The reference to the connection monitor to which to connect the imc monitor. As can be seen, the provided type is from namespace infr. However, the user does not have to use this namespace explicitly when using monitors, as the class infr::monitor is a base for the class ocp_connection_monitor.

filename The name of the file that shall contain the logger's output.

trace_extensions An integer that defines the extension log level. Currently two levels are supported:

- 0 : No extension logging.
- 1 : OCP extension logging.

trace_data An integer that defines if and how to log the data array. If it is zero, the data array will not be logged. If it is greater than 0, the data array bytes will be logged grouped into lines of trace_data items.

trace_be An integer that defines if and how to log the byte enable array. If it is zero, the byte enable array will not be logged. If it is greater than 0, the byte enable array bytes will be logged grouped into lines of trace_data items.

check_release If set to true, the logger will capture the moment at which the free() method of the transaction is called. If set to false this will not happen.

Example: Monitor a TL3 connection with the ocp_tl3_imc_logger, including data array, byte enable array, and extensions.

```
int sc_main(int, char**){
  // Creates masters and slaves
  ocp_tl3_slave  sl1("sl1");
  ocp_tl3_master ms1("ms1");

  //configure the sockets
  ocpip::ocp_parameters  config;
  ocpip::map_string_type config_map=get_config_map();
  config.set_ocp_configuration(sl1.ocp.name(), config_map);
  sl1.ocp.set_ocp_config(config);
  config.set_ocp_configuration(ms1.ocp.name(), config_map);
  ms1.ocp.set_ocp_config(config);

  // Connect masters and slaves via a connection monitor
  ocpip::ocp_connection_monitor<32> ocp_monitor_t_piece1(ms1.ocp, sl1.ocp);
  ocpip::ocp_tl3_imc_logger ocp_tl3_imc_logger(ocp_monitor_t_piece1, "tl3_log.txt", 1, 4, 4, true);

  // start simulation
  sc_start(100, sc_core::SC_NS);

  return 0;

}
```

A part of the gathered data can be seen below. It is a write followed by a read. Note how the data array is grouped into a line of four bytes, and how the extensions are logged.

```
@0 s(+11 ns), delta count=0
  CALL    nb_transport_fw(0xb084c0,BEGIN_REQ)
    Command value=TLM_WRITE_COMMAND
    Address value=0x1
    StreamingWidth value=4
    ResponseState value=TLM_INCOMPLETE_RESPONSE
    DMIHint value=Not allowed
    Data array (length=4)
      [0:3]=0x1 0 0 0
    Byte Enable array not used.
    Start of Extension List
      </extension name="burst_length" type="guarded_data" value="1">
    End of Extension List
@0 s(+16 ns), delta count=0
  RETURN nb_transport_fw(0xb084c0)->TLM_COMPLETED
    Command value=TLM_WRITE_COMMAND
    Address value=0x1
    StreamingWidth value=4
    ResponseState value=TLM_OK_RESPONSE
    DMIHint value=Not allowed
    Data array (length=4)
      [0:3]=0x1 0 0 0
    Byte Enable array not used.
    Start of Extension List
      </extension name="burst_length" type="guarded_data" value="1">
    End of Extension List

+++++ release of transaction 0xb084c0 ++++
@16 ns(+12 ns), delta count=1
  CALL    nb_transport_fw(0xb084c0,BEGIN_REQ)
    Command value=TLM_READ_COMMAND
    Address value=0x2
    StreamingWidth value=4
    ResponseState value=TLM_INCOMPLETE_RESPONSE
    DMIHint value=Not allowed
    Data array (length=4)
      [0:3]=0x1 0 0 0
    Byte Enable array not used.
    Start of Extension List
      </extension name="burst_length" type="guarded_data" value="1">
    End of Extension List
@16 ns(+19 ns), delta count=1
  RETURN nb_transport_fw(0xb084c0)->TLM_ACCEPTED

@35 ns(+0 s), delta count=2
  CALL    nb_transport_bw(0xb084c0,BEGIN_RESP)
    Command value=TLM_READ_COMMAND
    Address value=0x2
    StreamingWidth value=4
    ResponseState value=TLM_OK_RESPONSE
    DMIHint value=Not allowed
    Data array (length=4)
      [0:3]=0x1 0 0 0
    Byte Enable array not used.
    Start of Extension List
      </extension name="burst_length" type="guarded_data" value="1">
    End of Extension List
@35 ns(+6 ns), delta count=2
  RETURN nb_transport_bw(0xb084c0)->TLM_UPDATED :END_RESP
    Command value=TLM_READ_COMMAND
    Address value=0x2
    StreamingWidth value=4
    ResponseState value=TLM_OK_RESPONSE
    DMIHint value=Not allowed
```

```
65      Data array (length=4)
66        [0:3]=0x1 0 0 0
67      Byte Enable array not used.
68      Start of Extension List
69        </extension name="burst_length" type="guarded_data" value="1">
70      End of Extension List
71
72  +++++ release of transaction 0xb084c0 ++++
```

Bibliography

[1] J. Aynsley. OSCI TLM-2.0 User Manual. *OSCI TLM-WG*, June 2008.

[2] OCP-IP. Open Core Protocol Specification. 2006.