A SystemC™ Generic Transaction Level Communication Channel

V2.0.2 – May 11, 2004

Document version 1.1

Revision History

Version	Date	Comment
1.0	1/15/03	Initial Generic Transaction Channel
1.0.1	3/31/03	First revision for OCP 1.0 channel
1.1	7/18/03	OCP 1.0 Sideband and layer adapters included
2.0	12/11/03	Adds updated request, response, and data handshake phase methods. Also adds additional sideband single methods. Adds descriptions of configurable master and slave models. Adds descriptions of OCP TL1 specific Enum Types and Template Classes. Updates ParamCl parameter names to conform to the parameter names in the OCP specification. Adds information about the TL2 data class and TL2 specific channel model.
2.0.2	05/11/04	Adds pre-emptive release methods.

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

Table of Contents

List of Figures

1. INTRODUCTION

This document describes a generic SystemC transaction level communication channel, applied with Open Core Protocol (OCP). The generic model is an extension to the original *SystemC™ Generic Transaction Level Communication Channel* specification (See www.systemc.org, Contributions area for more information). The generic channel is maintained for providing backward compatibility with models designed with the old generic channel model released by Open SystemC Initiative (OSCI) and older OCP channel versions released by OCP-IP (www.ocpip.org). The current release contains an updated channel model and a data class, which implements data fields required by OCP protocol.

The generic channel contains protocol primitives (functions and events), which can be used to build protocol-specific channel models. Although OCP protocol support is included in the data class of the channel, it is not recommended that the naked generic channel be used directly in new designs. The generic channel is used as a basis for OCP specific transaction channel. This channel adds an OCP API on the generic channel and OCP data class. The OCP specific channel models are described in another document: *A SystemC™ OCP Transaction Level Communication Channel* specification.

This document categorizes the communication abstraction levels according to those introduced in the white paper *SystemC™ based SoC Communication Modeling for the OCP™ Protocol*. (You can obtain a copy of this paper at www.ocpip.org.) The abstraction levels are as follows:

1. Transaction Level

   - Layer-3: Message Layer
     Model untimed functionality
     Point-point communication

   - Layer-2: Transaction Layer
     Model/analyze SoC architecture
     Start SW development
     Estimate timing

- Layer-1: Transfer Layer
  Cycle true but faster than RTL
  Detailed analysis, develop low-level SW

2. Pin Level
   - Layer-0: Register Transfer Level

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

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

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

2. DIRECTORY STRUCTURE AND CLASS HIERACHY

The generic channel is a SystemC module (`sc_module`), which uses "request/update" methods for delta cycle delayed updates of the channel state. The base generic model contains a pointer to the type of data that moves through the channel. In this case, the data is in the Open Core Protocol (OCP) Transaction Layer One or Two (TL1, TL2) format. Any type of data, even non-OCP data, can move through the generic base channel.

The Figure 1 shows the internal class hierarchy for the generic channel, with TL1 data class. The channel becomes TL2 by using TL2 data class instead.

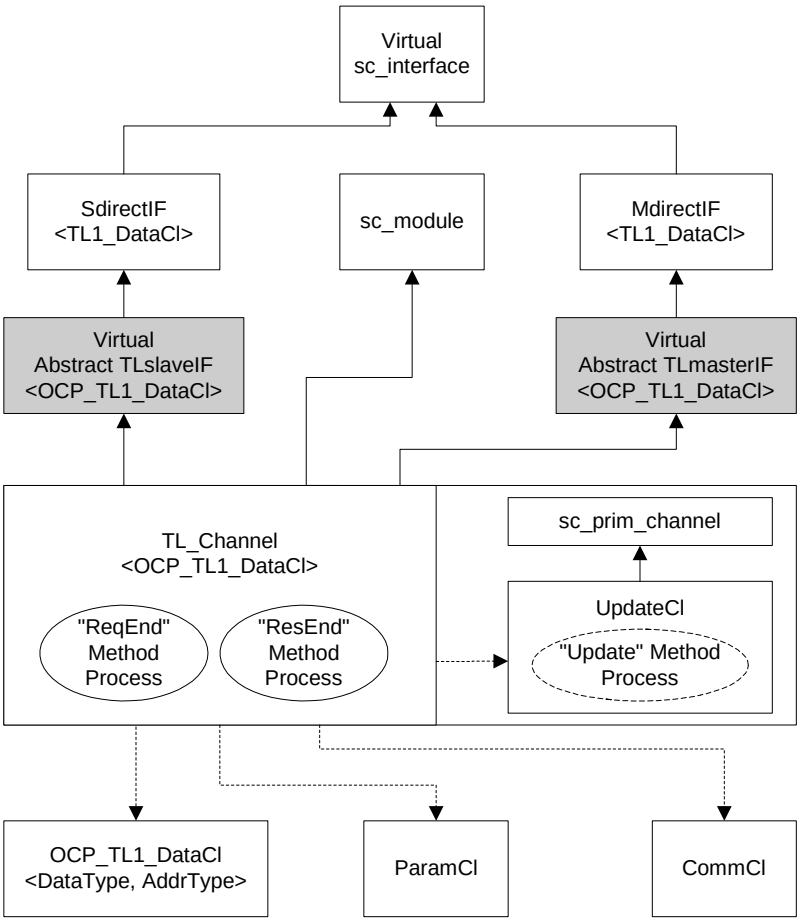

Figure 1. Generic Channel Class Hierarchy

3. TRANSACTION CHANNEL

This section describes the generic channel with the following goal: the channel maintains the "generic" interface of the OSCI transaction channel and adds the ability to move data in the OCP format across the channel. The OCP data class is included, since the generic channel does not do anything as such. The data class can be replaced with another one to support a different interface protocol.

3.1. Generic Channel Interfaces

The base generic transaction channel (in header file `tl_channel.h`) is a new version of the generic OSCI transaction channel, with minor additions explained in the next section. The channel provides synchronization for transactions, and it is used with all transaction layers. The channel can support both clocked and event-driven master and slave modules. The channel is templatized over a data class, which in this case contains member functions for implementing a version of the OCP protocol (in header files `ocp_tl1_data_cl.h` and `ocp_tl2_data_cl.h`).

The channel synchronization methods and events are visible to the masters and slaves through SystemC interface definitions (in files `tl_master_if.h`, `tl_slave_if.h and tl_direct_if.h`).

The implementation of the channel is somewhat different from the original generic channel. There is better support for clocked TL1 masters and slaves. Also, easier integration with RTL models is provided. The channel implementation is not meant to be user-modifiable and should not matter to users. For those interested, more information can be found in the channel heading and in-line comments.

4. Generic Channel Application Interface (API)

This section describes the API of the generic communication channel. In particular, it describes the constructor parameters of the channel and the methods of the channel's master and slave interface. This is followed by a description of the methods of the data classes for the OCP protocol.

4.1. Constructor Parameters of the Base Generic Channel

The base generic channel has the following constructor:

```
TL_Channel(sc_module_name name, bool Synchron = true,
           bool SyncEvent = true, bool DefaultEvent = true,
           EndTimes = false)
```

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

`Synchron` specifies whether the channel's internal states and events are updated synchronously (`Synchron = true`) or asynchronously (`Synchron = false`). OCP users always set `Synchron` to true. Asynchronous updating is slightly faster. In some applications, the update mechanism makes no difference.

`SyncEvent` specifies whether the channel's events for the synchronization of `Mput*()` and `Sget*()` as well as `Sput*()` and `Mget*()` methods are triggered (`SyncEvent = true`) or not. OCP users always set `SyncEvent` to true. The channel may be slightly faster if no synchronization events are used. Use the default value (`= true`) unless you know exactly what you are doing.).

`DefaultEvent` specifies whether the channel should trigger the default event. The channel may be slightly faster if no default event is triggered. `DefaultEvent` can be false if none of the attached modules are sensitive to port events. Use the default value (= true) unless you know exactly what you are doing.

`EndTimes` specifies whether the channel should record transaction end time. This information is used only by some OCP transactions, and the OCP Channel sets the parameter automatically if needed. Since this slows the channel down considerably, always use the default value when instantiating the generic channel.

4.1.1. Base Generic Class Definition

The base generic channel is templatized over the data class. The data class itself is templatized over the data type and the address type. In the following subsections, `TdataCl` denotes the template data class argument (*template<class TdataCl> class TL_Channel*).

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

4.1.2. Generic Master Interface (tl_master_if.h)

This section describes the interface methods for the master.

`TdataCl * GetDataCl()`

Purpose: Gets the pointer to the data class of the channel.

Return: Returns the pointer immediately.

`bool MgetSbusy()`

Purpose: Status of the slave-busy semaphore. `MgetSbusy()` indicates whether the slave has released the previous request.

Return: Immediately returns true if the slave has not released the last request. Returns false if it has.

Events: No event.

`bool MgetSbusyData()`

Purpose: Status of the data-busy semaphore. `MgetSbusyData()` indicates whether the slave has released the previous data request. Used only with TL protocols, which have separate data and request phases.

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

Events: No event.

`bool MputWriteRequestBlocking()`

Purpose: Issues a write request to the slave. The master may write to the channel's data class any time outside this call by either copying or pointer passing.

Return: Suspends calling thread. Resumes and returns after the slave has released the request channel. Returns true at success, false at failure.

Events: Triggers the default event and request start event.

`bool MputWriteRequest()`

Purpose: Issues a write request to the slave. `MputWriteRequest()` should be called only after `MgetSbusy()` returns false; that is, the slave is no longer using the channel data buffer. Both copy and pointer-passing put data methods (of the data class) can be used, but the master must not reuse the passed data buffer until `MgetSbusy()` is false. Must be called only once per clock cycle in TL1 models.

Return: Returns immediately. Master must suspend itself to allow slave process to run. Returns true if channel accepts the request, false if not.

Events: Triggers the default event and request start event. No event when the return value is false.

`bool MputDataRequestBlocking()`

Purpose: Issues a data write request to slave. The master may write to the data handshake group of the channel's data class any time outside this call by either copying, or pointer passing. Used only with TL protocols, which have separate data and request phases.

Return: Suspends the calling thread. Returns after the slave has released the data request channel. Returns true on success, false on failure.

Events: Triggers the default event and data request start event.

`bool MputDataRequest()`

Purpose: Issues a data write request to the slave. `MputDataRequest()` should be called only after `MgetSbusy()` returns false; that is, the slave is no longer using the channel data buffer. Both copy and pointer-passing put data methods can be used, but the passed data buffer must not be re-used by the master until `MgetSbusyData()` is false. `MputDataRequest()` is used only with TL protocols, which have separate data and request phases. Must be called only once per clock cycle in TL1 models.

Return: Returns immediately. The master must suspend itself to allow the slave process to run. Returns true if channel accepts the request, false if not.

Events: Triggers the default event and data request start event. No event when return-value is false.

`bool MputReadRequestBlocking()`

Purpose: Issues a read request to the slave. The master may read the response data only after `MgetResponse*()` call returns true.

Return: Returns after the slave has released the request channel. Returns true on success and false on failure. Suspends calling thread.

Events: Triggers the default event and request start event. No event when return-value is false.

`bool MputReadRequest()`

Purpose: Issues a read request to the slave. Should be called only when `MgetSbusy()` returns false. The master may read the response data only after an `MgetResponse*()` call returns true. Must be called only once per clock cycle in TL1 models.

Return: Returns immediately. The master must suspend itself to allow the slave process to run. Returns true if channel accepts the request, false if not.

Events: Triggers the default event and request start event. No event when return-value is false.

`bool MgetResponseBlocking(bool Release)`

Purpose: Gets the response from the slave and suspends the calling thread. `MgetResponseBlocking()` can only be called after `Mput*Request()`. After this command returns, the channel's data class will contain the response data.

Parameters: If `Release` is true, the response channel is released immediately; otherwise, the response channel is not released until `Mrelease()` is called. If channel parameter `respaccept` is 0, Release parameter has no effect; the channel is released a delta cycle after the request automatically.

Return: Returns true at success and false on failure. Returns after the slave has called the `SputResponse()` method.

Events: No event.

`bool MgetResponse(bool Release)`

Purpose: Gets the response from the slave. If this command returns true, the channel's data class will contain the response data.

Parameters: If `Release` is true, the response channel is released immediately. Otherwise the response channel is not released until `Mrelease*()` is called. The release mechanism is equal to calling `Mrelease()` after `MgetResponse(false)`. If called at clock edge, the channel is released for the next clock (two cycle transaction). If channel parameter `respaccept` is 0, Release parameter has no effect; the channel is released a delta cycle after the request automatically.

Return: Returns immediately. Returns true after the slave has responded to the read request, false before. Master must suspend itself to allow the slave process to run.

Events: No event.

`bool MgetResponsePE()` **deprecated**

Purpose: Gets the response from the slave. `MgetResponsePE()` can only be called after `Mput*Request()`. The channel's data class contains valid data, and the channel's data pointer can be used by the master when this call returns true, and before `Mput*Request()` is called again. Use only in conjunction with `MreleasePE()`. Use only with clocked processes.

Parameters: None.

Return: Returns immediately. Returns true after the slave has responded to the read request, false before. Master must suspend itself to allow the slave process to run.

Events: No event.

`void Mrelease()`

Purpose: Releases the response channel. Note that the calling thread is not suspended, and the slave thread cannot run until the master suspends itself. Can be called only after a response is detected on the channel. In fully clocked TL1 models, this means that the slave sees the release at the next clock edge, resulting to a minimum of two-cycle transaction.

Return: Returns immediately. No return value.

Events: No event.

`void MreleasePE()`

Purpose: Preemptively releases the response channel; that is, without knowing if there is a response. This is used in TL1 masters, which do not want to block the next response. The primary purpose of this method is to allow TL1 masters use single-cycle response handshake. Can be called at any time during a clock cycle. This causes a channel update, and the pending and all the following responses are released until `MunreleasePE()` is called. This method should not be used in TL2 modeling.

Return: Returns immediately. No return value.

Events: No event.

`void MunreleasePE()`

Purpose: Removes preemptive release on the response channel. This is used in TL1 masters, which do want to block the next response. Can be called at any time during a clock cycle. This causes a channel update, and the pending or the following response is blocked until `Mrelease*()` is called. This method should not be used in TL2 modeling.

Return: Returns immediately. No return value.

Events: No event.

`void Mrelease(sc_time Time)`

Purpose: Releases the response after `Time` time units. Note that the calling thread is not suspended, and the slave thread cannot run until the master suspends itself. This call is primarily used with TL2 and TL3.

Return: Returns immediately. No return value.

Events: No event.

`void MregisterDirectIF(MdirectIF<TdataCl > *MasterDirectIF)`

Purpose: This method registers the direct interface of the master at the channel. It must be called if the master has implemented the `SputDirect()` method, which can be used by the slave to directly read or write data to the master without affecting the timing of the system.

Return: Returns immediately. No return value.

Events: No event.

```
bool MputDirect(int MasterID, bool IsWrite,
                Td* DataPointer, Ta Address, int NumWords)
```

Purpose: This method belongs to the direct interface of the slave, and it must be implemented in the slave. This method allows the master to directly read or write data to the slave without affecting the timing of the system. If the slave has implemented this method, the slave must register the method at the channel by calling the `SregisterDirectIF()` method of the channel.

Return: Returns immediately, returning true on success and false on failure. A return value of false usually means that the slave has not implemented this method; that is, the slave does not support direct access.

Events: No event.

4.1.3. Generic Slave Interface (tl_slave_if.h)

This section describes the methods for the slave's interface.

`TdataCl * GetDataCl()`

Purpose: Gets the pointer to the data class of the channel.

Return: Returns the pointer immediately.

`bool IsWrite()`

Purpose: Indicates whether current request is a read or a write transfer. Other request types can be communicated over the data class (in case of OCP, through the `MCmd` field).

Return: false = Read transfer

true = Write transfer

`bool SgetMbusy()`

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

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

Events: No event.

`bool SgetRequestBlocking(bool Release)`

Purpose: Blocks execution until the master signals a request event. Channel data can be used until the slave thread suspends; that is, meets a `wait()` call.

Parameters: If `Release` is true, the request channel is released immediately; otherwise, the request channel is not released until `Srelease()` is called. If channel parameter `cmdaccept` is 0, Release parameter has no effect; the channel is released a delta cycle after the request automatically.

Return: Suspends the calling thread. Returns after a read or write event from the master.

Events: No events.

`bool SgetRequest(bool Release)`

Purpose: Gets a request. If true, channel data can be used until the slave thread suspends; that is, meets a `wait()` call.

Parameters: If `Release` is set true, the request channel is released immediately; otherwise it is not released until `Srelease*()` is called. The release mechanism is equal to calling `Srelease()` after `SgetRequest(false)`. If called at clock edge, the channel is released for the next clock (two cycle transaction). If channel parameter `cmdaccept` is 0, Release parameter has no effect; the channel is released a delta cycle after the request automatically.

Return: Returns true when the master request is pending, returns immediately.

Events: No events.

`bool SgetRequestPE()` **deprecated**

Purpose: Gets a request. If true, channel data can be used until the slave thread suspends; that is, meets a `wait()` call. Use only with `SreleasePE()`. Use only with clocked processes.

Parameters: None.

Return: Returns true when a master request is pending, returns immediately.

Events: No events.

`bool SgetDataRequest(bool Release)`

Purpose: Gets a data request. If true, channel data can be used until the slave thread suspends; that is, it meets a `wait()` call.

Parameters: If `Release` is set to true, the data request channel is released immediately; otherwise, it is not released until `SreleaseData()` is called. The release mechanism is equal to calling `SreleaseData()` after `SgetDataRequest(false)`. If called at clock edge, the channel is released for the next clock (two cycle transaction). If channel parameter `dataaccept` is 0, Release parameter has no effect; the channel is released a delta cycle after the request automatically. Used only with TL protocols, which have separate data and request phases.

Return: Returns true when master data request is pending, returns immediately.

Events: No events.

`bool SgetDataRequestPE()` **deprecated**

Purpose: Gets a data request. If true, channel data can be used until the slave thread suspends; that is, it meets a `wait()` call. Use this method only with `SreleaseDataPE()`. Use only with clocked processes.

Parameters: None.

Return: Returns true when a master data request is pending, returns immediately.

Events: No events.

`bool SgetDataRequestBlocking(bool Release)`

Purpose: Blocks execution until the master signals a data request event. Channel data can be used until the slave thread suspends; that is, meets a `wait()` call.

Parameters: If `Release` is true, the request channel is released immediately; otherwise, the data request channel is not released until `SreleaseData*()` is called. If channel parameter `dataaccept` is 0, Release parameter has no effect; the channel is released a delta cycle after the request automatically. Used only with TL protocols, which have separate data and request phases.

Return: Suspends calling thread. Returns after read or write event from the master.

Events: No events.

`bool SputResponseBlocking()`

Purpose: Issues a response to the master. `SputResponseBlocking()` can be called only after a request is detected by `SgetRequest()`. The response data can only be written between these get and put calls.

Return: Suspends calling thread. Returns after the master called `Mrelease*()`. Returns true at success and false at failure.

Events: Triggers default event, and response event. No event if returns false.

`bool SputResponse()`

Purpose: Issues a response to the master. `SputResponse()` can be called only after a request is detected by `SgetRequest()` or `SgetRequestBlocking()`. The response data can only be written between these get and put calls. Must be called only once per clock cycle in TL1 models.

Return: Returns immediately. Returns true if channel accepts the response, false if not.

Events: Triggers default event, and response event. No event if returns false.

`void Srelease()`

Purpose: Releases the request channel. Note that the calling thread is not suspended, and the master thread cannot run until the slave suspends itself.

Return: Returns immediately. No return value.

Events: No events.

`void SreleasePE()`

Purpose: Releases request channel preemptively. See `MreleasePE()`.

Return: Returns immediately. No return value.

Events: No events.

`void SunreleasePE()`

Purpose: Removes preemptive release from request channel. See `MunreleasePE()`.

Return: Returns immediately. No return value.

Events: No events.

`void Srelease(sc_time Time)`

Purpose: Releases the request channel after `Time` time units. Note that the calling thread is not suspended, and the master thread cannot run until the slave suspends itself.

Return: Returns immediately. No return value.

Events: No events.

`void SreleaseData()`

Purpose: Releases the data request channel. Note that the calling thread is not suspended, and the master thread cannot run until the slave suspends itself. Used only with TL protocols, which have separate data and request phases.

Return: Returns immediately. No return value.

Events: No events.

`void SreleaseDataPE()`

Purpose: Releases the data request channel preemptively. Used only with TL1 protocols, which have separate data and request phases. See `MreleasePE()`.

Return: Returns immediately. No return value.

Events: No events.

`void SunreleaseDataPE()`

Purpose: Removes preemptive release from the data request channel. Used only with TL1 protocols, which have separate data and request phases. See `MunreleasePE()`.

Return: Returns immediately. No return value.

Events: No events.

`void SreleaseData(sc_time Time)`

Purpose: Releases the data request channel after `Time` time units. Note that the calling thread is not suspended, and the master thread cannot run until the slave suspends itself. Used only with TL1 protocols, which have separate data and request phases.

Return: Returns immediately. No return value.

Events: No events.

`void SregisterDirectIF(SdirectIF<TdataCl > *SlaveDirectIF)`

Purpose: Registers the direct interface of the slave at the channel. This method must be called if the slave has implemented the `MputDirect()` method, which can be used by the master to directly read or write data to the slave without affecting the timing of the system

Return: Returns immediately. No return value.

Events: No event.

```
bool SputDirect(int SlaveID, bool IsWrite, Td* DataPointer,
                Ta Address,  int NumWords)
```

Purpose: This method belongs to the direct interface of the master and must be implemented in the master. This method allows the slave to directly read or write data to the master without affecting the timing of the system. If the master has implemented this method, the master must register the method at the channel by calling the `MregisterDirectIF()` method of the channel.

Return: Returns immediately, returning true on success and false on failure. A return of false usually means that the master has not implemented this method; that is, the master does not support direct access.

Events: No event.

4.1.4. Reset Methods (common for tl_master_if.h and tl_slave_if.h)

This section describes the methods for the reset methods

```
void reset()
```

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

Return: Void

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

```
void remove_reset()
```

Purpose: Removes reset state from the channel after a delta cycle delay.

Return: Void

Events: ResetEndEvent.

```
bool get_reset()
```

Purpose: Tests if channel is in reset state.

Return: True if reset, false otherwise.

Events: None.

4.2. OCP TL1 Data Class

A data class provides an implementation for a bus protocol. A user may create a data class that suits a particular purpose. The data class implements the data access methods for the protocol. The typical minimum data of any protocol are read data, write data, and address. The data class should also implement data protection, such as current-next copies of the data fields. The generic channel assumes that the data class implements public methods `update_Fw(int eventSelect)`, `update_FwD(int eventSelect)`, and `update_Bw(int eventSelect)`, which are called at request, data request, and response updates.

We provide a data class for OCP protocol with the generic channel release package. A user may modify this class for use with other protocols. A rudimentary OCP TL1 channel can be created by using the generic channel class template with an argument of the `OCP_TL1_DataCl` class, which is provided in the `include/ocp_tl1_data_cl.h` header file. The `OCP_TL1_DataCl` class contains private member variables and public access methods such that, by using them, OCP dataflow transactions and sideband signals can be exchanged between the master and slave ports.

Notice that the generic channel with the OCP_TL1_DataCl described in this section is not the same as the OCP specific TL1 channel. The generic channel and the OCP data class together provide a foundation for the OCP specific TL1 channel, which implements an easy-to-use API for the OCP. For more information on the OCP TL1 specific channel, see the document *A SystemC™ OCP Transaction Level Communication Channel*.

In general, for each master-driven OCP signal, `M*`, there is a corresponding `MputM*()` driving method. In addition, `SgetM*()` methods also exist for those signals and can be used by the slave to sample the signals. Similarly for slave-driven OCP signals, `S*`, corresponding `SputS*()` and `MgetS*()` methods are provided. As for the OCP control and status sideband signals, access methods for the system side and the core side are distinguished by having the "Sys" prefix and the "C" prefix, respectively.

This naming scheme makes the behavior of most of the channel methods obvious. For those that are not, some explanation is given in the following sections.

4.2.1. Enumerator Types

Two enumerator types `OCPMCmdType` and `OCPSRespType` are defined in an included header file with the encoding as specified for the OCP `MCmd` signal and `SResp` signal, respectively. The encodings are named according to the OCP specification (See www.ocpip.org).

The MCmd signal is encoded as follows:

```
enum OCPMCmdType {
      OCP_MCMD_IDLE = 0,
      OCP_MCMD_WR,
      OCP_MCMD_RD,
      OCP_MCMD_RDEX,
      OCP_MCMD_RESERVED4,
      OCP_MCMD_WRNP,
      OCP_MCMD_RESERVED6,
      OCP_MCMD_BCST
};
```

The SResp signal is encoded as follows:

```
enum OCPSRespType {
      OCP_SRESP_NULL = 0,
      OCP_SRESP_DVA,
      OCP_SRESP_RESERVED2,
      OCP_SRESP_ERR
};
```

4.2.2. Mandatory (and Generic) Data Class Member Functions

There are several methods used by the channel itself and hence must always be defined. These methods do not follow the general naming scheme described in the previous subsection because they are not used through masters or slaves. The mandatory methods are:

```
bool IsWriteRequest()
{
      // OCP implementation of the method
      return((m_MCmd[1 – m_ReqToggle] == OCP_MCMD_WR) ? true : false);
}

void SetWriteRequest()
{
      // OCP implementation of the method
      m_MCmd[m_ReqToggle] = OCP_MCMD_WR;
}

void SetReadRequest()
{
      // OCP implementation of the method
      m_MCmd[m_ReqToggle] = OCP_MCMD_RD;
}
```

4.2.3. OCP Request Group Signals

The request group signals can be written by the master (`Mput*`) when the `MgetSbusy()` channel call returns false before `Mput*Request*()`. Typically, the request group signals are written at a rising clock edge or a small delay later (the latter one with RTL/TLM cosimulation).

The signals can be read by the slave (`Sget*`) when the `SgetRequest()` channel function returns true at a clock rising edge or at channel default event. The methods for the OCP request group signals are as follows:

```
void MputMAddr(Ta a)
Ta   SgetMAddr()

void MputMAddrSpace(int a)
int  SgetMAddrSpace()

void MputMByteEn(int a)
int  SgetMByteEn()

void        MputMCmd(OCPMCmdType a)
OCPMCmdType SgetMCmd()
OCPMCmdType SreadMCmd(void) const

void MputMConnID(int a)
int  SgetMConnID()

void MputMData(Td d)
void SgetMData(Td &d)
Td   SgetMData() // This is a different form of SgetMData()

void MputMThreadID(int a)
int  SgetMThreadID()

void MputMBurstPrecise(bool a)
bool SgetMBurstPrecise(void) const
void MputMBurstSeq(OCPMBurstSeqType a)
OCPMBurstSeqType SgetMBurstSeq(void) const
void MputMBurstSingleReq(bool a)
bool SgetMBurstSingleReq(void) const
void MputMReqLast(bool a)
bool SgetMReqLast(void) const
```

Warning: `SgetMCmd()` returns the OCP `MCmd` field and resets the MCmd signal; that is, the `SgetMCmd()` is not persistent. This behavior, although different from the OCP MCmd signal in the RTL level, is alright because the data class calls are not used for synchronization. `SreadMCmd()` does not reset MCmd.

4.2.4. OCP Data Request Group Signals

The data request group signals can be written by the master (`Mput*`) when the `MgetSbusyData()` channel call returns false before `MputDataRequest*()`. Typically,

the request group signals are written at a rising clock edge or a small delay later (the latter one with RTL/TLM co-simulation). These signals are to be used with OCP interfaces, which have data handshake enabled. Notice that the `MputMDataHS()` and `SgetMDataHS()` are used instead of `MputMData()` and `SgetMData()`. Those calls should not be intermixed.

The signals can be read by the slave (`Sget*`) when the `SgetDataRequest()` channel function returns true, at a clock rising edge, or at channel default event. The methods for the OCP data request group signals is as follows:

```
void MputMDataHS(Td d)
void SgetMDataHS(Td &d)
Td   SgetMDataHS()

void MputMDataValid(bool a)  //Should not be used
bool SgetMDataValid()
bool SreadMDataValid(void) const

void MputMDataThreadID(int a)
int  SgetMDataThreadID()
```

Warning: `SgetMDataValid()` returns the OCP `MDataValid` field and resets the master's MDataValid signal; that is, the `SgetMDataValid()` is not persistent. This behavior, although different from OCP MDataValid signal in RTL level, is all right because the data class calls are not used for synchronization. `SreadMDataValid(void) const` does not reset the MDataValid.

4.2.5. OCP Response Group Signals

The response group signals can be written by the slave (`Sput*`) when the `SgetMbusy()` channel call returns false, before `Sput*Response*()`. Typically, the request group signals are written at a rising clock edge or a small delay later (the latter one with RTL/TLM co-simulation).

The signals can be read by the master (`Mget*`) when the `MgetResponse()` channel function returns true, at a clock rising edge, or at channel default event. The methods for the OCP response group signals is as follows:

```
void SputSData(Td d)
void MgetSData(Td &d)
Td   MgetSData()

void         SputSResp(OCPSRespType a)
OCPSRespType MgetSResp()
OCPSRespType MreadSResp(void) const
```

```
void SputSThreadID(int a)
int  MgetSThreadID()

void SputSDataInfo(unsigned int d)
void MgetSDataInfo(unsigned int &d) const

void SputSRespInfo(unsigned int d)
void MgetSRespInfo(unsigned int &d) const
void SputSRespLast(bool d)
void MgetSRespLast(bool &d) const
bool MgetSRespLast(void) const
```

Warning: `MgetSResp()` returns the OCP `SResp` field and changes the slave's `SResp` signal; that is. `MgetSResp()` is not persistent. This behavior, although different from OCP `SResp` signal in the RTL level, is alright because the data class calls are not used for synchronization. `MreadSResp()` does not reset SResp.

4.2.6. Example: Address Transfer Methods

The address transfer methods are shown here as an example to illustrate how the private member variables, *m_MAddr[0]* and *m_MAddr[1],* are set when their access methods are called. The `MputMAddr()` method is used to drive a new address onto the OCP `MAddr` signal. The `SgetMAddr()` method is used to sample the OCP `MAddr` signal. Note that the data class is templatized over the address type (`Ta`). This allows switching between, for example, 32-bit addresses and 64-bit addresses without rewriting code. The toggling happens at least one delta cycle after the transaction is initiated through the channel. This provides the necessary inertia so that data does not just trickle through the channel. The update process can be either an inertial update or an immediate update, depending on the setting of the `m_Synchron` variable. The inertial update ensures that current data members are never updated at the same clock edge they are read. This is required in TL1 for independence of thread execution order.

The following are the example methods.

```
void MputMAddr(Ta a) {
      m_MAddr[m_ReqToggle] = a;
}

Ta SgetMAddr() {
      return m_MAddr[1 – m_ReqToggle];
}
```

The update_Fw(int eventSelect) method is called by the generic channel at the request update() phase, toggling the toggle:

```
void update_Fw(int eventSelect) {

        .. .

        m_ReqToggle = 1 - m_ReqToggle;

        .. .

}
```

4.2.7. TL1 versus RTL

The TL1 protocol sequences are similar to RTL sequences, with only small differences. It is possible to recreate accurate OCP timing diagrams with correctly constructed masters and slaves. The OCP fields internal to the data class do not follow OCP timing accurately because the generic channel synchronization must be combined with the data class in order to implement the full protocol. For example, the `MCmd` field is reset when `SgetMCmd()` is called. Therefore, to recreate the MCmd signal for RTL, `SgetRequest()`, `Srelease()`, and the clock must be used.

Because the TL1 event model is far simpler than the RTL event model for purposes of simulation speed, arbitrary RTL delays and signal glitches cannot be fed into TL1 channel. The data fields must remain stable once the transaction is committed. This causes extra difficulties for creating RTL-TLM converters similar to co-simulation of RTL and cycle-based models. The difficulties arise from the very essence of the abstraction levels and are unavoidable in our opinion. The RTL side of the RTL-TLM converter must take care of glitch removal.

4.2.8. Example: Sending and Receiving Write Transactions

The following pseudo-code segments show, as an example, how an OCP write transaction can be sent over an OCP channel using the `OCP_TL1_DataCl` class' public methods. It is assumed that this is a posted write transaction, thus, no response is sent.

```
// ----------------------------
// on the master sending side
// ----------------------------
wait(); // Wait for clock rising edge

if (!MgetSbusy()) { // Check if slave can accept request
      MputMConnID(0);
      MputMThreadID(0);
      MputMAddr(0x30);
      MputMData((Td)wr_data);

      MputWriteRequest();
}

// ----------------------------
// on the slave receiving side
// ----------------------------

wait(); // Wait for clock rising edge
```

```
if (SgetRequest(true)) {
       int mconnid = SgetMConnID();
       int mthreadid = SgetMThreadID();
       Ta address = SgetMAddr();
       OCPMCmdType mcmd = SgetMCmd();
       Td wr_data = SgetMData();
}
```

Calling the `MputMConnID()`, `MputMThreadID()`, and `MputMAddr()` methods sets the OCP MConnID, MThreadID, and MAddr signals, respectively. The address passed into the `MputMAddr()` method should be on an OCP-word boundary. The `MputWriteRequest()` call indicates that an OCP write transaction is going to be delivered. The `MputMData()` method is used to send the write data onto the OCP channel.

On the slave side, the `SgetMConnID()`, `SgetMThreadID()`, `SgetMAddr()`, and `SgetMCmd()` methods are called to retrieve values of the OCP MConnID, MThreadID, MAddr, and MCmd signals, respectively. The `SgetMData()`method is used to receive the write data.

Note that the slave sees the request at the next clock edge after the master has called `MputWriteRequest(),` and the master sees the release at the next cycle after the slave has released the channel with `SgetRequest(true)`-method. A transaction last therefore two cycles in this example.

In the following pseudo-code, the slave uses pre-emptive release method to release all incoming requests. The slave must be prepared to get the request, and consume the data at every clock cycle. A transaction lasts therefore one cycle in this example.

```
// ----------------------------
// on the master sending side
// ----------------------------
wait(); // Wait for clock rising edge
if (!MgetSbusy()) { // Check if slave can accept request
      MputMConnID(0);
      MputMThreadID(0);
      MputMAddr(0x30);
      MputMData((Td)wr_data);

      MputWriteRequest();
}
// ----------------------------
// on the slave receiving side
// ----------------------------
SreleasePE(); // Release the channel for all transactions
wait(); // Wait for clock rising edge

if (SgetRequest(false)) { // the SreleasePE() overrides the argument
       int mconnid = SgetMConnID();
       int mthreadid = SgetMThreadID();
       Ta address = SgetMAddr();
       OCPMCmdType mcmd = SgetMCmd();
```

```
        Td wr_data = SgetMData();
}
```

4.2.9. Example: Sending and Receiving Read Responses

The following pseudo-code segments show, as an example, how a single-OCP-word read transaction can be exchanged between the master and slave of an OCP connection.

```
// ----------------------------
// on the master sending side
// ----------------------------
// sending a read request

wait(); // Wait for rising clock edge

MputMAddr(0x30);
MputReadRequest();
…..
while(true) {
      wait();
      // receiving a read response and data
      if (MgetResponse(1)) {
            OCPSRespType sresp = MgetSResp();
            if (sresp == OCP_SRESP_DVA) {
                 Td rd_data = MgetSData();
            }
            break;
      }
}
```

```
// ---------------------------
// on the slave side
// ---------------------------
wait(); // Wait for rising clock edge

if (SgetRequest(1)) {
      Ta address = SgetMAddr();
      OCPMCmdType mcmd = SgetMCmd();
      if (mcmd == OCP_MCMD_RD) {
           // Response code (data valid)
           SputSResp(OCP_SRESP_DVA);
           SputSData(rd_data);
           SputResponse();
      }
}
```

This time the `MputMCmd()` and `MputBurstLen()` calls together send a single-OCP-word read request to the OCP channel. When the slave receives a request, it first checks whether it is a read request. In this example, after the read data is obtained, the slave sends back an OCP DVA response and the single-OCP-word read data by calling the `SputSResp()` and `SputSData()` methods, respectively.

On the master side, when a read response has arrived, the master uses the `MgetSResp()` and `MgetSData()` methods to retrieve the status of the response and the read data, respectively.

4.3. OCP TL2 Data Class

In principle, the same data class could be used for TL2 and TL1. We have created separate classes, since the TL2 requirements are much simpler, and the data class can be implemented more efficiently. For example, current-next copies of data are not needed since there is no clock, and no sub-clock cycle events. The data integrity can be fully guaranteed with the flow control that the generic channel provides. Pointer passing is usually sufficient, without need to copy the data at all. (Destruction of buffers is the responsibility of the sender.)

We provide a TL2 data class for OCP protocol with the generic channel release package. A user may modify this class for use with other protocols. A rudimentary OCP TL2 channel can be created by using the generic channel class template with an argument of the `OCP_TL2_DataCl` class, which is provided in the `include/ocp_tl2_data_cl.h` header file. The `OCP_TL2_DataCl` class contains private member variables and public access methods such that, by using them, OCP dataflow transactions and sideband signals can be exchanged between the master and slave ports. (See also section 4.2 “OCP TL1 Data Class.”)

In general, for each master-driven OCP signal, M*, there is a corresponding `MputM*()` driving method. In addition, `SgetM*()` methods also exist for those signals and can be used by the slave to sample the signals. Similarly for slave-driven OCP signals, S*, corresponding `SputS*()` and `MgetS*()` methods are provided. As for the OCP control and status sideband

signals, the access methods for the system side signals are distinguished with “Sys” prefix, and the core side signals are distinguished with a “C” prefix. This naming scheme makes the behavior of most of the channel methods obvious. For those that are not, some explanation is given in the following sections.

To the user, the TL2 data class looks very similar as the TL1 with a few additional members. The TL2 transaction typically contains a full OCP burst. The data fields are set at the beginning of the burst and stay constant during the burst. For example, the address field of the transaction is the first address of the burst, and the slave derives the other addresses from the burst related fields.

Because some fields, like byte enable, may change at each transfer of a burst, it is possible to break the burst into several transactions (sometimes called “chunks”) when necessary. To do so, four additional data class members that are TL2-specific have been added: `MreqChunkLen`, `SrespChunkLen`, `MreqChunkLen`, `MReqChunkLast`, and ‘`SrespChunkLen`. The `MreqChunkLen` and `SrespChunkLen` members are used to specify the chunk length for both request and response transfers. The `MReqChunkLast` and `SRespChunkLast` members indicate if the current chunk is the last one of a complete OCP request/response burst or not. Usage of these members is illustrated on Figure 4.

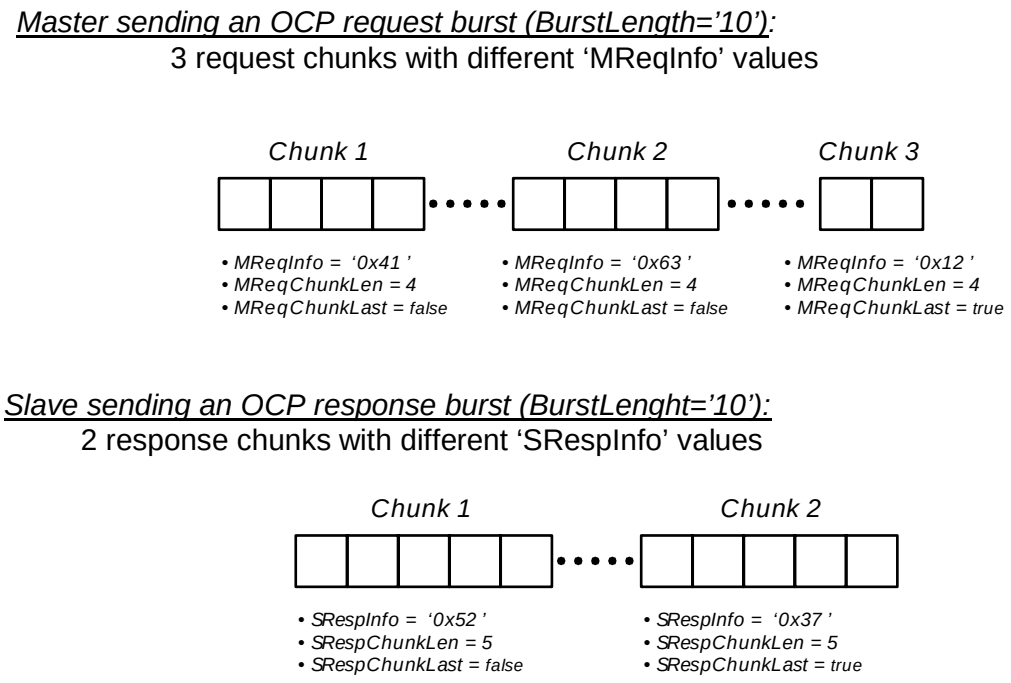

Figure 2. Usage of the chunk-related data class members

4.3.1. OCP Request Group Signals

The request group signals can be written by the master (`Mput*`) right before the `Mput*Request*()` call. The data protection toggle switches state at each request. The slave may read the signals (`Sget*`) right after the `SgetRequest*()` returns true.

The following are the methods for the OCP request group signals:

```
void MputMAddr(Ta a)
Ta   SgetMAddr()

void          MputMAddrSpace(unsigned int a)
unsigned int  SgetMAddrSpace()


void MputAtomicLen(unsigned int a)
```

Purpose: Sets Request Transaction data length. Can be used in lieu of w-parameter of `MputMData()` function. Not part of OCP 1.0 but necessary for TL2 to work.

```
unsigned int SgetAtomicLen()
```

Purpose: Returns the Request Transaction data length. Can be used in lieu of w-parameter of `SgetMData()` function. Not part of OCP 1.0, but necessary for TL2 to work.

```
void             MputMBurstSeq(OCPMBurstType a)
OCPMBurstSeqType SgetMBurstSeq()

void MputMByteEn(int a)
int  SgetMByteEn()

void        MputMCmd(OCPMCmdType a)
OCPMCmdType SgetMCmd()

void MputMConnID(int a)
int  SgetMConnID()

void         MputMThreadID(unsigned int a)
unsigned int SgetMThreadID()

void         MputMBurstLength(unsigned int a)
unsigned int SgetMBurstLength()

void MputMBurstPrecise (bool a)
bool SgetMBurstPrecise ()
```

```
void MputMBurstSingleReq(bool a)
bool SgetMBurstSingleReq()

void         MputMReqInfo (unsigned int a)
unsigned int SgetMReqInfo ()

void MputMReqLast(bool a)
bool SgetMReqLast()


void MputMData(Td* d, unsigned int w = 1,
               bool last_of_a_burst = true)
```

Parameters:

> `d` is the pointer to data array
> `w` is the data array length (request chunk length)
> `last_of_a_burst` is the last datum of a burst transfer is sent with this transaction

```
Td* SgetMData(unsigned int& w, bool& last_of_a_burst)
```

Parameters:

> `w` is the data array length (request chunk length)
> `last_of_a_burst` is the last datum of a burst transfer is sent with this chunk.

Return: Pointer to data array

```
Td* SgetMData(int& w)
```

Parameters: `w` is the data array length (request chunk length)

Return: Pointer to data array

```
void MputMReqChunkLen(unsigned int w)
```

Purpose: Sets the request chunk length. `MputMReqChunkLen()` can be used in instead of `w` parameter of the `MputMData()` function (for example,. to send a multiple-chunk READ request). This method is not part of OCP 2.0, but necessary for TL2 to work.

`unsigned int SgetMReqChunkLen()`

Purpose: Returns the request chunk length. `SgetMReqChunkLen()` can be used instead of `w` parameter of the SgetMData() function (for example, to get a multiple-chunk READ request). This method is not part of OCP 2.0 but is necessary for TL2 to work.

`void MputMReqChunkLast(bool w)`

Purpose: Useful for informing a slave that this chunk is the last of a complete request burst. `MputMReqChunkLast()` can be used instead of the `last_of_a_burst` parameter of the `MputMData()` function (for example, to send a multiple-chunk READ request). This method is not part of OCP 2.0 but is necessary for TL2 to work.

`unsigned int SgetMReqChunkLast()`

Purpose: Determine if this chunk is the last of a complete request burst. Can be used in lieu of last_of_a_burst-parameter of SgetMData() function (e.g. to get a multiple-chunk READ request). This method is not part of OCP 2.0 but is necessary for TL2 to work.

Notes: `SgetMCmd()` returns the OCP `MCmd` field and resets the MCmd signal; that is, `SgetMCmd()` is not persistent. This behavior, although different from the OCP MCmd signal in the RTL level, is alright because the data class calls are not used for synchronization

`MputMCmd()` should only be used in conjunction with `MputMRequest()` channel call because `MputMWriteRequest()` or `MputReadRequest()` calls overwrite the MCmd signal.

4.3.1.1. Timestamp Methods

These methods can be used by a master to indicate duration of a request packet. The `MputEndTime()` method simulates the time it takes the master to output the current transaction. In other words, the end time is the earliest time that the request packet is completely through the interface if the slave does not perform any throttling. The time stamps do not delay channel events. They are meant for additional information so that the master can calculate the release time instant. The timestamp methods are as follows:

```
void MputEndTime(sc_time tt)
{
      ReqEndTime = tt;
}
```

```
sc_time SgetEndTime()
{
      return ReqEndTime;
}
```

4.3.2. OCP Data Request Group Signals

At the TL2 level, Request and Data Request phases are merged, hence most of the data request signals are redundant and do no need to be accessed. However, users could use the two following methods to model the special `MDataInfo` signal that may be different from the `MReqInfo` in some implementations.

```
void         MputMDataInfo(unsigned int a)
unsigned int SgetMDataInfo()
```

4.3.3. OCP Response Group Signals

The response group signals can be written by the slave (Sput*) right before the `Sput*Request*()` call. The data protection toggle switches the state at each response. The master may read the signals (Mget*) right after the `MgetRequest*()` returns true.

The methods for the OCP response group signals are as follows:

```
void         SputSDataInfo (unsigned int a)
unsigned int MgetSDataInfo ()

void SputSResp (OCPSRespType a)
OCPSRespType MgetSResp ()

void SputSRespInfo (unsigned int a)
unsigned int MgetSRespInfo ()

void SputSRespLast(bool a)
bool MgetSRespLast()

void         SputSThreadID (unsigned int a)
unsigned int MgetSThreadID ()


void SputSData(Td* d, unsigned int w = 1,
               bool last_of_a_burst = true)
```

Parameters: `d` is the pointer to data array.
`w` is the data array length (response chunk length).
`last_of_a_burst` is the last datum of a burst transfer is sent with this transaction.

```
Td* MgetSData(int& w, bool& last_of_a_burst)
```

Parameters: `w` is the data array length (response chunk length)
`last_of_a_burst` is the last cell of a burst transfer sent with this transaction

Return: Pointer to data array

```
Td* MgetSData(unsigned int& w)
```

Parameters: `w` is the data array length (response chunk length)

Return: Pointer to data array

`void SputSRespChunkLen(unsigned int w)`

Purpose: Sets the response chunk length. `SputSRespChunkLen()` can be used instead of the `w` parameter of the `SputSData()` function (for example, to send a multiple-chunk WRITE non-post response). This method is not part of OCP 2.0 but is necessary for TL2 to work.

`unsigned int SgetSRespChunkLen()`

Purpose: Returns the response chunk length. `SgetSRespChunkLen()` can be used instead of the `w` parameter of the `MgetSData()` function (for example, to get a multiple-chunk WRITE non-post request). This method is not part of OCP 2.0 but is necessary for TL2 to work.

`void SputSRespChunkLast(bool w)`

Purpose: Useful for informing a master that this chunk is the last of a complete response burst. `SputSRespChunkLast()` can be used instead of the `last_of_a_burst` parameter of the `SputSData()` function (for example,. to send a multiple-chunk WRITE non-post response). This method is not part of OCP 2.0 but is necessary for TL2 to work.

`unsigned int MgetSRespChunkLast()`

Purpose: Determines if this chunk is the last of a complete response burst. `MgetSRespChunkLast()` can be used instead of the `last_of_a_burst` parameter of `MgetSData()` function (for example, to get a multiple-chunk WRITE non-post response). This method is not part of OCP 2.0 but is necessary for TL2 to work.

4.3.3.1. Timestamp Methods

These methods can be used by the slave to indicate duration of response packet. The `SputEndTime()` method simulates the time it takes the slave to output the current transaction. In other words, the end time is the earliest time the response packet is completely through the interface, if the master does not do any throttling. The time stamps do not delay channel events. They are meant for additional information so that the master can calculate the release time instant. The timestamp methods are as follows:

```
void SputEndTime(sc_time tt)
{
      ResEndTime = tt;
}

sc_time MgetEndTime()
{
      return ResEndTime;
}
```

4.3.4. Example: Sending and Receiving (Burst) Write Transactions

The following pseudo-code segments are an example of how a 16-OCP-word write burst transaction (made of only one chunk in this case) can be sent from (received by) the master (slave) of an OCP channel using the public methods of the `OCP_TL2_DataCl` class. Assuming this is a posted write transaction, thus, no response is exchanged.

```
// ----------------------------
// on the master sending side
// ----------------------------
MputMConnID(0);
MputMThreadID(0);
MputMAddr(0x30);
MputMCmd(OCP_MCMD_WR);  //Not mandatory since MputWriteRequestBlocking()
                        //is used

// assuming wr_data_ptr is pointed to the 16-word write data
unsigned int chunk_length = 16;
bool last_chunk_of_a_burst=true; // Burst is made of only one chunk
MputMData(wr_data_ptr, chunk_length, last_chunk_of_a_burst);

// the wr_data_ptr and its contest can only be changed after the transaction
// is committed
…..
MputWriteRequestBlocking();

// ----------------------------
// on the slave receiving side
// ----------------------------
if (SgetRequestBlocking(1)) {
      int mconnid = SgetMConnID();
      int mthreadid = SgetMThreadID();
      Ta address = SgetMAddr();
```

```
      OCPMCmdType mcmd = SgetMCmd();
      Td* wr_data_ptr = SgetMData(chunk_length,last_chunk_of_a_burst);

 // after done with the data pointer, need to commit this write transaction
      …
}
```

Calling the `MputMConnID()`, `MputMThreadID()`, and `MputMAddr()` methods sets up the OCP MConnID, MThreadID, and MAddr signals, respectively. The address passed into the `MputMAddr()` method should be on an OCP-word boundary. The `MputMCmd()` call indicates that an OCP write transaction is going to be delivered. The `MputMData()` method is used to send a 16-OCP-word chunk of write data of the write burst transaction onto the OCP channel; plus, it is the last chunk of the burst transfer. (This is indicated by setting the last actual argument to "true.")

On the slave side, the *SgetMConnID()*, *SgetMThreadID()*, *SgetMAddr()*, and `SgetMCmd()` methods are called to retrieve values of the OCP MConnID, MThreadID, MAddr, and MCmd signals, respectively. The `SgetMData()` method is used to receive the write data chunk pointer, plus, the data word length, and to tell whether or not this is the last chunk of the current write burst transaction.

Note that for the `MputMData()` method and the `SgetMData()` method, only pointer passing is allowed in this version. Therefore, the data pointer and its content should not be changed until the transaction is committed (that is, released). This does not cause any problems because the channel uses data toggling for protection. Copying transactions execute considerably slower and require dynamic memory allocation in the channel.

4.3.5. Example: Sending and Receiving (Burst) Read Responses

The following pseudo-code segments show as an example how a single-OCP-word read transaction can be exchanged between the master and slave of an OCP connection. It is also assumed that this OCP connection is configured without the MConnID and MThreadID signals.

```
// ----------------------------
// on the master sending side
// ----------------------------
// sending a read request
MputMAddr(0x30);
MputMCmd(OCP_MCMD_RD);
MputMReqChunkLen(1); // chunk length
MputMReqChunkLast(true); Burst is made of only one chunk
MputReadRequest();
…..

// receiving a read response and data
if (MgetResponseBlocking(1)) {
      OCPSRespType sresp = MgetSResp();
      if (sresp == OCP_SRESP_DVA) {
            Td* rd_data_ptr = MgetSData(chunk_length,last_chunk_of_a_burst);
      }
```

```
}

// ---------------------------
// on the slave receiving side
// ---------------------------
if (SgetRequestBlocking(1)) {
      Ta address = SgetMAddr();
      OCPMCmdType mcmd = SgetMCmd();
      if (mcmd == OCP_MCMD_RD) {
            int chunk_length = SgetMReqChunkLen();
            bool last_chunk_of_a_burst = SgetMReqChunkLast();

            // returning a read response and data
            SputSResp(OCP_SRESP_DVA);

            // assuming rd_data_ptr is pointed to the single-word data
            SputSData(rd_data_ptr, chunk_length, last_chunk_of_a_burst);
            SputResponseBlocking()l
      }
}
```

This time the `MputMCmd()` and `MputBurstLen()` calls together send a single-OCP-word read request to the OCP channel. When the slave receives a request, it checks whether it is a read request first. If a read request is received, the slave retrieves the word length of this read request using the `SgetMGreqChunkLen()` method. In this example, after the read data is obtained, the slave sends back an OCP DVA response and the single-OCP-word read data by calling the `SputSResp()` and `SputSData()` methods, respectively.

On the master side, when a read response arrives, the master uses the `MgetSResp()` and `MgetSData()` methods to retrieve the status of the response and the read data, respectively.

Note that for the `SputSData()` method and the `MgetSData()` method, only pointer passing is allowed for now.

5. GENERIC CHANNEL EXAMPLES

This section presents examples for each transaction layer. The implementations of the examples can be found in the directories `examples/generic_ocp_tl1` and `examples/generic_ocp_tl2`. Except for the `OCP_TL2_Bus`, the examples focus on the usage of the communication methods of the channel, not on functionality inside the master/slave modules. The `OC_TL2_Bus` is an example implementation of a non-cycle-true bus.

The code of the example descriptions is not duplicated. Explanations are focused on the different concepts. The code can be accessed in the directories mentioned above. The code for the generic TL channel can be found in directory `include`. The file names in that directory start with "tl_", indicating that this is generic code common to all layers. The user-written and protocol-specific files start with "ocp_tlx_", where "x" is 1, 2, or 3 depending on the transaction layer of the example. The top level C++ files are named top_x.cpp. The C++ files for the example master and slaves have a description at the beginning, which is worth reading.

5.1. Generic Channel TL1 Examples

TL1 examples are characterized by masters and slaves that have clock ports and use non-blocking methods.

5.1.1. TL1 Example #0

The top level C++ file is called `top_async.cpp`. The example master and slave files are `ocp_tl1_master_async.cpp` and `ocp_tl1_slave_async.cpp`. The example illustrates a simple point-to-point connection involving one master, one channel instance, and one slave. Use the `Make_tl1_sync.gcc` make file to build the example.

> **Note**: This example is for illustration purposes only. There is no guarantee that it will behave as expected. Use at your own risk.

5.1.1.1. Master implementation

The master uses two threads: a request sending thread and a response receiving thread. The request thread is clock driven and issues requests at pre-defined clock cycles. It uses the non-blocking calls `MputReadRequest()` and `MputWriteRequest()`. The master's response thread is sensitive to the master port and is triggered once the slave has issued the response. The response channel is released immediately, and because the thread is event-triggered, this results to a single-cycle response. The response thread can be thought to be level-triggered in RTL terms, sensitive to the SResp signal, and with the assumption that the response group is stable once the response is issued. If you want to sample the response data at the clock edge but handle the release mechanism asynchronously, see example #3 described in section 5.1.4.

5.1.1.2. Slave Implementation

The slave uses two threads: one request receiving thread and one response sending thread. The request thread is sensitive to the default event of the channel. This is an asynchronous response mechanism, allowing for responses in the same cycle as the request was issued. Once the master has sent a request, the slave gets triggered. The slave retrieves the request data and parameter. It stores them in a FIFO queue so that it can receive more than one request before issuing a response. The request thread releases the request channel using the `Srelease(Time)` call. The time between receiving the request and releasing the request channel models the slave's request acknowledge delay. Because the acknowledge delay is modeled through a channel method (as opposed to an explicit `wait()` call in the request thread), the slave request thread can continue execution. For read requests, the request thread computes the response time (based on the time it acknowledged the request) and activates the response thread. The request thread then waits for a new request. The response thread is activated by an event triggered in the request thread. Additionally, the response thread has a state variable indicating whether or not the event was triggered while the thread was busy (for example, sending the response). Once the response thread has been activated, it checks for correct timing and sends the response to the master.

This slave is totally asynchronous. It can be used as a model for implementing asynchronous RAM modules.

5.1.2. TL1 Example #1

The top level C++ file is called `top_async_hs.cpp`. The example master and slave files are `ocp_tl1_master_async_hs.cpp` and `ocp_tl1_slave_async_hs.cpp`. This example is nearly identical with the previous one. The only difference is that here a data handshake channel is used for transferring write data in a different phase from write commands. Both slave has an extra thread for this purpose. The data handshake is similar to request (command) handshake. This example uses asynchronous slave and separate threads, but the data handshake can also be implemented completely synchronously and with a single thread.

Use the `Make_tl1_async_hs.gcc` make file to build the example.

> **Note**: This example is for illustration purposes only. There is no guarantee that it will behave as expected. Use at your own risk.

5.1.3. TL1 Example #2

The top level C++ file is called `top_sync.cpp`. The example master and slave files are `ocp_tl1_master_sync.cpp` and `ocp_tl1_slave_sync.cpp`. The quirk in this example is that slave uses pre-emptive synchronous release. Use the `Make_tl1_sync.gcc` make file to build the example.

5.1.3.1. Master implementation

The master is similar to example #1, but the response thread is clocked, resulting to two cycle transactions, since pre-emptive release is not used.

5.1.3.2. Slave Implementation

The master is similar to example #1, but the release thread is clocked, resulting to two cycle transactions, since pre-emptive release is not used.

5.1.4. TL1 Example #3

The top level C++ file is still called `top_sync2.cpp`. The example master and slave files are `ocp_tl1_master_sync.cpp`, `ocp_tl1_slave_sync2.cpp`. Use the `Make_tl1_sync2.gcc` make file to build the example.

5.1.4.1. Master implementation

The master is the same as in example #2.

5.1.4.2. Slave Implementation

The slave is otherwise similar to example #2, but now pre-emptive release call is used resulting to one-cycle transactions.

5.2. Generic Channel TL2 Examples

TL2 examples are characterized by the following features:

- No clock ports
- Time is estimated
- Mostly blocking methods are used

There are two layer-2 examples: a point-to-point connection example and a 3-masters-1-bus-4-slaves system. Note that the point-to-point connection is produced by the same master and slave modules that are also used in the master-bus-slave system, proving that no bus is needed to connect masters with slaves.

5.2.1. TL2 Example #0

The top level C++ file is called `ocp_tl2_top0.cpp`. The example master and slave files are `ocp_tl2_master.cpp` and `ocp_tl2_slave.cpp`, respectively. The example has a simple point-to-point connection involving one master, one channel instance, and one slave.

Depending on a random number, the master sends either read or write requests for a burst of data to the slave.

5.2.1.1. Master Implementation

The master has the following constructor:

```
OCP_TL2_Master(   sc_module_name name,
                  int  ID,
                  int  Priority,
                  bool Pipelined           = false,
                  bool WriteResponse       = true,
                  int  ReadAcceptCycles    = 0,
                  int  WriteAcceptCycles   = 0,
                  int  ReadResponseCycles  = 0,
                  int  WriteResponseCycles = 0  )
```

- `ID` is a number identifying the master. `ID` must be unique among all masters attached to the same bus.
- `Priority` is a positive number specifying the priority for bus access relative to the other masters. Higher numbers means higher priority. Masters can have the same priority.
- `Pipelined` is a switch to change between non-pipelined and pipelined operation mode of the bus.
- `WriteResponse` is a switch to enable/disable the sending of a response to a master's write request. Note that all modules involved in a system must use the same value.
- `ReadAcceptCycles`, `WriteAcceptCycles`, `ReadResponseCycles`, and `WriteResponseCycles` specify the number of waiting cycles per OCP word by which the master delays the acceptance of a response or the sending of a request, respectively.

The master can be pipelined or non-pipelined, depending on a constructor parameter. In the non-pipelined case, the master uses one thread, which handles request sending and response receiving. In that case, a new request can only be sent if the response of the previous request has been received. In the pipelined case, the master uses two threads: a request sending thread and a response receiving thread. Sending requests and receiving responses are completely independent from each other. The master sends requests at predefined time instances and accepts responses whenever the slave sends one. Because blocking methods are used, no sensitive list is needed. Time delays between sending two consecutive requests are modeled through `wait()` statements. The same holds true for responses. The wait cycles are configurable through constructor parameters.

The example master has an additional thread, called `MasterD`, which shows how to use the direct access methods. To execute this thread, use the constructor of the master accordingly.

The master has the following timing:

- Write request transfer:
  - Accept: wait `NumWords * WriteAcceptCycles` cycles
  - Response: wait `WriteResponseCycles` cycles
- Read request transfer:
  - Accept: wait `ReadAcceptCycles` cycles
  - Response: wait `NumWords * ReadResponseCycles` cycles

Note that in the non-pipelined case the response is started after the request, while in the pipelined case request and response are started in parallel.

5.2.1.2. Slave Implementation

The slave has the following constructor:

```
OCP_TL2_Slave(    sc_module_name name,
                  int  ID,
                  Ta   StartAddress,
                  Ta   EndAddress,
                  bool Pipelined           = false,
                  bool WriteResponse       = true,
                  int  ReadAcceptCycles    = 0,
                  int  WriteAcceptCycles   = 0,
                  int  ReadResponseCycles  = 0,
                  int  WriteResponseCycles = 0  )
```

- `ID` is a number identifying the slave. `ID` must be unique among all slaves attached to the same bus.
- `StartAddress` is the start address of the slave's memory region.
- `EndAddress` is the end address of the slave's memory region. The bus requires 1K-address alignment.
- `Pipelined` is a switch to change between non-pipelined and pipelined operation mode of the bus.
- `WriteResponse` is a switch to enable/disable the sending of a response to a master's write request. Note that all modules involved in a system must use the same value.
- `ReadAcceptCycles`, `WriteAcceptCycles`, `ReadResponseCycles`, and `WriteResponseCycles` specify the number of waiting cycles per OCP word that the slave delays the acceptance of a request or the sending of a response, respectively.

The slave can be pipelined or non-pipelined, depending on a constructor parameter. In the non-pipelined case, the slave uses one thread, which handles request receiving and response sending. In that case, a new request can only be received after the response of the previous request has been sent. In the pipelined case, the slave uses two threads: a request receiving thread and a response sending thread. The slave uses blocking methods in both cases, so no sensitivity list is necessary. In the pipelined case, the request thread accepts a request whenever the master sends one. It then activates the response thread and is ready to receive another request. Because requests and responses are not synchronized, the request parameters are stored in a FIFO. This enables the slave to process several requests before sending a response. If the FIFO is full, the request thread is suspended until a couple of responses have been sent, and the FIFO is ready to store the new request parameters. The response thread checks a state variable, indicating whether or not there are pending responses. If there are none, the response thread waits for an event triggered by the request thread. Otherwise, the response thread keeps sending responses.

The slave has the following timing:

- Write request transfer:
  - Accept: wait `NumWords * WriteAcceptCycles` cycles
  - Response: wait `WriteResponseCycles` cycles
- Read request transfer:
  - Accept: wait `ReadAcceptCycles` cycles
  - Response: wait `NumWords * ReadResponseCycles` cycles

Note that in the non-pipelined case the response is started after the request, while in the pipelined case request and response are started in parallel.

5.2.2. TL2 Example #1

The top level C++ file is called `ocp_tl2_top1.cpp`. The example master and slave files are again `ocp_tl2_master.cpp` and `ocp_tl2_slave.cpp`, respectively, which are described in the previous section. Additionally, there is an example bus (file `ocp_tl2_bus.cpp`). Example #1 models a master-bus-slave system with three master instances and four slave instances. Depending on a random number, the masters send either read or write requests for a burst of data to the slave. The master ID is a constructor parameter, which controls the random number, the address that the masters send the requests to, and the burst length.

5.2.2.1. Bus Implementation

The bus has the following constructor:

```
OCP_TL2_Bus( sc_module_name name,
             int  BusID,
             bool Pipelined       = false,
             bool WriteResponse   = true,
             int  ReadWaitCycles  = 0,
             int  WriteWaitCycles = 0 )
```

- `BusID` is a number identifying the bus. `BusID` must be unique among all busses in a system.
- `Pipelined` is a switch that changes between non-pipelined and pipelined operation mode of the bus.
- `WriteResponse` is a switch to enable/disable the sending of a response to a master's write request. Note that all modules involved in a system must use the same value.
- `ReadWaitCycles` specifies the number of waiting cycles per OCP word by which the bus delays the transport of the read request/response.
- `WriteWaitCycles` specifies the number of waiting cycles per OCP word by which the bus delays the transport of the write request/response.

The bus is a module that acts as master and slave. For the masters attached to the bus, the bus is a slave. For the slaves attached to the bus, the bus is a master. The bus can be pipelined or non-pipelined, depending on a constructor parameter. In the non-pipelined case, the bus uses one thread that handles the receiving of requests from the master, sending requests to the slave, receiving responses from the slave, and sending responses to the master. In that case, a new request from a master can only be processed after the response of the previous request has been sent to the master. In the pipelined case, the bus uses two threads: a request thread which handles receiving requests from the master and sending requests to the slave and a response thread that performs receiving responses from the slave and sending responses to the master. These two threads are not synchronized, so the bus can process several requests before sending a response. The basic functionality of the pipelined bus is as follows:

- Masters send requests to bus.
  - The master-bus request channels are locked.
  - The bus is triggered that there are pending requests.

- Bus collects all pending requests.
  - Bus performs arbitration to select one master.
  - Bus performs address decoding to select the addressed slave.
  - Bus copies the data from the master-bus request channel to the bus-slave request channel.
  - Bus sends request to the addressed slave and locks the bus-slave-request channel.
  - Bus frees the master-bus request channel.
  - Bus waits for an answer coming from a slave.
- The slaves send responses and unlock the corresponding bus-slave request channel.
  - Bus-slave response channel is locked.
  - Bus is triggered that there are pending responses.
- The bus collects all pending responses.
  - Bus selects the first pending response.
  - Bus copies the data from the bus-slave response channel to the master-bus response channel.
  - Bus sends response to the master.
  - Bus frees the bus-slave response channel.
- master processes the response data and frees the master-bus response channel.

The functionality in the non-pipelined case is similar.

The bus uses a two-tier arbitration scheme. The first tier arbitration scheme selects the master with the highest priority. If there is more than one master with the highest priority, the second tier arbitration scheme is performed. That is a fair-among-equals algorithm based on the number of processed requests for each master.

The timing of the bus is as follows:

- Write request transfer: wait `NumWords * WriteWaitCycles` cycles
- Read request transfer: wait 1 `* ReadWaitCycles` cycles
- Write response transfer: wait 1 `* WriteWaitCycles` cycles
- Read response transfer: wait `NumWords * ReadWaitCycles` cycles

6. AUXILIARY CLASSES

6.1. CommCl (tl_comm_cl.h)

This class contains the states and events used by the communication mechanism of the Channel. Access is provided to this class for base generic channel users. Users of OCP specific commands never need to worry about the `CommCl` class because their commands handle all interactions for them.

The states and events in the `CommCl` class must not be changed by masters and slaves, although the generic channel gives full access to this class. The purpose of exporting these states and events is to give masters and slaves read access. Again, these accesses must be read only. For the intended normal use of the Channel, this class should not be changed.

6.2. ParamCl (ocp_tl_param.h)

The `ParamCl` class is a Transaction Level parameter class. This parameter class provides a means for storing parameters like master priorities or slave addresses. When the channel is used for OCP commands, the parameter class also stores all of the OCP parameter settings for the channel. Its basic usage model is to write values to this class in the elaboration phase and read these values from the parameter class at the beginning of the simulation.

> **Note**: These parameter names exactly match those described in the *Open Channel Protocol Specification*. For more detail and information about these parameters, refer to the specification.

6.2.1. Constructor

The `ParamCl()` constructor takes no arguments and is called automatically by the channel when a new channel is created. When a new *`ParamCl`* object is created, all of the OCP parameters are set to their default values as defined by the *Open Core Protocol Specification* document. When the channel's `setConfiguration()` function is called, it uses the passed parameter map to set the values in the *`ParamCl`* object. The constructor is defined as

```
ParamCl()
```

6.2.2. Parameter Member Variables

The master or slave can read the parameters of the channel by issuing the command:

```
ParamCl<TdataCl> *GetParamCl()
```

This returns the *`ParamCl`* object used to hold the channel's parameters. To be compatible with the base generic class, this is a non constant pointer. As a result, the core write operation could use this pointer to change the parameter values of the channel. However, you should avoid this,

especially during the simulation run. The values in the *ParamCl* object should be considered to be read-only by the core.

The current parameters include the following.

`string name`

Purpose: The name of the OCP channel. Set, but not used, by the OCP specific TL1 channel. May be used by the cores to identify the channel they are attached to.

Default: "unnamed_ocp20_channel"

`int MasterID`

Purpose: A non-negative integer, which indicates the identification number of the master core that is connected to an OCP channel. In generic TL2 example #1 (see section 5.1.2), which models a multi-master-single-bus-and-multi-slave system, the `MasterID` parameter is set by a master module and is used by the bus module to identify its master-core interfaces.

Default: -1 (which is illegal)

`int Priority`

Purpose: A non-negative integer (higher value means higher priority), which indicates the bus arbitration priority for the master core that is connected to an OCP channel. In generic TL2 example #1 (see section 5.2.2), which models a multi-master-single-bus- and-multi-slave system, the `Priority` parameter is set by a master module and is used by the slave bus module during arbitration.

Default: -1 (which is illegal)

`int SlaveID`

Purpose: A non-negative integer, which indicates the identification number of the slave core that is connected to an OCP channel. In generic TL2 example #1 (see section 5.2.2), which models a multi-master-single-bus-and-multi-slave system, the `SlaveID` parameter is set by a slave module and is used by the master bus module to identify its slave-core interfaces.

Default: -1 (which is illegal)

`Ta StartAddress`

Purpose: Indicates the beginning of the address space (region) of the slave core that is connected to an OCP channel. In the generic TL2 example #1 (see section 5.2.2, which models a multi-master-single-bus-and-multi-slave system, the `StartAddress` and `EndAddress` parameters are set by a slave module, and they are used by the master bus module for address decoding; that is, to dispatch requests to their proper slave-core interfaces (based on each request's MAddr value). No address regions of slave-core interfaces on the bus module can overlap with each other.

Default: 0

`Ta EndAddress`

Purpose: Indicate the end of the address space (region) of the slave core that is connected to an OCP channel. In the generic TL2 example #1 (see section 5.2.2, which models a multi-master-single-bus-and-multi-slave system, the `StartAddress` and `EndAddress` parameters are set by a slave module, and they are used by the master bus module for address decoding; that is, to dispatch requests to their proper slave-core interfaces (based on each request's MAddr value). No address regions of slave-core interfaces on the bus module can overlap with each other.

Default: 0

`float ocp20version`

Purpose: Specifies the version of OCP.

Default: 2.0

`bool broadcast_enable`

Purpose: Enables the broadcast command when set to true.

Default: false

`bool burst_aligned`

Purpose: Forces burst to be aligned by a power of two when set to true.

Default: false

`bool burstseq_dflt1_enable`

Purpose: Enables DFLT1 burst mode.

Default: false

`bool burstseq_dflt2_enable`

Purpose: Enables DFLT2 burst mode.

Default: false

`bool burstseq_incr_enable`

Purpose: Allows incrementing bursts.

Default: true

`bool burstseq_strm_enable`

Purpose: Allows streaming bursts.

Default: false

`bool burstseq_unkn_enable`

Purpose: Enables UNKN burst mode.

Default: false

`bool burstseq_wrap_enable`

Purpose: Enables WRAP burst mode.

Default: false

`bool burstseq_xor_enable`

Purpose: Enables XOR burst.

Default: false

`string endian`

Purpose: Specifies the endianess of the channel. The values for this parameter are: `"little"`, `"big"`, `"both"`, and `"neutral"`.

Default: `"little"`

`bool force_aligned`

Purpose: Forces the byte-enable patterns to be powers of two.

Default: false

`bool mthreadbusy_exact`

Purpose: Specifies that the slave must use the MTheadbusy signal to send responses, and the master must accept immediately on non-busy threads.

Default: false

`bool rdlwrc_enable`

Purpose: Enables both the `ReadLinked` command and the `WriteConditional` command on the channel.

Default: false

`bool read_enable`

Purpose: Enables support of the Read command.

Default: true

`bool readex_enable`

Purpose: Enables support of the `ReadEx` command.

Default: false

`bool sdatathreadbusy_exact`

Purpose: Specifies that the master must use SDataThreadBusy signal to send new data, and the slave must accept new data immediately on non-busy threads.

Default: false

`bool sthreadbusy_exact`

Purpose: Specifies the master must use SThreadBusy signal to send a new request, and the slave must accept new request immediately on non-busy threads.

Default: false

`bool write_enable`

Purpose: Enables support of the `Write` command.

Default: true

`bool writenonpost_enable`

Purpose: Enables support of the `WriteNonPost` command.

Default: false

`bool datahandshake`

Purpose: Indicates whether there is a separate channel for request data when set to true.

Default: false

`bool reqdata_together`

Purpose: Specifies whether the master always puts a request and data in the same cycle, and the slave always accepts them together in the same cycle.

Default: false

`bool writeresp_enable`

Purpose: Indicates whether responses are sent for write commands

Default: false

`bool addr`

Purpose: Indicates whether MAddr (Request Address) is part of the OCP.

Default: true

`int addr_wdth`

Purpose: The user must set the address width if the addr parameter is set to true.

Default: None.

`bool addrspace`

Purpose: Indicates whether the MAddrSpace signal is part of the OCP.

Default: None

`int addrspace_wdth`

Purpose: Indicates the width of the address space.

Default: None.

`bool atomiclength`

Purpose: Specifies whether there are a minimum number of transfers to hold together during a burst.

Default: false

`int atomiclength_wdth`

Purpose: Specifies the minimum number of transfers to be held together during a burst when the `atomiclength` parameter is set to true.

Default: None.

`bool burstlength`

Purpose: Specifies whether there is a set number of transfers in a burst

Default: false

`int burstlength_wdth`

Purpose: Specifies the number of transfers in a burst.

Default: None.

`bool burstprecise`

Purpose: Specifies whether the length of a burst is known at the start of the burst.

Default: false

`bool burstseq`

Purpose: Specifies whether there is a sequence of addresses in a burst

Default: false

`bool burstsinglereq`

Purpose: Specifies whether a single request is allowed to generate multiple data transfers in a burst.

Default: false

`bool byteen`

Purpose: Specifies whether MByteEn is part of the OCP.

Default: false

`bool cmdaccept`

Purpose: Specifies whether the SCmdAccept is part of the channel. If false, all requests are automatically accepted.

Default: true

`bool connid`

Purpose: Specifies whether the `MConnID` connection identifier is part of the Request group.

Default: 0

`int connid_wdth`

Purpose: Specifies the width of `MConnID`.

Default: None

`bool dataaccept`

Purpose: Specifies whether the SDataAccept is part of the channel. If false, all data requests are automatically accepted.

Default: true

`bool datalast`

Purpose: Specifies whether the `MDataLast` burst signal is part of the OCP.

Default: false

`int data_wdth`

Purpose: Specifies the width of MData.

Default: None

`bool mdata`

Purpose: Specifies whether MData is part of the OCP.

Default: true

`bool mdatabyteen`

Purpose: Specifies whether the MDataByteEn signal is in the OCP

Default: false

`bool mdatainfo`

Purpose: Specifies whether the MDataInfo signal is in the OCP

Default: false

`int mdatainfo_wdth`

Purpose: Specifies the width of the MDataInfo signal when the `mdatainfo` parameter is true.

Default: None

`int mdatainfobyte_wdth`

Purpose: Specifies the number of bits of `MDataInfo` that are associated with each data byte of MData.

Default: 1

`bool sdatathreadbusy`

Purpose: Specifies whether SDataThreadBusy is part of the OCP channel.

Default: false

`bool mthreadbusy`

Purpose: Specifies whether the MThreadBusy signal is part of the OCP channel.

Default: false

`bool reqinfo`

Purpose: Specifies whether the MReqInfo signal is part of the OCP channel.

Default: false

`int reqinfo_wdth`

Purpose: Specifies the width of MReqInfo signal.

Default: None.

`bool reqlast`

Purpose: Specifies whether the MReqLast burst signal part of the OCP channel.

Default: false

`bool resp`

Purpose: Specifies whether the SResp signal part of the OCP channel.

Default: true

`bool respaccept`

Purpose: Specifies whether the MRespAccept is part of the channel. If false, all responses are automatically accepted.

Default: false

`bool respinfo`

Purpose: Specifies whether the SRespInfo signal is part of the OCP channel.

Default: false

`int respinfo_wdth`

Purpose: Specifies the width of the SRespInfo signal.

Default: None

`bool resplast`

Purpose: Specifies whether the SRespLast burst signal is part of the OCP channel.

Default: false

`bool sdata`

Purpose: Specifies whether the SData signal part of the OCP channel.

Default: false

`bool sdatainfo`

Purpose: Specifies whether the SDataInfo signal is supported.

Default: false

`int sdatainfo_wdth`

Purpose: Specifies the width of SDataInfo signal.

Default: None. The OCP specification states that the user must set this parameter; however, if the user does not specify a value, the channel will set it to 1.

`int sdatainfobyte_wdth`

Purpose: Specifies the number of bits in the SDataInfo signal devoted to each byte of `SData`.

Default: None. The OCP specification requires that the user set this parameter. Thus, there is no standard default value; however, if the user fails to provide a value for this parameter, the channel will set it to 1.

`bool sthreadbusy`

Purpose: Specifies whether the SThreadBusy signal is supported

Default: false

`int threads`

Purpose: Specifies the number of threads allowed.

Default: 1

`bool control`

Purpose: Specifies whether the sideband Control signal is supported.

Default: false

`bool controlbusy`

Purpose: Specifies whether the sideband ControlBusy signal supported.

Default: false

`int control_wdth`

Purpose: Specifies the width of the ControlBusy signal.

Default: None. The OCP specification requires that the user set this parameter. Thus, there is no standard default value; however, if the user fails to provide a value for this parameter, the channel will set it to 1.

`bool controlwr`

Purpose: Specifies whether the sideband ControlWr signal is supported

Default: false

`bool interrupt`

Purpose: Specifies whether the sideband SInterrupt signal supported.

Default: false

`bool merror`

Purpose: Specifies whether the sideband MError signal is supported.

Default: false

`bool mflag`

Purpose: Specifies whether the sideband MFlag signal is supported.

Default: false

`int mflag_wdth`

Purpose: Specifies the width of sideband MFlag signal.

Default: None. The OCP specification requires that the user set this parameter. Thus, there is no standard default value; however, if the user fails to provide a value for this parameter, the channel will set it to 1.

`bool mreset`

Purpose: Specifies whether the sideband MReset signal supported.

Default: None. The OCP specification requires that the user set this parameter. Thus, there is no standard default value; however, if the user fails to provide a value for this parameter, the channel will set it to false.

`bool serror`

Purpose: Specifies whether the sideband SError signal is supported.

Default: false

`bool sflag`

Purpose: Specifies whether the sideband SFlag signal is supported.

Default: false

`int sflag_wdth`

Purpose: Specifies the width of the sideband SFlag signal.

Default: None. The OCP specification requires that the user set this parameter. Thus, there is no standard default value; however, if the user fails to provide a value for this parameter, the channel will set it to 1.

`bool sreset`

Purpose: Specifies whether the SReset signal is part of the OCP channel.

Default: None. The OCP specification does not specify a default value. However, if the user does not specify a value, the channel sets it to false.

`bool status`

Purpose: Specifies whether the sideband Status signal is supported.

Default: false

`bool statusbusy`

Purpose: Specifies whether the sideband StatusBusy signal is supported

Default: false

`bool statusrd`

Purpose: Specifies whether the sideband StatusRd signal supported.

Default: false

`int status_wdth`

Purpose: Width of the `Status` signal.

Default: None. The OCP specification requires that the user set this parameter. Thus, there is no standard default value; however, if the user fails to provide a value for this parameter, the channel will set it to 1.