

## **White Paper for**

# **SystemC™ based SoC Communication Modeling for the OCP™ Protocol**

**V1.0 - October 14, 2002**

**Anssi Haverinen, Nokia**

**Maxime Leclercq, Texas Instruments**

**Norman Weyrich, Synopsys**

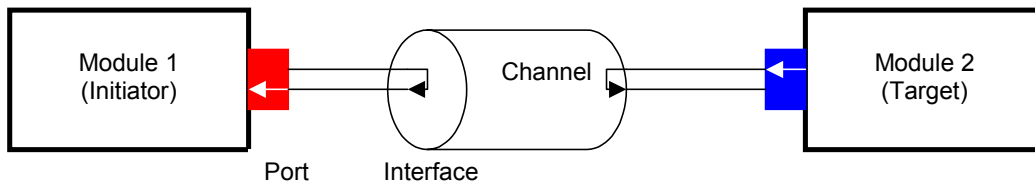
**Drew Wingard, Sonics**

## 1. INTRODUCTION

This paper outlines a conceptual framework for modeling communication in complex systems in various layers of abstraction. It aims to provide a methodology and re-usable concepts for SoC hardware-software co-development. It promotes 'system-level' modeling and design as an essential component of SoC design flow. The principles presented for abstracting data and control flows are generic, and can be used to express both *function* and *performance* of the various hardware and software components implementing a given system. These principles can be implemented with various event-driven modeling languages. To present a practical application for these concepts, we will use Open Core Protocol for hardware communication framework, and SystemC for modeling language. Open Core Protocol provides a rich, configurable protocol set, which is completely bus-independent and can express a large variety of hardware communication behaviors. SystemC provides an easily extendable language platform, which can model all the abstraction layers proposed.

The first section gives a brief overview to the motivation for our work, and an overlook to the basic concepts. In particular it defines the different communication abstraction layers. The second section gives detailed requirements for each communication abstraction layer. The third section presents an API for a generic point-to-point communication channel (Figure 1. ). This channel implements the communication between two modules, where a module is an Initiator or Target or both. The channel is generic in the following aspects:

- No assumptions on the communication protocol between the two modules are being made. The channel just implements the communication; the protocol must be implemented in the modules.
- The same channel can be used at different abstraction layers.



**Figure 1. Initiator and Target connected through a communication channel**

Note that the API described in this section is under development and may change in later releases. However it is included to help outline our ideas for better understanding of the scope of this proposal.

### 1.1. Motivation

The reason for pursuing standards for SoC communication modeling is that the plethora of more or less proprietary methodologies used in system-level modeling and executable specifications does not foster re-use of abstract models, or even development of novel system-level EDA tools. Various useful methodologies and tools have sprung up for tasks such as interface synthesis, real-time performance estimation, low-level driver generation, executable specification, and hardware-software co-simulation, but they share little in modeling methodology. Also, moving from one abstraction level to another is often cumbersome, due to the lack of overall communication modeling strategy. Verification and performance analysis often become bottlenecks with complex embedded systems. The methodology and modeling concepts described here are an attempt to make the link between the up-front architecture exploration and the implementation. System level design with a scalable range of abstract models is opening new possibilities for improving design cycle and enhance the overall quality of the designs.

The largest hurdle in SoC integration has been a lack of hardware interconnect standard. It is not practically feasible for the system architect to support and exercise all possible bus and on-chip network architectures when defining and benchmarking a new system. The OCP interface standard lowers this hurdle, since it isolates interconnect from interface, making possible the use of only one set of transactions that are independent of the selected interconnect architecture. It provides a universal view from a socket to another, and modeling the communication pipe between them by a transaction channel enables the architect to quickly exercise and validate new design partitioning.

Another hurdle, almost as severe, is a mental one. So-called system-level modeling languages and environments often concern themselves too much with RTL implementation, neglecting full support for above-RTL analysis. This strategy backfires as RTL methodology is well established, and can be improved only incrementally. The biggest gains in SoC design can be achieved in specification phase, by giving designers tools for quick trade-off analyses, and functional experiments. Although such tools do exist, few offer reusability of high-level models and test cases with implementation models disconnecting specification from implementation and verification. In our experience, SoC design organizations appreciate most single source, and verification reuse. What they abhor is re-writing the same behavior for specification, implementation and verification. We define single source as follows: A certain behavior is defined only once, but more details can be added or behavior can be refined during the design process. More useful, and maybe much more productive method is to allow mixing abstraction layers in simulation instead of forcing re-implementation of code. An example of the latter is typical DSP ASIC design method, where a high-level model and its test environment are built in a signal processing simulation tool, and implementation and its test bench are re-coded in VHDL or Verilog environment. In the "single-source" method as loosely defined here, the high-level model can still have C-core, but its interfaces are expressed with our high-level transactions. The RTL model uses OCP interfaces. This makes both the test benches and the core models interchangeable and interoperable. Thus, the single-source does not mean that the RTL code must be automatically generated from the golden model C-code, but all behavior (including tests) can be defined with the methodology, which suits the problem best and the various models can be mixed and matched, and system-level tests need not be redesigned for RTL level.

The following flow diagram depicts a top-down approach for system level design. At the early stage of the specification, the architect using SystemC and appropriate communication layers can refine his system while keeping a close link with the implementation. The interoperability given by OCP and its model with different layer of abstraction allow simple hand-off between the architects and the designers.

Our target is to present layers of communication abstraction that can be used for all modeling purposes. The abstractions are selected so that they naturally support the roles of various organizational units doing different aspect of SoC and embedded software design.

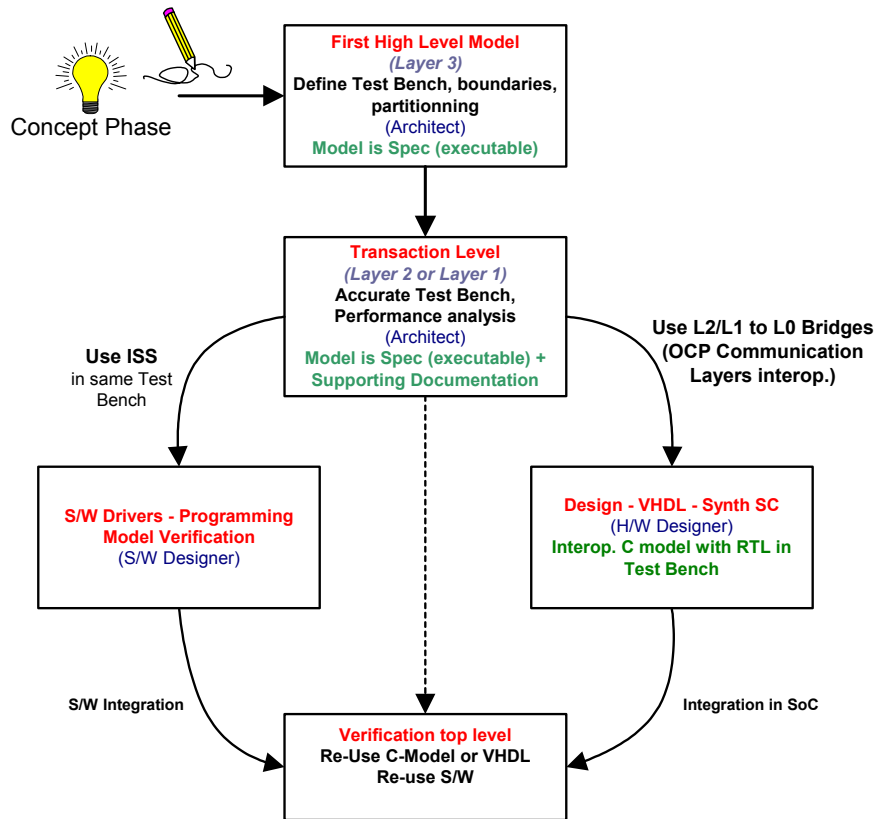
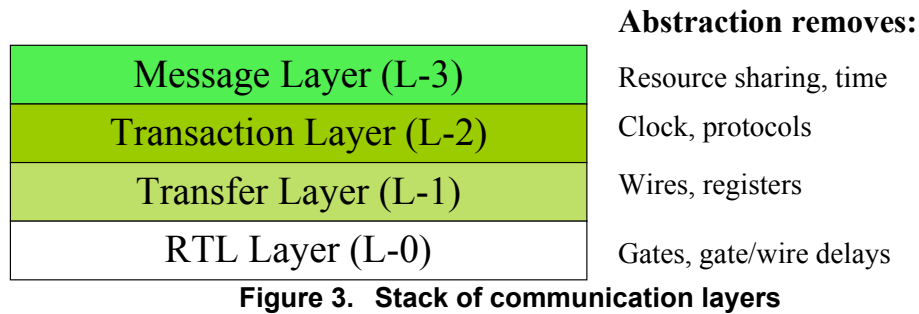


Figure 2. Top-Down design approach using SystemC Communication Layer

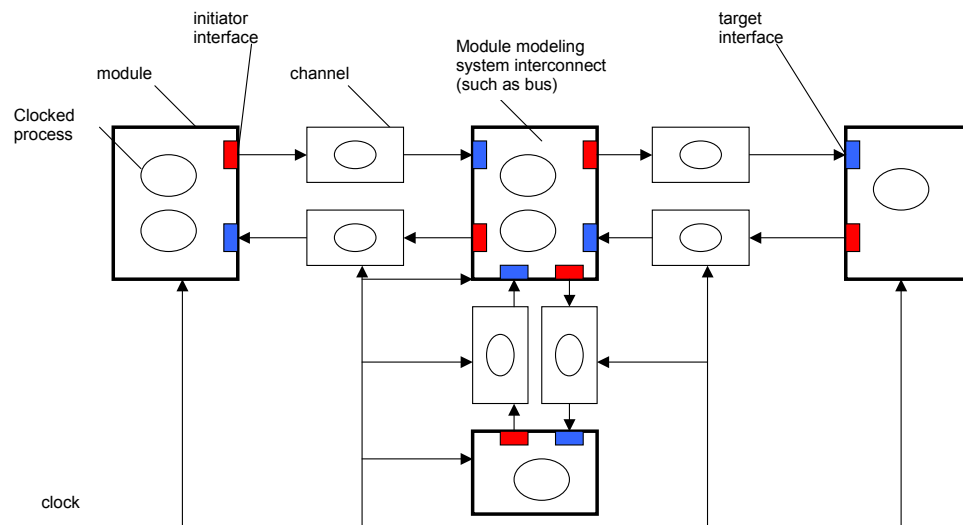
## 1.2. Introduction to Communication Abstraction Layers

Each communication layer consists of three agents: *Initiator*, *Channel*, and *Target*. The *Initiator* is connected to the *Target* with the *Channel*, through an *Interface*. The *Interface* presents the *Target* and the *Initiator* with the services the *Channel* offers (a view), thus the channel implementation can be modified without the *Target* and the *Initiator* knowing as long as the minimum services required by the *Initiator* and *Target* are provided. For example, the system consisting of number of *Initiators* and *Targets* can use either point-to-point connections, or multipoint-to-multipoint connections for communication medium (or interconnect network) without functional changes from the peers' perspective. In both cases the communication would be handled by (several instances of) the same point-to-point communication channel. A point-to-point connection between two modules can be realized directly with a point-to-point communication channel, while a multipoint-to-multipoint connection requires an additional module (namely a so-called multipoint-to-multipoint interconnection module), which is both *Target* (to collect all requests) and *Initiator* (to distribute the requests to the destination). In a typical system analysis the usage of point-to-point or multipoint-to-multipoint connections would be a trade-off of performance versus complexity or size. The communication layers support true interface-based design methodology. The interoperability of *Initiators* and *Targets* from different communication abstraction layers can be implemented with adapter components such as a layer-wrapper or multi-layer capable channels.



The Figure 3. presents the stack of communication abstraction layers, with the main features that each layer abstracts away. We describe the usage model and main features of the different communication abstraction layers in the following.

### 1.2.1. Layer-1; Transfer Layer



**Figure 4. A system with L1 communication channels**

Layer-1 systems are characterized by cycle-true behavior. That means they behave cycle-true the same as the corresponding RTL system. Hence Layer-1 systems must follow a specific communication protocol. For Master-Bus-Slave systems, that could be e.g. the AMBA protocol. However in order to be more general (and independent from a bus standard) we select the Open Core Protocol (OCP) in our example implementation. Normally Layer-1 systems are clocked. A transaction between Initiator and Target involves the transport of one data item (with necessary control information).

The Layer-1 is not meant for writing synthesizable implementation code; although it is perfectly valid for that use, in case of a straightforward interconnect. Instead, the Layer-1 channels provide a fully cycle and protocol accurate connectivity for stub modules and fast-simulating test benches. Layer-1 modules and channels can connect to RTL modules (Layer-0) seamlessly with proper RTL adaptors, and can connect and interact with Layer-2 channels as well. Also, Layer-1 semantics suit implementing ISS model interfaces to cycle accurate world, and providing cycle-accurate adaptors (BFM) for software simulation models to hardware, without using a processor simulator.

One can argue that most Layer-1 functionality can be achieved in RTL level, and this is absolutely true. However, the benefits of the Layer-1 over RTL are:

- Simpler netlist; only single wire for the whole communication interface. The netlist does not need to be changed with parameter (communication protocol or functionality) changes.
- Faster simulation; the communication channel implementation does not need signals, but operates with events, variables and functions, which are faster to simulate.
- Simpler interface code between the core and the communication channel; the core uses only the channel function calls it needs and does not need to bother knowing all the interface signals.

The usage model and main features of Layer-1 communication systems are summarized in the following subsections.

### Use

- Bridging Layer-3 or Layer-2 components to cycle accurate world (cycle accurate wrapper)
- Modeling cycle-accurate interfaces for abstract simulation models of IP blocks such as embedded processors
- Cycle-accurate test bench modeling
- Cycle-accurate performance simulation
- Apples-to-apples performance comparison with RTL

### Main Features

- Clock-accurate protocols mapped to the chosen H/W interfaces and bus structure
- Interface pins are hidden
- Byte-accurate data
- Transactions have internal structure (protocols, data, clock)
- Transactions map directly to bus cycles
- Parametizable to model different bus protocols and signal interfaces

#### 1.2.2. Layer-2; Transaction Layer

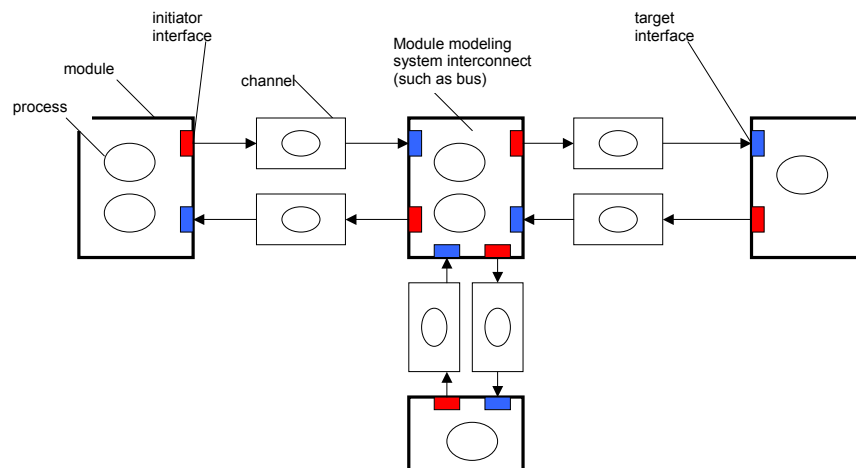


Figure 5. A system with L2 communication channels

Layer-2 systems are timed, but not cycle-accurate. The system executes event-driven. A single transaction between Initiator and Target involves the transfer of several datus (i.e. a burst, or a partial burst of data). The elapsed time to execute the transaction is estimed inside the Initiator and Target. Normally Layer-2 systems are independent of bus protocols, since bus protocols can only be implemented with cycle-true systems. However certain protocol properties like pipelined access and split transactions can be modeled. Specific bus protocols may also be taken into account when timing estimations are being made. The usage model and main features of Layer-2 communication systems are summarized in the following subsections.

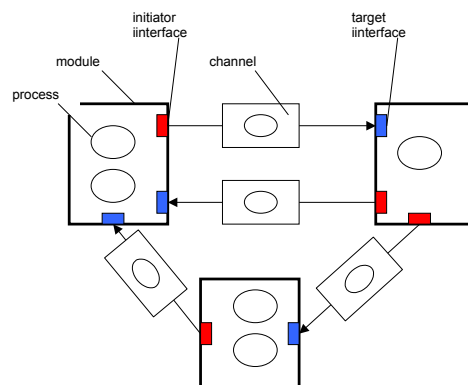
### Use

- Hardware architectural performance and detailed behavior analysis
- HW/SW partitioning and co-development
- Cycle performance estimation
- Interfacing low-level device drivers with hardware simulation models
- Integration of OS simulators, HW emulators
- Test benches for cycle-accurate, and transaction models
- Source of golden test patterns or a system scenario for the implementation

### Main Features

- Essence of layer 2 is mapping ideal architecture into resource-constrained world
- Memory/Register map accurate
- Allows multi-threaded communication
- Bit-width and transfer-size constrained data types to allow mapping to bus bursts, or fragments of bursts
- Event driven simulation with time estimation
- Capable of cycle-performance estimation based on transfer sizes and latency constraints. Performance estimation can be report based, ie. the event driven channel counts and reports how many cycles a transaction would take based on bus width and protocol, or it can insert delays (specified by the Initiator and Target) to simulation time
- Parametizable to fine tune bottlenecks or relax over-constrained specifications
- Split, pipelined with time delays (See Appendix B).
- Parametizable to model different bus protocols and signal interfaces

#### 1.2.3. Layer-3; Messaging Layer



**Figure 6. A system with L3 communication channels**

Layer-3 systems are untimed. The system executes event-driven. A single transaction between Initiator and Target involves the transfer of several datums, which may be of very abstract data type.

The usage model and main features of Layer-3 communication systems are summarized in the following subsections.

### Use

System concept proof, executable specifications, first level of functional partitioning and high-level trade-off:

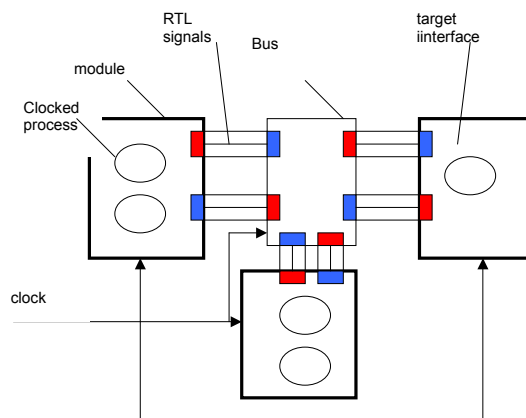
- Functional partitioning to data / control
- Communication definition and first conceptual system test bench
- Integration to higher-level system simulations (SDL, UML)
- Algorithmic performance, behavior, control

### Main Features

- Implementation architecture-abstract
- Event-driven simulation semantics
- Point-to-point Initiator-Target connections
- Primitive support for RTOS-like communication and synchronization methods. Copy and pointer passing data transfer (see Appendix A).
- Abstract data types (e.g. any type expressible with C++ *typedef*)
- Not address mapped but process mapped (static binding of channel end-points to execution processes)
- Blocking or non-blocking message passing interface, no requirement for message queues (queuing is a model feature, not in channel)
- Ordered events, if this is a system requirement. E.g. a slave may ignore events in some systems, if it cannot wait for them, and the master will have to deal with that.
- Non-pipelined, event-order pipelined, non-split, and split interface methods (See Appendix B)

#### 1.2.4. Layer-0; RTL Layer

The RTL layer is not strictly speaking part of our proposal, but added here for the sake of completeness.



**Figure 7. A system with L0 communication channels**

**Use**

SoC HW implementation

**Main Features**

Standard RTL implementation

- Pin/bit accurate
- Register transfer accurate
- Final VHDL/Verilog/Synthesizable SystemC
- Estimated propagation timing can be backannoted

Since plenty of literature exists for RTL, we will leave further study to the reader.

## 2. OPERATIONAL SEMANTIC REQUIREMENTS

This section presents the requirements for control and data semantics for any implementation of the communication layers. We first outline the main requirements and then describe the requirements of each layer in more details in the sub sections.

The main requirements, imposed to more than one layer, are:

1. Layer-1 and 2 use the same address mapping and data types
2. Layer-2 and 3 support copying and pointer-passing data transfers
3. Layer-2 and 3 support blocking and non-blocking flow control methods.

In the following we describe the addressing scheme, the data interface and the flow control interface of the three communication layers

The data interface specifies the type of the data (and methods to access it), transferred between Initiator and Target. The data transfer between Initiator and Target supports passing along pointers, as well as explicitly copying the data to the channel. For passing along pointers, the channel contains a pointer that can be shared by the Initiator and the Target. Both must fetch the pointer before they can access the data. The party, which is the data source, must set the channel pointer to point to its internal data buffer. In case of a write transfer, the buffer is in the Initiator, and in case of a read transfer, the buffer is in the Target. Alternatively the data can be copied to and from the channel. A mixture of both concepts is possible. E.g. for a write transfer, the Initiator sets the data pointer to his data buffer while the Target uses a copy method to copy the Initiators data to the Targets data buffer. The data transfer scheme can be selected by the Initiator and the Target.

The control interface gives synchronization and sequencing mechanisms, which are used with the data interface to implement complete communication methods. The flow of control assumes a threaded computation model without pre-emption, i.e. a task, or a thread must explicitly suspend to let other tasks run. The flow of control may exhibit some pipelining features with separate request and response calls, even though the time abstraction is event ordered, without regard to real-time. The treatment of pipelined or posted requests is upto the Initiator and Target models. If the Target does not support split request-responses, the Initiator may not issue a new request before the current one has been served.

The data interface can only be used in conjunction with the control interface. These interfaces together form the communication API.

### 2.1. Layer-1

#### 2.1.1. Addressing

Since multiple Initiators and Targets may be connected through multipoint-to-multipoint interconnect modules (see Figure 4. ), each transfer must be addressed system address-map accurately. To support the OCP addressing scheme the address information may be divided into fields base address and address space (see OCP chapter in the Appendix C).

#### 2.1.2. Data Interface

Payload data is a scalar number, transferable over a constrained interface in one clock cycle. The other data structure fields depend on the bus protocol used.

The following access mechanisms must be available:

- A way for Initiator to write data for Target
- A way for Target to read data written by Initiator
- A way for Initiator to request data from Target
- A way for Target to write data for Initiator
- A way for Initiator to read data written by Target
- A way for Initiator to write the Address field
- A way for Target to read the Address field

In addition, other protocol-specific fields may be made available, such as burst information, error flags, etc. These are treated the same as data with respect to flow of control.

### **2.1.3. Control Interface**

The Layer-1 is completely cycle-driven (with exception of asynchronous Initiators and Targets). Interface functions may be called only at clock edge, or after a channel event in asynchronous case. The channel is clocked, and it stores state information between clock cycles, and updates the channel state at clock edge. The primitive transfers are fully split, i.e. the request and response transfers are decoupled. It is possible to express non-split transfers with split primitives. The main benefit for using split primitives is that the flow control can be made symmetric; only one communication mechanism is needed to express request and response transfers (although the data fields are different).

The following synchronization mechanisms must be available. Flag-based synchronization is used with clocked interfaces, and event-based with asynchronous interfaces.

- A request notification mechanism that the Initiator has data available to the Target (event and/or flag)
- A response notification mechanism that the Target has data available to the Initiator (event and/or flag)
- An acknowledge notification mechanism that the Target has not processed request (event and/or flag)
- An acknowledge notification mechanism that the Initiator has not processed response (event and/or flag)
- A flag mechanism to indicate to the Target whether the Initiator requests a read or a write transfer.

Protocols with separate data and address handshake may require additional request and acknowledge mechanisms.

## **2.2. Layer-2**

### **2.2.1. Addressing**

Like Layer-1, Layer-2 supports multipoint-to-multipoint interconnect modules. Therefore the same system address mapping as in Layer-1 is used.

### **2.2.2. Data Interface**

The data type on Layer-2 is the same as on Layer-1. As a difference to Layer-1, a complete data burst consisting of N datums (words) can be transferred with one request. To transfer the same amount of data on Layer-1 would require N Layer-1 requests.

The data interface of Layer-2 must contain the following information:

- Number of words to be transmitted from Initiator to Target.
- Number of words to be transmitted from Target to Initiator.
- Start address.
- Initiator data pointer.
- Target data pointer.

Additional members, which are normally protocol specific, can be defined by the user.

### **2.2.3. Control Interface**

Layer-2 is event driven. As a difference to Layer-1, Initiators and Targets must explicitly add timing information, so that the events occur at the right time. The following synchronization mechanisms must be available in addition to the ones listed in section 2.1.3:

- A mechanism to model the time in the Initiator between initiating two consecutive requests as well as between receiving two consecutive responses from the Target.
- A mechanism to model the time in the Target between receiving the Initiators request and responding to that request.

## **2.3. Layer-3**

### **2.3.1. Addressing**

Since the Initiator and the Target have statically defined point-to-point connections (channels), there is no need for addressing. Only data and flow control is passed with the message. Any module can have any number of Initiator and Target ports. A process can act as a master and slave to several channels (with some limitations caused by event sensitivity).

Nevertheless, it is possible to include message data fields, which have specific meaning to the Initiator and the Target. Address can thus be implemented as a special data field for a given simulation model.

### **2.3.2. Data Interface**

Any composite or scalar data type must be supported; interface needs to support only pre-defined data types. This means that different message types require different interface instances. The difference to Layer-2 is that the Layer-3 data type may contain any message, whereas the Layer-2 data type contains physical interface signal fields, which are relevant for transferring a byte burst over a multipoint-to-multipoint interconnection module. The Layer-3 data type can be mapped to burst data transfer of Layer-2. The process is similar to overlaying a data structure to a memory buffer, and will be explained later.

Data is transferred between Initiator and Target through shared memory in a channel, like on Layer-2. The memory sharing can employ pointer-passing or copy semantics, or both. This arrangement allows fast implementation for simulator, and can be used with flow control commands to model behavior of such system-implementation concepts as Message Passing and Remote Procedure Call. Notice that even though the Layer-3 does not actually implement RPC, the effects of RPC in system-level communication can still be modeled with a suitable combination of blocking flow control and data copying. The shared memory is only a simulator implementation concept, with no reference to hardware that is being modeled.

The channel contains a pointer that can be shared by the Initiator and the Target. Both must fetch the pointer before they can access the data. The party, which is the data source, must set the channel pointer to point to its internal data buffer (in case of write message, the buffer is in the Initiator, and in case of read, the buffer is in the Target).

The data interface can only be used in conjunction with the flow control interface. These interfaces together form the Layer-3.

The following mechanism must be available:

- Initiator data pointer.
- Target data pointer.

### **2.3.3. Control Interface**

The following synchronization mechanisms must be available:

- A notification mechanism that the Initiator has data available to the Target (event and/or flag)
- A notification mechanism that the Target has data available to the Initiator (event and/or flag)
- A blocking mechanism such that the Target can block execution of the Initiator and the Initiator can block the execution of the Target.
- A flag mechanism to indicate to the Target whether the Initiator requests a read or a write transfer.
- A flag mechanism to indicate to the Initiator that the Target has not processed the last request. This is necessary for flow control with non-blocking Initiator requests.
- A flag mechanism to indicate to the Target that the Initiator has not processed the last request. This is necessary for flow control with non-blocking Target responses.

### 3. OCP-SYSTEMC API

In this section we provide the API of the communication channel as well as pseudo code examples for the different transfers schemes on each layer. **Note that the API of the communication channel is not yet fixed and may change in later versions.** Some OCP features, like data handshake and thread busy signaling are missing in the current version. Also, some simulation performance enhancements may yet change the API.

The channel implementation class is shared among all abstraction layers, but the interfaces differ slightly for the parameters of the channel methods and visibility of some members.

#### 3.1. Layer-1

The communication channel on Layer-1 is independent from the interconnection protocol, but can be configured to support any specific protocol. For a reference implementation the Open Core Protocol (OCP) was selected. For more information about the OCP standard see the Appendix C.

##### 3.1.1. Data Interface

The data interface methods are implemented in template class **TL1\_Bus\_Signals<DataType>**. They are accessible through interface method **GetDataIF()**. The data interface gives an access to the data and address fields of the interface. Basic fields like data and address have generically named methods, which can be used also with other protocols than OCP. All OCP fields can also be accessed directly with their OCP name. For example to set MData, the Initiator calls **GetDataIF->MData(new\_value)**, and to read MData, the Target calls **value = GetDataIF->MData()**. The data interface methods should be called according to rules set in the control interface. All data interface methods return immediately.

##### Initiator Data Interface

```
void Address(int Address)
```

Purpose: Set target address for transfer (MAddr field of OCP)

```
void MputData(DataType Data)
```

Purpose: Set MData field OCP.

```
DataType MgetData()
```

Purpose: Return SData field of OCP.

##### Target Data Interface:

```
int Address()
```

Purpose: Return MAddr field of OCP.

*void SputData(DataType Data)*

Purpose: Set SData field of OCP.

*DataType SgetData()*

Purpose: Return MData field of OCP.

### 3.1.2. Control Interface

The control interface supports both pipelined and non-pipelined access. The control methods are visible directly at the Initiator and Target interfaces, and they provide access to OCP handshake protocols. The Layer-1 is clocked, and the control interface methods are sequenced with a clock signal.

#### Initiator Control Interface

*TL1\_Bus\_Signals<DataType> \* GetDataIF()*

Purpose: Returns a pointer to channel data interface.

*bool MgetSBusy ()*

Purpose: Target busy semaphore. An initiator may call data interface methods (that modify data) only when this method returns false.

Return: Returns immediately true if Target has not acknowledged the last request, and false if it has.

*MputWriteRequest ()*

Purpose: Issues a write request to Target. Should be called only, when MgetSBusy() returns false.

Return: Return value same as MgetSBusy(). No action when return false.

Events: Causes a write event to asynchronous Target, none to synchronous.

*bool MputReadRequest ()*

Purpose: Issues a read request to Target. Should be called only when MgetSBusy() returns false.

Return: Returns immediately. Return value same as in MgetSBusy(). No action when return false.

Events: Causes a read event to asynchronous Target, none to synchronous.

***bool MgetResponse (bool Release)***

Purpose: Tests if Target has send response. Channel data is valid, and channel data can be read by the Initiator when this call returns true the first time, and the following clock cycle.

Parameters: If Release is true, the Target is acknowledged immediately. Further calls will return false, until the next response is sent by the target.

Events: No events.

***void MRelease ()***

Purpose: Acknowledge target response. Can be used instead of MgetResponse's Release parameter.

Return: Returns immediately. No return value.

Events: No events.

**Target Control Interface*****TL1\_Bus\_Signals<DataType> \* GetDataIF()***

Purpose: Returns pointer to channel data interface.

***Bool SgetRequest(bool Release)***

Purpose: Test if Initiator has sent request. Channel data is valid, and channel data can be read by the Target when this call returns true the first time, and the following clock cycle.

Parameters: If Release is true, the Initiator is acknowledged immediately. Further calls will return false, until the next request is sent by the Initiator.

Events: No events.

***int IsWrite()***

Purpose: Get direction of request.

Return: Returns 1, when pending Initiator request is Write, and 0 when it is true. All other values are reserved for later use.

Events: No events.

**Bool SputResponse()**

Purpose: Issues a response to Target. Can be called only after a request is detected by SGetRequest().

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

**void SRelease()**

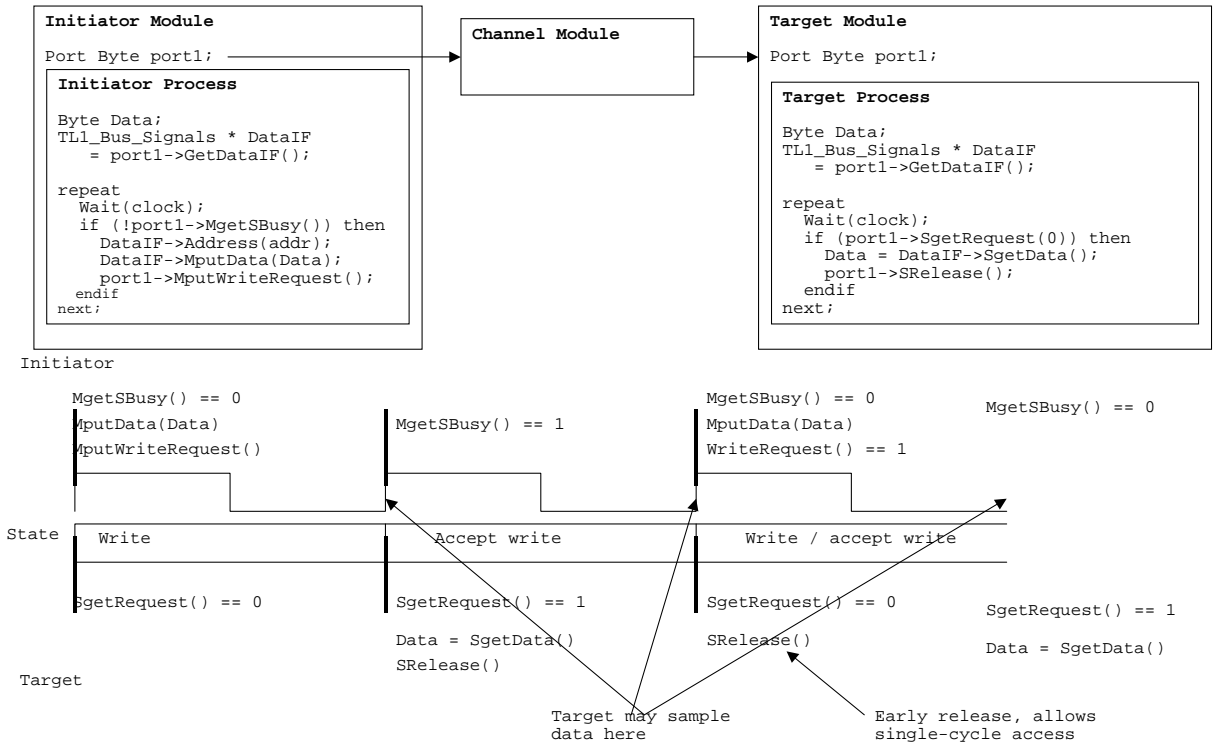
Purpose: Acknowledge initiator request. Can be used instead of SgetRequest's Release parameter

Return: Returns immediately. No return value.

Events: No events.

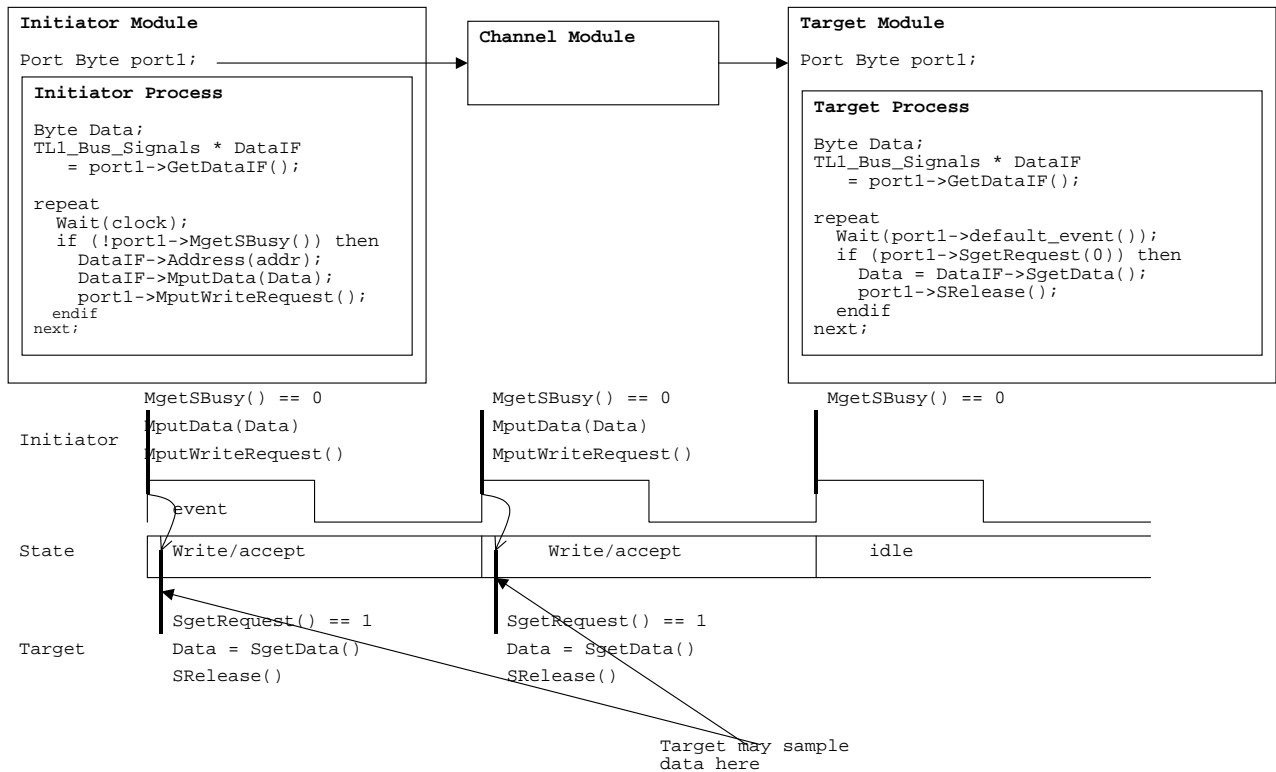
**3.1.3. Pseudo-code Examples**

The following figure shows how the control and data interface are used with a clock to achieve data write transfer over the interface. The mechanism is the same for response sequence. The first write completes in two cycles, since the Target calls SRelease() after it detects a request. The second write completes in one cycle, since the Target calls SRelease() pre-emptively.



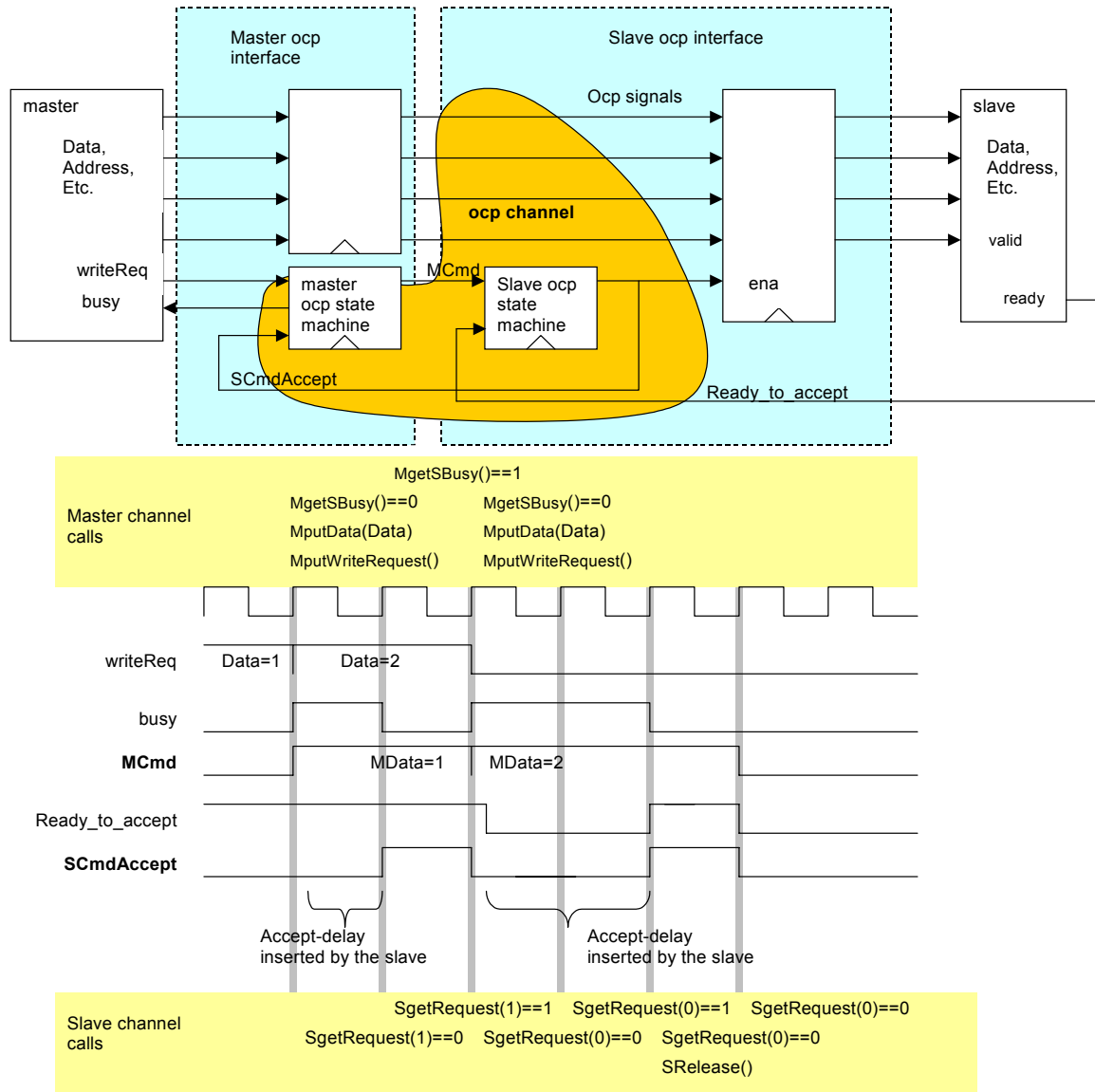
**Figure 8. Control and data sequencing (Synchronous)**

Figure 9. shows a case where the Target is sensitive to the channel event, and reacts to the request asynchronously. This can be used for example to model asynchronous memories. The release and data sampling can be also triggered independently, release asynchronously and data sampling synchronously to indicate asynchronous SCmdAccept() and MRespAccept() behavior of OCP.



**Figure 9. Control and data sequencing (Asynchronous)**

Figure 10. overlays the transaction behavior seen by Initiator and Target with equivalent RTL block diagram and signal sequence. All signal names not beginning with capital M or S are internal to the modules, and not part of OCP interface. They illustrate how the Layer-1 might be mapped to RTL signals. The channel (that implements the OCP state machine) has a minimum slave command accept delay of one cycle. In this example the master requests write two times (two consecutive requests). At the first request, the channel is idle and the request is accepted immediately. The second request is accepted after the channel becomes non-busy while the first request cycle ends. The Target needs an extra cycle to process the first request, so it does not call `SRelease()` until two cycles after the first one has completed, thus holding the channel state machine.



**Figure 10. RTL Equivalent of Layer-1 Fully Synchronous Channel**

The registers and the state machines are implemented in the master and the slave in RTL design though in some cases the interconnect scheme can add a number of cycles of latency. In the SystemC Layer-1 design, the registers are partially within the master and the slave and partially within the OCP channel model, and the state machines are within the OCP channel model (the dark blob). The channel calls provide the Initiator and the Target effectively with interface functionality similar to what an RTL core would have to its OCP interface module. The reason for all this is to isolate OCP protocol from master and slave SystemC models, providing only limited control and limited visibility to the OCP, thus making the core interface independent of the OCP interface configuration. By changing the channel parameters, the OCP functionality and performance can be greatly modified without touching the cores.

The `SputResponse()` – `MgetResponse()` sequencing is identical, and independent of the request sequence. They can be used together to implement pipelined and non-pipelined sequences.

### 3.2. Layer-2

The Layer-2 is not cycle-accurate and therefore most of the protocol specific signals and parameters of Layer-1 are redundant.

#### 3.2.1. Data Interface

The data interface methods are implemented in template class **Bus\_Signals<DataType>**. They are accessible through the interface method **GetDataIF()**. The data interface gives access to the data and address fields of the interface. Basic fields like data and address have generically named methods, which can be used also with other protocols than OCP. All OCP fields can also be accessed directly with their OCP name. For example to set MData, call **GetDataIF->MData(new\_value)**, and to read MData, call **value = GetDataIF->MData()**. The data interface methods should be called according to rules set in the control interface. All data interface methods return immediately.

#### Initiator Data Interface

*void Address(int Address)*

Purpose: Set target address for transfer (MAddr field of OCP)

*int MNumWords*

Purpose: Number of words to be transmitted from Initiator to Target.  
A word is one element of the data burst containing one or more bytes.

*bool MputData(DataType &IData, int NumWords)*

Purpose: Allocates memory and copies the data (NumWords words) from Initiator to the channel. Sets MNumWords = NumWords.

Return: True at success, false at failure.

*void MputData(DataType \*IDataPointer, int NumWords)*

Purpose: Sets the Initiator data pointer. Used for passing data pointers to the Target.  
Sets MNumWords = NumWords.

*DataType \* MgetData( int &NumWords)*

Purpose: Return pointer to target data. Sets NumWords = SNumWords.

## Target Data Interface

*int SNumWords*

Purpose: Number of words to be transmitted from Target to Initiator. A word is one element of the data burst containing one or more bytes.

*int Address()*

Purpose: Returns the start address of a burst.

*void SputData(DataType &TData, int NumWords)*

Purpose: Allocates memory and copies the data from Target to the channel. Sets SNumWords = NumWords.

*void SputData(DataType \*TDataPointer, int NumWords)*

Purpose: Sets the Target data pointer. Used for passing data pointers to the Initiator. Sets SNumWords = NumWords.

*DataType \* SgetData(int &NumWords)*

Purpose: Returns a pointer to Initiator data. Sets NumWords = MNumWords.

Additional members, which are normally protocol specific, can be defined by the user. OCP fields can be accessed directly with OCP names.

### 3.2.2. Control Interface

The control interface supports both pipelined and non-pipelined access. The control methods are executed after certain events, or with blocking semantics.

## Initiator Interface

*Bus\_Signals<DataType> \* GetDataIF()*

Purpose: Gets the channel data pointer.

Return: Returns immediately with pointer to channel data.

*int IsWrite()*

Purpose: Specifies read or write transfer.  
 Return: 0 = Read transfer  
 1 = Write transfer  
 other = Reserved

*bool MgetSBusy ()*

Purpose: Data busy semaphore. Initiator may access the channel data only when the busy returns false.  
 Return: Returns immediately true if Target has not responded to the last request event, and false if it has.  
 Events: No events.

*bool MputWriteRequestBlocking ()*

Purpose: Issues a write request to Target. The Initiator may write channel data any time outside this call by either copying or pointer passing. MgetSbusy() returns always false when this call is used.  
 Return: Suspends calling thread. Returns after the Target has released data semaphore. Returns true at success, false at failure.  
 Events: Causes a write event to Target.

*Bool MputWriteRequest()*

Purpose: Issues a write request to Target. Should be called only after MgetSbusy() returns false, i.e. the Target is not using the channel data buffer anymore. Both copy and pointer-passing data interface can be used, but the passed data buffer must not be re-used by the Initiator until Sbusy() is false.  
 Return: Returns immediately. Initiator must suspend itself to allow Target process to run. Returns true if channel accepts the request, false if not.  
 Events: Causes a write event to Target. No event when return value is false.

*Bool MputReadRequestBlocking ()*

Purpose: Issues a read request to Target. Initiator may read the response data only after an MgetResponse\*() call. MgetSBusy() returns always false when this call is used.  
 Return: Suspends calling thread. Returns after the Target has released data semaphore. Returns true at success and false at failure.

Events: Causes a read event to Target.

*bool MputReadRequest ()*

Purpose: Issues a read request to Target. Can be called only when MgetSBusy() returns false. The Initiator may read the response data only after an MgetResponse\*() call.

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

Events: Causes a read event to Target. No event if return value is false.

*Bool MgetResponseBlocking (bool Release)*

Purpose: Gets response from Target. Can only be called after Mput\*Request\*(). Channel data is valid, and channel data pointer can be used by the Initiator any time outside this call, before Mput\*Request\*() is called again.

Parameters: If Release is true, the Target SputResponseBlocking() is released immediately. Otherwise it is not released until MRelease() is called and there is no effect to SputResponse().

Return: Suspends calling thread. Returns after the Target has released data semaphore. True at success, false at failure.

Events: No events

*bool MgetResponse (bool Release)*

Purpose: Gets response from Target. Can only be called after Mput\*RequestBlocking() or Mput\*Request\*(). Channel data is valid, and channel data pointer can be used by the Initiator when this call returns true, and before MPut\*Request is called again.

Parameters: If Release is true, the Target SputResponseBlocking() is released immediately. Otherwise it is not released until MRelease() is called. There is no effect to SputResponse().

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

Events: No events.

*void MRelease ()*

Purpose: Releases SputResponseBlocking(). No effect on SputResponse(). Notice that calling thread is not suspended, and Target thread cannot run until Initiator suspends itself.

Return: Returns immediately. No return value.

Events: No events.

*void MRelease (int Time)*

Purpose: Releases SputResponseBlocking() after *Time* -time units. No effect on SputResponse(). Notice that calling thread is not suspended, and Target thread cannot run until Initiator suspends itself. The time unit size is set in management interface of the channel.

Return: Returns immediately. No return value.

Events: No events.

**Target Interface***Bool SgetRequestBlocking(bool Release)*

Purpose: Block execution until Initiator signals a request event. Channel data can be used, until Target thread suspends (meets a wait() call).

Parameters: If *Release* is true, the Target Mput\*RequestBlocking() is released immediately. Otherwise it is not released until SRelease() is called. There is no effect to Mput\*Request().

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

Events: No events.

*Bool SgetRequest(bool Release)*

Purpose: Get request. If true, channel data can be used, until Target thread suspends (meets a wait() call).

Parameters: If *Release* is set true, the Target Mput\*RequestBlocking() is released immediately. Otherwise it is not released until SRelease() is called. There is no effect to Mput\*Request().

Return: Return true when Initiator request is pending, return immediately.

Events: No events.

*Bool SputResponseBlocking ()*

Purpose: Issues a response to Target. Can be called only after a request is detected by SgetRequest(). The response data can only be written between get and put calls.

Return: Suspends calling thread. Returns after the Initiator called MRelease(). Returns true at success and false at failure.

Events: Causes a response event to Initiator.

*Bool SputResponse()*

Purpose: Issues a response to Target. Can be called only after a request is detected by SgetRequest() or SgetRequestBlocking(). The response data can only be written between get and put calls.

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

Events: Causes a response event to Initiator. No event if return value is false.

*void SRelease()*

Purpose: Releases Mput\*RequestBlocking(). No effect on Mput\*Request(). Releases data semaphore. Notice that calling thread is not suspended, and Initiator thread cannot run until Target suspends itself.

Return: Returns immediately. No return value.

Events: No events.

*void SRelease(int Time)*

Purpose: Release Mput\*RequestBlocking() after Time –time units. No effect on Mput\*Request(). Releases data semaphore. Notice that calling thread is not suspended, and Initiator thread cannot run until Target suspends itself.

Return: Returns immediately. No return value.

Events: No events.

### 3.2.3. Pseudo Code Examples

The Layer-2 sequencing is similar to Layer-3 (see section 3.3.3), with one exception: The Layer-2 `xRelease()` calls may insert a specific time delay, after which the `MputRequestBlocking()` or `SputResponseBlocking()` method will be released. This makes it possible to model duration of data transfer over the interface. If request and response use different execution threads, it is possible to model timing of a pipelined system. The Layer-2 is not fully cycle-accurate, but rather cycle-approximate, since only the caller of `xRelease()` may throttle the transfer once it has started. Anyhow, the time abstraction allows modeling of shared resources.

The data access is also similar to that of Layer-3, but with more limited data types (see above). The number of interface words transferred in one transaction is communicated over the interface with the `MNumWords` and `SNumWords` data interface fields. These may be different from protocol specific burst size, if a burst is divided into several Layer-2 'packets'.

All the examples in 3.3.3 are valid at Layer-2, with the addition that on Layer-2 timing estimations are included in the `xRelease()` calls. This is shown in Figure 11. :

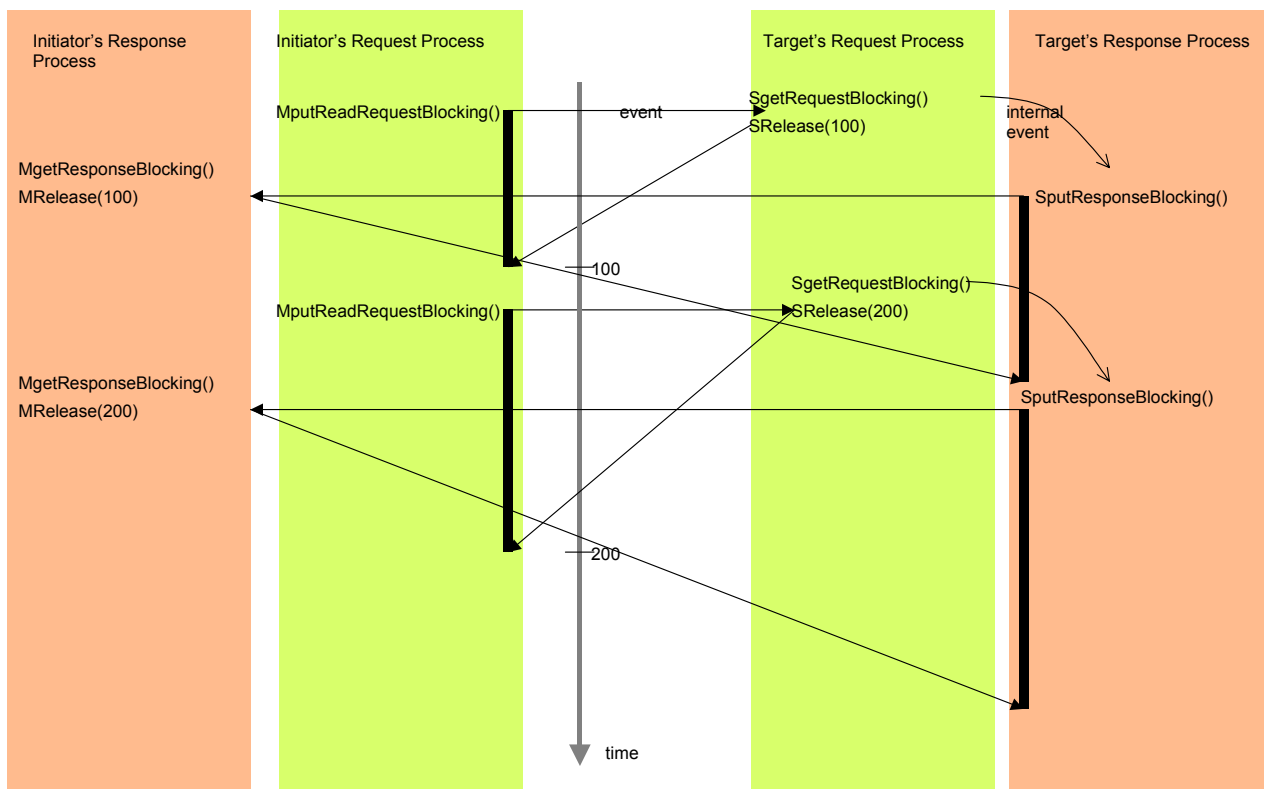
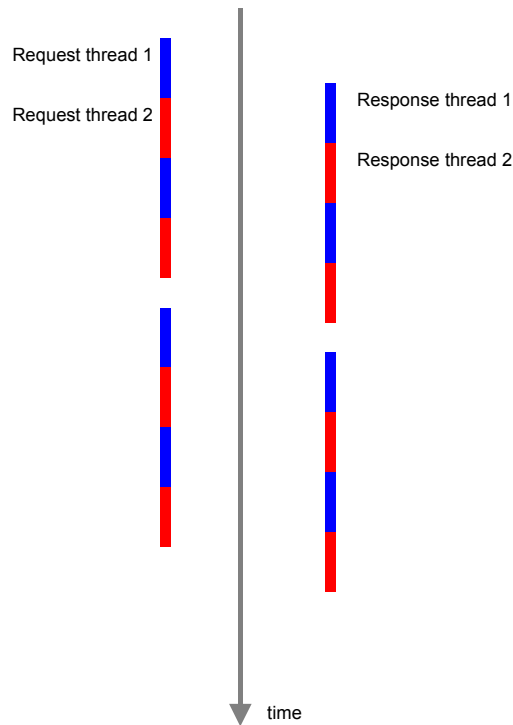


Figure 11. A Pipelined system with estimated timing



**Figure 12. A Pipelined, threaded system with estimated timing**

The Figure 12. shows the thread activation time of a multi-threaded simulation where the threading effect is achieved by alternating transactions with different thread identifiers. This simulates an OCP interface with different threads.

### 3.3. Layer-3

#### 3.3.1. Data Interface

The Layer-3 data interface differs from the Layer-2 in the allowed data types and addressing. The Layer-3 data type can be any legal C++ type, whereas the Layer-2 data type must always be an array of words that can be transferred over a bus interface in a burst or a fraction of a burst. The Layer-3 does not use explicit address. If an address is needed in the system, it must be embedded in the data type. The data access calls are similar to Layer-2, but without data array size parameter.

#### Initiator Data Interface

```
bool MputData(DataType &IData)
```

Purpose: Allocates memory and copies the data (sizeof(IData)) from Initiator to the channel.

Return: Returns true at success, false at failure.

*void MputData(DataType \*IDataPointer)*

Purpose: Sets the Initiator data pointer. Used for passing data pointers to the Target.

*DataType \* MgetData()*

Purpose: Returns pointer to target data.

### **Target Data Interface**

*void SputData(DataType &TData)*

Purpose: Allocates memory and copies the data from Target to the channel.

*void SputData(DataType \*TDataPointer)*

Purpose: Sets the Target data pointer. Used for passing data pointers to the Initiator.

*DataType \* SgetData()*

Purpose: Returns pointer to Initiator data.

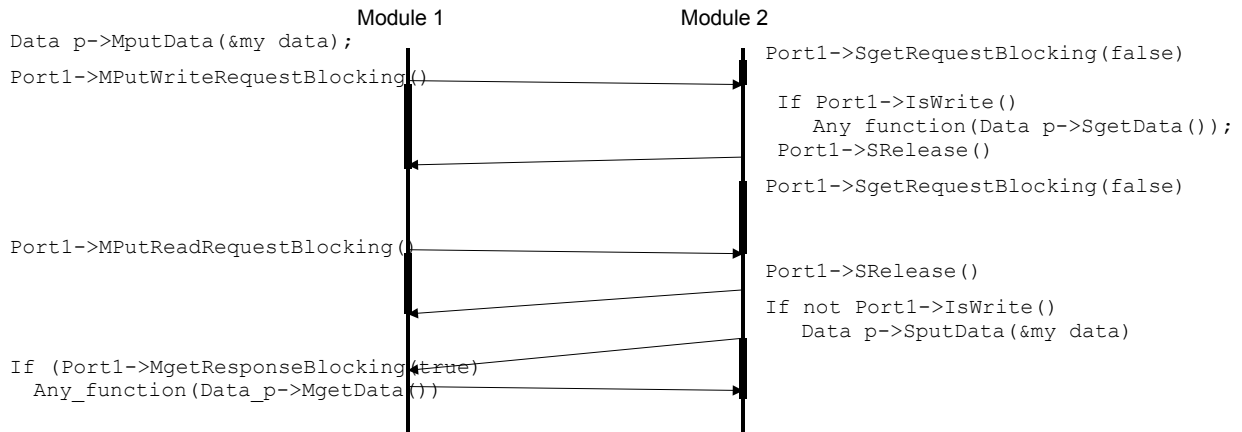
### **3.3.2. Control Interface**

The control interface is the same as in Layer-2, but SRelease(int Time) and MRelease(int Time) are not allowed. The only difference to Layer-2 sequencing is that there is no time, only event ordering.

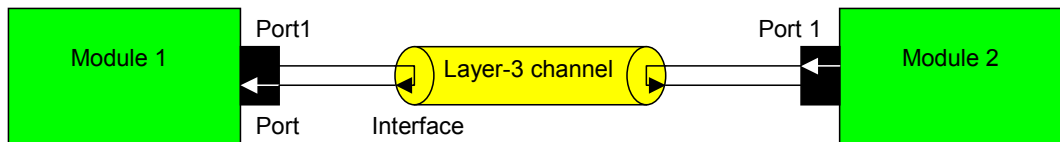
### 3.3.3. Pseudo-code Examples

```
Datatype my_data;
Bus Signals<Datatype> * data p;
data p = Port1->GetDataIF();
```

```
Datatype my_data;
Bus Signals<Datatype> * data p;
Data p = Port1->GetDataIF();
```

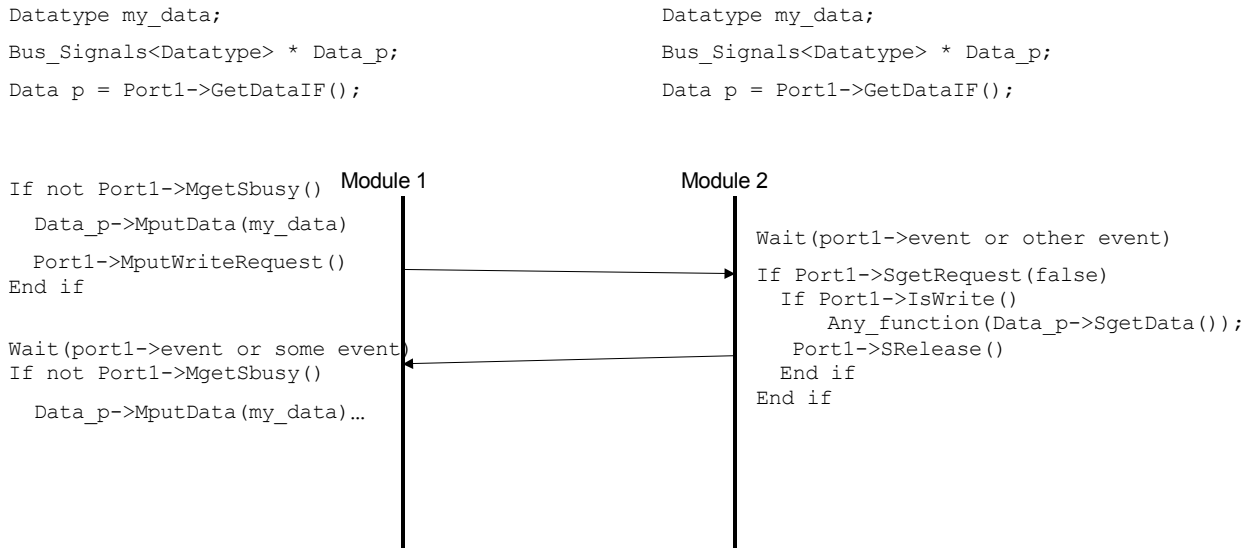


**Figure 13. A non-pipelined system with blocking messages and pointer passing**



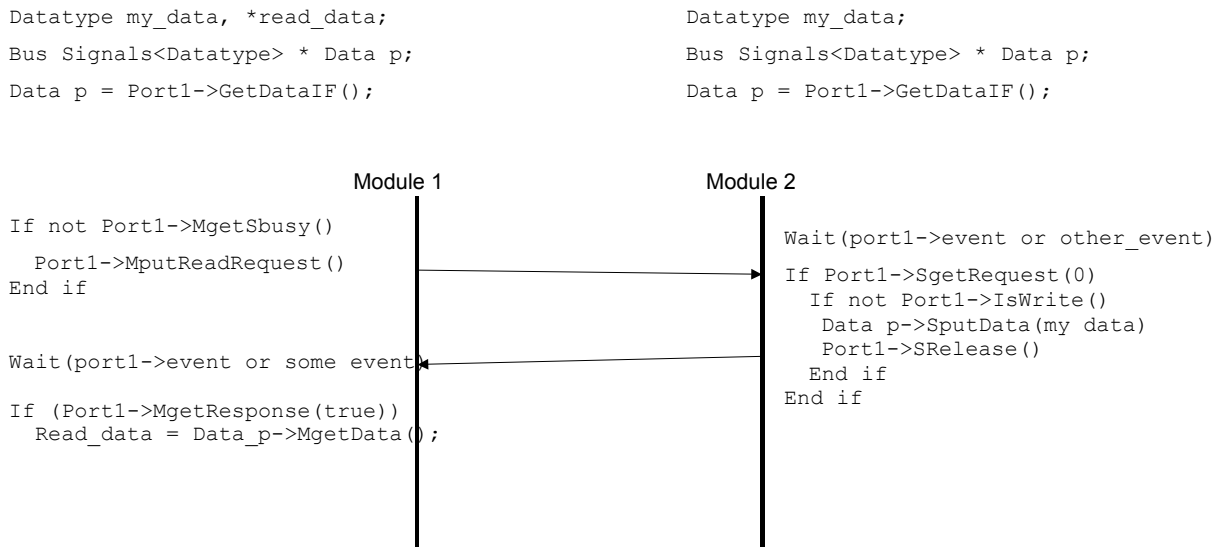
**Figure 14. Initiator and Target connected through a (Layer-3) communication channel**

Figure 13. : The Module 1 is connected to Module 2 through a Layer-3 channel, visible to both through a port (both named 'Port1' here, see Figure 14. ) Both modules fetch the channel pointers before starting message processing. The Module 1 sets the data pointer to its internal variable 'my\_data', and issues a blocking write call to Module 2. The blocking get request returns as a result, and the Module 2 can check the direction of the message and use the data it sees in the channel data pointer (now Module 1's my\_data). Once the Module 2 is through with using the data, it calls the Srelease() method, which causes the Module 1 to resume execution, after the Module 2 has called blocking get request anew. A similar sequence is shown for blocking read requests. Since both modules use only blocking calls, only one of them can execute at a time, and data integrity is guaranteed even though both modules manipulate pointers to the same variable. This sequence is analogous to Module 1 issuing RPC to Module 2, or synchronous message passing with pointer passing semantics.



**Figure 15. A write in non-pipelined system with non-blocking messages and data copying**

The Figure 15. example shows both modules using non-blocking calls. The Module 1 must check channel data semaphore (MgetSbusy()) before it can touch the channel data. If the MgetSbusy() is false, the Module 1 copies the variable my\_data to channel data pointer, and issues a non-blocking write request. Since the write request returns immediately, and the data were copied, the Module 1 may continue manipulating my\_data without regard to data semaphore. It may not write again to the channel, or call write request until the semaphore has cleared. The Module 2 starts executing in response to the event caused by write request, or some other event in the system. It checks that there is a non-served request with non-blocking get request call, checks the direction of the request (write), and uses the data. The slave does not need to copy the data from channel data pointer, since it holds the channel semaphore. The Module 2 releases the semaphore by calling the SRelease() method. Alternatively, the Module 2 may copy the data, and release the semaphore immediately. Both ways maintain data integrity. This sequence is analogous to shared memory with semaphore and event, or to asynchronous message passing with copy semantics.



**Figure 16. A read in non-pipelined system with non-blocking messages and data copying**

The Figure 16. example shows a sequence of non-blocking read transactions. The sequence is similar to the previous example, with data direction reversed. The Module 1 uses the Module 2's data once the semaphore has cleared.

### **3.4. Management Plane**

Each abstraction layer implementation provides a management interface that configures the channel during the elaboration time. The management plane can configure parameters like protocol subsets, latencies, burst sizes, register shielding, address map relocation, etc. Most importantly, the management plane can be used as a link between low-level software generation and system-level modeling, if they use the same hardware description database.

Each abstraction layer has its own subset of the management plane parameters (since not all parameters make sense in all layers). These parameters are provided to the SystemC channel models as a structure, through a special `ConfigureChannel()` member function, called at elaboration time. The contents of the data structure can be parsed from an external file (e.g. XML format) at compilation time, or hard coded in design time.

#### 4. APPENDIX A: BUFFER MECHANICS

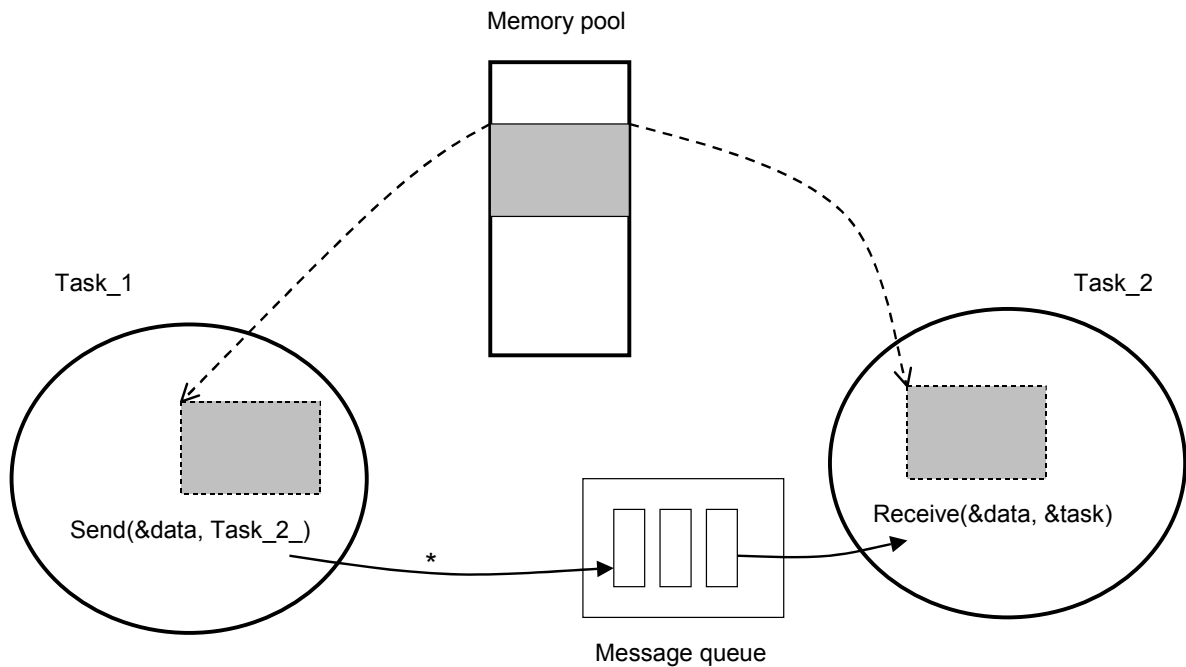
Here we discuss buffer handling in software and hardware modelling.

##### 4.1. Software Messages

Software processes may communicate with signals (messages), which can perform both task synchronization and data sharing. Signals either copy the data buffer attached to them, or pass a pointer to that buffer to the receiving task. Pointer passing executes faster, but copying may have to be performed

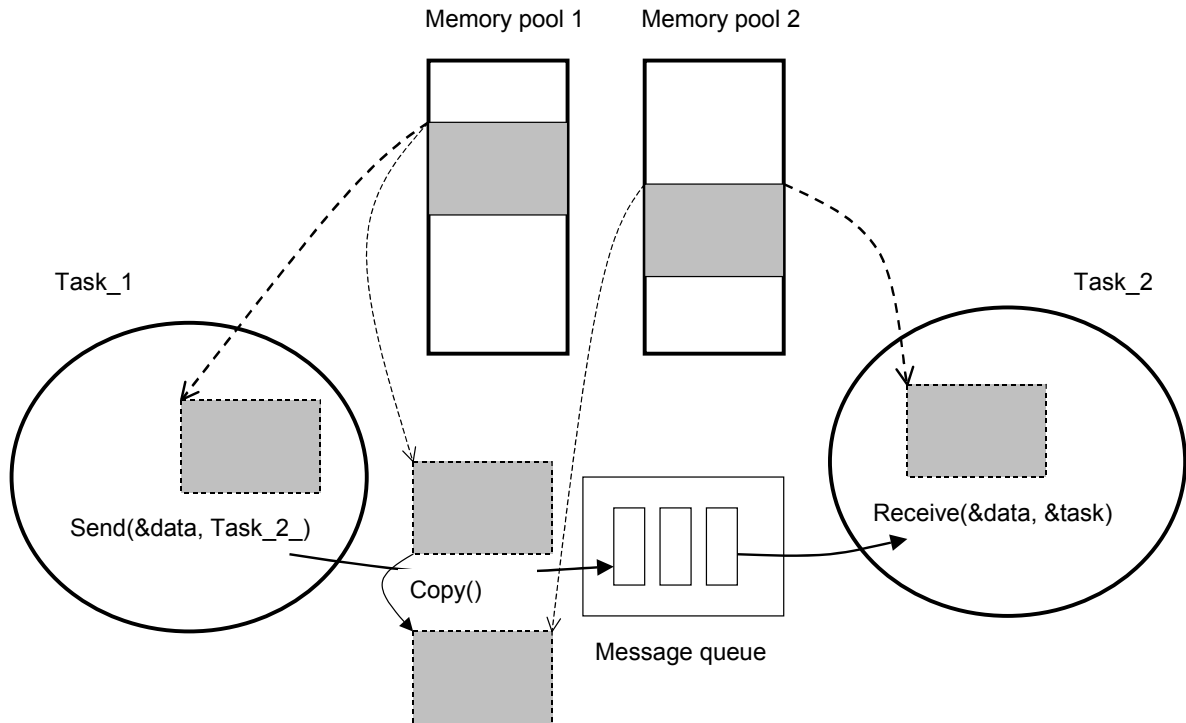
- A) for security or system integrity reasons,
- B) because the sending and receiving task do not share common memory pool,
- C) the sending and receiving tasks reside in different hosts.

An OS signal `Send()` typically contains a parameter telling whether the data should be copied. The copying may also be forced by the operating system, in cases B) and C).



**Figure 17. Pointer passing signal in RTOS**

Figure 17. shows a signal with pointer passing semantics. Sending and receiving tasks both have access to the same memory pool. The sending task gives ownership of the buffer to the receiving task after it sends the signal, and may not do any operations to the buffer after the sending. The receiving task must explicitly free the buffer and signal the sender before the buffer can be reused, unless the memory protection can be left to the operating system. In either case, the sender may not use the `*data`-pointer unless it is reassigned to point elsewhere.

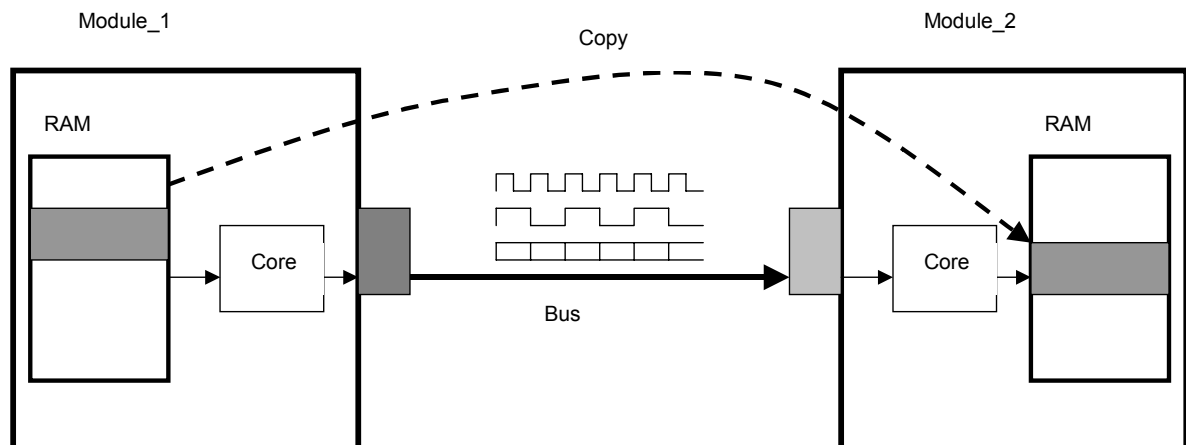


**Figure 18. Copy signal in RTOS**

Figure 18. shows a signal, which copies data buffer from Task\_1's memory pool to Task\_2's. The execution is slower than in pointer passing, but the data buffers are protected and system integrity guaranteed. Note that how the signal allocates the buffer in the receiving task is not covered here.

#### 4.2. Hardware Messages

Message abstraction for communication can also be used with hardware modeling. Hardware modules cannot allocate memory dynamically, and therefore we are not covering shared-memory arrangements in this text. The premise here is that each module can access a static memory buffer, and an explicit transfer mechanism (such as a bus) must exist between modules wanting to share data. The following Figure shows the setup under inspection. Next we will present how message abstraction can simulate the data transfer.



**Figure 19. Bus communication between hardware modules**

The modules in Figure 19. rely on a bus protocol to copy the data from one module to another, to the correct location in the receiver's RAM. No pointers can be shared in this setup, as in the case of software tasks. The process of copying the data can be abstracted, or bundled, in a single transaction or message capturing the intent of the system. This is shown with the dashed arrow indicating a copy operation between the two memories. The abstract copy operation hides the actual mechanism for data transfer, and the only indication is that after the copy is finished, the sender's data buffer content appears in the receiver's memory.

Different mechanisms can be used to capture this intent in a simulation model that implements the protocol hiding. The actual copying can be done in three different places, in Sender, Bus or Receiver model. The location of the copying is dictated by the time mode of the simulation. Since the actual hardware that is modeled may or may not use mechanisms to protect memory, also the simulation model must exhibit similar behavior. For example, the Sender's memory may be shared by its Core and its Bus interface, which can access the memory simultaneously. It is thus possible that the Core corrupts the memory buffer while the Bus Interface is transmitting the data. We will now present the call mechanisms and copy locations, which allow different memory protection schemes.

The Sender and Receiver models may use two function call return mechanisms, blocking and non-blocking. Blocking means that the call does not return until the communication counterpart has released a lock. A blocking write-call to a Receiver returns after the data buffer has been used (copied). The blocking call offers automatic memory protection. A non-blocking call returns immediately, and leaves buffers vulnerable to corruption if the data is not copied, just as in the case of a non-copying software signal, described in the previous chapter.

In the non-blocking case, we can utilize either a semaphore or a copy to protect the data. A semaphore is simply a shared variable between the Sender and the Receiver, which the Sender can check before attempting to reuse its data buffer. The Receiver (or the Bus model) will release the semaphore after the data is safe. This simulates a bus state "transfer in progress" in a real hardware implementation. The semaphore is conveniently implemented in the Bus model, for example so that e.g. a write()-call returns false if the semaphore is not released. As stated earlier, the copy can happen in Sender, Bus or Receiver. If the Sender copies the data, it allocates a new buffer, which it passes along. The Receiver must take ownership of this buffer, replace its target buffer with this new copy, and delete the buffer after it receives the next. This allocation process cannot create new memory, since the hardware memory is static. Alternatively, the allocate-copy process may be implemented in the bus model. The benefit of this is the allocation may not be necessary at all, if the Receiver gives the Bus model a pointer to its memory, where the Bus model copies the data directly. If the Receiver copies the data from the pointer to its static buffer (as it does in hardware implementation, from the bus interface to the RAM), the only available protection method for the Sender is the semaphore.

### 4.3. Conclusion

We conclude that the minimum set of mechanisms that can express all hardware memory protection methods is a blocking call with pointer passing, and a non-blocking copying call with semaphore.

The simulation models for software signals require an additional mechanism; copying the data buffer in the model of the signal itself, as explained in the chapter 4.1.

5. APPENDIX B: SYNCHRONIZATION MECHANISMS

5.1. Layer-1 Mechanisms

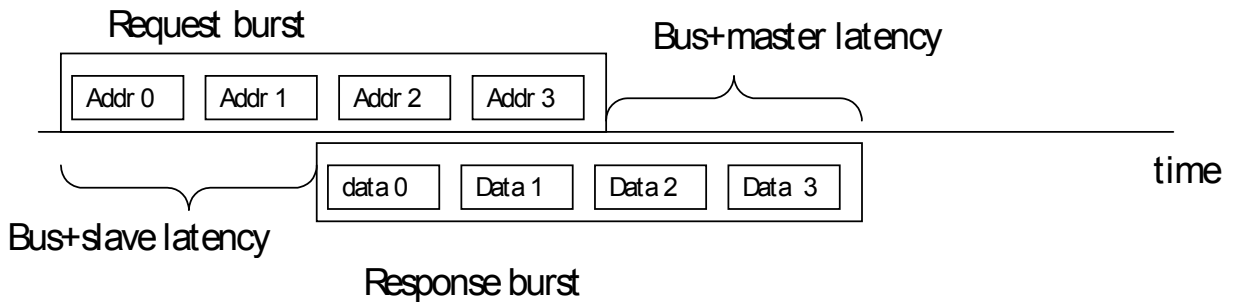


Figure 20. Pipelined burst transactions

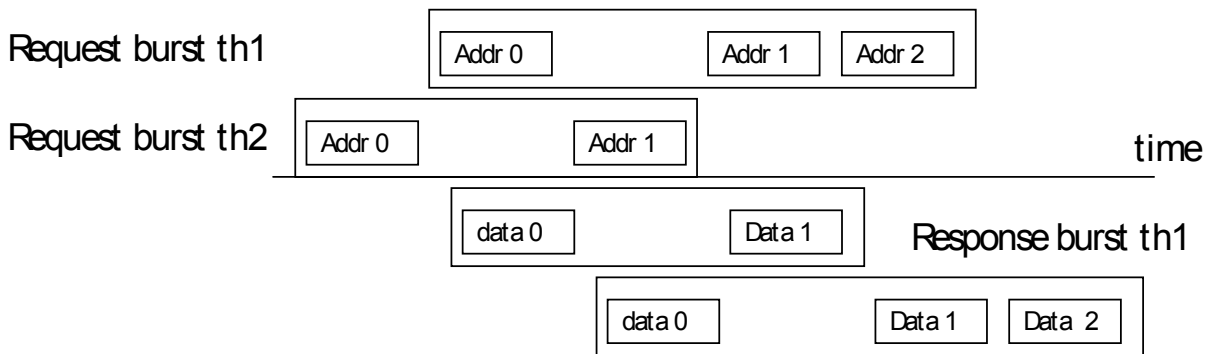


Figure 21. Pipelined, multi-threaded burst transactions

5.2. Layer-2 Mechanisms

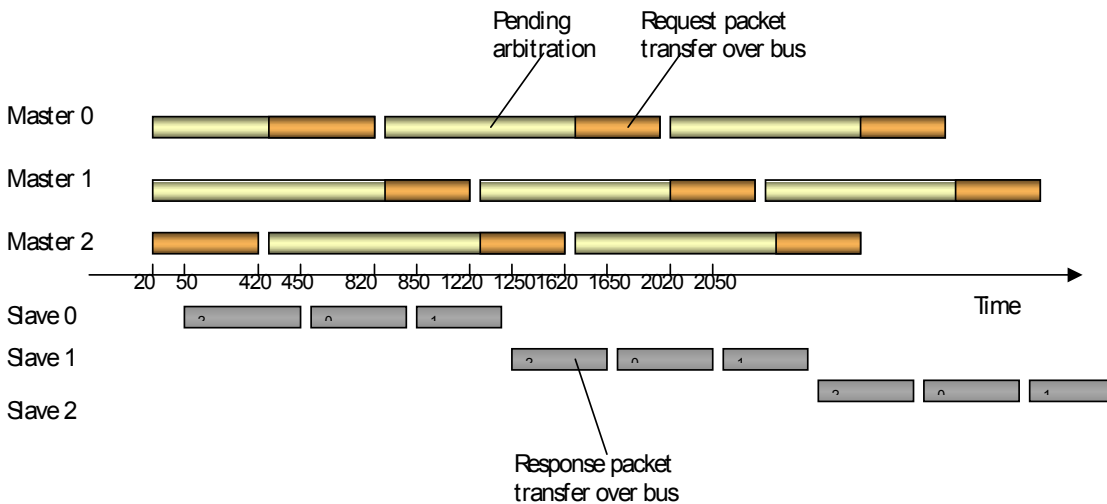
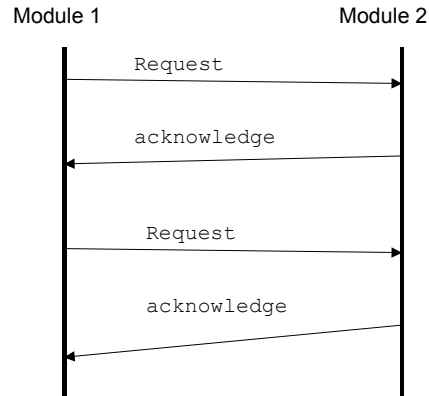


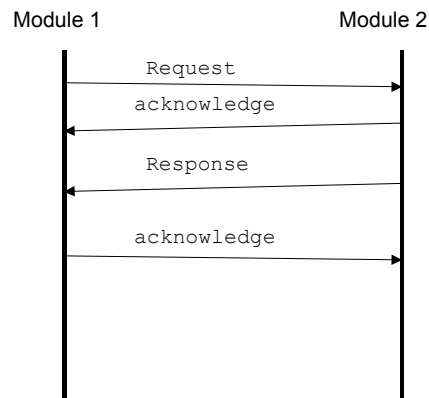
Figure 22. Pipelined, arbitrated multi-master, multi-slave system

### 5.3. Layer-3 Mechanisms



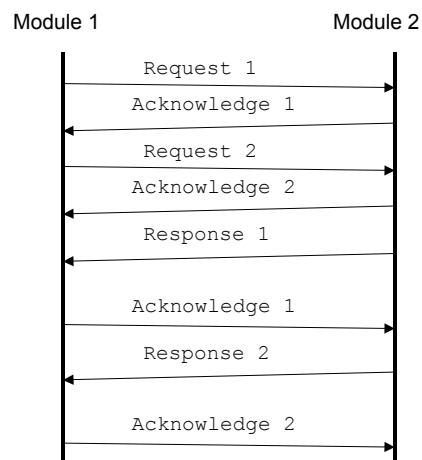
**Figure 23. Event-ordered non-split communication**

Non-split communication is synchronized with only single event (or semaphore) pair. The response is always ready before next request.



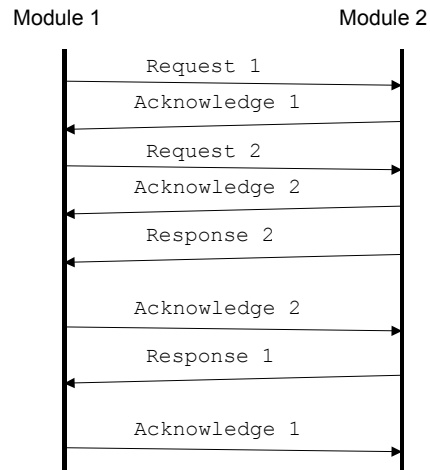
**Figure 24. Event-ordered split communication**

Split communication separates acknowledgement of request from signaling response. Thus two event (or semaphore) pairs are required.



**Figure 25. Event-ordered pipelined communication**

Pipelined communication extends split one with possibility to issue several requests (pipeline depth) before expecting response. Responses must come in the same order as requests.



**Figure 26. General pipelined communication**

General form of pipelined communication allows response re-ordering. The responses must be tagged, or the system must maintain knowledge of the re-ordering other ways.

## 6. APPENDIX C: INTRODUCTION TO OCP

The Open Core Protocol (OCP) defines a high-performance, bus-independent interface between IP cores that reduces design time, design risk, and manufacturing costs for SoC designs.

An IP core can be a simple peripheral core, a high-performance microprocessor, or an on-chip communication subsystem such as a wrapped on-chip bus. An IP core utilizing OCP can be interfaced to any bus. One test of a bus-independent interface is to connect a master to a slave without an intervening on-chip bus. This test not only drives the specification towards a fully symmetric interface, but also helps isolate the IP cores from bus specific decode/selection logic.

The benefits of using OCP are:

- Increased IP design reuse since cores can be made independent of the architecture and design of the SoC interconnect
- Simplified system verification by providing a controllable and observable boundary around the IP cores
- Low overhead as OCP supports a high degree of configurability to adapt to the core's functionality.

### 6.1. Configuration

To address the wide range of IP core functionality, performance and interface requirements, OCP defines a highly configurable interface. OCP includes all of the signals required to describe a IP core communications including data flow, control, and verification and test signals. Only the features needed by the cores communicating over an OCP are configured into the interface.

To increase transfer efficiency, many IP cores have wide data fields. OCP directly supports up to 128-bit data fields, allowing transfer of 16 bytes simultaneously. OCP refers to the chosen data field width as the *word size* of the OCP. The term word is used in the traditional computer system context; that is, a *word* is the natural transfer unit of the block.

### 6.2. Transfers

There are two basic OCP commands, Read and Write and three command extensions. The basic Write command is posted (i.e. has no response phase). The WriteNonPost and Broadcast commands have semantics that are similar to the Write command. A WriteNonPost explicitly instructs the slave not to post a write. For the Broadcast command, the master indicates that it is attempting to write to several or all remote target devices that are connected on the other side of the slave. As such, Broadcast is typically useful only for slaves that are in turn a master on another communication medium (such as an attached bus).

The third command extension, Read Exclusive (ReadEx) has protocol semantics that are similar to Read, but guarantees sufficient resource locking to support atomic read-modify-write or swap semantics. Upon receiving a ReadEx command, the slave attempts to acquire exclusive access to the addressed resource. Once the slave returns data from that address, the master can assume that it has obtained exclusive access and issue a Write or WriteNonPost command. The command notifies the slave to update the address (which must match the ReadEx address), and then to release exclusive access to the addressed resource.

Transfers of less than a full word of data are supported by providing byte enable information that specifies which bytes are to be transferred.

To provide high transfer efficiency, burst support is essential for many IP cores. OCP supports annotation of transfers with burst information but requires that appropriately sequenced addresses

accompany each successive command in the burst. This simplifies the requirements for address sequencing/burst count processing in the slave.

### 6.3. Ordering/Threading

OCP separates requests from responses; in other words transactions are split. A slave can accept a request from a master on one cycle and respond in a later cycle as long as the responses are presented in the same order as the originating requests. The separation of request from response allows OCP to have multiple outstanding requests thus enabling pipelining of transfers.

To support concurrency and out-of-order processing of transfers, OCP supports the notion of multiple threads. Transactions from different threads have no ordering requirements, and so can be processed out of order. This ordering flexibility can be especially beneficial at a memory interface such as an SDRAM. Within a single thread of data flow, all OCP transfers must remain ordered.

While the notion of a thread is a local concept between a master and a slave communicating over an OCP, it is possible to globally pass information from initiator to target using connection identifiers. Connection information helps to identify the initiator and can be used to determine priorities at the target.

### 6.4. Control Flow

While moving data between devices is a central requirement of on-chip communication systems, other types of communications are also important. Different types of control signaling are required to coordinate data transfers (for instance, high-level flow control) or signal system events (such as interrupts or power management). Dedicated point-to-point data communication is sometimes required. Many devices also require the ability to notify the system of errors that may be unrelated to address/data transfers.

OCP refers to all such communication as *sideband* (or out-of-band) signaling, since it is not directly related to the protocol state machines of the dataflow portion. OCP provides support for such signals through sideband signaling extensions.

Errors are reported across the OCP using two mechanisms. The error response code in the response field describes errors resulting from OCP transfers that provide responses. Posted write-type commands do not generate a response and cannot use the in-band reporting mechanism. The second method for reporting errors across the OCP is using out-of-band error fields. These signals report more generic sideband errors, including those associated with posted write commands.

OCP also includes basic hardware/software boundary signaling. For example, exchange of control and status information can be implemented over OCP. This allows the system to drive control information toward an IP core, and the IP core to report status information to the system. Control and status field widths are OCP configurable and their encoding is core specific.