July 2009

OSCI TLM-2.0 LANGUAGE REFERENCE MANUAL

Software version: TLM 2.0.1

Document version: JA31

Contributors

The TLM-2.0 standard was created under the leadership of the following individuals:

Bart Vanthournout, CoWare, TLM Working Group Chair
James Aldis, Texas Instruments, TLM Working Group Vice-Chair

Previous TLM Working Group Chairs:

Trevor Wieman, Intel
Frank Ghenassia, ST Microelectronics
Mark Burton, GreenSocs

This document was authored by:

John Aynsley, Doulos

The following is a list of active technical participants in the OSCI TLM Working Group at the time of release of this standard:

Tom Aernoudt, CoWare
James Aldis, Texas Instruments
Guillaume Audeon, ARM
John Aynsley, Doulos
David Black, XtremeEDA
Mark Burton, GreenSocs
Jerome Cornet, ST Microelectronics
Ross Dickson, Virtutech
Jakob Engblom, Virtutech
Alan Fitch, Doulos

Robert Guenzel, GreenSocs
Andrea Kroll, Jeda
Laurent Maillet-Contoz, ST Microelectronics
Kiyoshi Makino, Mentor Graphics
Marcelo Montoreano, Synopsys
Bart Vanthournout, CoWare
Yossi Veller, Mentor Graphics
Trevor Wieman, Intel
Charles Wilson, XtremeEDA

The following people have also contributed to the development of this standard within the OSCI TLM Working Group:

Mike Andrews, Mentor Graphics
Matthew Ballance, Mentor Graphics
Geoff Barrett, Broadcom
Ryan Bedwell, Freescale
Bishnupriya Bhattacharya, Cadence

Bobby Bhattacharya, ARM
Axel Braun, University of Tuebingen
Herve Broquin, ST Microelectronics
Bill Bunton, ESLX
Jack Donovan, XtremeEDA

Adam Erickson, Cadence
Othman Fathy, Mentor Graphics
George Frazier, Cadence
Michel Genard, Virtutech
Frank Ghenassia, ST Microelectronics
Mark Glasser, Mentor Graphics
Andrew Goodrich, Forte Design
Serge Goosens, CoWare
Thorsten Groetker, Synopsys
Karthick Gururaj, NXP
Kamal Hashmi, SpiraTech
Gino Van Hauwermeiren, NXP
Atsushi Kasuya, Jeda
Holger Keding, Synopsys
Devon Kehoe, Mentor Graphics
Anna Keist, ESLX
Wolfgang Klingauf, GreenSocs
Tim Kogel, CoWare
David Long, Doulos
Mike Meredith, Forte Design

Rishiyur Nikhil, Bluespec
David Pena, Cadence
Victor Reyes, NXP
Nizar Romdhane, ARM
Adam Rose, Mentor Graphics
Olaf Scheufen, Synopsys
Stefan Schmermbeck, Chipvision
Kolja Schneider, Fraunhofer
Shiri Shem-Tov, Freescale
Jean-Philippe Strassen, ST Microelectronics
Alan Su, ITRI & Springsoft
Stuart Swan, Cadence
Tsutomu Takei, STARC
Jos Verhaegh, NXP
Maurizio Vitale, Philips Semiconductors
Vincent Viteau, Summit Design
Thomas Wilde, Infineon
Hiroyuki Yagi, STARC
Kaz Yoshinaga, Starc
Eugene Zhang, Jeda

Contents

1   Overview

This document is the Reference Manual for the OSCI Transaction Level Modeling standard, version 2.0. This version of the standard supersedes versions 2.0-draft-1 and 2.0-draft-2, and is not generally compatible with either. This version of the standard includes the core interfaces from TLM-1.

TLM-2.0 consists of a set of core interfaces, the global quantum, initiator and target sockets, the generic payload and base protocol, and the utilities. The TLM-1 core interfaces, analysis interface and analysis ports are also included, although they are separate from the main body of the TLM-2.0 standard. The TLM-2.0 core interfaces consist of the blocking and non-blocking transport interfaces, the direct memory interface (DMI), and the debug transport interface. The generic payload supports the abstract modeling of memory-mapped buses, together with an extension mechanism to support the modeling of specific bus protocols whilst maximizing interoperability.

The TLM-2.0 classes are layered on top of the SystemC class library as shown in the diagram below. For maximum interoperability, and particularly for memory-mapped bus modeling, it is recommended that the TLM-2.0 core interfaces, sockets, generic payload and base protocol be used together in concert. These classes are known collectively as the *interoperability layer*. In cases where the generic payload is inappropriate, it is possible for the core interfaces and the initiator and target sockets, or the core interfaces alone, to be used with an alternative transaction type. It is even technically possible for the generic payload to be used directly with the core interfaces without the initiator and target sockets, although this approach is not recommended.

It is not strictly necessary to use the utilities to achieve interoperability between bus models. Nonetheless, these classes should be used where possible for consistency of style and are documented and maintained as part of the TLM-2.0 standard.

TLM 2.0 Classes

Interoperability layer

Generic payload & base protocol

Initiator & target sockets

Global quantum

TLM-1:
TLM-1 core interfaces
tlm_fifo
Analysis interface
Analysis ports

TLM-2 core interfaces:
Blocking transport interface
Non-blocking transport interface
Direct memory interface
Debug transaction interface

Utilities:
Convenience sockets
Payload event queues
Quantum keeper
Instance-specific extensions

IEEE 1666™ SystemC

Figure 1

The generic payload is primarily intended for memory-mapped bus modeling, but may also be used to model other non-bus protocols with similar attributes. The attributes and phases of the generic payload can be extended to model specific protocols, but such extensions may lead to a reduction in interoperability depending on the degree of deviation from the standard non-extended generic payload.

A fast, loosely-timed model is typically expected to use the blocking transport interface, the direct memory interface, and temporal decoupling. A more accurate, approximately-timed model is typically expected to use the non-blocking transport interface and the payload event queues. These statements are just coding style suggestions, however, and are not a normative part of the TLM-2.0 standard.

1.1   Scope

This document is the definitive reference manual for the TLM-2.0 standard. The main focus of this document is the key concepts and semantics of the TLM-2.0 classes, including the utilities. It does not describe all the supporting code, examples, and unit test. It lists the TLM-1 core interfaces, but does not define their semantics.

1.2   Source code and documentation

The TLM-2.0 release has a hierarchical directory structure as follows:

include/tlm   The C++ source code of the TLM-2.0 standard, with readme files and release notes

  ./tlm_h   TLM-2.0 interoperability layer
  ./tlm_h/tlm_2_interfaces   TLM-2 core interfaces
  ./tlm_h/tlm_generic_payload   TLM-2 generic payload
  ./tlm_h/tlm_sockets   TLM-2 sockets
  ./tlm_h/tlm_quantum   TLM-2 global quantum

  ./tlm_1   TLM-1 and analysis
  ./tlm_1/tlm_req_rsp   The TLM-1 standard
  ./tlm_1/tlm_req_rsp/tlm_1_interfaces   TLM-1 core interfaces
  ./tlm_1/tlm_req_rsp/tlm_channels   TLM-1 fifo and req-rsp channels
  ./tlm_1/tlm_req_rsp/tlm_ports   TLM-1 non-blocking ports with event finders
  ./tlm_1/tlm_req_rsp/tlm_adapters   TLM-1 slave-to-transport & transport-to-master adapters
  ./tlm_1/tlm_analysis   Analysis interface and ports

  ./tlm_utils   TLM-2 standard utility classes not essential for interoperability

docs   Documentation, including User Manual, white papers, and Doxygen
examples   A set of application-oriented examples with their own documentation
unit_test   A set of regression tests

The **docs** directory includes HTML documentation for the C++ source code created with Doxygen. This gives comprehensive text-based and graphical views of the code structured by class and by file. The entry point for this documentation is the file **docs/doxygen/html/index.html**.

2 References

This standard shall be used in conjunction with the following publications:

ISO/IEC 14882:2003, Programming Languages—C++

IEEE Std 1666-2005, SystemC Language Reference Manual

Requirements Specification for TLM 2.0, Version 1.1, September 16, 2007

2.1 Bibliography

The following books may provide useful background information:

Transaction-Level Modeling with SystemC, TLM Concepts and Applications for Embedded Systems, edited by Frank Ghenassia, published by Springer 2005, ISBN 10 0 387-26232-6(HB), ISBN 13 978-0-387-26232-1(HB)

Integrated System-Level Modeling of Network-on-Chip enabled Multi-Processor Platforms, by Tim Kogel, Rainer Leupers, and Heinrich Meyr, published by Springer 2006, ISBN 10 1-4020-4825-4(HB), ISBN 13 978-1-4020-4825-4(HB)

ESL Design and Verification, by Brian Bailey, Grant Martin and Andrew Piziali, published by Morgan Kaufmann/Elsevier 2007, ISBN 10 0 12 373551-3, ISBN 13 978 0 12 373551-5

3 Introduction

3.1 Background

The TLM-1 standard defined a set of core interfaces for transporting transactions by value or const reference. This set of interfaces is being used successfully in some applications, but has three shortcomings with respect to the modeling of memory-mapped buses and other on-chip communication networks:

a) TLM-1 has no standard transaction class, so each application has to create its own non-standard classes, resulting in very poor interoperability between models from different sources. TLM-2.0 addresses this shortcoming with the generic payload.

b) TLM-1 has no support for timing annotation, so no standard way of communicating timing information between models. TLM-1 models would typically implement delays by calling **wait**, which slows down simulation. TLM-2.0 addresses this shortcoming with the addition of timing annotation to the blocking and non-blocking transport interface.

c) The TLM-1 interfaces require all transaction objects and data to be passed by value or const reference, which slows down simulation. Some applications work around this restriction by embedded pointers in transaction objects, but this is non-standard and non-interoperable. TLM-2.0 addresses this shortcoming with transaction objects whose lifetime extends across several transport calls, supported by a new transport interface.

3.2 Transaction-level modeling, use cases and abstraction

There has been a longstanding discussion in the ESL community concerning what is the most appropriate taxonomy of abstraction levels for transaction level modeling. Models have been categorized according to a range of criteria, including granularity of time, frequency of model evaluation, functional abstraction, communication abstraction, and use cases. The TLM-2.0 activity explicitly recognizes the existence of a variety of use cases for transaction-level modeling (see the Requirements Specification for TLM-2.0), but rather than defining an abstraction level around each use case, TLM-2.0 takes the approach of distinguishing between interfaces (APIs) on the one hand, and coding styles on the other. The TLM-2.0 standard defines a set of interfaces which should be thought of as low-level programming mechanisms for implementing transaction-level models, then describes a number of coding styles that are appropriate for, but not locked to, the various use cases.

The definitions of the standard TLM-2.0 interfaces stand apart from the descriptions of the coding styles. It is the TLM-2.0 interfaces which form the normative part of the standard and ensure interoperability. Each coding style can support a range of abstraction across functionality, timing and communication. In principle users can create their own coding styles.

An untimed functional model consisting of a single software thread can be written as a C function or as a single SystemC process, and is sometimes termed an *algorithmic* model. Such a model is not *transaction-level* per se, because by definition a transaction is an abstraction of communication, and a single-threaded model has no inter-process communication. A transaction-level model requires multiple SystemC processes to simulate concurrent execution and communication.

An abstract transaction-level model containing multiple processes (multiple software threads) requires some mechanism by which those threads can yield control to one another. This is because SystemC uses a co-operative multitasking model where an executing process cannot be pre-empted by any other process. SystemC processes yield control by calling *wait* in the case of a thread process, or returning to the kernel in the case of a method process. Calls to *wait* are usually hidden behind a programming interface (API), which may model a particular abstract or concrete protocol that may or may not rely on timing information.

Synchronization may be *strong* in the sense that the sequence of communication events is precisely determined in advance, or *weak* in the sense that the sequence of communication events is partially determined by the detailed timing of the individual processes. Strong sychronization is easily implemented in SystemC using FIFOs or semaphores, allowing a completely untimed modeling style where in principle simulation can run without advancing simulation time. Untimed modeling in this sense is outside the scope of TLM-2.0. On the other hand, a fast virtual platform model allowing multiple embedded software threads to run in parallel may use either strong or weak synchronization. In this standard, the appropriate coding style for such a model is termed *loosely-timed*.

A more detailed transaction-level model may need to associate multiple protocol-specific timing points with each transaction, such as timing points to mark the start and the end of each phase of the protocol. By choosing an appropriate number of timing points, it is possible to model communication to a high degree of timing accuracy without the need to execute the component models on every single clock cycle. In this standard, such a coding style is termed *approximately-timed*.

Figure 2
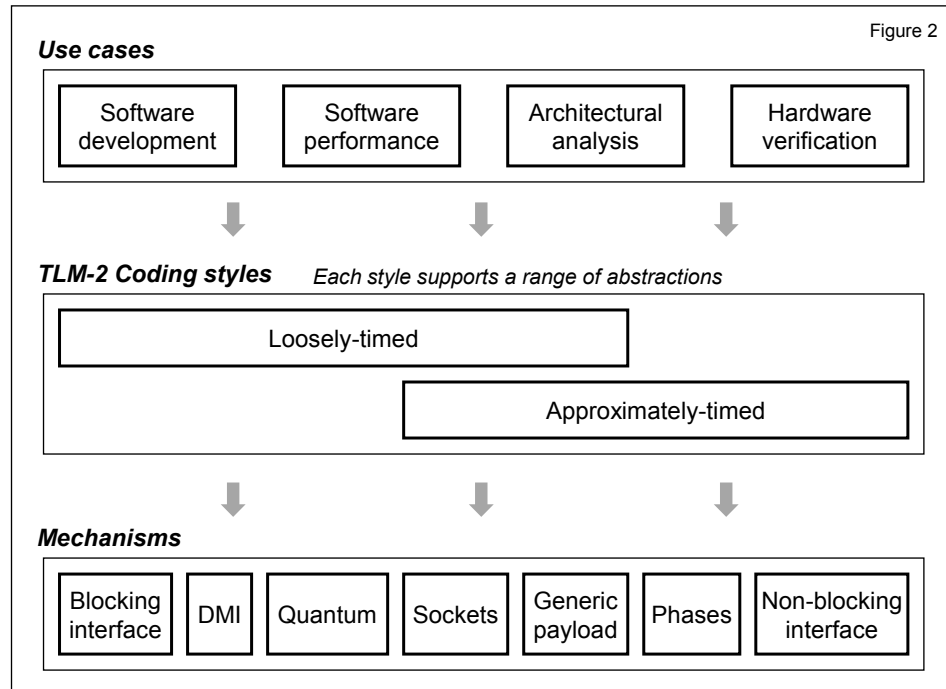
3.3 Coding styles

A coding style is a set of programming language idioms that work well together, not a specific abstraction level or software programming interface. For simplicity and clarity, this document restricts itself to elaborating two specific named coding styles; *loosely-timed* and *approximately-timed*. By their nature the

coding styles are not precisely defined, and the rules governing the TLM-2.0 core interfaces are defined independently from these coding styles. In principle, it would be possible to define other coding styles based on the TLM-1 and TLM-2.0 mechanisms.

3.3.1   Untimed coding style

TLM-2.0 does not make explicit provision for an untimed coding style, because all contemporary bus-based systems require some notion of time in order to model software running on one or more embedded processors. However, untimed modeling is supported by the TLM-1 core interfaces. (The term *untimed* is sometimes used to refer to models that contain a limited amount of timing information of unspecified accuracy. In TLM-2.0, such models would be termed loosely-timed.)

3.3.2   Loosely-timed coding style and temporal decoupling

The loosely-timed coding style makes use of the blocking transport interface. This interface allows only two timing points to be associated with each transaction, corresponding to the call to and return from the blocking transport function. In the case of the base protocol, the first timing point marks the beginning of the request, and the second marks the beginning of the response. These two timing points could occur at the same simulation time or at different times.

The loosely-timed coding style is appropriate for the use case of software development using a virtual platform model of an MPSoC, where the software content may include one or more operating systems. The loosely-timed coding style supports the modeling of timers and interrupts, sufficient to boot an operating system and run arbitrary code on the target machine.

The loosely-timed coding style also supports *temporal decoupling*, where individual SystemC processes are permitted to run ahead in a local "time warp" without actually advancing simulation time until they reach the point when they need to synchronize with the rest of the system. Temporal decoupling can result in very fast simulation for certain systems because it increases the data and code locality and reduces the scheduling overhead of the simulator. Each process is allowed to run for a certain time slice or *quantum* before switching to the next, or instead may yield control when it reaches an explicit synchronization point.

Just considering SystemC itself, the SystemC scheduler keeps a tight hold on simulation time. The scheduler advances simulation time to the time of the next event, then runs any processes due to run at that time or sensitive to that event. SystemC processes only run at the current simulation time (as obtained by calling the method **sc_time_stamp**), and whenever a SystemC process reads or writes a variable, it accesses the state of the variable as it would be at the current simulation time. When a process finishes running it must pass control back to the simulation kernel. If the simulation model is written at a fine-grained level, then the overhead of event scheduling and process context switching becomes the dominant factor in simulation speed. One way to speed up simulation is to allow processes to run ahead of the current simulation time, or temporal decoupling.

When implementing temporal decoupling in SystemC, a process can be allowed to run ahead of simulation time until it needs to interact with another process, for example to read or update a variable belonging to another process. At that point, the process may either access the current value and continue (with some possible loss of timing accuracy) or may return control to the simulation kernel, only resuming the process when simulation time has caught up with the local "time warp". Each process is responsible for determining whether it can run ahead of simulation time without breaking the functionality of the model. When a process encounters an external dependency it has two choices: either force synchronization, which means yielding to allow all other processes to run as normal until simulation time catches up, or sample or update the current

value and continue. The synchronization option guarantees functional congruency with the standard SystemC simulation semantics. Continuing with the current value relies on making assumptions concerning communication and timing in the modeled system. It assumes that no damage will be done by sampling or updating the value too early or too late. This assumption is usually valid in the context of a virtual platform simulation, where the software stack should not be dependent on the low-level details of the hardware timing anyway.

Temporal decoupling is characteristic of the loosely-timed coding style.

If a process were permitted to run ahead of simulation time with no limit, the SystemC scheduler would be unable to operate and other processes would never get the chance to execute. This may be avoided by reference to the *global quantum*, which imposes an upper limit on the time a process is allowed to run ahead of simulation time. The quantum is set by the application, and the quantum value represents a tradeoff between simulation speed and accuracy. Too small a quantum forces processes to yield and synchronize very frequently, slowing down simulation. Too large a quantum might introduce timing inconsistencies across the system, possibly to the point where the system ceases to function.

For example, consider the simulation of a system consisting of a processor, a memory, a timer, and some slow external peripherals. The software running on the processor spends most of its time fetching and executing instructions from system memory, and only interacts with the rest of the system when it is interrupted by the timer, say every 1ms. The ISS that models the processor could be permitted to run ahead of SystemC simulation time with a quantum of up to 1ms, making direct accesses to the memory model, but only synchronizing with the peripheral models at the rate of timer interrupts. The point here is that the ISS does not have to be locked to the simulation time clock of the hardware part of the system, as would be the case with more traditional hardware-software co-simulation. Depending on the detail of the models, temporal decoupling alone could give a simulation speed improvement of approximately 10X, or 100X when combined with DMI.

It is quite possible for some processes to be temporally decoupled and others not, and also for different processes to use different values for the time quantum. However, any process that is not temporally decoupled is likely to become a simulation speed bottleneck.

In TLM-2.0, temporal decoupling is supported by the **tlm_global_quantum** class and the timing annotation of the blocking and non-blocking transport interface. The utility class **tlm_quantumkeeper** provides a convenient way to access the global quantum.

3.3.3 Synchronization in loosely-timed models

An untimed model relies on the presence of explicit synchronization points (that is calls to **wait** or blocking method calls) in order to pass control between initiators at predetermined points during execution. A loosely-timed model can also benefit from explicit synchronization in order to guarantee deterministic execution, but a loosely-timed model is able to make progress even in the absence of explicit synchronization points (calls to **wait**), because each initiator will only run ahead as far as the end of the time quantum before yielding control. A loosely-timed model can increase its timing accuracy by using synchronization-on-demand, that is, yielding control to the scheduler before reaching the end of the time quantum.

The time quantum mechanism is not intended to ensure correct system synchronization, but rather is a simulation mechanism that allows multiple system initiators to make progress in a scheduler-based

simulation environment. The time quantum mechanism is not an alternative to designing an explicit synchronization scheme at the system level.

3.3.4　Approximately-timed coding style

The approximately-timed coding style is supported by the non-blocking transport interface, which is appropriate for the use cases of architectural exploration and performance analysis. The non-blocking transport interface provides for timing annotation and for multiple phases and timing points during the lifetime of a transaction.

For approximately-timed modeling, a transaction is broken down into multiple phases, with an explicit timing point marking the transition between phases. In the case of the base protocol there are exactly four timing points marking the beginning and the end of the request and the beginning and the end of the response. Specific protocols may need to add further timing points, which may possibly cause the loss of direct compatibility with the generic payload.

Although it is possible to use the non-blocking transport interface with just two phases to indicate the start and end of a transaction, the blocking transport interface is generally preferred for loosely-timed modeling.

The approximately-timed coding style cannot generally exploit temporal decoupling because of the need for timing accuracy. Instead, each process typically executes in lock step with the SystemC scheduler. Process interactions are annotated with specific delays. To create an approximately-timed model, it is generally sufficient to annotate delays representing the data transfer times for write and read commands and the latency of the target. For the base protocol, the data transfer times are effectively same as the minimum initiation interval or accept delay between two successive requests or two successive responses. The annotated delays are implemented by making calls to the SystemC scheduler, that is, **wait**(delay) or **notify**(delay).

3.3.5　Characterization of loosely-timed and approximately-timed coding styles

The coding styles can be characterized in terms of timing points and temporal decoupling.

Loosely-timed. Each transaction has just two timing point, marking the start and the end of the transaction. Simulation time is used, but processes may be temporally decoupled from simulation time. Each process keeps a tally of how far it has run ahead of simulation time, and may yield because it reaches an explicit synchronization point or because it has consumed its time quantum.

Approximately-timed. Each transaction has multiple timing points. Processes typically need to run in lock-step with SystemC simulation time. Delays annotated onto process interactions are implemented using timeouts (wait) or timed event notifications.

Untimed. The notion of simulation time is unnecessary. Processes yield at explicit pre-determined synchronization points.

3.3.6　Switching between loosely-timed and approximately-timed modeling

A model may switch between the loosely-timed and approximately-timed coding style during simulation. The idea is to run rapidly through the reset and boot sequence at the loosely-timed level, then switch to approximately timed modeling for more detailed analysis once the simulation has reached an interesting stage.

3.3.7   Cycle-accurate modeling

Cycle-accurate modeling is beyond the scope of TLM-2.0 at present. It is possible to create cycle-accurate models using SystemC and TLM-1 as it stands, but the requirement for the standardization of a cycle-accurate coding style still remains an open issue, possibly to be addressed by a future OSCI standard.

In principle only, the approximately-timed coding style might be extended to encompass cycle-accurate modeling by defining an appropriate set of phases and rules. The TLM-2.0 release includes sufficient machinery for this, but the details have not been worked out.

3.3.8   Blocking versus non-blocking transport interfaces

The blocking and non-blocking transport interfaces are distinct interfaces that exist in TLM-2.0 to support different levels of timing detail. The blocking transport interface is only able to model the start and end of a transaction, with the transaction being completed within a single function call. The non-blocking transport interface allows a transaction to be broken down into multiple timing points, and typically requires multiple function calls for a single transaction.

For interoperability, the blocking and non-blocking transport interfaces are combined into a single interface. A model that initiates transactions may use the blocking or non-blocking transport interfaces (or both) according to coding style. Any model that provides a TLM-2.0 transport interface is obliged to provide both the blocking and non-blocking forms for maximal interoperability, although such a model is not obliged to implement both interfaces internally.

TLM-2.0 provides a mechanism (the so-called *convenience socket*) to automatically convert incoming blocking or non-blocking transport calls to non-blocking or blocking transport calls, respectively. Converting transport call types does incur some cost, particularly converting an incoming non-blocking call to a blocking implementation. However, the cost overhead is mitigated by the fact that the presence of an approximately-timed model is likely to have a negative impact on simulation speed anyway.

The C++ static typing rules enforce the implementation of both interfaces, but an initiator can choose dynamically whether to call the blocking or the non-blocking transport method. It is possible for different initiators to call different methods, or for a given initiator to switch between blocking and non-blocking calls on-the-fly. This standard includes rules governing the mixing and ordering of blocking and non-blocking transport calls to the same target.

The strength of the blocking transport interface is that it allows a simplified coding style for models that are able to complete a transaction in a single function call. The strength of the non-blocking transport interface is that it supports the association of multiple timing points with a single transaction. In principle, either interface supports temporal decoupling, but the speed benefits of temporal decoupling are likely to be nullified by the presence of multiple timing points for approximately-timed models.

3.3.9   Use cases and coding styles

The table below summarizes the mapping between use cases for transaction-level modeling and coding styles:

Use Case	Coding style
Software application development	Loosely-timed
Software performance analysis	Loosely-timed
Hardware architectural analysis	Loosely-timed or approximately-timed
Hardware performance verification	Approximately-timed or cycle-accurate
Hardware functional verification	Untimed (verification environment), loosely-timed or approximately-timed

3.4   Initiators, targets, sockets, and transaction bridges

The TLM-2.0 core interfaces pass transactions between initiators and targets. An initiator is a module that can initiate transactions, that is, create new transaction objects and pass them on by calling a method of one of the core interfaces. A target is a module that acts as the final destination for a transaction. In the case of a write

Figure 3
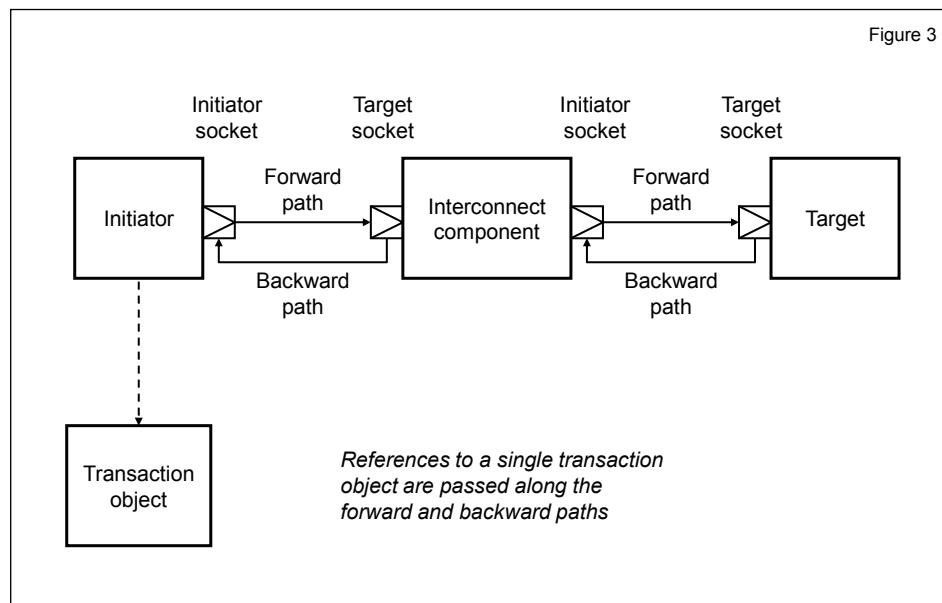
transaction, an initiator (such as a processor) writes data to a target (such as a memory). In the case of a read transaction, an initiator reads data from a target. An interconnect component is a module that accesses a

transaction but does not act as an initiator or a target for that transaction, typical examples being arbiters and routers. The roles of initiator, interconnect and target can change dynamically. For example, a given component may act as an interconnect for some transactions but as a target for other transactions.

In order to illustrate the idea, this paragraph will describe the lifetime of a typical transaction object. The transaction object is created by an initiator and passed as an argument of a method of the transport interface (blocking or non-blocking). That method is implemented by an interconnect component such as an arbiter, which may read attributes of the transaction object before passing it on to a further transport call. That second transport method is implemented by a second interconnect component, such as a router, which in turn passes on the transaction through a third transport call to a target such as a memory, the final destination for the transaction object. (The actual number of interconnect components will vary from transaction to transaction. There may be none.) This sequence of method calls is known as the *forward path*. The transaction is executed in the target, and the transaction object may be returned to the initiator in one of two ways, either carried with the return from the transport method calls as they unwind, known as the *return path*, or passed by making explicit transport method calls on the opposite path from target back to initiator, known as the *backward path*. This choice is determined by the return value from the non-blocking transport method. (Strictly speaking there are two return paths corresponding to the forward and backward paths, but the meaning is usually clear from the context.)

Figure 4
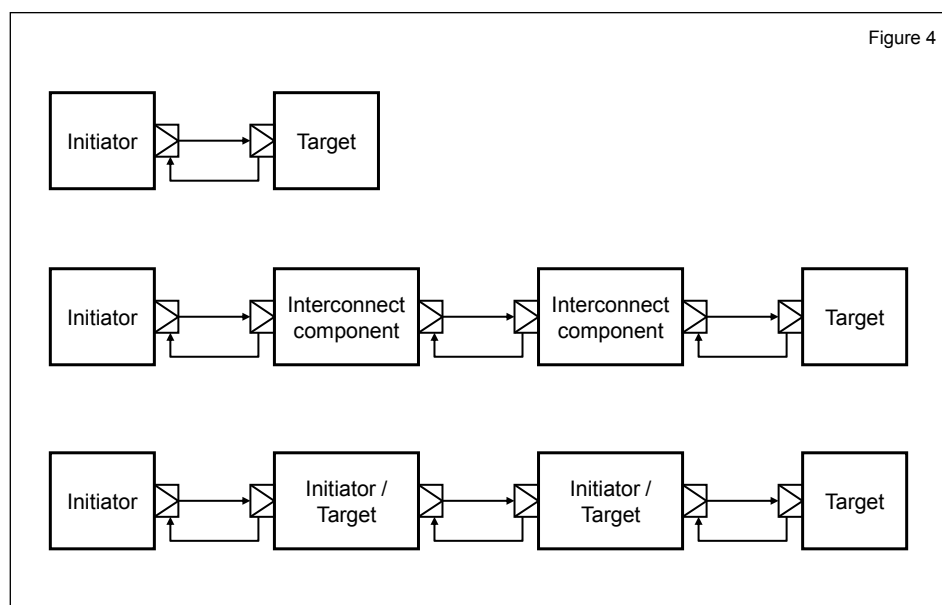
The forward path is the calling path by which an initiator or interconnect component makes interface method calls forward in the direction of another interconnect component or the target. The backward path is the calling path by which a target or interconnect component makes interface method calls back in the direction of another interconnect component or the initiator. The entire path between an initiator and a target consists of a number of *hops*, each hop connecting two adjacent components. The number of hops from initiator to target is one greater than the number of interconnect components on that path. When using the generic payload, the forward and backward paths should always pass through the same set of components and sockets, obviously in reverse order.

In order to support both forward and backward paths, each connection between components requires a port and an export, both of which have to be bound. This is facilitated by the *initiator socket* and the *target socket*. An initiator socket provides a port for interface method calls on the forward path and an export for interface method calls on the backward path. A target socket provides the opposite. (More specifically, an initiator socket is derived from class **sc_port** and *has* an **sc_export**, and vice versa for a target socket.) The initiator and target socket classes overload the SystemC port binding operator to implicitly bind both forward and backward paths.

As well as the transport interfaces, the sockets also encapsulate the DMI and debug transport interfaces (see below).

When using sockets, an initiator component will have at least one initiator socket, a target component at least one target socket, and an interconnect component at least one of each. A component may have several sockets transporting different transaction types, in which case a single component may act as initiator or target for multiple independent transactions.

In order to model a bus bridge there are two alternatives. Either model the bus bridge as an interconnect component, or model the bus bridge as a transaction bridge between two separate TLM-2.0 transactions. An interconnect component would pass on a pointer to a single transaction object, which is the best approach for simulation speed. A transaction bridge would require the transaction object to be copied, which gives much more flexibility because the two transactions could have different attributes.

The use of TLM-2.0 sockets is recommended for maximal interoperability, convenience, and a consistent coding style. Whilst it is possible for components to use SystemC ports and exports directly with the TLM-2.0 core interfaces, this is not recommended.

3.5   DMI and debug transport interfaces

The direct memory interface (DMI) and debug transport interface are specialized interfaces distinct from the transport interface, providing direct access and debug access to an area of memory owned by a target. Once a DMI request has been granted, the DMI interface enables an initiator to bypass the usual path through the interconnect components used by the transport interface. DMI is intended to accelerate regular memory transactions in a loosely-timed simulation, whereas the debug transport interface is for debug access free of the delays or side-effects associated with regular transactions.

The DMI has both forward (initiator-to-target) and backward (target-to-initiator) interfaces, whereas debug only has a forward interface. See 4.2 Direct memory interface and 4.3 Debug transport interface

3.6   Combined interfaces and sockets

The blocking and non-blocking transport interfaces are combined with the DMI and debug transport interfaces in the standard initiator and target sockets. All four interfaces (the two transport interfaces, DMI, and debug) can be used in parallel to access a given target (subject to the rules described in this standard). These combined interfaces are one of the keys to ensuring interoperability between components using the TLM-2.0 standard, the other key being the generic payload. See 6.1 Combined interfaces

The standard target sockets provide all four interfaces, so each target component must effectively implement the methods of all four interfaces. However, the design of the blocking and non-blocking transport interfaces together with the provision of convenience sockets to convert between the two means that a given target need

only implement either the blocking or the non-blocking transport method, not both, according to the speed and accuracy requirements of the model.

A given initiator may choose to call methods through any or all of the core interfaces, again according to the speed and accuracy requirements. The coding styles mentioned above help guide the choice of an appropriate set of interface features. Typically, a loosely-timed initiator will call blocking transport, DMI and debug, whereas an approximately-timed initiator will call non-blocking transport and debug.

3.7 Namespaces

The TLM-2.0 classes shall be declared in a two top-level C++ namespaces, **tlm** and **tlm_utils**. Particular implementations of the TLM-2.0 classes may choose to nest further namespaces within these two namespaces, but such nested namespaces shall not be used in applications.

Namespace **tlm** contains the classes that comprise the interoperability interface for memory-mapped bus modeling.

Namespace **tlm_utils** contains utility classes that are not strictly necessary for interoperability at the interface between memory-mapped bus models, but which are nevertheless a proper part of the TLM-2.0 standard.

3.8 Header files and version numbers

Applications should #include the header file **tlm.h** from the **include/tlm** directory of the software distribution. Applications should also #include any header files they may require from the **include/tlm/tlm_utils** directory.

Applications compiling the simple sockets with current released versions of the OSCI proof-of-concept simulator should define the macro SC_INCLUDE_DYNAMIC_PROCESSES before including the SystemC header file.

3.8.1 Software version information

The header file **include/tlm/tlm_h/tlm_version.h** shall contain a set of macros, constants, and functions that provide information concerning the version number of the OSCI TLM-2.0 software distribution. Applications may use these macros and constants.

3.8.1.1 Definitions

```
namespace tlm
{

#define  TLM_VERSION_MAJOR          implementation_defined_number // For example, 2
#define  TLM_VERSION_MINOR          implementation_defined_number // 0
#define  TLM_VERSION_PATCH          implementation_defined_number // 1
#define  TLM_VERSION_ORIGINATOR     implementation_defined_string   // "OSCI"
#define  TLM_VERSION_RELEASE_DATE   implementation_defined_date     // "20090329"
#define  TLM_VERSION_PRERELEASE     implementation_defined_string   // "beta"
#define  TLM_IS_PRERELEASE          implementation_defined_bool     // TRUE
#define  TLM_VERSION                implementation_defined_string   // "2.0.1_beta-OSCI"
```

```
#define  TLM_COPYRIGHT                implementation_defined_string

const unsigned int  tlm_version_major;
const unsigned int  tlm_version_minor;
const unsigned int  tlm_version_patch;
const std::string       tlm_version_originator;
const std::string       tlm_version_release_date;
const std::string       tlm_version_prerelease;
const bool              tlm_is_prerelease;
const std::string       tlm_version_string;
const std::string       tlm_copyright_string;

inline  const char*   tlm_release();
inline  const char*   tlm_version();
inline  const char*   tlm_copyright();

} // namespace tlm
```

3.8.1.2   Rules

a) Each *implementation_defined_number* shall consist of a sequence of decimal digits from the character set [0-9] not enclosed in quotation marks.

b) The originator and pre-release strings shall each consist of a sequence of characters from the character set [A-Z][a-z][0-9]_ enclosed in quotation marks.

c) The version release date shall consist of an ISO 8601 basic format calendar date of the form YYYYMMDD, where each of the 8 characters is a decimal digit, enclosed in quotation marks.

d) The IS_PRERELEASE flag shall be either FALSE or TRUE, not enclosed in quotation marks.

e) The version string shall be set to the value "major.minor.patch_prerelease-originator" or "major.minor.patch-originator", where major, minor, patch, prerelease and originator are the values of the corresponding strings (without enclosing quotation marks), and the presence or absence of the prerelease string shall depend on the value of the IS_PRERELEASE flag.

f) The copyright string should be set to a copyright notice.

g) Each constant shall be initialized with the value defined by the macro of the corresponding name converted to the appropriate data type.

h) The methods **tlm_release** and **tlm_version** shall each return the value of the version string converted to a C string.

i) The method **tlm_copyright** shall return the value of the copyright string converted to a C string.

4   TLM-2.0 Core Interfaces

In addition to the core interfaces from TLM-1, TLM-2.0 adds blocking and non-blocking transport interfaces, a direct memory interface (DMI), and a debug transport interface.

4.1   Transport interfaces

The transport interfaces are the primary interfaces used to transport transactions between initiators, targets and interconnect components. Both the blocking and non-blocking transport interfaces support timing annotation and temporal decoupling, but only non-blocking transport supports multiple phases within the lifetime of a transaction. Blocking transport does not have an explicit phase argument, and any association between blocking transport and the phases of the non-blocking transport interface is purely notional. Only the non-blocking transport method returns a value indicating whether or not the return path was used.

The transport interfaces and the generic payload were designed to be used together for the fast, abstract modeling of memory-mapped buses. The transport interface templates are specialized with the transaction type allowing them to be used separately from the generic payload, although many of the interoperability benefits would be lost.

The rules governing memory management of the transaction object, transaction ordering, and the permitted function calling sequence depend on the specific transaction type passed as a template argument to the transport interface, which in turn depends on the protocol traits class passed as a template argument to the socket (if a socket is used).

4.1.1   Blocking transport interface

4.1.1.1   Introduction

The TLM-2.0 blocking transport interface is intended to support the loosely-timed coding style. The blocking transport interface is appropriate where an initiator wishes to complete a transaction with a target during the course of a single function call, the only timing points of interest being those that mark the start and the end of the transaction.

The blocking transport interface only uses the forward path from initiator to target.

The TLM-2.0 blocking transport interface has deliberate similarities with the transport interface from TLM-1, which is still part of the TLM-2.0 standard, but the TLM-1 transport interface and the TLM-2.0 blocking transport interface are not identical. In particular, the new **b_transport** method has a single transaction argument passed by non-const reference and a second argument to annotate timing, whereas the TLM-1 **transport** method takes a request as a single const reference request argument, has no timing annotation, and returns a response by value. TLM-1 assumes separate request and response objects passed by value (or const reference), whereas TLM-2.0 assumes a single transaction object passed by reference, whether using the blocking or the non-blocking TLM-2.0 interfaces.

The **b_transport** method has a timing annotation argument. This single argument is used on both the call to and the return from **b_transport** to indicate the time of the start and end of the transaction, respectively, relative to the current simulation time.

4.1.1.2   Class definition

```
namespace tlm {

template <typename TRANS = tlm_generic_payload>
class tlm_blocking_transport_if : public virtual sc_core::sc_interface {
public:
  virtual void b_transport(TRANS& trans, sc_core::sc_time& t) = 0;
};

} // namespace tlm
```

4.1.1.3   The TRANS template argument

The intent is that this core interface may be used to transport transactions of any type. A specific transaction type, **tlm_generic_payload**, is provided to ease interoperability between models where the precise details of the transaction attributes are less important.

For maximum interoperability, applications should use the default transaction type **tlm_generic_payload** with the base protocol. See 8.2 Base protocol. In order to model specific protocols, applications may substitute their own transaction type. Sockets that use interfaces specialized with different transaction types cannot be bound together, providing compile-time checking but restricting interoperability.

4.1.1.4   Rules

a)   The **b_transport** method may call **wait**, directly or indirectly.

b)   The **b_transport** method shall not be called from a method process.

c)   The initiator may re-use a transaction object from one call to the next and across calls to the transport interfaces, DMI, and the debug transport interface

d)   The call to **b_transport** marks the first timing point of the transaction. The return from **b_transport** marks the final timing point of the transaction.

e)   The timing annotation argument allows the timing points to be offset from the simulation times (value returned by **sc_time_stamp()**) at which the function call and return are executed.

f)   The callee may modify or update the transaction object, subject to any constraints imposed by the transaction class **TRANS**.

g)   It is recommended that the transaction object should not contain timing information. Timing should be annotated using the **sc_time** argument to **b_transport**.

h)   Typically, an interconnect component should pass the **b_transport** call along the forward path from initiator to target. In other words, the implementation of **b_transport** for the target socket of the interconnect component may call the **b_transport** method of an initiator socket.

i)   Whether or not the implementation of **b_transport** is permitted to call **nb_transport_fw** depends on the rules associated with the protocol. For the base protocol, the convenience socket **simple_target_socket** is able to make this conversion automatically. See 9.1.2 Simple sockets.

j)   The implementation of **b_transport** shall not call **nb_transport_bw**.

4.1.1.5   Message sequence chart – blocking transport

The blocking transport method may return immediately (that is, in the current SystemC evaluation phase) or may yield control to the scheduler and only return to the initiator at a later point in simulation time. Although the initiator thread may be blocked, another thread in the initiator may be permitted to call **b_transport** before the first call has returned, depending on the protocol.

Blocking Transport

Figure 5

Copyright © 2007-2009 by the Open SystemC Initiative (OSCI)
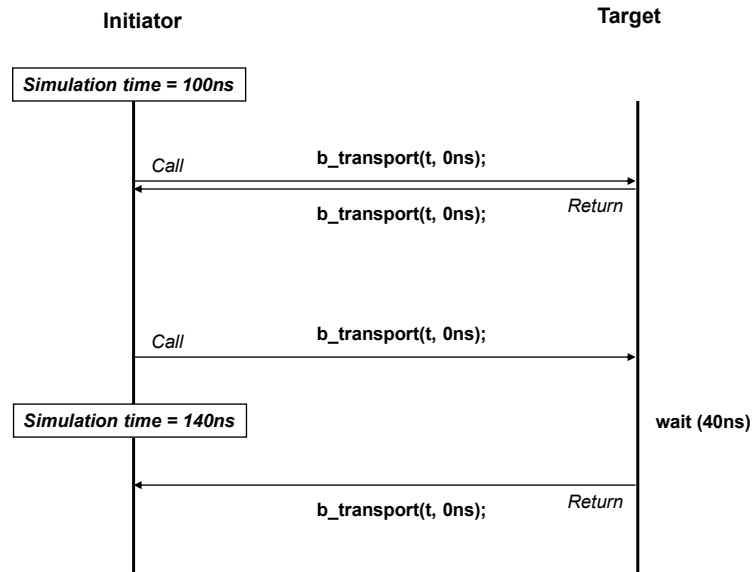
4.1.1.6   Message sequence chart – temporal decoupling

A temporally decoupled initiator may run at a notional local time in advance of the current simulation time, in which case it should pass a non-zero value for the time argument to **b_transport**, as shown below. The initiator and target may each further advance the local time offset by increasing the value of the time argument. Adding the time argument returned from the call to the current simulation time gives the notional time at which each the transaction completes, but simulation time itself cannot advance until the initiator thread yields.

The body of **b_transport** may itself call **wait**, in which case the local time offset should be reset to zero. In the diagram below, the final return from the initiator happens at simulation time 140ns, but with an annotated delay of 5ns, giving an effective local time of 145ns.

Temporal decoupling

Figure 6

| | Initiator | | Target |

| Simulation time = 100ns |
| Local time offset |
+0ns	Call	**b_transport(t, 0ns);**
+5ns		**b_transport(t, 5ns);**	Return
+20ns	Call	**b_transport(t, 20ns);**
+25ns		**b_transport(t, 25ns);**	Return
	Call	**b_transport(t, 30ns);**
+30ns			
Simulation time = 140ns			wait (40ns)
+5ns		**b_transport(t, 5ns);**	Return
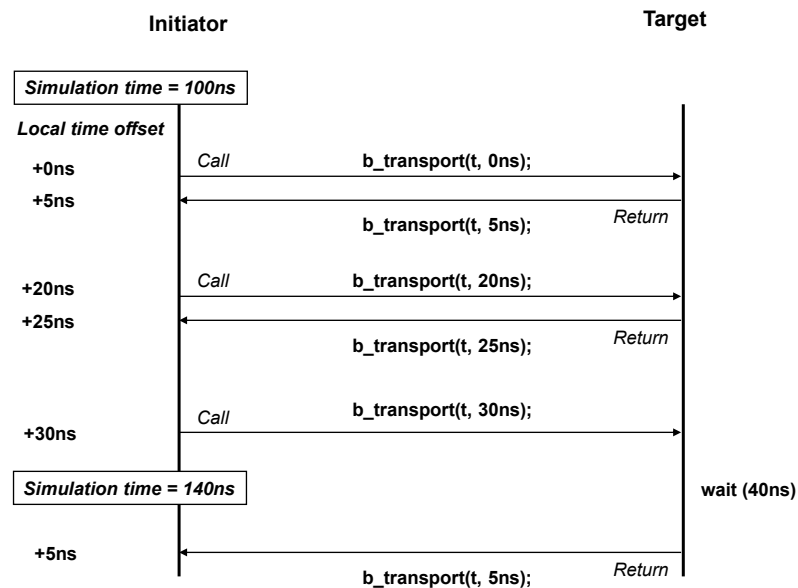
4.1.1.7   Message sequence chart – the time quantum

A temporally decoupled initiator will continue to advance local time until the time quantum is exceeded. At that point, the initiator is obliged to synchronize by suspending execution, directly or indirectly calling the **wait** method with the local time as an argument. This allows other initiators in the model to run and to catch up, which effectively means that the initiators execute in turn, each being responsible for determining when to hand back control by keeping track of its own local time. The original initiator should only run again after simulation time has advanced to the next quantum.

The primary purpose of delays in the loosely-timed coding style is to allow each initiator to determine when to hand back control. It is best if the model does not rely on the details of the timing in order to function correctly.

Within each quantum, the transactions generated by a given initiator happen in strict sequential order, but without advancing simulation time. The local time is not tracked by the SystemC scheduler.

The time quantum

Figure 7
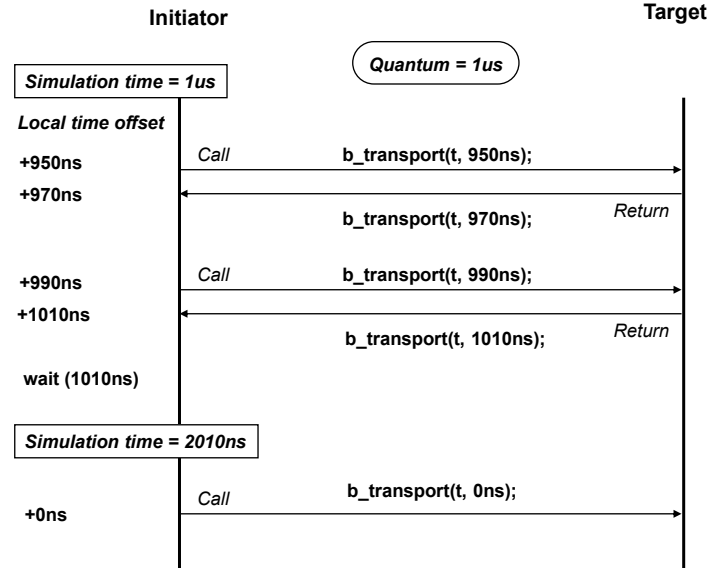
4.1.2 Non-blocking transport interface

4.1.2.1 Introduction

The non-blocking transport interface is intended to support the approximately-timed coding style. The non-blocking transport interface is appropriate where it is desired to model the detailed sequence of interactions between initiator and target during the course of each transaction. In other words, to break down a transaction into multiple phases, where each phase transition is associated with a timing point. Each call to and return from the non-blocking transport method may correspond to a phase transition.

By restricting the number of timing points to two, it is possible to use the non-blocking transport interface with the loosely-timed coding style, but this is not generally recommended. For loosely-timed modeling, the blocking transport interface is generally preferred for its simplicity. The non-blocking transport interface is particularly suited for modeling pipelined transactions, which would be awkward to model using blocking transport.

The non-blocking transport interface uses both the forward path from initiator to target and the backward path from target to initiator. There are two distinct interfaces, **tlm_fw_nonblocking_transport_if** and **tlm_bw_nonblocking_transport_if**, for use on opposite paths.

The non-blocking transport interface uses a similar argument-passing mechanism to the blocking transport interface in that the non-blocking transport methods pass a non-const reference to the transaction object and a timing annotation, but there the similarity ends. The non-blocking transport method also passes a phase to indicate the state of the transaction, and returns an enumeration value to indicate whether the return from the function also represents a phase transition.

Both blocking and non-blocking transport support timing annotation, but only non-blocking transport supports multiple phases within the lifetime of a transaction. The blocking and non-blocking transport interface and the generic payload were designed to be used together for the fast, abstract modeling of memory-mapped buses. However, the transport interfaces can be used separately from the generic payload to model specific protocols. Both the transaction type and the phase type are template parameters of the non-blocking transport interface.

4.1.2.2 Class definition

```
namespace tlm {

enum tlm_sync_enum { TLM_ACCEPTED, TLM_UPDATED, TLM_COMPLETED };

template <typename TRANS = tlm_generic_payload, typename PHASE = tlm_phase>
class tlm_fw_nonblocking_transport_if : public virtual sc_core::sc_interface {
public:
  virtual tlm_sync_enum nb_transport_fw(TRANS& trans, PHASE& phase, sc_core::sc_time& t) = 0;
};

template <typename TRANS = tlm_generic_payload, typename PHASE = tlm_phase>
class tlm_bw_nonblocking_transport_if : public virtual sc_core::sc_interface {
public:
  virtual tlm_sync_enum nb_transport_bw(TRANS& trans, PHASE& phase, sc_core::sc_time& t) = 0;
```

```
    };

    } // namespace tlm
```

4.1.2.3   The TRANS and PHASE template arguments

The intent is that the non-blocking transport interface may be used to transport transactions of any type and with any number of phases and timing points. A specific transaction type, **tlm_generic_payload**, is provided to ease interoperability between models where the precise details of the transaction attributes are less important, and a specific type **tlm_phase** is provided for use with the base protocol. See 8.2 Base protocol

For maximum interoperability, applications should use the default transaction type **tlm_generic_payload** and the default phase type **tlm_phase** with the base protocol. In order to model specific protocols, applications may substitute their own transaction type and phase type. Sockets that use interfaces specialized with different transaction types cannot be bound together, providing compile-time checking but restricting interoperability.

4.1.2.4   The nb_transport_fw and nb_transport_bw calls

a)   There are two non-blocking transport methods, **nb_transport_fw** for use on the forward path, and **nb_transport_bw** for use on the backward path. Aside from their names and calling direction these two methods have similar semantics. In this document, the italicized term *nb_transport* is used to describe both methods in situations where there is no need to distinguish between them.

b)   In the case of the base protocol, the forward and backward paths should pass through exactly the same sequence of components and sockets in opposing order. It is the responsibility of each component to route any transaction returning toward the initiator using the target socket through which that transaction was first received.

c)   **nb_transport_fw** shall only be called on the forward path, and **nb_transport_bw** shall only be called on the backward path.

d)   An **nb_transport_fw** call on the forward path shall under no circumstances directly or indirectly make a call to **nb_transport_bw** on the backward path, and vice versa.

e)   The *nb_transport* methods shall not call **wait**, directly or indirectly.

f)   The *nb_transport* methods may be called from a thread process or from a method process.

g)   *nb_transport* is not permitted to call **b_transport**. One solution would be to call **b_transport** from a separate thread process, spawned or notified by the original **nb_transport_fw** method. For the base protocol, a convenience socket **simple_target_socket** is provided, which is able to make this conversion automatically. See 9.1.2 Simple sockets.

h)   The non-blocking transport interface is explicitly intended to support pipelined transactions. In other words, several successive calls to **nb_transport_fw** from the same process could each initiate separate transactions without having to wait for the first transaction to complete.

i)   In principle, the final timing point of a transaction may be marked by a call to or a return from *nb_transport* either on the forward path or the backward path.

4.1.2.5   The trans argument

a)   The lifetime of a given transaction object may extend beyond the return from *nb_transport* such that a series of calls to *nb_transport* may pass a single transaction object forward and backward between initiators, interconnect components, and targets.

b)   If there are multiple calls to *nb_transport* associated with a given transaction instance, one and the same transaction object shall be passed as an argument to every such call. In other words, a given transaction instance shall be represented by a single transaction object.

c)   An initiator may re-use a given transaction object to represent more than one transaction instance, or across calls to the transport interfaces, DMI, and the debug transport interface.

d)   Since the lifetime of the transaction object may extend over several calls to *nb_transport*, either the caller or the callee may modify or update the transaction object, subject to any constraints imposed by the transaction class **TRANS**. For example, for the generic payload, the target may update the data array of the transaction object in the case of a read command, but shall not update the command field. See 7.7 Default values and modifiability of attributes

4.1.2.6   The phase argument

a)   Each call to *nb_transport* passes a reference to a phase object. In the case of the base protocol, successive calls to *nb_transport* with the same phase are not permitted. Each phase transition has an associated timing point. A timing annotation using the **sc_time** argument shall delay the timing point relative to the phase transition.

b)   The phase argument is passed by reference. Either caller or callee may modify the phase.

c)   The intent is that the phase argument should be used to inform components as to whether and when they are permitted to read or modify the attributes of a transaction. If the rules of a protocol allow a given component to modify the value of a transaction attribute during a particular phase, then that component may modify the value at any time during that phase and any number of times during that phase. The protocol should forbid other components from reading the value of that attribute during that phase, only permitting the value to be read after the next phase transition.

d)   The value of the phase argument represents the current state of the protocol state machine for the given hop. Where a single transaction object is passed between more than two components (initiator, interconnect, target), each hop requires (notionally, at least) a separate protocol state machine.

e)   Whereas the transaction object has a lifetime and a scope that may extend beyond any single call to *nb_transport*, the phase object is normally local to the caller. Each *nb_transport* call for a given transaction may have a separate phase object. Corresponding phase transitions on different hops may occur at different points in simulation time.

f)   The default phase type **tlm_phase** is specific to the base protocol. Other protocols may use or extend type **tlm_phase** or may substitute their own phase type (with a corresponding loss of interoperability). See 8.1 Phases.

4.1.2.7   The tlm_sync_enum return value

a) The concept of sychronization is referred to in several places. To *synchronize* is to yield control to the SystemC scheduler in order that other processes may run, but has additional connotations for temporal decoupling. This is discussed more fully elsewhere. See 9.2.4 General guidelines for processes using temporal decoupling.

b) In principle, synchronization can be accomplished by yielding (calling **wait** in the case of a thread process or returning to the kernel in the case of a method process), but a temporally decoupled initiator should synchronize by calling the **sync** method of class **tlm_quantum_keeper**. In general, it is necessary for an initiator to synchronize from time-to-time in order to allow other SystemC processes to run.

c) The following rules apply to both the forward and backward paths.

d) The meaning of the return value from *nb_transport* is fixed, and does not vary according to the transaction type or phase type. Hence the following rules are not restricted to the base protocol but apply to every transaction and phase type used to parameterize the non-blocking transport interface.

e) **TLM_ACCEPTED**. The callee shall not have modified the state of the transaction object, the phase, or the time argument during the call. In other words, TLM_ACCEPTED indicates that the return path is not being used. The caller may ignore the values of the *nb_transport* arguments following the call, since the callee is obliged to leave them unchanged. In general, the caller will have to yield before the component containing the callee can respond to the transaction. For the base protocol, a callee that is ignoring an ignorable phase should return TLM_ACCEPTED.

f) **TLM_UPDATED**. The callee has updated the transaction object. The callee may have modified the state of the phase argument, may have modified the state of the transaction object, and may have increased the value of the time argument during the call. In other words, TLM_UPDATED indicates that the return path is being used, and the callee has advanced the state of the protocol state machine associated with the transaction. Whether or not the callee is actually obliged to modify each of the arguments depends on the protocol. Following the call to *nb_transport*, the caller should inspect the phase, transaction and time arguments and take the appropriate action.

g) **TLM_COMPLETED**. The callee has updated the transaction object, and the transaction is complete. The callee may have modified the state of the transaction object, and may have increased the value of the time argument during the call. The value of the phase argument is undefined. In other words, TLM_COMPLETED indicates that the return path is being used and the transaction is complete with respect to a particular socket. Following the call to *nb_transport*, the caller should inspect the transaction object and take the appropriate action, but should ignore the phase argument. There shall be no further transport calls associated with this particular transaction through the current socket along either the forward or backward paths. Completion in this sense does not necessarily imply successful completion, so depending on the transaction type, the caller may need to inspect a response status embedded in the transaction object.

h) In general there is no obligation to complete a transaction by having *nb_transport* return TLM_COMPLETED. A transaction is in any case complete with respect to a particular socket when the final phase of the protocol is passed as an argument to *nb_transport*. (For the base protocol, the final phase is END_RESP.) In other words, TLM_COMPLETED is not mandatory.

i) For any of the three return values, and depending on the protocol, following the call to *nb_transport* the caller may need to yield in order to allow the component containing the callee to generate a response or to release the transaction object.

4.1.2.8   tlm_sync_enum summary

tlm_sync_enum	Transaction object	Phase on return	Timing annotation on return
TLM_ACCEPTED	Unmodified	Unchanged	Unchanged
TLM_UPDATED	Updated	Changed	May be increased
TLM_COMPLETED	Updated	Ignored	May be increased

4.1.2.9　Message sequence chart – using the backward path

The following message sequence charts illustrate various calling sequences to *nb_transport*. The arguments and return value passed to and from *nb_transport* are shown using the notation *return, phase, delay*, where return is the value returned from the function call, phase is the value of the phase argument, and delay is the value of the **sc_time** argument The notation '-' indicates that the value is unused.

The following message sequence charts use the phases of the base protocol as an example, that is, BEGIN_REQ, END_REQ and so on. With the approximately-timed coding style and the base protocol, a transaction is passed back-and-forth twice between initiator and target. For other protocols, the number of phases and their names may be different.

If the recipient of an *nb_transport* call is unable immediately to calculate the next state of the transaction or the delay to the next timing point, it should return a value of TLM_ACCEPTED. The caller should yield control to the scheduler and expect to receive a call to *nb_transport* on the opposite path when the callee is ready to respond. Notice that in this case, unlike the loosely-timed case, the caller could be the initiator or the target.

Transactions may be pipelined. The initiator could call *nb_transport* to send another transaction to the target before having seen the final phase transition of the previous transaction.

Because processes are regularly yielding control to the scheduler in order to allow simulation time to advance, the approximately-timed coding style is expected to simulate a lot more slowly than the loosely-timed coding style.

Figure 8
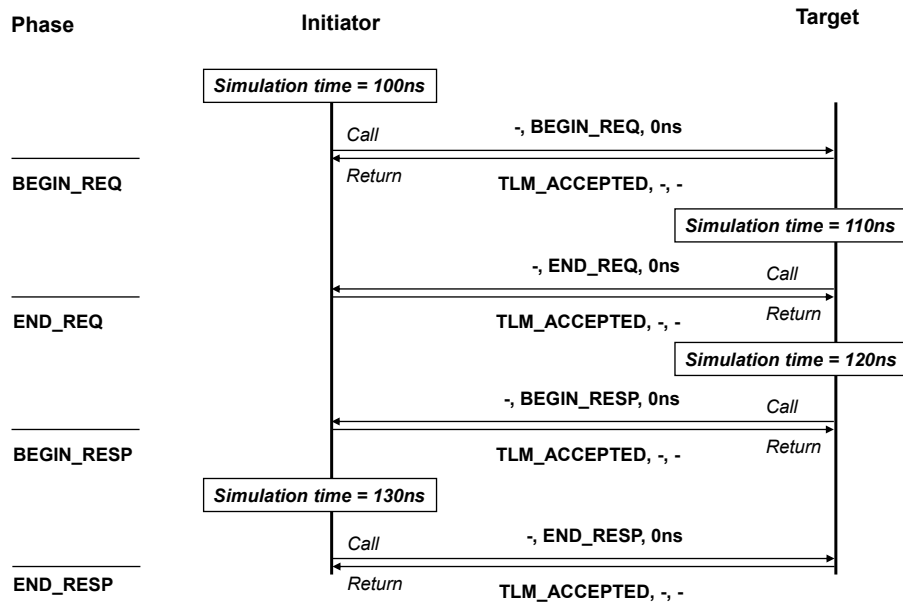
4.1.2.10 Message sequence chart – using the return path

If the recipient of an *nb_transport* call can immediately calculate the next state of the transaction and the delay to the next timing point, it may return the new state on return from *nb_transport* rather than using the opposite path. If the next timing point marks the end of the transaction, the recipient can return either TLM_UPDATED or TLM_COMPLETED. A callee can return TLM_COMPLETED at any stage (subject to the rules of the protocol) to indicate to the caller that it has pre-empted the other phases and jumped to the final phase, completing the transaction. This applies to initiator and target alike.

With TLM_UPDATED, the callee should update the transaction, the phase, and the timing annotation.

In the diagram below, the non-zero timing annotation argument passed on return from the function calls indicates to the caller the delay between the phase transition on the hop and the corresponding timing point.

Figure 9

Using the return path

Phase	Initiator	Target

Simulation time = 100ns

Call — -, BEGIN_REQ, 0ns

BEGIN_REQ   Return — TLM_UPDATED, END_REQ, 10ns

END_REQ   *Simulation time = 110ns*

Simulation time = 150ns

-, BEGIN_RESP, 0ns   Call

BEGIN_RESP   TLM_UPDATED, END_RESP, 5ns   Return

END_RESP   *Simulation time = 155ns*
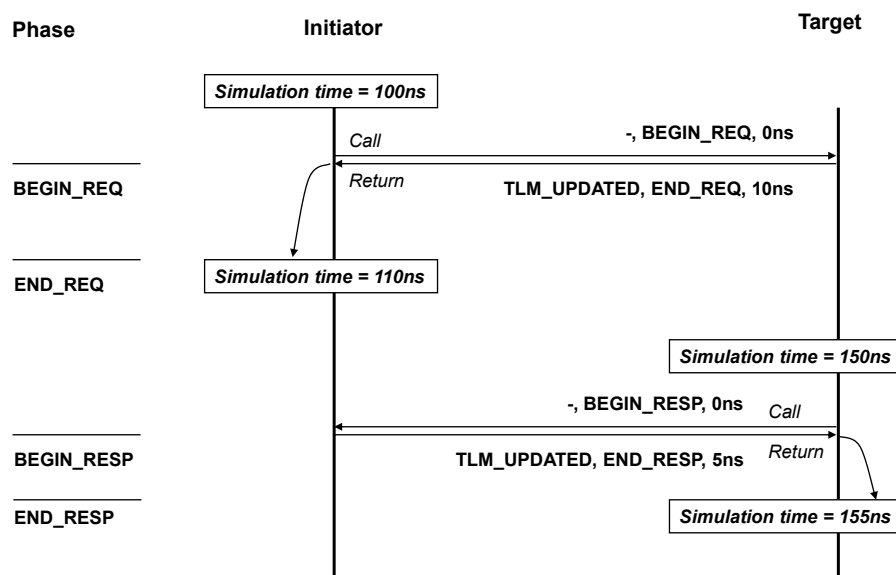
4.1.2.11 Message sequence chart – early completion

Depending on the protocol, an initiator or a target may return TLM_COMPLETED from *nb_transport* at any point in order to complete the transaction early. Neither initiator nor target may make any further *nb_transport* calls for this particular transaction instance. Whether or not an initiator or target is actually permitted to shortcut a transaction in this way depends on the rules of the specific protocol.

In the diagram below, the timing annotation on the return path indicates to the initiator that the final timing point is to occur after the given delay. The phase transitions from BEGIN_REQ through END_REQ and BEGIN_RESP to END_RESP are implicit, rather than being passed explicitly as arguments to nb_transport.

Early completion

Figure 10

Phase	Initiator		Target

Simulation time = 100ns

Call   -, BEGIN_REQ, 0ns

BEGIN_REQ   TLM_COMPLETED, -, 10ns   *Return*

(END_RESP)   Simulation time = 110ns
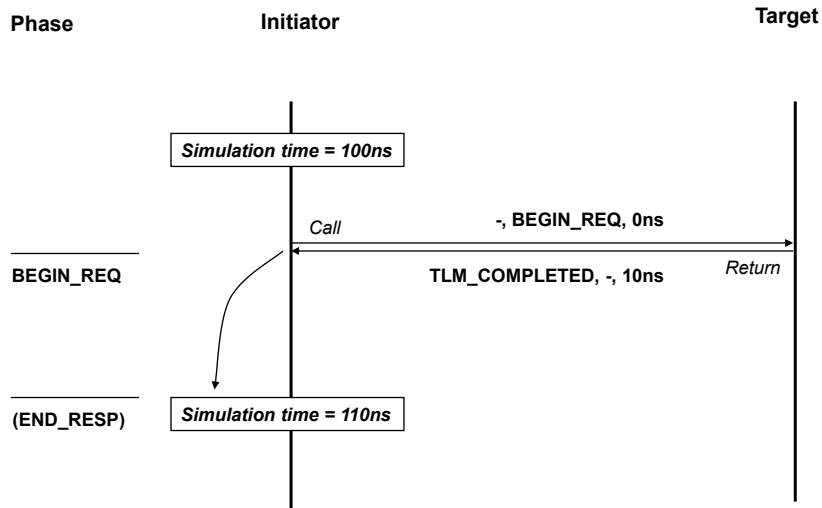
4.1.2.12 Message sequence chart – timing annotation

A caller may annotate a delay onto an *nb_transport* call. This is an indication to the callee that the transaction should be processed *as if* it had been received after the given delay. An approximately-timed callee would typically handle this situation by putting the transaction into a payload event queue for processing when simulation time has *caught up* with the annotated delay. Depending on the implementation of the payload event queue, this processing may occur either in a SystemC process sensitive to an event notification from the payload event queue or in a callback registered with the payload event queue.

Delays can be annotated onto calls on the forward and backward paths and the corresponding return paths. An approximately-timed initiator would be expected to handle incoming transactions on both the forward return path and the backward path in the same way. Similarly, an approximately-timed target would be expected to handle incoming transactions on both the backward return path and the forward path in the same way.

Figure 11
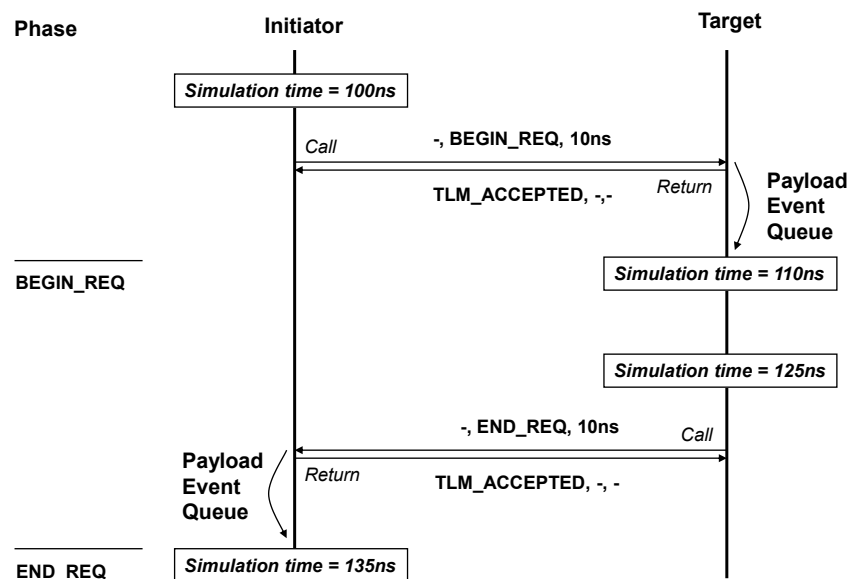
Timing annotation

4.1.3 Timing annotation with the transport interfaces

Timing annotation is a shared feature of the blocking and non-blocking transport interfaces, expressed using the **sc_time** argument to the **b_transport**, **nb_transport_fw** and **nb_transport_bw** methods. In this document, the italicized term *transport* is used to denote the three methods **b_transport**, **nb_transport_fw**, and **nb_transport_bw**.

Transaction ordering is governed by a combination of the core interface rules and the protocol rules. The rules in the following clause apply to the core interfaces regardless of the choice of protocol. For the base protocol, the rules given here should be read in conjunction with those in 8.2.7 Base protocol rules concerning timing annotation.

4.1.3.1 The sc_time argument

a)   It is recommended that the transaction object should not contain timing information. Any timing annotation should be expressed using the **sc_time** argument to *transport*

b)   The time argument shall be non-negative, and shall be expressed relative to the current simulation time **sc_time_stamp()**.

c)   The time argument shall apply on both the call to and return from *transport* (subject to the rules of the **tlm_sync_enum** return value of *nb_transport*).

d)   The *nb_transport* method may itself increase the value of the time argument, but shall not decrease the value. The **b_transport** method may increase the value of the time argument, or may decrease the value in the case that it has called **wait** and thus synchronized with simulation time, but only by an amount that is less than or equal to the time for which the process was suspended. This rule is consistent with time not running backward in a SystemC simulation.

e)   In the following description, the *recipient* of the transaction on the call to *transport* is the callee, and the *recipient* of the transaction on return from *transport* is the caller.

f)   The *recipient* shall behave as if it had received the transaction at an effective local time of **sc_time_stamp() + t**, where **t** is the value of the time argument. In other words, the recipient shall behave as if the timing point associated with the interface method call is to occur at the effective local time.

g)   Given a sequence of calls to *transport*, the effective local times at which the transactions are to be processed may or may not be in increasing time order in general. For transactions created by different initiators, it is fundamental to temporal decoupling that interface method call order may be different from effective local time order. The responsibility to handle transactions with out-of-order timing annotations lies with the *recipient*.

h)   Upon receipt of a transaction with a non-zero timing annotation, any recipient always has choices which reflect the modeling tradeoff between speed and accuracy. The recipient can execute any state changes caused by the transaction immediately and pass on the timing annotation, possibly increased, or it can schedule some internal process to resume after part or all of the annotated time has elapsed and execute the state changes only then. The choice is not enforced by the transport interface, but may be documented as part of a protocol traits class or coding style.

i) If the recipient is not concerned with timing accuracy or with processing a sequence of incoming transactions in the order given by their timing annotations, it may process each transaction immediately, without delay. Having done so, the recipient may also choose to increase the value of the timing annotation to model the time needed to process the transaction. This scenario assumes that the system design can tolerate out-of-order execution because of the existence of some explicit mechanism (over and above the TLM-2.0 interfaces) to enforce the correct causal chain of events.

j) If the recipient is to implement an accurate model of timing and execution order, it should ensure that the transaction is indeed processed at the correct time relative to any other SystemC processes with which it may interact. In SystemC, the appropriate mechanism to schedule an event at a future time is the timed event notification. For convenience, TLM-2.0 provides a family of utility classes, know as payload event queues, which can be used to queue transactions for processing at the proper simulation time according to the natural semantics of SystemC (see 9.3 Payload event queue). In other words, an approximately-timed recipient should typically put the transaction into a payload event queue.

k) Rather than processing the transaction directly, the recipient may pass the transaction on with a further call to or return from a *transport* method without modification to the transaction and using the same phase and timing annotation (or with an increased timing annotation).

l) With the loosely-timed coding style, transactions are typically executed immediately such that execution order matches interface method call order, and the **b_transport** method is recommended.

m) With the approximately-timed coding style, transactions are typically delayed such that their execution order matches the effective local time order, and the *nb_transport* method is recommended.

n) Each component can make the above choice dynamically on a call-by-call basis. For example, a loosely-timed component may execute a series of transactions immediately in call order, passing on the timing annotations, but may then choose to schedule the very next transaction for execution only after the delay given by the timing annotation has elapsed (known as *synchronization on demand*). This is a matter of coding style.

o) The above choice exists for both blocking and non-blocking transport. For example, **b_transport** may increase the timing annotation and return immediately or may **wait** for the timing annotation to elapse before returning. *nb_transport* may increase the timing annotation and return TLM_COMPLETED or may return TLM_ACCEPTED and schedule the transaction for execution later.

p) As a consequence of the above rules, if a component is the recipient of a series of transactions where the order of the incoming interface method calls is different from the effective local time order, the component is free to choose the mutual execution order of those particular transactions. The recommendation is to execute all transactions in call order or to execute all transactions in effective local time order, but this is not an obligation.

q) Note that the order in which incoming transactions are executed by a component should *in effect* always be the same as interface method call order, because a component will either execute an incoming transaction before returning from the interface method call regardless of the timing annotation (loosely-timed), or will schedule the transaction for execution at the proper future time and return TLM_ACCEPTED (approximately-timed), thus indicating to the caller that it should wait before issuing the next transaction. (TLM_ACCEPTED alone does not forbid the caller from issuing the next transaction, but in the case of the base protocol, the request and response exclusion rules may do so.)

r)   Timing annotation can also be described in terms of temporal decoupling. A non-zero timing annotation can be considered as an invitation to the recipient to "warp time". The recipient can choose to enter a time warp, or it can put the transaction in a queue for later processing and yield. In a loosely-timed model, time warping is generally acceptable. On the other hand, if the target has dependencies on other asynchronous events, the target may have to wait for simulation time to advance before it can predict the future state of the transaction with certainty.

s)   For a general description of temporal decoupling, see 3.3.2 Loosely-timed coding style and temporal decoupling

t)   For a description of the quantum, see 9.2 Quantum keeper

Example

The following pseudo-code fragments illustrate just some of the many possible coding styles:

```
// ---------------------------------------------
// Various interface method definitions
// ---------------------------------------------

void b_transport( TRANS& trans, sc_core::sc_time& t )
{
   // Loosely-timed coding style
   execute_transaction( trans );
   t = t + latency;
}

void b_transport( TRANS& trans, sc_core::sc_time& t )
{
   // Loosely-timed with synchronization at the target or synchronization-on-demand
   wait( t );
   execute_transaction( trans );
   t = SC_ZERO_TIME;
}

tlm_sync_enum nb_transport_fw( TRANS& trans, PHASE& phase, sc_core::sc_time& t )
{
   // Pseudo-loosely-timed coding style using non-blocking transport (not recommended)
   execute_transaction( trans );
   t = t + latency;
   return TLM_COMPLETED;
}

tlm_sync_enum nb_transport_fw( TRANS& trans, PHASE& phase, sc_core::sc_time& t )
{
   // Approximately-timed coding style
   // Post the transaction into a payload event queue for execution at time sc_time_stamp() + t
```

```
      payload_event_queue->notify( trans, phase, t );
      // Increment the transaction reference count
      trans.acquire();
      return TLM_ACCEPTED;
}

tlm_sync_enum nb_transport_fw( TRANS& trans, PHASE& phase, sc_core::sc_time& t )
{
      // Approximately-timed coding style making use of the backward path
      payload_event_queue->notify( trans, phase, t );
      trans.acquire();
      // Modify the phase and time arguments
      phase = END_REQ;
      t = t + accept_delay;
      return TLM_UPDATED;
}


// ---------------------------------------------------------------------------
// b_transport interface method calls, loosely-timed coding style
// ---------------------------------------------------------------------------

initialize_transaction( T1 );
socket->b_transport( T1, t );  // t may increase
process_response( T1 );

initialize_transaction( T2 );
socket->b_transport( T2, t );  // t may increase
process_response( T2 );

// Initiator may sync after each transaction or after a series of transactions
quantum_keeper->set( t );
if ( quantum_keeper->need_sync() )
      quantum_keeper->sync();


// --------------------------------------------------------------------------------
// nb_transport interface method call, approximately-timed coding style
// --------------------------------------------------------------------------------

initialize_transaction( T3 );
status = socket->nb_transport_fw( T3, phase, t );
if ( status == TLM_ACCEPTED )
{
      // No action, but expect an incoming nb_transport_bw method call
}
else if ( status == TLM_UPDATED )       // Backward path is being used
{
```

```
    payload_event_queue->notify( T3, phase, t );
}
else if ( status == TLM_COMPLETED )    // Early completion
{
    // Timing annotation may be taken into account in one of several ways
    // Either (1) by waiting, as shown here
    wait ( t );
    process_response( T3 );
    // or (2) by creating an event notification
    //    response_event.notify( t );
    // or (3) by being passed on to the next transport method call (code not shown here)
}
```

4.1.4   Migration path from TLM-1

The old TLM-1 and the new TLM-2.0 interfaces are both part of the TLM-2.0 standard. The TLM-1 blocking and non-blocking interfaces are still useful in their own right. For example, a number of vendors have used these interfaces in building functional verification environments for HDL designs.

The intent is that the similarity between the old and new blocking transport interfaces should ease the task of building adapters between legacy models using the TLM-1 interfaces and the new TLM-2.0 interfaces.

4.2 Direct memory interface

4.2.1 Introduction

The Direct Memory Interface, or DMI, provides a means by which an initiator can get direct access to an area of memory owned by a target, thereafter accessing that memory using a direct pointer rather than through the transport interface. The DMI offers a large potential increase in simulation speed for memory access between initiator and target because once established it is able to bypass the normal path of multiple **b_transport** or *nb_transport* calls from initiator through interconnect components to target.

There are two direct memory interfaces, one for calls on the forward path from initiator to target, and a second for calls on the backward path from target to initiator. The forward path is used to request a particular mode of DMI access (e.g. read or write) to a given address, and returns a reference to a DMI descriptor of type **tlm_dmi**, which contains the bounds of the DMI region. The backward path is used by the target to invalidate DMI pointers previously established using the forward path. The forward and backward paths may pass through zero, one or many interconnect components, but should be identical to the forward and backward paths for the corresponding transport calls through the same sockets.

A DMI pointer is requested by passing a transaction along the forward path. The default DMI transaction type is **tlm_generic_payload**, where only the command and address attributes of the transaction object are used. DMI follows the same approach to extension as the transport interface, that is, a DMI request may contain ignorable extensions, but any non-ignorable or mandatory extension requires the definition of a new protocol traits class (see 7.2.2 Define a new protocol traits class containing a typedef for tlm_generic_payload).

The DMI descriptor returns latency values for use by the initiator, and so provides sufficient timing accuracy for loosely-timed modeling.

DMI pointers may be used for debug, but the debug transport interface itself is usually sufficient because debug traffic is usually light, and usually dominated by I/O rather than memory access. Debug transactions are not usually on the critical path for simulation speed. If DMI pointers were used for debug, the latency values should be ignored.

4.2.2 Class definition

```
namespace tlm {

class tlm_dmi
{
public:
   tlm_dmi() { init(); }

   void init();

   enum dmi_access_e {
      DMI_ACCESS_NONE     = 0x00,
      DMI_ACCESS_READ     = 0x01,
      DMI_ACCESS_WRITE    = 0x02,
      DMI_ACCESS_READ_WRITE = DMI_ACCESS_READ | DMI_ACCESS_WRITE
```

```
    };

    unsigned char* get_dmi_ptr() const;
    sc_dt::uint64 get_start_address() const;
    sc_dt::uint64 get_end_address() const;
    sc_core::sc_time get_read_latency() const;
    sc_core::sc_time get_write_latency() const;
    dmi_access_e get_granted_access() const;
    bool is_none_allowed() const;
    bool is_read_allowed() const;
    bool is_write_allowed() const;
    bool is_read_write_allowed() const;

    void set_dmi_ptr(unsigned char* p);
    void set_start_address(sc_dt::uint64 addr);
    void set_end_address(sc_dt::uint64 addr);
    void set_read_latency(sc_core::sc_time t);
    void set_write_latency(sc_core::sc_time t);
    void set_granted_access(dmi_access_e t);
    void allow_none();
    void allow_read();
    void allow_write();
    void allow_read_write();
};

template <typename TRANS = tlm_generic_payload>
class tlm_fw_direct_mem_if : public virtual sc_core::sc_interface
{
public:
    virtual bool get_direct_mem_ptr(TRANS& trans, tlm_dmi& dmi_data) = 0;
};

class tlm_bw_direct_mem_if : public virtual sc_core::sc_interface
{
public:
    virtual void invalidate_direct_mem_ptr(sc_dt::uint64 start_range, sc_dt::uint64 end_range) = 0;
};

} // namespace tlm
```

4.2.3 get_direct_mem_ptr method

a) The **get_direct_mem_ptr** method shall only be called by an initiator or by an interconnect component, not by a target.

b) The **trans** argument shall pass a reference to a DMI transaction object constructed by the initiator.

c) The **dmi_data** argument shall be a reference to a DMI descriptor constructed by the initiator.

d) Any interconnect component should pass the **get_direct_mem_ptr** call along the forward path from initiator to target. In other words, the implementation of **get_direct_mem_ptr** for the target socket of the interconnect component may call the **get_direct_mem_ptr** method of an initiator socket.

e) Each **get_direct_mem_ptr** call shall follow exactly the same path from initiator to target as a corresponding set of transport calls. In other words, each DMI request shall involve an interaction between one initiator and one target, where those two components also serve the role of initiator and target for a single transaction object passed through the transport interface. DMI cannot be used on a path through a component that initiates a second transaction object, such as a non-trivial width converter. (If DMI is an absolute requirement for simulation speed, the simulation model may need to be restructured to permit it.)

f) Any interconnect components shall pass on the **trans** and **dmi_data** arguments in the forward direction, the only permitted modification being to the value of the address attribute of the DMI transaction object as described below. The address attribute of the transaction and the DMI descriptor may both be modified on return from the **get_direct_mem_ptr** method, that is, when unwinding the function calls from target back to initiator.

g) If the target is able to support DMI access to the given address, it shall set the members of the DMI descriptor as described below and set the return value of the function to **true**. When a target grants DMI access, the DMI descriptor is used to indicate the details of the access being granted.

h) If the target is not able to support DMI access to the given address, it need set only the address range and type members of the DMI descriptor as described below and set the return value of the function to **false**. When a target denies DMI access, the DMI descriptor is used to indicate the details of the access being denied.

i) A target may grant or deny DMI access to any part or parts of its memory region, including non-contiguous regions, subject to the rules given in this clause.

j) In the case that a target has granted DMI access and has set the return value of the function to **true**, an interconnect component may deny DMI access by setting the return value of the function to **false** on return from the **get_direct_mem_ptr** method. The reverse is not permitted; in the case that a target has denied DMI access, an interconnect component shall not grant DMI access.

k) Given multiple calls to **get_direct_mem_ptr**, a target may grant DMI access to multiple initiators for the same memory region at the same time. The application is responsible for synchronization and coherency.

l) Since each call to **get_direct_mem_ptr** can only return a single DMI pointer to a contiguous memory region, each DMI request can only be fulfilled by a single target in practice. In other words, if a memory region is scattered across multiple targets, then even though the address range is contiguous, each target will likely require a separate DMI request.

m) If read or write access to a certain region of memory causes side effects in a target (that is, causes some other change to the state of the target over and above the state of the memory), the target should not grant DMI access of the given type to that memory region. But if, for example, only write access causes side effects in a target, the target may still grant DMI read access to a given region.

n) The implementation of **get_direct_mem_ptr** may call **invalidate_direct_mem_ptr**.

o) The implementation of **get_direct_mem_ptr** shall not call **wait**, directly or indirectly.

4.2.4   template argument and tlm_generic_payload class

a)   The **tlm_fw_direct_mem_if** template shall be parameterized with the type of a DMI transaction class.

b)   The transaction object shall contain attributes to indicate the address for which direct memory access is requested and the type of access requested, namely read access or write access to the given address. In the case of the base protocol, these shall be the command and address attributes of the generic payload.

c)   The default value of the TRANS template argument shall be the class **tlm_generic_payload**.

d)   For maximal interoperability, the DMI transaction class should be the **tlm_generic_payload** class. The use of non-ignorable extensions or other transaction types will restrict interoperability.

e)   The initiator shall be responsible for constructing and managing the DMI transaction object, and for setting the appropriate attributes of the object before passing it as an argument to **get_direct_mem_ptr**.

f)   The command attribute of the transaction object shall be set by the initiator to indicate the kind of DMI access being requested, and shall not be modified by any interconnect component or target. For the base protocol, the permitted values are TLM_READ_COMMAND for read access, and TLM_WRITE_COMMAND for write access.

g)   For the base protocol, the command attribute is forbidden from having the value TLM_IGNORE_COMMAND. However, this value may be used by other protocols.

h)   The address attribute of the transaction object shall be set by the initiator to indicate the address for which direct memory access is being requested.

i)   An interconnect component passing the DMI transaction object along the forward path should decode and where necessary modify the address attribute of the transaction exactly as it would for the corresponding transport interface of the same socket. For example, an interconnect component may need to mask the address (reducing the number of significant bits) according to the address width of the target and its location in the system memory map.

j)   An interconnect component need not pass on the **get_direct_mem_ptr** call in the event that it detects an addressing error.

k)   In the case of the base protocol, the initiator is not obliged to set any other attributes of the generic payload aside from command and address, and the target and any interconnect components may ignore all other attributes. In particular, the response status attribute and the DMI allowed attribute may be ignored. The DMI allowed attribute is only intended for use with the transport interfaces.

l)   The initiator may re-use a transaction object from one DMI call to the next and across calls to DMI, the transport interfaces, and the debug transport interface.

m) If an application needs to add further attributes to a DMI transaction object for use by the target when determining the kind of DMI access being requested, the recommended approach is to add extensions to the generic payload rather than substituting an unrelated transaction class. For example, the DMI transaction might include a CPU ID to allow the target to service DMI requests differently depending on the kind of CPU making the request. In the case that such extensions are non-ignorable, this will require the definition of a new protocol traits class.

4.2.5   tlm_dmi class

a)   A DMI descriptor is an object of class **tlm_dmi**. DMI descriptors shall be constructed by initiators, but their members may be set by interconnect components or targets.

b)   A DMI descriptor shall have the following attributes: the DMI pointer attribute, the granted access type attribute, the start address attribute, the end address attribute, the read latency attribute, and the write latency attribute. The default values of these attributes shall be as follows: DMI pointer attribute = 0, access type = DMI_ACCESS_NONE, start address = 0, end address = the maximum value of type **sc_dt::uint64,** read latency = SC_ZERO_TIME, and write latency = SC_ZERO_TIME.

c)   Method **init** shall initialize the members of the DMI descriptor to their default values.

d)   A DMI descriptor shall be in its default state whenever it is passed as an argument to **get_direct_mem_ptr** by the initiator. If DMI descriptor objects are pooled, the initiator shall reset the DMI descriptor to its default state before passing it as an argument to **get_direct_mem_ptr**. Method **init** may be called for this purpose.

e)   Since an interconnect component is not permitted to modify the DMI descriptor as it is passed on towards the target, the DMI descriptor shall be in its initial state when it is received by the target.

f)   The method **set_dmi_ptr** shall set the DMI pointer attribute to the value passed as an argument. The method **get_dmi_ptr** shall return the current value of the DMI pointer attribute

g)   The DMI pointer attribute shall be set by the target to point to the storage location corresponding to the value of the start address attribute. This shall be less than or equal to the address requested in the call to **get_direct_mem_ptr**. The initial value shall be 0.

h)   The storage in the DMI region is represented with type **unsigned char***. The storage shall have the same organization as the data array of the generic payload. If a target is unable to return a pointer to a memory region with that organization, the target is unable to support DMI and **get_direct_mem_ptr** should return the value false. For a full description of how memory organization and endianness are handled in TLM-2.0, see 7.17 Endianness

i)   An interconnect component is permitted to modify the DMI pointer attribute on the return path from the **get_direct_mem_ptr** function call in order to restrict the region to which DMI access is being granted.

j)   The method **set_granted_access** shall set the granted access type attribute to the value passed as an argument. The method **get_granted_access** shall return the current value of the granted access type attribute. The initial value shall be DMI_ACCESS_NONE.

k)   The methods **allow_none**, **allow_read**, **allow_write** and **allow_read_write** shall set the granted access type attribute to the value DMI_ACCESS_NONE, DMI_ACCESS_READ, DMI_ACCESS_WRITE or DMI_ACCESS_READ_WRITE respectively.

l)   The method **is_none_allowed** shall return true if and only if the granted access type attribute has the value DMI_ACCESS_NONE. The method **is_read_allowed** shall return true if and only if the granted access type attribute has the value DMI_ACCESS_READ or DMI_ACCESS_READ_WRITE. The method **is_write_allowed** shall return true if and only if the granted access type attribute has the value DMI_ACCESS_WRITE or DMI_ACCESS_READ_WRITE. The method **is_read_write_allowed** shall return true if and only if the granted access type attribute has the value DMI_ACCESS_READ_WRITE.

m) The target shall set the granted access type attribute to the type of access being granted or being denied. A target is permitted to respond to a request for read access by granting (or denying) read or read/write access, and to a request for write access by granting (or denying) write or read/write access. An interconnect component is permitted to restrict the granted access type by overwriting a value of DMI_ACCESS_READ_WRITE with DMI_ACCESS_READ or DMI_ACCESS_WRITE on the return path from the **get_direct_mem_ptr** function call.

n) A target wishing to deny read and write access to the DMI region should set the granted access type to DMI_ACCESS_READ_WRITE, not to DMI_ACCESS_NONE.

Example

```
bool get_direct_mem_ptr( TRANS& trans, tlm::tlm_dmi&  dmi_data )
{
    // Deny DMI access to entire memory region
    dmi_data.allow_read_write();
    dmi_data.set_start_address( 0x0 );
    dmi_data.set_end_address( (sc_dt::uint64)-1 );
    return false;
}
```

o) The target should set the granted access type to DMI_ACCESS_NONE to indicate that it is not granting (or denying) read, write, or read/write access to the initiator, but is granting (or denying) some other kind of access as requested by an extension to the DMI transaction object. This value should only be used in cases where an extension to the DMI transaction object makes the pre-defined access types read, write and read/write unnecessary or meaningless. This value should not be used in the case of the base protocol.

p) The initiator is responsible for using only those modes of DMI access which have been granted by the target (and possibly modified by the interconnect) using the granted access type attribute (or in cases other than the base protocol, granted using extensions to the generic payload or using other DMI transaction types).

q) The methods **set_start_address** and **set_end_address** shall set the start and end address attributes, respectively, to the values passed as arguments. The methods **get_start_address** and **get_end_address** shall return the current values of the start and end address attributes, respectively.

r) The start and end address attributes shall be set by the target (or modified by the interconnect) to point to the addresses of the first and the last bytes in the DMI region. The DMI region is either being granted or being denied, as determined by the value returned from the **get_direct_mem_ptr** method (**true** or **false**). A target wishing to deny access to its entire memory region may set the start address to **0** and the end address to the maximum value of type **sc_dt::uint64**.

s) A target can only grant or deny a single contiguous memory region for each **get_direct_mem_ptr** call. A target can set the DMI region to a single address by having the start and end address attributes be equal, or can set the DMI region to be arbitrarily large.

t)   Having been granted DMI access of a given type to a given region, an initiator may perform access of the given type anywhere in that region until it is invalidated. In other words, access is not restricted to the address given in the DMI request.

u)   Any interconnect components that pass on the **get_direct_mem_ptr** call are obliged to transform the start and end address attributes as they do the address argument. Any transformations on the addresses in the DMI descriptor shall occur as the descriptor is passed along the return path from the **get_direct_mem_ptr** function call. For example, the target may set the start address attribute to a relative address within the memory map known to that target, in which case the interconnect component is obliged to transform the relative address back to an absolute address in the system memory map. The initial values shall be 0 and the maximum value of type **sc_dt::uint64**, respectively.

v)   An interconnect component is permitted to modify the start and end address attributes in order to restrict the region to which DMI access is being granted, or expand the range to which DMI access is being denied.

w)   If **get_direct_mem_ptr** returns the value **true**, the DMI region indicated by the start and end address attributes is a region for which DMI access is allowed. On the other hand, if **get_direct_mem_ptr** return the value **false,** it is a region for which DMI access is disallowed.

x)   A target or interconnect component receiving two or more calls to **get_direct_mem_ptr** may return two or more overlapping allowed DMI regions or two or more overlapping disallowed DMI regions.

y)   A target or interconnect component shall not return overlapping DMI regions where one region is allowed and the other is disallowed for the same access type, for example both read or read/write or both write or read/write, without making an intervening call to **invalidate_direct_mem_ptr** to invalidate the first region.

z)   In other words, the definition of the DMI regions shall not be dependent upon the order in which they were created unless the first region is invalidated by an intervening call to **invalidate_direct_mem_ptr**. Specifically, the creation of a disallowed DMI region shall not be permitted to punch a hole in an existing allowed DMI region for the same access type, or vice versa.

aa) A target may disallow DMI access to the entire address space (start address attribute = 0, end address attribute = maximum value), perhaps because the target does not support DMI access at all, in which case an interconnect component should clip this disallowed region down to the part of the memory map occupied by the target. Otherwise, if an interconnect component fails to clip the address range, then an initiator would be misled into thinking that DMI was disallowed across the entire system address space.

bb) The methods **set_read_latency** and **set_write_latency** shall set the read and write latency attributes, respectively, to the values passed as arguments. The methods **get_read_latency** and **get_write_latency** shall return the current values of the read and write latency attributes, respectively.

cc) The read and write latency attributes shall be set to the average latency per byte for read and write memory transactions, respectively. In other words, the initiator performing the direct memory operation shall calculate the actual latency by multiplying the read or write latency from the DMI descriptor by the number of bytes that would have been transferred by the equivalent *transport* transaction. The initial values shall be SC_ZERO_TIME. Both interconnect components and the target may increase the value of either latency such that the latency accumulates as the DMI descriptor is passed back from target to initiator on return from the **get_direct_mem_ptr** method. One or both latencies will be valid, depending on the value of the granted access type attribute.

dd) The initiator is responsible for respecting the latencies whenever it accesses memory using the direct memory pointer. If the initiator chooses to ignore the latencies, this may result in timing inaccuracies.

4.2.6 invalidate_direct_mem_ptr method

a) The **invalidate_direct_mem_ptr** method shall only be called by a target or an interconnect component.

b) A target is obliged to call **invalidate_direct_mem_ptr** before any change that would modify the validity or the access type of any existing DMI region. For example, before restricting the address range of an existing DMI region, before changing the access type from read/write to read, or before re-mapping the address space.

c) The **start_range** and **end_range** arguments shall be the first and last addresses of the address range for which DMI access is to be invalidated.

d) An initiator receiving an incoming call to **invalidate_direct_mem_ptr** shall immediately invalidate and discard any DMI region (previously received from a call to **get_direct_mem_ptr**) that overlaps with the given address range.

e) In the case of a partial overlap, that is, only part of an existing DMI region is invalidated, an initiator may adjust the boundaries of the existing region or may invalidate the entire region.

f) Each DMI region shall remain valid until it is explicitly invalidated by a target using a call to **invalidate_direct_mem_ptr**. Each initiator may maintain a table of valid DMI regions, and continue to use each region until it is invalidated.

g) Any interconnect components are obliged to pass on the **invalidate_direct_mem_ptr** call along the backward path from target to initiator, decoding and where necessary modifying the address arguments as they would for the corresponding transport interface. Because the transport interface transforms the address on the forward path and DMI on the backward path, the transport and DMI transformations should be the inverse of one another.

h) Given a single **invalidate_direct_mem_ptr** call from a target, an interconnect component may make multiple **invalidate_direct_mem_ptr** calls to initiators. Since there may be multiple initiators each getting direct memory pointers to the same target, a safe implementation is for an interconnect component to call **invalidate_direct_mem_ptr** for every initiator.

i) An interconnect component can invalidate all direct memory pointers in an initiator by setting **start_range** to 0 and **end_range** to the maximum value of the type **sc_dt::uint64**.

j) The implementation of any TLM-2.0 core interface method may call **invalidate_direct_mem_ptr**.

k) The implementation of **invalidate_direct_mem_ptr** shall not call **get_direct_mem_ptr**, directly or indirectly.

l) The implementation of **invalidate_direct_mem_ptr** shall not call **wait**, directly or indirectly.

4.2.7 DMI versus transport

a) By definition, the direct memory interface provides a direct interface between initiator and target that bypasses any interconnect components. The transport interfaces, on the other hand, cannot bypass interconnect components.

b) Care must be taken to ensure correct behavior when an interconnect component retains state or has side effects, such as buffered interconnects or interconnects modeling cache memory. The transport interfaces may access and update the state of the interconnect component, whereas the direct memory interface will bypass the interconnect component. The safest alternative is for such interconnect components always to deny DMI access.

c) It is possible for an initiator to switch back and forth between calling the transport interfaces and using a direct memory pointer. It is also possible that one initiator may use DMI while another initiator is using the transport interfaces. Care must be taken to ensure correct behavior, particularly considering that transport calls may carry a timing annotation. This is the responsibility of the application. For example, a given target could support DMI and transport simultaneously, or could invalidate every DMI pointer whenever transport is called.

4.2.8   DMI and temporal decoupling

a) A DMI region can only be invalidated by means of a target or interconnect component making a call to **invalidate_direct_mem_ptr**.

b) An initiator is responsible for checking that a DMI region is still valid before using the associated DMI pointer, subject to the following considerations.

c) The co-routine semantics of SystemC guarantee that once an initiator has started running, no other SystemC process will be able to run until the initiator yields. In particular, no other SystemC process would be able to invalidate a DMI pointer (although the current process might). As a consequence, a temporally decoupled initiator does not necessarily need to check repeatedly that a given DMI region is still valid each time it uses the associated DMI pointer.

d) It is possible that an interface method call made from an initiator may cause another component to call **invalidate_direct_mem_ptr**, thus invalidating a DMI region being used by that initiator. This could be true of a temporally decoupled initiator that runs without yielding.

e) While an initiator is running without interacting with any other components and without yielding, any valid DMI region will remain valid.

f) It is possible that after a temporally decoupled initiator using DMI has yielded, another temporally decoupled initiator may cause that same DMI region to be invalidated within the current time quantum. This reflects the fundamental inaccuracy intrinsic to temporal decoupling in general, but does not represent a violation of the rules given in this clause.

4.2.9   Optimization using a DMI hint

a) The DMI hint, otherwise known as the DMI allowed attribute, is a mechanism to optimize simulation speed by avoiding the need to repeatedly poll for DMI access. Instead of calling **get_direct_mem_ptr** to check for the availability of a DMI pointer, an initiator can check the DMI allowed attribute of a normal transaction passed through the transport interface.

b) The generic payload provides a DMI allowed attribute. User-defined transactions could implement a similar mechanism, in which case the target should set the value of the DMI allowed attribute appropriately.

c) Use of the DMI allowed attribute is optional. An initiator is free to ignore the DMI allowed attribute of the generic payload.

d) For an initiator wishing to take advantage of the DMI allowed attribute, the recommended sequence of actions is as follows:

  i. The initiator should check the address against its cache of valid DMI regions

  ii. If there is no existing DMI pointer, the initiator should perform a normal transaction through the transport interface

  iii. Following that, the initiator should check the DMI allowed attribute of the transaction

  iv. If the attribute indicates DMI is allowed, the initiator should call **get_direct_mem_ptr**

  v. The initiator should modify its cache of valid DMI regions according to the values returned from the call.

4.3   Debug transport interface

4.3.1   Introduction

The debug transport interface provides a means to read and write to storage in a target, over the same forward path from initiator to target as is used by the transport interface, but without any of the delays, waits, event notifications or side effects associated with a regular transaction. In other words, the debug transport interface is non-intrusive. Because the debug transport interface follows the same path as the transport interface, the implementation of the debug transport interface can perform the same address translation as for regular transactions.

For example, the debug transport interface could permit a software debugger attached to an ISS to peek or poke an address in the memory of the simulated system from the point of view of the simulated CPU. The debug transport interface could also allow an initiator to take a snapshot of system memory contents during simulation for diagnostic purposes, or to initialize some area of system memory at the end of elaboration.

The default debug transaction type is **tlm_generic_payload**, where only the command, address, data length and data pointer attributes of the transaction object are used. Debug transactions follow the same approach to extension as the transport interface, that is, a debug transaction may contain ignorable extensions, but any non-ignorable or mandatory extension requires the definition of a new protocol traits class (see 7.2.2 Define a new protocol traits class containing a typedef for tlm_generic_payload).

4.3.2   Class definition

```
namespace tlm {

template <typename TRANS = tlm_generic_payload>
class tlm_transport_dbg_if : public virtual sc_core::sc_interface
{
public:
    virtual unsigned int transport_dbg(TRANS& trans) = 0;
};

} // namespace tlm
```

4.3.3   TRANS template argument and tlm_generic_payload class

a)   The **tlm_transport_dbg_if** template shall be parameterized with the type of a debug transaction class.

b)   The debug transaction class shall contain attributes to indicate to the target the command, address, data length and date pointer for the debug access. In the case of the base protocol, these shall be the corresponding attributes of the generic payload.

c)   The default value of the TRANS template argument shall be the class **tlm_generic_payload**.

d)   For maximal interoperability, the debug transaction class should be **tlm_generic_payload**. The use of non-ignorable extensions or other transaction types will restrict interoperability.

e) If an application needs to add further attributes to a debug transaction, the recommended approach is to add extensions to the generic payload rather than substituting an unrelated transaction class. In the case that such extensions are non-ignorable or mandatory, this will require the definition of a new protocol traits class.

4.3.4 Rules

a) Calls to **transport_dbg** shall follow the same forward path as the transport interface used for normal transactions.

b) The **trans** argument shall pass a reference to a debug transaction object.

c) The initiator shall be responsible for constructing and managing the debug transaction object, and for setting the appropriate attributes of the object before passing it as an argument to **transport_dbg**.

d) The command attribute of the transaction object shall be set by the initiator to indicate the kind of debug access being requested, and shall not be modified by any interconnect component or target. For the base protocol, the permitted values are TLM_READ_COMMAND for read access to the target, TLM_WRITE_COMMAND for write access to the target, and TLM_IGNORE_COMMAND.

e) On receipt of a transaction with the command attribute equal to TLM_IGNORE_COMMAND, the target should not execute a read or a write, but may use the value of any attribute in the generic payload, including any extensions, in executing an extended debug transaction.

f) As is the case for the transport interface, the use of any non-ignorable or mandatory generic payload extension with the debug transport interface requires the definition of a new protocol traits class.

g) The address attribute shall be set by the initiator to the first address in the region to be read or written.

h) An interconnect component passing the debug transaction object along the forward path should decode and where necessary modify the address attribute of the transaction object exactly as it would for the corresponding transport interface of the same socket. For example, an interconnect component may need to mask the address (reducing the number of significant bits) according to the address width of the target and its location in the system memory map.

i) An interconnect component need not pass on the **transport_dbg** call in the event that it detects an addressing error.

j) The address attribute may be modified several times if a debug payload is forwarded through several interconnect components. When the debug payload is returned to the initiator, the original value of the address attribute may have been overwritten.

k) The data length attribute shall be set by the initiator to the number of bytes to be read or written and shall not be modified by any interconnect component or target. The data length attribute may be 0, in which case the target shall not read or write any bytes, and the data pointer attribute may be null.

l) The data pointer attribute shall be set by the initiator to the address of an array from which values are to be copied to the target (for a write), or to which values are to be copied from the target (for a read), and shall not be modified by any interconnect component or target. This array shall be allocated by the initiator, and shall not be deleted before the return from **transport_dbg**. The size of the array shall be at least equal to the value of the data length attribute. If the data length attribute is 0, the data pointer attribute may be the null pointer and the array need not be allocated.

m) The implementation of **transport_dbg** in the target shall read or write the given number of bytes using the given address (after address translation through the interconnect), if it is able. In the case of a write command, the target shall not modify the contents of the data array.

n) The data array shall have the same organization as the data array of the generic payload when used with the transport interface. The implementation of **transport_dbg** shall be responsible for converting between the organization of the local data storage within the target and the generic payload organization.

o) In the case of the base protocol, the initiator is not obliged to set any other attributes of the generic payload aside from command, address, data length and data pointer, and the target and any interconnect components may ignore all other attributes. In particular, the response status attribute may be ignored.

p) The initiator may re-use a transaction object from one call to the next and across calls to the debug transport interface, the transport interfaces, and DMI.

q) **transport_dbg** shall return a count of the number of bytes actually read or written, which may be less than the value of the data length attribute. If the target is not able to perform the operation, it shall return a value of 0.

r) Directly or indirectly, **transport_dbg** shall not call **wait**, and should not cause any state changes in any interconnect component or in the target aside from the immediate effect of executing a debug write command.

5   Global quantum

5.1   Introduction

Temporal decoupling permits SystemC processes to run ahead of simulation time for an amount of time known as the time quantum, and is associated with the loosely-timed coding style. Temporal decoupling permits a significant simulation speed improvement by reducing the number of context switches and events. The use of a time quantum is not strictly necessary in the presence of explicit synchronization between temporally decoupled processes, in which case processes may run arbitrarily far ahead to the point when the next synchronization point is reached. However, any processes that do require a time quantum should use the global quantum.

When using temporal decoupling, the delays annotated to the **b_transport** and *nb_transport* methods are to be interpreted as local time offsets defined relative to the current simulation time as returned by **sc_time_stamp()**, also known as the quantum boundary. The global quantum is the default time interval between successive quantum boundaries. The value of the global time quantum is maintained by the singleton class **tlm_global_quantum**. It is recommended that each process should use the global time quantum, but a process is permitted to calculate its own local time quantum.

For a general description of temporal decoupling, see 3.3.2 Loosely-timed coding style and temporal decoupling

For a description of timing annotation, see 4.1.3 Timing annotation with the transport interfaces

The utility class **tlm_quantumkeeper** provides a set of methods for managing and interacting with the time quantum. For a description of how to use a quantum keeper, see 9.2 Quantum keeper

5.2   Header file

The class definition for the global quantum shall be in the header file **tlm.h**

5.3   Class definition

```
namespace tlm {

class tlm_global_quantum
{
public:
    static tlm_global_quantum& instance();
    virtual ~tlm_global_quantum();
    void set( const sc_core::sc_time& );
    const sc_core::sc_time& get() const;
    sc_core::sc_time compute_local_quantum();

protected:
```

```
    tlm_global_quantum();
};

} // namespace tlm
```

5.4   Class tlm_global_quantum

a)   There is a unique global quantum maintained by the class **tlm_global_quantum**. This should be considered the default time quantum. The intent is that all temporally decoupled initiators should synchronize on integer multiples of the global quantum, or more frequently where required.

b)   It is possible for each initiator to use a different time quantum, but more typical for all initiators to use the global quantum. An initiator that only requires infrequent synchronization could conceivably have a longer time quantum than the rest, but it is usually the shortest time quantum that has the biggest negative impact on simulation speed.

c)   The method **instance** shall return a reference to the singleton global quantum object.

d)   The method **set** shall set the value of the global quantum to the value passed as an argument.

e)   The method **get** shall return the value of the global quantum.

f)   The method **compute_local_quantum** shall calculate and return the value of the local quantum based on the unique global quantum. The local quantum shall be calculated by subtracting the value of **sc_time_stamp** from the next larger integer multiple of the global quantum. The local quantum shall equal the global quantum in the case where **compute_local_quantum** is called at a simulation time that is an integer multiple of the global quantum. Otherwise, the local quantum shall be less than the global quantum.

6   Combined interfaces and sockets

6.1   Combined interfaces

6.1.1   Introduction

The combined forward and backward transport interfaces group the core TLM-2.0 interfaces for use by the initiator and target sockets. Note that the combined interfaces include the transport, DMI and debug transport interfaces, but do not include any TLM-1 core interfaces. The forward interface provides method calls on the forward path from initiator socket to target socket, and the backwards interface on the backward path from target socket to initiator socket. Neither the blocking transport interface nor the debug transport interface require a backward calling path.

It would be technically possible to define new socket class templates unrelated to the standard initiator and target sockets and then to instantiate those class templates using the combined interfaces as template arguments, but for the sake of interoperability this is not recommended. On the other hand, deriving new socket classes from the standard sockets is recommended for convenience.

The combined interface templates are parameterized with a *protocol traits class* that defines the types used by the forward and backward interfaces, namely the payload type and the phase type. A protocol traits class is associated with a specific protocol. The default protocol type is the class **tlm_base_protocol_types**. See 8.2 Base protocol.

6.1.2   Class definition

```
namespace tlm {

// The default protocol traits class:
struct tlm_base_protocol_types
{
    typedef  tlm_generic_payload  tlm_payload_type;
    typedef  tlm_phase            tlm_phase_type;
};

// The combined forward interface:
template< typename TYPES = tlm_base_protocol_types >
class tlm_fw_transport_if
    : public virtual tlm_fw_nonblocking_transport_if<typename TYPES::tlm_payload_type ,
                                                     typename TYPES::tlm_phase_type>
    , public virtual tlm_blocking_transport_if<      typename TYPES::tlm_payload_type>
    , public virtual tlm_fw_direct_mem_if<           typename TYPES::tlm_payload_type>
    , public virtual tlm_transport_dbg_if<           typename TYPES::tlm_payload_type>
{};

// The combined backward interface:
template < typename TYPES = tlm_base_protocol_types >
```

```
class tlm_bw_transport_if
    : public virtual tlm_bw_nonblocking_transport_if<typename TYPES::tlm_payload_type ,
                                          typename TYPES::tlm_phase_type >
    , public virtual tlm_bw_direct_mem_if
{};

} // namespace tlm
```

6.2   Initiator and target sockets

6.2.1   Introduction

A socket combines a port with an export. An initiator socket has a port for the forward path and an export for the backward path, whilst a target socket has an export for the forward path and a port for the backward path. The sockets also overload the SystemC port binding operators to bind both the port and export to the export and port in the opposing socket. When binding sockets hierarchically, parent to child or child to parent, it is important to carefully consider the binding order.

Both the initiator and target sockets are coded using a C++ inheritance hierarchy. Only the most derived classes **tlm_initiator_socket** and **tlm_target_socket** are typically used directly by applications. These two sockets are parameterized with a *protocol traits class* that defines the types used by the forward and backward interfaces. Sockets can only be bound together if they have the identical protocol type. The default protocol type is the class **tlm_base_protocol_types**. If an application defines a new protocol it should instantiate combined interface templates with a new protocol traits class, whether or not the new protocol is based on the generic payload.

The initiator and target sockets provide the following benefits:

a) They group the transport, direct memory and debug transport interfaces for both the forward and backward paths together into a single object.

b) They provide methods to bind port and export of both the forward and backward paths in a single call.

c) They offer strong type checking when binding sockets parameterized with incompatible protocol types.

d) They include a bus width parameter that may be used to interpret the transaction.

The socket classes **tlm_initiator_socket** and **tlm_target_socket** belong to the interoperability layer of the TLM-2.0 standard. In addition, there is a family of derived socket classes provided in the utilities namespace, collectively known as *convenience sockets*.

6.2.2   Class definition

```
namespace tlm {

// Abstract base class for initiator sockets
template <
```

```
      unsigned int BUSWIDTH = 32,
      typename FW_IF = tlm_fw_transport_if<>,
      typename BW_IF = tlm_bw_transport_if<>
   >
   class tlm_base_initiator_socket_b
   {
   public:
      virtual ~tlm_base_initiator_socket_b() {}

      virtual sc_core::sc_port_b<FW_IF> & get_base_port() = 0;
      virtual                    BW_IF & get_base_interface() = 0;
      virtual sc_core::sc_export<BW_IF> & get_base_export() = 0;
   };


   // Abstract base class for target sockets
   template <
      unsigned int BUSWIDTH = 32,
      typename FW_IF = tlm_fw_transport_if<>,
      typename BW_IF = tlm_bw_transport_if<>
   >
   class tlm_base_target_socket_b
   {
   public:
      virtual ~tlm_base_target_socket_b();

      virtual sc_core::sc_port_b<BW_IF> &  get_base_port() = 0;
      virtual sc_core::sc_export<FW_IF> &  get_base_export() = 0;
      virtual                     FW_IF &  get_base_interface() = 0;
   };


   // Base class for initiator sockets, providing binding methods
   template <
      unsigned int BUSWIDTH = 32,
      typename FW_IF = tlm_fw_transport_if<>,
      typename BW_IF = tlm_bw_transport_if<>,
      int N = 1,
      sc_core::sc_port_policy POL = sc_core::SC_ONE_OR_MORE_BOUND
   >
   class tlm_base_initiator_socket : public tlm_base_initiator_socket_b<BUSWIDTH, FW_IF, BW_IF>,
                                     public sc_core::sc_port<FW_IF, N, POL>
   {
   public:
      typedef FW_IF                              fw_interface_type;
      typedef BW_IF                              bw_interface_type;
```

```
    typedef sc_core::sc_port<fw_interface_type, N, POL>     port_type;
    typedef sc_core::sc_export<bw_interface_type>           export_type;
    typedef tlm_base_target_socket_b<BUSWIDTH, fw_interface_type, bw_interface_type>
                                                   base_target_socket_type;
    typedef tlm_base_initiator_socket_b<BUSWIDTH, fw_interface_type, bw_interface_type>
                                                   base_type;

    tlm_base_initiator_socket();
    explicit tlm_base_initiator_socket(const char* name);
    virtual const char* kind() const;

    unsigned int get_bus_width() const;

    void bind(base_target_socket_type& s);
    void operator() (base_target_socket_type& s);
    void bind(base_type& s);
    void operator() (base_type& s);
    void bind(bw_interface_type& ifs);
    void operator() (bw_interface_type& s);

    // Implementation of pure virtual functions of base class
    virtual sc_core::sc_port_b<FW_IF> &  get_base_port()      { return *this;   }
    virtual                    BW_IF &  get_base_interface() { return m_export; }
    virtual sc_core::sc_export<BW_IF> &  get_base_export()     { return m_export; }

protected:
    export_type m_export;
};


// Base class for target sockets, providing binding methods
template <
    unsigned int BUSWIDTH = 32,
    typename FW_IF = tlm_fw_transport_if<>,
    typename BW_IF = tlm_bw_transport_if<>,
    int N = 1,
    sc_core::sc_port_policy POL = sc_core::SC_ONE_OR_MORE_BOUND
>
class tlm_base_target_socket : public tlm_base_target_socket_b<BUSWIDTH, FW_IF, BW_IF>,
                               public sc_core::sc_export<FW_IF>
{
public:
    typedef  FW_IF                                    fw_interface_type;
    typedef  BW_IF                                    bw_interface_type;
    typedef sc_core::sc_port<bw_interface_type, N, POL>   port_type;
    typedef sc_core::sc_export<fw_interface_type>        export_type;
```

```
typedef tlm_base_initiator_socket_b<BUSWIDTH, fw_interface_type, bw_interface_type>
                                          base_initiator_socket_type;
typedef tlm_base_target_socket_b<BUSWIDTH, fw_interface_type, bw_interface_type>
                                          base_type;

tlm_base_target_socket();
explicit tlm_base_target_socket(const char* name);
virtual const char* kind() const;

unsigned int get_bus_width() const;

void bind(base_initiator_socket_type& s);
void operator() (base_initiator_socket_type& s);
void bind(base_type& s);
void operator() (base_type& s);
void bind(fw_interface_type& ifs);
void operator() (fw_interface_type& s);

int size() const;
bw_interface_type* operator-> ();
bw_interface_type* operator[] (int i);

// Implementation of pure virtual functions of base class
virtual sc_core::sc_port_b<BW_IF> & get_base_port()      { return m_port;  }
virtual                    FW_IF &  get_base_interface() { return *this;  }
virtual sc_core::sc_export<FW_IF> & get_base_export()    { return *this;  }

protected:
  port_type m_port;
};


// Principal initiator socket, parameterized with protocol traits class
template <
  unsigned int BUSWIDTH = 32,
  typename TYPES = tlm_base_protocol_types,
  int N = 1,
  sc_core::sc_port_policy POL = sc_core::SC_ONE_OR_MORE_BOUND
>
class tlm_initiator_socket : public tlm_base_initiator_socket <
  BUSWIDTH, tlm_fw_transport_if<TYPES>, tlm_bw_transport_if<TYPES>, N, POL>
{
public:
  tlm_initiator_socket();
  explicit tlm_initiator_socket(const char* name);
  virtual const char* kind() const;
```

```
    };


    // Principal target socket, parameterized with protocol traits class
    template <
        unsigned int BUSWIDTH = 32,
        typename TYPES = tlm_base_protocol_types,
        int N = 1,
        sc_core::sc_port_policy POL = sc_core::SC_ONE_OR_MORE_BOUND
    >
    class tlm_target_socket : public tlm_base_target_socket <
        BUSWIDTH, tlm_fw_transport_if<TYPES>, tlm_bw_transport_if<TYPES>, N, POL>
    {
    public:
        tlm_target_socket();
        explicit tlm_target_socket(const char* name);
        virtual const char* kind() const;
    };


    } // namespace tlm
```

6.2.3   Classes tlm_base_initiator_socket_b and tlm_base_target_socket_b

a)   The abstract base classes **tlm_base_initiator_socket_b** and **tlm_base_target_socket_b** declare pure virtual functions that should be overridden in any derived class to return the port, export and interface objects associated with the socket.

b)   These sockets are not typically used directly by applications.

6.2.4   Classes tlm_base_initiator_socket and tlm_base_target_socket

a)   For class **tlm_base_initiator_socket**, the constructor with a name argument shall pass the character string argument to the constructor belonging to the base class **sc_port** to set the string name of the instance in the module hierarchy, and shall also pass the same character string to set the string name of the corresponding **sc_export** on the backward path, adding the suffix **"_export"** and calling **sc_gen_unique_name** to avoid name clashes. For example, the call **tlm_initiator_socket("foo")** would set the port name to **"foo"** and the export name to **"foo_export"**. In the case of the default constructor, the names shall be created by calling **sc_gen_unique_name("tlm_base_initiator_socket")** for the port, and **sc_gen_unique_name("tlm_base_initiator_socket_export")** for the export.

b)   For class **tlm_base_target_socket**, the constructor with a name argument shall pass the character string argument to the constructor belonging to the base class **sc_export** to set the string name of the instance in the module hierarchy, and shall also pass the same character string to set the string name of the corresponding **sc_port** on the backward path, adding the suffix **"_port"** and calling **sc_gen_unique_name** to avoid name clashes. For example, the call **tlm_target_socket("foo")** would set the export name to **"foo"** and the port name to **"foo_port"**. In the case of the default constructor, the

names shall be created by calling **sc_gen_unique_name("tlm_base_target_socket")** for the export, and **sc_gen_unique_name("tlm_base_target_socket_port")** for the port.

c) The method **kind** shall return the class name as a C string, that is, "tlm_base_initiator_socket" or "tlm_base_target_socket" respectively.

d) The method **get_bus_width** shall return the value of the BUSWIDTH template argument.

e) Template argument BUSWIDTH shall determine the word length for each individual data word transferred through the socket, expressed as the number of bits in each word. For a burst transfer, BUSWIDTH shall determine the number of bits in each beat of the burst. The precise interpretation of this attribute shall depend on the transaction type. For the meaning of BUSWIDTH with the generic payload, see 7.11 Data length attribute.

f) When binding socket-to-socket, the two sockets shall have identical values for the BUSWIDTH template argument. Executable code in the initiator or target may get and act on the BUSWIDTH.

g) Each of the methods **bind** and **operator()** that take a socket as an argument shall bind the socket instance to which the method belongs to the socket instance passed as an argument to the method.

h) Each of the methods **bind** and **operator()** that take an interface as an argument shall bind the export of the socket instance to which the method belongs to the channel instance passed as an argument to the method. (A channel is the SystemC term for a class that implements an interface.)

i) When binding initiator socket to target socket, the **bind** method and **operator()** shall each bind the port of the initiator socket to the export of the target socket, and the port of the target socket to the export of the initiator socket. This is for use when binding socket-to-socket at the same level in the hierarchy.

j) An initiator socket can be bound to a target socket by calling the **bind** method or **operator()** of either socket, with precisely the same effect. In either case, the forward path lies in the direction from the initiator socket to the target socket.

k) When binding initiator socket to initiator socket or target socket to target socket, the **bind** method and **operator()** shall each bind the port of one socket to the port of the other socket, and the export of one socket to the export of the other socket. This is for use in hierarchical binding, that is, when binding a socket on a child module to a socket on a parent module, or a socket on a parent module to a socket on a child module, passing transactions up or down the module hierarchy.

l) For hierarchical binding, it is necessary to bind sockets in the correct order. When binding initiator socket to initiator socket, the socket of the child must be bound to the socket of the parent. When binding target socket to target socket, the socket of the parent must be bound to the socket of the child. This rule is consistent with the fact the **tlm_base_initiator_socket** is derived from **sc_port**, and **tlm_base_target_socket** from **sc_export**. Port must be bound to port going up the hierarchy, port-to-export across the top, and export-to-export going down the hierarchy.

m) In order for two sockets of classes **tlm_base_initiator_socket** and **tlm_base_target_socket** to be bound together, they must share the same forward and backward interface types and bus widths

n) The method **size** of the target socket shall call method **size** of the port in the target socket (on the backward path), and shall return the value returned by **size** of the port.

o) The method **operator->** of the target socket shall call method **operator->** of the port in the target socket (on the backward path), and shall return the value returned by **operator->** of the port.

p) The method **operator[]** of the target socket shall call method **operator[]** of the port in the target socket (on the backward path) with the same argument, and shall return the value returned by **operator[]** of the port.

q) Class **tlm_base_initiator_socket** and class **tlm_base_target_socket** each act as multi-sockets, that is, a single initiator socket may be bound to multiple target sockets, and a single target socket may be bound to multiple initiator sockets. The two class templates have template parameters specifying the number of bindings and the port binding policy, which are used within the class implementation to parameterize the associated **sc_port** template instantiation.

r) If an object of class **tlm_base_initiator_socket** or **tlm_base_target_socket** is bound multiple times, then the method **operator[]** can be used to address the corresponding object to which the socket is bound. The index value is determined by the order in which the methods **bind** or **operator()** were called to bind the sockets. However, any incoming interface method calls received by such a socket will be *anonymous* in the sense that there is no mechanism provided to identify the caller. On the other hand, such a mechanism is provided by the convenience sockets. See 9.1.4 Multi-sockets.

s) For example, consider a socket bound to two separate targets. The calls **socket[0]->nb_transport_fw(...)** and **socket[1]->nb_transport_fw()** would address the two targets, but there is no way to identify the caller of in incoming **nb_transport_bw()** method from one of those two targets.

t) The implementations of the virtual methods **get_base_port** and **get_base_export** shall return the port and export objects of the socket, respectively. The implementation of the virtual method **get_base_interface** shall return the export object in the case of the initiator port, or the socket object itself in the case of the target socket.

6.2.5   Classes tlm_initiator_socket and tlm_target_socket

a) The socket **tlm_initiator_socket** and **tlm_target_socket** take a protocol traits class as a template parameter. These sockets (or convenience sockets derived from them) should typically be used by an application rather than the base sockets.

b) The constructors of the classes **tlm_initiator_socket** and **tlm_target_socket** shall call the corresponding constructors of their respective base classes, passing the **char*** argument where it exists.

c) In order for two sockets of classes **tlm_initiator_socket** and **tlm_target_socket** to be bound together, they must share the same protocol traits class (default **tlm_base_protocol_types**) and bus width. Strong type checking between sockets can be achieved by defining a new protocol traits class for each distinct protocol, whether or not that protocol is based on the generic payload.

d) The method **kind** shall return the class name as a C string, that is, "tlm_initiator_socket" or "tlm_target_socket" respectively.

Example

```
#include <systemc>
#include "tlm.h"
using namespace sc_core;
using namespace std;
```

```
struct Initiator: sc_module, tlm::tlm_bw_transport_if<>  // Initiator implements the bw interface
{
    tlm::tlm_initiator_socket<32> init_socket;        // Protocol types default to base protocol

    SC_CTOR(Initiator) : init_socket("init_socket") {
        SC_THREAD(thread);
        init_socket.bind( *this );                     // Initiator socket bound to the initiator itself
    }

    void thread() {                                    // Process generates one dummy transaction
        tlm::tlm_generic_payload trans;
        sc_time delay = SC_ZERO_TIME;
        init_socket->b_transport(trans, delay);
    }

    virtual tlm::tlm_sync_enum nb_transport_bw(
        tlm::tlm_generic_payload& trans,
        tlm::tlm_phase& phase,
        sc_core::sc_time& t) {
        return tlm::TLM_COMPLETED;                      // Dummy implementation
    }

    virtual void invalidate_direct_mem_ptr(sc_dt::uint64 start_range, sc_dt::uint64 end_range)
    { }                                                 // Dummy implementation
};

struct Target: sc_module, tlm::tlm_fw_transport_if<>    // Target implements the fw interface
{
    tlm::tlm_target_socket<32> targ_socket;            // Protocol types default to base protocol

    SC_CTOR(Target) : targ_socket("targ_socket") {
        targ_socket.bind( *this );                      // Target socket bound to the target itself
    }

    virtual tlm::tlm_sync_enum nb_transport_fw(
        tlm::tlm_generic_payload& trans, tlm::tlm_phase& phase, sc_core::sc_time& t) {
            return tlm::TLM_COMPLETED;                   // Dummy implementation
    }

    virtual void b_transport( tlm::tlm_generic_payload& trans, sc_time& delay )
    { }                                                 // Dummy implementation

    virtual bool get_direct_mem_ptr(tlm::tlm_generic_payload& trans, tlm::tlm_dmi&  dmi_data)
    { return false; }                                   // Dummy implementation

    virtual unsigned int transport_dbg(tlm::tlm_generic_payload& trans)
```

```
    { return 0; }                                        // Dummy implementation
};

SC_MODULE(Top1)    // Showing a simple non-hierarchical binding of initiator to target
{
    Initiator *init;
    Target    *targ;

    SC_CTOR(Top1) {
        init = new Initiator("init");
        targ = new Target("targ");
        init->init_socket.bind(targ->targ_socket);     // Bind initiator socket to target socket
    }
};

struct Parent_of_initiator: sc_module                 // Showing hierarchical socket binding
{
    tlm::tlm_initiator_socket<32> init_socket;

    Initiator* initiator;

    SC_CTOR(Parent_of_initiator) : init_socket("init_socket") {
        initiator = new Initiator("initiator");
        initiator->init_socket.bind( init_socket );      // Bind initiator socket to parent initiator socket
    }
};

struct Parent_of_target: sc_module
{
    tlm::tlm_target_socket<32> targ_socket;

    Target* target;

    SC_CTOR(Parent_of_target) : targ_socket("targ_socket") {
        target = new Target("target");
        targ_socket.bind( target->targ_socket );         // Bind parent target socket to target socket
    }
};

SC_MODULE(Top2)
{
    Parent_of_initiator *init;
    Parent_of_target    *targ;

    SC_CTOR(Top2) {
        init = new Parent_of_initiator("init");
```

```
        targ = new Parent_of_target("targ");
        init->init_socket.bind(targ->targ_socket);      // Bind initiator socket to target socket at top level
    }
};
```

7 Generic payload

7.1 Introduction

The generic payload is the class type offered by the TLM-2.0 standard for transaction objects passed through the core interfaces. The generic payload is closely related to the base protocol, which itself defines further rules to ensure interoperability when using the generic payload. See 8.2 Base protocol

The generic payload is intended to improve the interoperability of memory-mapped bus models, which it does at two levels. Firstly, the generic payload provides an off-the-shelf general-purpose payload that guarantees immediate interoperability when creating abstract models of memory-mapped buses where the precise details of the bus protocol are unimportant, whilst at the same time providing an extension mechanism for *ignorable* attributes. Secondly, the generic payload can be used as the basis for creating detailed models of specific bus protocols, with the advantage of reducing the implementation cost and increasing simulation speed when there is a need to bridge or adapt between different protocols, sometimes to the point where the bridge becomes trivial to write.

The generic payload is specifically aimed at modeling memory-mapped buses. It includes some of the attributes found in typical memory-mapped bus protocols such as command, address, data, byte enables, single word transfers, burst transfers, streaming, and response status. The generic payload may also be used as the basis for modeling protocols other than memory-mapped buses.

The generic payload does not include every attribute found in typical memory-mapped bus protocols, but it does include an extension mechanism so that applications can add their own specialized attributes.

For specific protocols, whether bus-based or not, modeling and interoperability are the responsibility of the protocol owners and are outside the scope of OSCI. It is up to the protocol owners or subject matter experts to proliferate models or coding guidelines for their own particular protocol. However, the generic payload is still applicable here, because it provides a common starting point for model creation, and in many cases will reduce the cost of bridging between different protocols in a transaction-level model.

It is recommended that the generic payload be used with the initiator and target sockets, which provide a bus width parameter used when interpreting the data array of the generic payload as well as forward and backward paths and a mechanism to enforce strong type checking between different protocols whether or not they are based on the generic payload.

The generic payload can be used with both the blocking and non-blocking transport interfaces. It can also be used with the direct memory and debug transport interfaces, in which case only a restricted set of attributes is used.

7.2 Extensions and interoperability

The goal of the generic payload is to enable interoperability between memory-mapped bus models, but all buses are not created equal. Given two transaction-level models that use different protocols and that model those protocols at a detailed level, then just as in a physical system, an adapter or bridge must be inserted between those models to perform protocol conversion and allow them to communicate. On the other hand, many transaction level models produced early in the design flow do not care about the specific details of any

particular protocol. For such models it is sufficient to copy a block of data starting at a given address, and for those models the generic payload can be used directly to give excellent interoperability.

The generic payload extension mechanism permits any number of extensions of any type to be defined and added to a transaction object. Each extension represents a new set of attributes, transported along with the transaction object. Extensions can be created, added, written and read by initiators, interconnect components, and targets alike. The extension mechanism itself does not impose any restrictions. Of course, undisciplined use of this extension mechanism would compromise interoperability, so disciplined use is strongly encouraged. But the flexibility is there where you need it!

The use of the extension mechanism represents a trade-off between increased coding convenience when binding sockets, and decreased compile-time type checking. If the undisciplined use of generic payload extensions were allowed, each application would be obliged to detect any incompatibility between extensions by including explicit run-time checks in each interconnect component and target, and there would be no mechanism to enforce the existence of a given extension. The TLM-2.0 standard prescribes specific coding guidelines to avoid these pitfalls.

There are three, and only three, recommended alternatives for the transaction template argument TRANS of the blocking and non-blocking transport interfaces and the template argument TYPES of the combined interfaces:

a)   Use the generic payload directly, with ignorable extensions

b)   Define a new protocol traits class containing a **typedef** for **tlm_generic_payload**.

c)   Define a new protocol traits class and a new transaction type

These three alternatives are defined below in order of decreasing interoperability.

It should be emphasized that although deriving a new class from the generic payload is possible, it is not the recommended approach for interoperability

It should also be emphasized that these three options may be mixed in a single system model. In particular, there is value in mixing the first two options, since the extension mechanism has been designed to permit efficient interoperability.

7.2.1   Use the generic payload directly, with ignorable extensions

a)   In this case, the transaction type is **tlm_generic_payload**, the phase type is **tlm_phase**, and the protocol traits class for the combined interfaces is **tlm_base_protocol_types**. These are the default values for the TRANS argument of the transport interfaces and TYPES argument of the combined interfaces, respectively. Any model that uses the standard initiator and target sockets with the base protocol will be interoperable with any other such model, provided that those models respect the semantics of the generic payload and the base protocol. See 8.2 Base protocol

b)   In this case, any generic payload extension or extended phase shall be ignorable. *Ignorable* means that any component other than the component that added the extension is permitted to behave as if the extension were absent. See 7.20.1.1 Ignorable extensions

c)   If an extension is ignorable, then by definition compile-time checking to enforce support for that extension in a target is not wanted, and indeed, the ignorable extension mechanism does not support compile-time checking.

d) The generic payload intrinsically supports minor variations in protocol. As a general principle, a target is recommended to support every feature of the generic payload. But, for example, a particular component may or may not support byte enables. A target that is unable to support a particular feature of the generic payload is obliged to generate the standard error response. This should be thought of as being part of the specification of the generic payload.

7.2.2 Define a new protocol traits class containing a typedef for tlm_generic_payload

a) In this case, the transaction type is **tlm_generic_payload** and the phase type **tlm_phase**, but the protocol traits class used to specialize the socket is a new application-defined class, not the default **tlm_base_protocol_types**. This ensures that the extended generic payload is treated as a distinct type, and provides compile-time type checking when the initiator and target sockets are bound.

b) The new protocol type may set its own rules, and these rules may extend or contradict any of the rules of the base protocol, including the generic payload memory management rules (see 7.5 Generic payload memory management) and the rules for the modifiability of attributes (see 7.7 Default values and modifiability of attributes). However, for the sake of consistency and interoperability it is recommended to follow the rules and coding style of the base protocol as far as possible. See 8.2 Base protocol

c) The generic payload extension mechanism may be used for ignorable, non-ignorable or mandatory extensions with no restrictions. The semantics of any extensions should be thoroughly documented with the new protocol traits class.

d) Because the transaction type is **tlm_generic_payload**, the transaction can be transported through interconnect components and targets that use the generic payload type, and can be cloned in its entirety, including all extensions. This provides a good starting point for building interoperable components and for creating adapters or bridges between different protocols, but the user should consider the semantics of the extended generic payload very carefully.

e) It is usual to use one and the same protocol traits class along the entire length of the path followed by a transaction from an initiator through zero or more interconnect components to a target. However, it may be possible to model an adapter or bus bridge as an interconnect component that takes incoming transactions of one protocol type and converts them to outgoing transactions of another protocol type. It is also possible to create a transaction bridge, which acts as a target for incoming transactions and as an initiator for outgoing transactions.

f) When passing a generic payload transaction between sockets specialized using different protocol traits classes, the user is obliged to consider the semantics of each extension very carefully to ensure that the transaction can be transported through components that are aware of the generic payload but not the extensions. There is no general rule. Some extensions can be transported through components ignorant of the extension without mishap, for example an attribute specifying the security level of the data. Other extensions will require explicit adaption or might not be supportable at all, for example an attribute specifying that the interconnect is to be locked.

7.2.3 Define a new protocol traits class and a new transaction type

a) In this case, the transaction type may be unrelated to the generic payload.

b) A new protocol traits class will need to be defined to parameterize the combined interfaces and the sockets.

c) This choice may be justified when the new transaction type is significantly different from the generic payload or represents a very specific protocol.

d) If the intention is to use the generic payload for maximal interoperability, the recommended approach is to use the generic payload as described in one of the previous two clauses rather than use it in the definition of a new class.

7.3   Generic payload attributes and methods

The generic payload class contains a set of private attributes, and a set of public access functions to get and set the values of those attributes. The exact implementation of those access functions is implementation-defined.

The majority of the attributes are set by the initiator and shall not be modified by any interconnect component or target. Only the address, DMI allowed, response status and extension attributes may be modified by an interconnect component or by the target. In the case of a read command, the target may also modify the data array.

7.4   Class definition

```
namespace tlm {

class tlm_generic_payload;

class tlm_mm_interface {
public:
   virtual void free(tlm_generic_payload*) = 0;
   virtual ~tlm_mm_interface() {}
};

unsigned int max_num_extensions();

class tlm_extension_base
{
public:
   virtual tlm_extension_base* clone() const = 0;
   virtual void free() { delete this; }
   virtual void copy_from(tlm_extension_base const &) = 0;
protected:
   virtual ~tlm_extension_base() {}
};

template <typename T>
class tlm_extension : public tlm_extension_base
{
public:
```

```
    virtual tlm_extension_base* clone() const = 0;
    virtual void copy_from(tlm_extension_base const &) = 0;
    virtual ~tlm_extension() {}
    const static unsigned int ID;
};


enum tlm_command {
  TLM_READ_COMMAND,
  TLM_WRITE_COMMAND,
  TLM_IGNORE_COMMAND
};


enum tlm_response_status {
  TLM_OK_RESPONSE = 1,
  TLM_INCOMPLETE_RESPONSE = 0,
  TLM_GENERIC_ERROR_RESPONSE = -1,
  TLM_ADDRESS_ERROR_RESPONSE = -2,
  TLM_COMMAND_ERROR_RESPONSE = -3,
  TLM_BURST_ERROR_RESPONSE = -4,
  TLM_BYTE_ENABLE_ERROR_RESPONSE = -5
};


#define TLM_BYTE_DISABLED 0x0
#define TLM_BYTE_ENABLED 0xff


class tlm_generic_payload {
public:
    // Constructors and destructor
    tlm_generic_payload();
    explicit tlm_generic_payload( tlm_mm_interface* );
    virtual ~tlm_generic_payload();

private:
    // Disable copy constructor and assignment operator
    tlm_generic_payload( const tlm_generic_payload& );
    tlm_generic_payload& operator= ( const tlm_generic_payload& );

public:
    // Memory management
    void set_mm( tlm_mm_interface* );
    bool has_mm() const;
    void acquire();
    void release();
    int get_ref_count() const;
    void reset();
    void deep_copy_from( const tlm_generic_payload & );
```

void **update_original_from**(const tlm_generic_payload & , bool use_byte_enable_on_read = true);
void **update_extensions_from**(const tlm_generic_payload &);
void **free_all_extensions**();

// Access methods
tlm_command **get_command**() const;
void **set_command**(const tlm_command);
bool **is_read**();
void **set_read**();
bool **is_write**();
void **set_write**();

sc_dt::uint64 **get_address**() const;
void **set_address**(const sc_dt::uint64);

unsigned char* **get_data_ptr**() const;
void **set_data_ptr**(unsigned char*);

unsigned int **get_data_length**() const;
void **set_data_length**(const unsigned int);

unsigned int **get_streaming_width**() const;
void **set_streaming_width**(const unsigned int);

unsigned char* **get_byte_enable_ptr**() const;
void **set_byte_enable_ptr**(unsigned char*);
unsigned int **get_byte_enable_length**() const;
void **set_byte_enable_length**(const unsigned int);

// DMI hint
void **set_dmi_allowed**(bool);
bool **is_dmi_allowed**() const;

tlm_response_status **get_response_status**() const;
void **set_response_status**(const tlm_response_status);
std::string **get_response_string**();
bool **is_response_ok**();
bool **is_response_error**();

// Extension mechanism
template <typename T> T* **set_extension**(T*);
tlm_extension_base* **set_extension**(unsigned int , tlm_extension_base*);

template <typename T> T* **set_auto_extension**(T*);
tlm_extension_base* **set_auto_extension**(unsigned int , tlm_extension_base*);

```
    template <typename T> void get_extension( T*& ) const;
    template <typename T> T* get_extension() const;
    tlm_extension_base* get_extension( unsigned int ) const;

    template <typename T> void clear_extension( const T* );
    template <typename T> void clear_extension();

    template <typename T> void release_extension( T* );
    template <typename T> void release_extension();

    void resize_extensions();
};

} // namespace tlm
```

7.5   Generic payload memory management

a)   The initiator shall be responsible for setting the data pointer and byte enable pointer attributes to existing storage, which could be static, automatic (stack) or dynamically allocated (new) storage. The initiator shall not delete this storage before the lifetime of the transaction is complete. The generic payload destructor does not delete these two arrays.

b)   This clause should be read in conjunction with the rules on generic payload extensions. See 7.20 Generic payload extensions.

c)   The generic payload supports two distinct approaches to memory management; reference counting with an explicit memory manager and ad hoc memory management by the initiator. The two approaches can be combined. Any memory management approach should manage both the transaction object itself and any extensions to the transaction object.

d)   The construction and destruction of objects of type **tlm_generic_payload** is expected to be expensive in terms of CPU time due to the implementation of the extension array. As a consequence, repeated construction and destruction of generic payload objects should be avoided. There are two recommended strategies; either use a memory manager that implements a pool of transaction objects, or if using ad hoc memory management, re-use the very same generic payload object across successive calls to **b_transport** (effectively a transaction pool with a size of one). In particular, having a generic payload object constructed and destructed once per call to *transport* would be prohibitively slow and should be avoided.

e)   A memory manager is a user-defined class that implements at least the **free** method of the abstract base class **tlm_mm_interface**. The intent is that a memory manager would provide a method to allocate a generic payload transaction object from a pool of transactions, would implement the **free** method to return a transaction object to that same pool, and would implement a destructor to delete the entire pool. The **free** method is called by the **release** method of class **tlm_generic_payload** when the reference count of a transaction object reaches 0. The **free** method of class **tlm_mm_interface** would typically call the **reset** method of class **tlm_generic_payload** in order to delete any extensions marked for automatic deletion.

f) The methods **set_mm**, **acquire**, **release**, **get_ref_count** and **reset** of the generic payload shall only used in the presence of a memory manager. By default, a generic payload object does not have a memory manager set.

g) Ad hoc memory management by the initiator without a memory manager requires the initiator to allocate memory for the transaction object before the TLM-2.0 core interface call, and delete or pool the transaction object and any extension objects after the call.

h) When the generic payload is used with the blocking transport interface, the direct memory interface or the debug transport interface, either approach may be used. Ad hoc memory management by the initiator is sufficient. In the absence of a memory manager, the **b_transport**, **get_direct_mem_ptr**, or **transport_dbg** method should assume that the transaction object and any extensions will be invalidated or deleted on return.

i) When the generic payload is used with the non-blocking transport interface, a memory manager shall be used. Any transaction object passed as an argument to *nb_transport* shall have a memory manager already set. This applies whether the caller is the initiator, an interconnect component, or a target.

j) A blocking-to-non-blocking transport adapter shall set a memory manager for a given transaction if none existed already, in which case it shall remove that same memory manager from the transaction before returning control to the caller. A memory manager cannot be removed until the reference count has returned to 0, so the implementation will necessarily require that the method **free** of the memory manager does not delete the transaction object. The **simple_target_socket** provides an example of such an adapter.

k) When using a memory manager, the transaction object and any extension objects shall be allocated from the heap (ultimately by calling **new** or **malloc**).

l) When using ad hoc memory management, the transaction object and any extensions may be allocated from the heap or from the stack. When using stack allocation, particular care needs to be taken with the memory management of extension objects in order to avoid memory leaks and segmentation faults.

m) The method **set_mm** shall set the memory manager of the generic payload object to the object whose address is passed as an argument. The argument may be null, in which case any existing memory manager would be removed from the transaction object, but not itself deleted. **set_mm** shall not be called for a transaction object that already has a memory manager and a reference count greater than 0.

n) The method **has_mm** shall return true if and only if a memory manager has been set. When called from the body of an *nb_transport* method, **has_mm** should return true.

o) When called from the body of the **b_transport**, **get_direct_mem_ptr**, or **transport_dbg** methods, **has_mm** may return true or false. An interconnect component may call **has_mm** and take the appropriate action depending on whether or not a transaction has a memory manager. Otherwise, it shall assume all the obligations of a transaction with a memory manager (for example, heap allocation), but shall not call any of the methods that require the presence of a memory manager (for example, **acquire**).

p) Each generic payload object has a reference count. The default value of the reference count is 0.

q) The method **acquire** shall increment the value of the reference count. If **acquire** is called in the absence of a memory manager, a run-time error will occur.

r) The method **release** shall decrement the value of the reference count, and if this leaves the value equal to 0, shall call the method **free** of the memory manager object, passing the address of the transaction object as an argument. If **release** is called in the absence of a memory manager, a run-time error will occur.

s) The method **get_ref_count** shall return the value of the reference count. In the absence of a memory manager, the value returned would be 0.

t) In the presence of a memory manager, each initiator should call the **acquire** method of each transaction object before first passing that object as an argument to an interface method call, and should call the **release** method of that transaction object when the object is no longer required.

u) In the presence of a memory manager, each interconnect component and target should call the **acquire** method whenever they need to extend the lifetime of a transaction object beyond the current interface method call, and call the **release** method when the object is no longer required.

v) In the presence of a memory manager, a component may call the **release** method from any interface method call or process. Thus, a component cannot assume a transaction object is still valid after making an interface method call or after yielding control unless it has previously called the **acquire** method. For example, an initiator may call **release** from its implementation of **nb_transport_bw**, or a target from its implementation of **nb_transport_fw**.

w) If an interconnect component or a target wishes to extend the lifetime of a transaction object indefinitely for analysis purposes, it should make a clone of the transaction object rather than using the reference counting mechanism. In other words, the reference count should not be used to extend the lifetime of a transaction object beyond the normal phases of the protocol.

x) In the presence of a memory manager, a transaction object shall not be re-used to represent a new transaction or re-used with a different interface until the reference count indicates that no component other than the initiator itself still has a reference to the transaction object. That is, assuming the initiator has called **acquire** for the transaction object, until the reference count equals 1. This rule applies when re-using transactions with the same interface or across the transport, direct memory and debug transport interfaces. When reusing transaction objects to represent different transaction instances, it is best practice not to reuse the object until the reference count equals 0, that is, until the object has been freed.

y) The method **reset** shall delete any extensions marked for automatic deletion, and shall set the corresponding extension pointers to null. Each extension shall be deleted by calling the method **free** of the extension object, which could conceivably be overloaded if a user wished to provide explicit memory management for extension objects. The method **reset** should typically be called from the method **free** of class **tlm_mm_interface** in order to delete extensions at the end of the lifetime of a transaction.

z) An extension object added by calling **set_extension** may be deleted by calling **release_extension**. Calling **clear_extension** would only clear the extension pointer, not delete the extension object itself. This latter behavior would be required in the case that transaction objects are stack-allocated without a memory manager, and extension objects pooled.

aa) In the absence of a memory manager, whichever component allocates or sets a given extension should also delete or clear that same extension before returning control from **b_transport**, **get_direct_mem_ptr**, or **transport_dbg**. For example, an interconnect component that implements **b_transport** and calls **set_mm** to add a memory manager to a transaction object shall not return from **b_transport** until it has removed from the transaction object all extensions added by itself (and assuming

that any downstream components will already have removed any extensions added by themselves, by virtue of this very same rule).

bb) In the presence of a memory manager, extensions can be added by calling **set_auto_extension,** and thus deleted or pooled automatically by the memory manager. Alternatively, extensions added by calling **set_extension** and not explicitly cleared are so-called *sticky* extensions, meaning that they will not be automatically deleted when the transaction reference count reaches 0 but may remain associated with the transaction object even when it is pooled. Sticky extensions are a particularly efficient way to manage extension objects because the extension object need not be deleted and re-constructed between transport calls. Sticky extensions rely on transaction objects being pooled (or re-used singly).

cc) If it is unknown whether or not a memory manager is present, extensions should be added by calling **set_extension** and deleted by calling **release_extension**. This calling sequence is safe in the presence or absence of a memory manager. This circumstance can only occur within an interconnect component or target that chooses not to call **has_mm**. (Within an initiator, it is always known whether or not a memory manager is present, and a call to **has_mm** will always reveal whether or not a memory manager is present.)

dd) The method **free_all_extensions** shall delete all extensions, including but not limited to those marked for automatic deletion, and shall set the corresponding extension pointers to null. Each extension shall be deleted by calling the method **free** of the extension object. The **free** method could conceivably be overloaded if a user wished to provide explicit memory management for extension objects.

ee) **free_all_extensions** would be useful when removing the extensions from a pooled transaction object that does not use a memory manager. With a memory manager, extensions marked for automatic deletion would indeed have been deleted automatically, while sticky extensions would not need to be deleted.

ff) The method **deep_copy_from** shall modify the attributes and extensions of the current transaction object by copying those of another transaction object, which is passed as an argument to the method. The command, address, data length, byte enable length, streaming width, response status, and DMI allowed attributes shall be copied. The data and byte enable arrays shall be deep copied if and only if the corresponding pointers in both transactions are non-null. The application is responsible for ensuring that the arrays in the current transaction are sufficiently large. If an extension on the other transaction already exists on the current transaction, it shall be copied by calling the **copy_from** method of the extension class. Otherwise, a new extension object shall be created by calling the **clone** method of the extension class, and set on the current transaction. In the case of cloning, the new extension shall be marked for automatic deletion if and only if a memory manager is present for the current transaction.

gg) In other words, in the presence of a memory manager **deep_copy_from** will mark for automatic deletion any new extensions that were not already on the current object. Without a memory manager, extensions cannot be marked for auto-deletion.

hh) The method **update_original_from** shall modify certain attributes and extensions of the current transaction object by copying those of another transaction object, which is passed as an argument to the method. The intent is that **update_original_from** should be called to pass back the response for a transaction created using **deep_copy_from**. The response status and DMI allowed attributes of the current transaction object shall be modified. The data array shall be deep copied if and only if the command attribute of the current transaction is TLM_READ_COMMAND and the data pointers in the two transactions are both non-null and are unequal. The byte enable array shall be used to mask the copy operation, as per the read command, if and only if the byte enable pointer is non-null and the

use_byte_enable_on_read argument is **true**. Otherwise, the entire data array shall be deep copied. The extensions of the current transaction object shall be updated as per the **update_extensions_from** method.

ii) The method **update_extensions_from** shall modify the extensions of the current transaction object by copying from another transaction object only those extensions that were already present on the current object. The extensions shall be copied by calling the **copy_from** method of the extension class.

jj) The typical use case for **deep_copy_from**, **update_original_from** and **update_extensions_from** is within a transaction bridge where they are used to deep copy an incoming request, send the copy out through an initiator socket, then on receiving back the response copy the appropriate attributes and extensions back to the original transaction object. The transaction bridge may choose to deep copy the arrays or merely to copy the pointers.

kk) These obligations apply to the generic payload. In principle, similar obligations might apply to transaction types unrelated to the generic payload

7.6 Constructors, assignment, and destructor

a) The default constructor shall set the generic payload attributes to their default values, as defined in the following clauses.

b) The constructor **tlm_generic_payload(tlm_mm_interface*)** shall set the generic payload attributes to their default values, and shall set the memory manager of the generic payload object to the object whose address is passed as an argument. This is equivalent to calling the default constructor then immediately calling **set_mm**.

c) The copy constructor and assignment operators are disabled.

d) The virtual destructor ~**tlm_generic_payload** shall delete all extensions, including but not limited to those marked for automatic deletion. Each extension shall be deleted by calling the method **free** of the extension object. The destructor shall not delete the data array or the byte enable array.

7.7 Default values and modifiability of attributes

The default values and modifiability of the generic payload attributes and arrays are summarized in the following table:

Attribute	Default value	Modifiable by interconnect?	Modifiable by target?
Command	TLM_IGNORE_COMMAND	No	No
Address	0	**Yes**	No
Data pointer	0	No	No
Data length	0	No	No
Byte enable pointer	0	No	No
Byte enable length	0	No	No
Streaming width	0	No	No
DMI allowed	false	**Yes**	**Yes**
Response status	TLM_INCOMPLETE_RESPONSE	No	**Yes**
Extension pointers	0	**Yes**	**Yes**

Arrays	Default value	Modifiable by interconnect?	Modifiable by target?
Data array	-	No	**Read command only**
Byte enable array	-	No	No

a) It is the responsibility of the initiator to set the value of every generic payload attribute (with the exception of extension pointers) prior to passing the transaction object through an interface method call. Care should be taken to ensure the attributes are set correctly in the case where transaction objects are pooled and reused.

b) In the case that a transaction object is returned to a pool or otherwise re-used, these modifiability rules cease to apply at the end of the lifetime of that transaction instance. In the presence of a memory manager this is the point at which the reference count reaches 0, or otherwise, on return from the interface method call. The modifiability rules would apply afresh where the transaction object to be re-used for a new transaction.

c) After passing the transaction object as an argument to an interface method call (**b_transport**, **nb_transport_fw**, **get_direct_mem_ptr**, or **transport_dbg**), the only generic payload attributes that the initiator is permitted to modify during the lifetime of the transaction are the extension pointers.

d) An interconnect component is permitted to modify the address attribute, but only before passing the transaction concerned as an argument to any TLM-2.0 core interface method on the forward path. Once an interconnect component has passed a reference to the transaction to a downstream component, it is not permitted to modify the address attribute of that transaction object again throughout the entire lifetime of the transaction.

e) As a consequence of the previous rule, the address attribute is valid immediately upon entering any of the forward path interface method calls **b_transport**, **get_direct_mem_ptr**, or **transport_dbg**. In the case of **nb_transport_fw** the address attribute is valid immediately upon entering the function but only when the phase is BEGIN_REQ. Following the return from any forward path TLM-2.0 interface method call, the address attribute will have the value set by the interconnect component lying furthest downstream, and so should be regarded as being undefined for the purposes of transaction routing.

f) The interconnect and target are not permitted to modify the data array in the case of a write command, but the target alone is permitted to modify the data array in the case of a read command.

g) For a given transaction object, the target is permitted to modify the DMI allowed attribute, the response status attribute and (for a read command) the data array at any time between having first received the transaction object and the time at which it passes a response in the upstream direction. A target is not permitted to modify these attributes after having sent a response in the upstream direction. A target sends a response in this sense whenever it returns control from the **b_transport**, **get_direct_mem_ptr** or **transport_dbg** methods, whenever it passes the BEGIN_RESP phase as an argument to *nb_transport*, or whenever it returns the value TLM_COMPLETED from *nb_transport*.

h) If the DMI allowed attribute is **false**, an interconnect component is not permitted to modify the DMI allowed attribute. But if the target sets the DMI allowed attribute to **true**, an interconnect component is permitted to reset the DMI allowed attribute to **false** as it passes the response in an upstream direction. In other words, an interconnect component is permitted to clear the DMI allowed attribute, despite the DMI allowed attribute having been set by the target.

i) The initiator is permitted to assume it is seeing the values of the DMI allowed attribute, the response status attribute and (for a read command) the data array as modified by the target only after it has received the response.

j) If the above rules permit a component to modify the value of a transaction attribute within a particular window of time, that attribute may be modified at any time during that window and any number of times during that window. Any other component shall only read the value of the attribute as it is left at the end of the time window (with the exception of extensions).

k) The roles of initiator, interconnect, and target may change dynamically. For example, although an interconnect component is not permitted to modify the response status attribute, that same component could modify the response status attribute by taking on the role of target for a given transaction. In its role as a target, the component would be forbidden from passing that particular transaction any further downstream.

l) In the case where the generic payload is used as the transaction type for the direct memory and debug transport interfaces, the modifiability rules given in this section shall apply to the appropriate attributes,

that is, the command and address attributes in the case of direct memory, and the command, address, data pointer and data length attributes in the case of debug transport.

7.8   Command attribute

a)   The method **set_command** shall set the command attribute to the value passed as an argument. The method **get_command** shall return the current value of the command attribute.

b)   The methods **set_read** and **set_write** shall set the command attribute to TLM_READ_COMMAND and TLM_WRITE_COMMAND respectively. The methods **is_read** and **is_write** shall return **true** if and only if the current value of the command attribute is TLM_READ_COMMAND and TLM_WRITE_COMMAND respectively.

c)   A read command is a generic payload transaction with the command attribute equal to TLM_READ_COMMAND. A write command is a generic payload transaction with the command attribute equal to TLM_WRITE_COMMAND. An ignore command is a generic payload transaction with the command attribute equal to TLM_IGNORE_COMMAND

d)   On receipt of a read command, the target shall copy the contents of a local array in the target to the array pointed to be the data pointer attribute, honoring all the semantics of the generic payload as defined by this standard.

e)   On receipt of a write command, the target shall copy the contents of the array pointed to by the data pointer attribute to a local array in the target, honoring all the semantics of the generic payload as defined by this standard.

f)   If the target is unable to execute a read or write command, it shall generate a standard error response. The recommended response status is TLM_COMMAND_ERROR_RESPONSE.

g)   An ignore command is a null command. The intent is that an ignore command may be used as a vehicle for transporting generic payload extensions without the need to execute a read or a write command, although the rules concerning extensions are the same for all three commands.

h)   On receipt of an ignore command, the target shall not execute a write command or a read command. In particular, it shall not modify the value of the local array that would be modified by a write command, or modify the value of the array pointed to by the data pointer attribute. The target may, however, use the value of any attribute in the generic payload, including any extensions.

i)   On receipt of an ignore command, a component that usually acts as an interconnect component may either forward the transaction onward toward the target (that is, act as an interconnect), or may return an error response (that is, act as a target). A component that routes read and write commands differently would be expected to return an error response.

j)   A target is deemed to have executed an ignore command successfully if it has received the transaction and has checked the values of the generic payload attributes to its own satisfaction. See 7.16 Response status attribute

k)   The command attribute shall be set by the initiator, and shall not be overwritten by any interconnect component or target.

l)   The default value of the command attribute shall be TLM_IGNORE_COMMAND.

7.9   Address attribute

a)   The method **set_address** shall set the address attribute to the value passed as an argument. The method **get_address** shall return the current value of the address attribute.

b)   For a read command or a write command, the target shall interpret the current value of the address attribute as the start address in the system memory map of the contiguous block of data being read or written. This address may or may not correspond to the first byte in the array pointed to by the data pointer attribute, depending on the endianness of the host computer.

c)   The address associated with any given byte in the data array is dependent upon the address attribute, the array index, the streaming width attribute, the endianness of the host computer and the width of the socket. See 7.17 Endianness

d)   The value of the address attribute need not be word-aligned (although address calculations can be considerably simplified if the address attribute is a multiple of the local socket width expressed in bytes).

e)   If the target is unable to execute the transaction with the given address attribute (because the address is out-of-range, for example) it shall generate a standard error response. The recommended response status is TLM_ADDRESS_ERROR_RESPONSE.

f)   The address attribute shall be set by the initiator, but may be overwritten by one or more interconnect components. This may be necessary if an interconnect component performs address translation, for example to translate an absolute address in the system memory map to a relative address in the memory map known to the target. Once the address attribute has been overwritten in this way, the old value is lost (unless it was explicitly saved somewhere).

g)   The default value of the address attribute shall be 0.

7.10   Data pointer attribute

a)   The method **set_data_ptr** shall set the data pointer attribute to the value passed as an argument. The method **get_data_ptr** shall return the current value of the data pointer attribute. Note that the data pointer attribute is a pointer to the data array, and these methods set or get the value of the pointer, not the contents of the array.

b)   For a read command or a write command, the target shall copy data to or from the data array, respectively, honoring the semantics of the remaining attributes of the generic payload.

c)   The initiator is responsible for allocating storage for the data and byte enable arrays. The storage may represent the final source or destination of the data in the initiator, such as a register file or cache memory, or may represent a temporary buffer used to transfer data to and from the transaction level interface.

d)   In general, the organization of the generic payload data array is independent of the organization of local storage within the initiator and the target. However, the generic payload has been designed so that data can be copied to and from the target with a single call to **memcpy** in most circumstances. This assumes that the target uses the same storage organization as the generic payload. This assumption is made for simulation efficiency, but does not restrict the expressive power of the generic payload: the target is free to transform the data in any way it wishes as it copies the data to and from the data array.

e) It is an error to call the transport interface with a transaction object having a null data pointer attribute.

f) The length of the data array shall be greater than or equal to the value of the data length attribute, in bytes.

g) The data pointer attribute shall be set by the initiator, and shall not be overwritten by any interconnect component or target.

h) For a write command or TLM_IGNORE_COMMAND, the contents of the data array shall be set by the initiator, and shall not be overwritten by any interconnect component or target

i) For a read command, the contents of the data array may be overwritten by the target (honoring the semantics of the byte enable) but by no other component and only before the target sends a response. A target sends a response in this sense whenever it returns control from the **b_transport**, **get_direct_mem_ptr** or **transport_dbg** methods, whenever it passes the BEGIN_RESP phase as an argument to *nb_transport*, or whenever it returns the value TLM_COMPLETED from *nb_transport*.

j) The default value of the data pointer attribute shall be 0, the null pointer.

7.11 Data length attribute

a) The method **set_data_length** shall set the data length attribute to the value passed as an argument. The method **get_data_length** shall return the current value of the data length attribute.

b) For a read command or a write command, the target shall interpret the data length attribute as the number of bytes to be copied to or from the data array, inclusive of any bytes disabled by the byte enable attribute.

c) The data length attribute shall be set by the initiator, and shall not be overwritten by any interconnect component or target.

d) The data length attribute shall not be set to 0. In order to transfer zero bytes, the command attribute should be set to TLM_IGNORE_COMMAND.

e) When using the standard socket classes of the interoperability layer (or classes derived from these) for burst transfers, the word length for each transfer shall be determined by the BUSWIDTH template parameter of the socket. BUSWIDTH is independent of the data length attribute. BUSWIDTH shall be expressed in bits. If the data length is less than or equal to the BUSWIDTH / 8, the transaction is effectively modeling a single-word transfer, and if greater, the transaction is effectively modeling a burst. A single transaction can be passed through sockets of different bus widths. The BUSWIDTH may be used to calculate the latency of the transfer.

f) The target may or may not support transactions with data length greater than the word length of the target, whether the word length is given by the BUSWIDTH template parameter or by some other value.

g) If the target is unable to execute the transaction with the given data length, it shall generate a standard error response, and it shall not modify the contents of the data array. The recommended response status is TLM_BURST_ERROR_RESPONSE.

h) The default value of the data length attribute shall be 0, which is an invalid value. Hence, the data length attribute shall be set explicitly before the transaction object is passed through an interface method call.

7.12 Byte enable pointer attribute

a)   The method **set_byte_enable_ptr** shall set the pointer to the byte enable array to the value passed as an argument. The method **get_byte_enable_ptr** shall return the current value of the byte enable pointer attribute.

b)   The elements in the byte enable array shall be interpreted as follows. A value of 0 shall indicate that that corresponding byte is disabled, and a value of 0xff shall indicate that the corresponding byte is enabled. The meaning of all other values shall be undefined. The value 0xff has been chosen so that the byte enable array can be used directly as a mask. The two macros TLM_BYTE_DISABLED and TLM_BYTE_ENABLED are provided for convenience.

c)   Byte enables may be used to create burst transfers where the address increment between each beat is greater than the number of significant bytes transferred on each beat, or to place words in selected byte lanes of a bus. At a more abstract level, byte enables may be used to create "lacy bursts" where the data array of the generic payload has an arbitrary pattern of holes punched in it.

d)   The byte enable mask may be defined by a small pattern applied repeatedly or by a large pattern covering the whole data array. See 7.13 Byte enable length attribute

e)   The number of elements in the byte enable array shall be given by the byte enable length attribute.

f)   The byte enable pointer may be set to 0, the null pointer, in which case byte enables shall not be used for the current transaction, and the byte enable length shall be ignored.

g)   If byte enables are used, the byte enable pointer attribute shall be set by the initiator, the storage for the byte enable array shall be allocated by the initiator, the contents of the byte enable array shall be set by the initiator, and neither the byte enable pointer nor the contents of the byte enable array shall be overwritten by any interconnect component or target.

h)   If the byte enable pointer is non-null, the target shall either implement the semantics of the byte enable as defined below or shall generate a standard error response. The recommended response status is TLM_BYTE_ENABLE_ERROR_RESPONSE.

i)   In the case of a write command, any interconnect component or target should ignore the values of any disabled bytes in the data array. It is recommended that disabled bytes have no effect on the behavior of any interconnect component or target. The initiator may set those bytes to any values, since they are going to be ignored.

j)   In the case of a write command, when a target is doing a byte-by-byte copy from the transaction data array to a local array, the target should not modify the values of bytes in the local array corresponding to disabled bytes in the generic payload.

k)   In the case of a read command, any interconnect component or target should not modify the values of disabled bytes in the data array. The initiator can assume that disabled bytes will not be modified by any interconnect component or target.

l)   In the case of a read command, when a target is doing a byte-by-byte copy from a local array to the transaction data array, the target should ignore the values of bytes in the local array corresponding to disabled bytes in the generic payload.

m)   If the application needs to violate these semantics for byte enables, or to violate any other semantics of the generic payload as defined in this document, the recommended approach would be to create a new

77

protocol traits class. See 7.2.2 Define a new protocol traits class containing a typedef for tlm_generic_payload

n)   The default value of the byte enable pointer attribute shall be 0, the null pointer.

7.13   Byte enable length attribute

a)   The method **set_byte_enable_length** shall set the byte enable length attribute to the value passed as an argument. The method **get_byte_enable_length** shall return the current value of the byte enable length attribute.

b)   For a read command or a write command, the target shall interpret the byte enable length attribute as the number of elements in the byte enable array.

c)   The byte enable length attribute shall be set by the initiator, and shall not be overwritten by any interconnect component or target.

d)   The byte enable to be applied to a given element of the data array shall be calculated using the formula **byte_enable_array_index = data_array_index % byte_enable_length**. In other words, the byte enable array is applied repeatedly to the data array.

e)   The byte enable length attribute may be greater than the data length attribute, in which case any superfluous byte enables should not affect the behavior of a read or write command, but could be used by extensions.

f)   If the byte enable pointer is 0, the null pointer, then the value of the byte enable length attribute shall be ignored by any interconnect component or target. If the byte enable pointer is non-0, the byte enable length shall be non-0.

g)   If the target is unable to execute the transaction with the given byte enable length, it shall generate a standard error response. The recommended response status is TLM_BYTE_ENABLE_ERROR_RESPONSE.

h)   The default value of the byte enable length attribute shall be 0.

7.14   Streaming width attribute

a)   The method **set_streaming_width** shall set the streaming width attribute to the value passed as an argument. The method **get_streaming_width** shall return the current value of the streaming width attribute.

b)   For a read command or a write command, the target shall interpret and act upon the current value of the streaming width attribute

c)   Streaming affects the way a component should interpret the data array. A stream consists of a sequence of data transfers occurring on successive notional beats, each beat having the same start address as given by the generic payload address attribute. The streaming width attribute shall determine the width of the stream, that is, the number of bytes transferred on each beat. In other words, streaming affects the local address associated with each byte in the data array. In all other respects, the organization of the data array is unaffected by streaming.

d) The bytes within the data array have a corresponding sequence of local addresses within the component accessing the generic payload transaction. The lowest address is given by the value of the address attribute. The highest address is given by the formula **address_attribute + streaming_width − 1**. The address to or from which each byte is being copied in the target shall be set to the value of the address attribute at the start of each beat.

e) With respect to the interpretation of the data array, a single transaction with a streaming width shall be functionally equivalent to a sequence of transactions each having the same address as the original transaction, each having a data length attribute equal to the streaming width of the original, and each with a data array that is a different subset of the original data array on each beat. This subset effectively steps down the original data array maintaining the sequence of bytes.

f) A streaming width of 0 shall be invalid. If a streaming transfer is not required, the streaming width attribute should be set to a value greater than or equal to the value of the data length attribute.

g) The value of the streaming width attribute shall have no affect on the length of the data array or the number of bytes stored in the data array.

h) Width conversion issues may arise when the streaming width is different from the width of the socket (when measured as a number of bytes). See 7.17 Endianness

i) If the target is unable to execute the transaction with the given streaming width, it shall generate a standard error response. The recommended response status is TLM_BURST_ERROR_RESPONSE.

j) Streaming may be used in conjunction with byte enables, in which case the streaming width would typically be equal to the byte enable length. It would also make sense to have the streaming width a multiple of the byte enable length. Having the byte enable length a multiple of the streaming width would imply that different bytes were enabled on each beat.

k) The streaming width attribute shall be set by the initiator, and shall not be overwritten by any interconnect component or target.

l) The default value of the streaming width attribute shall be 0.

7.15 DMI allowed attribute

a) The method **set_dmi_allowed** shall set the DMI allowed attribute to the value passed as an argument. The method **is_dmi_allowed** shall return the current value of the DMI allowed attribute.

b) The DMI allowed attribute provides a hint to an initiator that it may try to obtain a direct memory pointer. The target should set this attribute to true if the transaction at hand could have been done through DMI. See 4.2.9 Optimization using a DMI hint

c) The default value of the DMI allowed attribute shall be **false**.

7.16 Response status attribute

a) The method **set_response_status** shall set the response status attribute to the value passed as an argument. The method **get_response_status** shall return the current value of the response status attribute.

b) The method **is_response_ok** shall return **true** if and only if the current value of the response status attribute is TLM_OK_RESPONSE. The method **is_response_error** shall return **true** if and only if the current value of the response status attribute is not equal to TLM_OK_RESPONSE.

c) The method **get_response_string** shall return the current value of the response status attribute as a text string.

d) As a general principle, a target is recommended to support every feature of the generic payload, but in the case that it does not, it shall generate the standard error response. See 7.16.1 The standard error response

e) The response status attribute shall be set to TLM_INCOMPLETE_RESPONSE by the initiator, and may or may not be overwritten by the target. The response status attribute shall not be overwritten by an interconnect component. The value TLM_INCOMPLETE_RESPONSE should be used to indicate that the component acting as the target did not attempt to execute the command, as might be the case if the response was returned from a component that usually acts as an interconnect component. But note that such a component would be allowed to set the response status attribute to any error response, because it is acting as a target.

f) If the target is able to execute the command successfully, it shall set the response status attribute to TLM_OK_RESPONSE. Otherwise, the target may set the response status to any of the six error responses listed in the table below. The target should choose the appropriate error response depending on the cause of the error.

Error response	Interpretation
TLM_INCOMPLETE_RESPONSE	Target did not attempt to execute the command
TLM_ADDRESS_ERROR_RESPONSE	Target was unable to act upon the address attribute, or address out-of-range
TLM_COMMAND_ERROR_RESPONSE	Target was unable to execute the command
TLM_BURST_ERROR_RESPONSE	Target was unable to act upon the data length or streaming width
TLM_BYTE_ENABLE_ERROR_RESPONSE	Target was unable to act upon the byte enable
TLM_GENERIC_ERROR_RESPONSE	Any other error

g) If a target detects an error but is unable to select a specific error response, it may set the response status to TLM_GENERIC_ERROR_RESPONSE.

h) The default value of the response status attribute shall be TLM_INCOMPLETE_RESPONSE.

i) In the case of TLM_IGNORE_COMMAND, a target that has received the transaction and would have been in a position to execute a read or write command should return TLM_OK_RESPONSE. Otherwise,

the target may choose, at its discretion, to set an error response using the same criteria it would have applied for a read or write command. For example, a target that does not support byte enables would be permitted (but not obliged) to return TLM_BYTE_ENABLE_ERROR_RESPONSE.

j) The presence of a generic payload extension or extended phase may cause a target to return a different response status, provided that the rules concerning ignorable extensions are honored. In other words, within the base protocol it is allowable that an extension may cause a command to fail, but it is also allowable that the target may ignore the extension and thus have the command succeed.

k) The target shall be responsible for setting the response status attribute at the appropriate point in the lifetime of the transaction. In the case of the blocking transport interface, this means before returning control from **b_transport**. In the case of the non-blocking transport interface and the base protocol, this means before sending the BEGIN_RESP phase or returning a value of TLM_COMPLETED.

l) It is recommended that the initiator should always check the response status attribute on receiving a transition to the BEGIN_RESP phase or after the completion of the transaction. An initiator *may* choose to ignore the response status if it is known in advance that the value will be TLM_OK_RESPONSE, perhaps because it is known in advance that the initiator is only connected to targets that always return TLM_OK_RESPONSE, but in general this will not be the case. In other words, the initiator ignores the response status at its own risk.

m) A target has some latitude when selecting an error response. For example, if the command and address attributes are in error, a target may be justified in setting any of TLM_ADDRESS_ERROR_RESPONSE, TLM_COMMAND_ERROR_RESPONSE, or TLM_GENERIC_ERROR_RESPONSE. When using the response status to determine its behavior an initiator should not rely on the distinction between the six categories of error response alone, although an initiator may use the response status to determine the content of diagnostic messages printed for the benefit of the user.

7.16.1 The standard error response

When a target receives a generic payload transaction, the target should perform one and only one of the following actions:

a) Execute the command represented by the transaction, honoring the semantics of the generic payload attributes, and honoring the publicly documented semantics of the component being modeled, and set the response status to TLM_OK_RESPONSE.

b) Set the response status attribute of the generic payload to one of the five error responses as described above.

c) Generate a report using the standard SystemC report handler with any of the four standard SystemC severity levels indicating that the command has failed or been ignored, and set the response status to TLM_OK_RESPONSE.

It is recommended that the target should perform exactly one of these actions, but an implementation is not obliged or permitted to enforce this recommendation.

It is recommended that a target for a transaction type other than the generic payload should follow this same principle, that is, execute the command as expected, or generate an error response using an attribute of the transaction, or generate a SystemC report. However, the details of the semantics and the error response mechanism for such a transaction are outside the scope of this standard.

The conditions for satisfying point a) above are determined by the expected behavior of the target component as would be visible to a user of that component. The attributes of the generic payload have defined semantics which correspond to conventional usage in the context of memory-mapped buses, but which do not necessarily assume that the target behaves as a random-access memory. There are many subtle corner cases. For example:

i.   A target may have a memory-mapped register that supports both read and write commands, but the write command is non-sticky, that is, write modifies the state of the target, but a write followed by read will not return the data just written but some other value determined by the state of the target. If this is the normal expected behavior of the component, it is covered by point a).

ii.   A target may implement the write command to set a bit whilst totally ignoring the value of the data attribute. If this is the normal expected behavior of the target, it is covered by point a)

iii.   A read-only memory may ignore the write command without signalling an error to the initiator using the response status attribute. Since the write command is not changing the state of the target but is being ignored altogether, the target should at least generate a SystemC report with severity SC_INFO or SC_WARNING.

iv.   A target should not under any circumstances implement the write command by performing a read, or vice versa. That would be a fundamental violation of the semantics of the generic payload.

v.   A target may implement the read command according to the intent of the generic payload, but with additional side-effects. This is covered by point a).

vi.   A target with a set of memory-mapped registers forming an addressable register file receives a write command with an out-of-range address. The target should either set the response status attribute of the transaction to TLM_ADDRESS_ERROR_RESPONSE or generate a SystemC report.

vii.   A passive simulation bus monitor target receives a transaction with an address that is outside the physical range of the bus being modeled. The target may log the erroneous transaction for post-processing under point a) and not generate an error response under points b) or c). Alternatively, the target may generate a report under point c).

In other words, the distinction between points a), b) and c) is ultimately a pragmatic judgement to be made on a case-by-case basis, but the definitive rule for the generic payload is that a target should always perform exactly one of these actions.

Example

   // **Showing generic payload with command, address, data, and response status**

   // **The initiator**
   void thread() {
      tlm::tlm_generic_payload trans;   // Construct default generic payload
      sc_time delay;

      trans.**set_command**(tlm::TLM_WRITE_COMMAND);   // A write command
      trans.**set_data_length**(4);   // Write 4 bytes
      trans.**set_byte_enable_ptr**(0);   // Byte enables unused
      trans.**set_streaming_width**(4);   // Streaming unused


```
    for (int i = 0; i < RUN_LENGTH; i += 4) {                 // Generate a series of transactions
        int word = i;
        trans.set_address(i);                                 // Set the address
        trans.set_data_ptr( (unsigned char*)(&word) );        // Write data from local variable 'word'
        trans.set_dmi_allowed(false);                         // Clear the DMI hint
        trans.set_response_status( tlm::TLM_INCOMPLETE_RESPONSE ); // Clear the response status

        init_socket->b_transport(trans, delay);

        if (trans.is_response_error() )                       // Check return value of b_transport
            SC_REPORT_ERROR("TLM-2.0", trans.get_response_string().c_str());
        ...
    }
    ...


    // The target
    virtual void b_transport( tlm::tlm_generic_payload& trans, sc_core::sc_time& t)
    {
        tlm::tlm_command   cmd   = trans.get_command();
        sc_dt::uint64      adr   = trans.get_address();
        unsigned char*     ptr   = trans.get_data_ptr();
        unsigned int       len   = trans.get_data_length();
        unsigned char*     byt   = trans.get_byte_enable_ptr();
        unsigned int       wid   = trans.get_streaming_width();

        if (adr+len > m_length) {                      // Check for storage address overflow
            trans.set_response_status( tlm::TLM_ADDRESS_ERROR_RESPONSE );
            return;
        }
        if (byt) {                                     // Target unable to support byte enable attribute
            trans.set_response_status( tlm::TLM_BYTE_ENABLE_ERROR_RESPONSE );
            return;
        }
        if (wid < len) {                               // Target unable to support streaming width attribute
            trans.set_response_status( tlm::TLM_BURST_ERROR_RESPONSE );
            return;
        }

        if (cmd == tlm::TLM_WRITE_COMMAND)             // Execute command
            memcpy(&m_storage[adr], ptr, len);
        else if (cmd == tlm::TLM_READ_COMMAND)
            memcpy(ptr, &m_storage[adr], len);

        trans.set_response_status( tlm::TLM_OK_RESPONSE );  // Successful completion
    }
```

// **Showing generic payload with byte enables**

```
// The initiator
void thread() {
    tlm::tlm_generic_payload trans;
    sc_time delay;

    static word_t byte_enable_mask = 0x0000fffful;  // MSB..LSB regardless of host-endianness

    trans.set_command(tlm::TLM_WRITE_COMMAND);
    trans.set_data_length(4);
    trans.set_byte_enable_ptr( reinterpret_cast<unsigned char*>( &byte_enable_mask ) );
    trans.set_byte_enable_length(4);
    trans.set_streaming_width(4);
    ...
...
// The target
virtual void b_transport(
    tlm::tlm_generic_payload& trans, sc_core::sc_time& t)
{
    tlm::tlm_command   cmd   = trans.get_command();
    sc_dt::uint64      adr   = trans.get_address();
    unsigned char*     ptr   = trans.get_data_ptr();
    unsigned int       len   = trans.get_data_length();
    unsigned char*     byt   = trans.get_byte_enable_ptr();
    unsigned int       bel   = trans.get_byte_enable_length();
    unsigned int       wid   = trans.get_streaming_width();

    if (cmd == tlm::TLM_WRITE_COMMAND) {
        if (byt) {
            for (unsigned int i = 0; i < len; i++)          // Byte enable applied repeatedly up data array
                if ( byt[i % bel] == TLM_BYTE_ENABLED )
                    m_storage[adr+i] = ptr[i];              // Byte enable [i] corresponds to data ptr [i]
        }
        else
            memcpy(&m_storage[adr], ptr, len);          // No byte enables
    } else if (cmd == tlm::TLM_READ_COMMAND) {
        if (byt) {                                      // Target does not support read with byte enables
            trans.set_response_status( tlm::TLM_BYTE_ENABLE_ERROR_RESPONSE );
            return;
        }
        else
            memcpy(ptr, &m_storage[adr], len);
    }
```

```
        trans.set_response_status( tlm::TLM_OK_RESPONSE );
    }
```

7.17 Endianness

7.17.1 Introduction

When using the generic payload to transfer data between initiator and target, both the endianness of the host machine (host endianness) and the endianness of the initiator and target being modeled (modeled endianness) are relevant. This clause defines rules to ensure interoperability between initiators and targets using the generic payload, so is specifically concerned with the organization of the generic payload data array and byte enable array. However, the rules given here may have an impact on some of the choices made in modeling endianness beyond the immediate scope of the generic payload.

A general principle in the TLM-2.0 approach to endianness is that the organization of the generic payload data array depends only on information known locally within each initiator, interconnect component or target. In particular, it depends on the width of the local socket through which the transaction is sent or received, the endianness of the host computer, and the endianness of the component being modeled.

The organization of the generic payload and the approach to endianness has been chosen to maximize simulation efficiency in certain common system scenarios, particularly mixed-endian systems. The rules given below dictate the organization of the generic payload, and this is independent of the organization of the system being modeled. For example, a "word" within the generic payload need not necessarily correspond in internal representation with any "word" within the modeled architecture.

At a macroscopic level, the main principle is that the generic payload assumes components in a mixed-endian system to be wired up MSB to MSB (most-significant byte), and LSB to LSB (least-significant byte). In other words, if a word is transferred between components of differing endianness, the MSB ... LSB relationship is preserved, but the *local address* of each byte as seen within each component will necessarily change using the transformation generally called *address swizzling*. This is true within both the modeled system and the TLM-2.0 model. On the other hand, if a mixed-endian system is wired such that the local addresses are invariant within each component (that is, each byte has the same local address when seen from any component), then an explicit byte swap would need to be inserted in the TLM-2.0 model.

In order to achieve interoperability with respect to the endianness of the generic payload arrays, it is only necessary to obey the rules given in this clause. A set of helper functions is provided to assist with the organization of the data array. See 7.19 Helper functions for endianness conversion

7.17.2 Rules

a)   In the following rules, the generic payload data array is denoted as **data** and the generic payload byte enable array as **be**.

b)   When using the standard socket classes of the interoperability layer (or classes derived from these), the contents of the data and byte enable arrays shall be interpreted using the BUSWIDTH template parameter of the socket through which the transaction is sent or received locally. The effective word length shall be calculated as (BUSWIDTH + 7)/8 bytes, and in the following rules is denoted as **W**.

c)   This quantity **W** defines the length of a *word* within the data array, each word being the amount of data that could be transferred through the local socket on a single beat. The data array may contain a single word, a part-word, or several contiguous words or part-words. Only the first and last words in the data

array may be part-words. This description refers to the internal organization of the generic payload, not to the organization of the architecture being modeled.

d)   If a given generic payload transaction object is passed through sockets of different widths, the data array word length would appear different when calculated from the point of view of different sockets (see the description of width conversion below).

e)   The order of the bytes within each word of the data array shall be host-endian. That is, on a little-endian host processor, within any given word **data[n]** shall be less significant than **data[n+1]**, and on a big-endian host processor, **data[n]** shall be the more significant than **data[n+1]**.

f)   The word boundaries in the data array shall be address-aligned, that is, they shall fall on addresses that are integer multiples of the word length **W**. However, neither the address attribute nor the data length attribute are required to be multiples of the word length. Hence the possibility that the first and last words in the data array could be part-words.

g)   The order of the words within the data array shall be determined by their addresses in the memory map of the modeled system. For array index values less than the value of the streaming width attribute, the local addresses of successive words shall be in increasing order, and (excluding any leading part-word) shall equal **address_attribute - (address_attribute % W) + NW**, where **N** is a non-negative integer, and **%** indicates remainder on division.

h)   In other words, using the notation {a,b,c,d} to list the elements of the data array in increasing order of array index, and using LSB_N to denote the least significant byte of the Nth word, on a little-endian host bytes are stored in the order {..., MSB_0, LSB_1, ..., MSB_1, LSB_2, ...}, and on a big-endian host {... LSB_0, MSB_1, ... LSB_1, MSB_2, ...}, where the number of bytes in each full word is given by **W**, and the total number of bytes is given by the **data_length** attribute.

i)   The above rules effectively mean that initiators and targets are connected LSB-to-LSB, MSB-to-MSB. The rules have been chosen to give optimal simulation speed in the case where the majority of initiators and targets are modeled using host endianness whatever their native endianness, also known as "arithmetic mode".

j)   It is strongly recommended that applications should be independent of host endianness, that is, should model the same behavior when run on a host of either endianness. This may require the use of helper functions or conditional compilation.

k)   If an initiator or target is modeled using its native endianness and that is different from host endianness, it will be necessary to swap the order of bytes within a word when transferring data to or from the generic payload data array. Helper functions are provided for this purpose.

l)   For example, consider the following SystemC code fragment, which uses the literal value 0xAABBCCDD to initialize the generic payload data array:

```
int data = 0xAABBCCDD;
trans.set_data_ptr( reinterpret_cast<unsigned char*>( &data ) );
trans.set_data_length(4);
trans.set_address(0);
socket->b_transport(trans, delay);
```

m) The C++ compiler will interpret the literal 0xAABBCCDD in host-endian form. In either case, the MSB has value 0xAA and the LSB has value 0xDD. Assuming this is the intent, the code fragment is valid and is independent of host endianness. However, the array index of the four bytes will differ depending on host endianness. On a little-endian host, data[0] = 0xDD, and on a big-endian host, data[0] = 0xAA. The correspondence between local addresses in the modeled system and array indexes will differ depending whether modeled endianess and host endianness are equal:

Little-endian model and little-endian host: data[0] is 0xDD and local address 0
Big-endian model   and little-endian host: data[0] is 0xDD and local address 3
Little-endian model and big-endian host: data[0] is 0xAA and local address 3
Big-endian model   and big-endian host: data[0] is 0xAA and local address 0

n) Code such as the fragment shown above would not be portable to a host computer that uses neither little nor big endianness. In such a case, the code would have to be re-written to access the generic payload data array using byte addressing only.

o) When a little-endian and a big-endian model interpret a given generic payload transaction, then by definition they will agree on which is the MSB and LSB of a word, but they will each use different local addresses to access the bytes of the word.

p) Neither the data length attribute nor the address attribute are required to be integer multiples of **W**. However, having address and data length aligned with word boundaries and having **W** be a power of 2 considerably simplifies access to the data array. Just to emphasize the point, it would be perfectly in order for a generic payload transaction to have an address and data length that indicated three bytes in the middle of a 48-bit socket. If a particular target is unable to support a given address attribute or data length, it should generate a standard error response. See 7.16 Response status attribute

q) For example, on a little-endian host and with **W = 4**, **address = 1**, and **data_length =** 4, the first word would contain 3 bytes at addresses 1...3, and the second word 1 byte at address 4.

r) Single byte and part-word transfers may be expressed using non-aligned addressing. For example, given **W** = 8, address = 5, and **data** = {1,2}, the two bytes with local addresses 5 and 6 are accessed in an order dependent on endianness.

s) Part-word and non-aligned transfers can always be expressed using integer multiples of **W** together with byte enables. This implies that a given transaction may have several equally valid generic payload representations. For example, given a little-endian host and a little-endian initiator,

address = 2, **W** = 4, **data** = {1} is equivalent to
address = 0, **W** = 4, **data** = {x, x, 1, x}, and **be** = {0, 0, 0xff, 0}

address = 2, **W** = 4, **data** = {1,2,3,4} is equivalent to
address = 0, **W** = 4, **data** = {x, x, 1, 2, 3, 4, x, x}, and **be** = {0, 0, 0xff, 0xff, 0xff, 0xff, 0, 0}.

t) For part-word access, the necessity to use byte enables is dependent on endianness. For example, given the intent to access the whole of the first word and the LSB of the second word, given a little-endian host this might be expressed as

address = 0, **W** = 4, **data** = {1,2,3,4,5}

Given a big-endian host, the equivalent would be

address = 0, **W** = 4, **data** = {4,3,2,1,x,x,x,5}, **be** = {0xff, 0xff, 0xff, 0xff, 0, 0, 0, 0xff }.

u) When two sockets are bound together, they necessarily have the same BUSWIDTH. However, a transaction may be forwarded from a target socket to an initiator socket of a different bus width. In this case, width conversion of the generic payload transaction must be considered. Any width conversion has its own intrinsic endianness, depending on whether the least- or most significant byte of the wider socket is picked out first.

Width conversion

Figure 12

Little-endian host
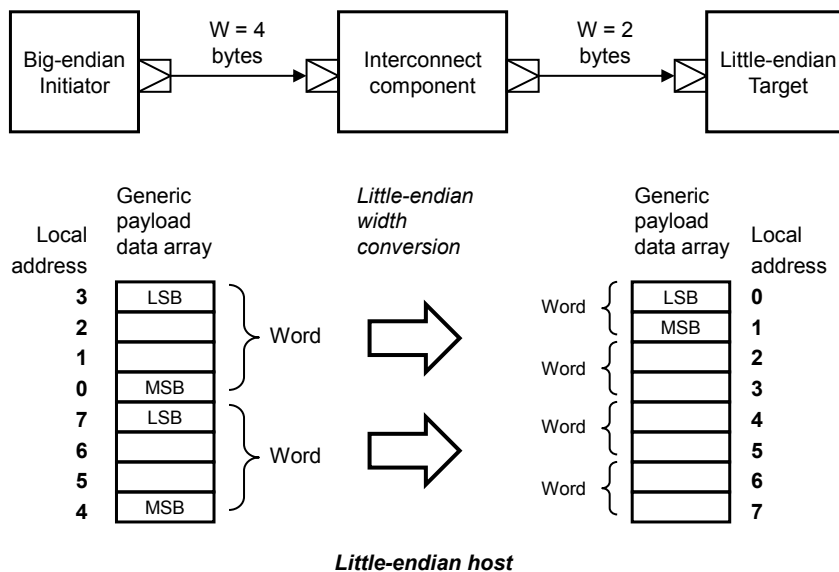
v) When the endianness chosen for a width conversion matches the host endianness, the width conversion is effectively free, meaning that a single transaction object can be forwarded from socket-to-socket without modification. Otherwise, two separate generic payload transaction objects would be required. In figure 12, the width conversion between the 4-byte socket and the 2-byte socket uses host-endianness, moving the less-significant bytes to lower addresses whilst retaining the host-endian byte order within each word. The initiator and target both access the same sequence of bytes in the data array, but their local addressing schemes are quite different.

w) If a width conversion is performed from a narrower socket to a wider socket, the choice has to be made as to whether or not to perform address alignment on the outgoing transaction. Performing address alignment will always necessitate the construction of a new generic payload transaction object.

x) Similar width conversion issues arise when the streaming width attribute is non-zero but different from **W**. A choice has to be made as to the order in which to read off the bytes down the data array depending on host endianness and the desired endianness of the width conversion.

7.18 Helper functions to determine host endianness

7.18.1 Introduction

A set of helper functions is provided to determine the endianness of the host computer. These are intended for use when creating or interpreting the generic payload data array.

7.18.2 Definition

```
namespace tlm {

enum tlm_endianness {
    TLM_UNKNOWN_ENDIAN, TLM_LITTLE_ENDIAN, TLM_BIG_ENDIAN };

inline tlm_endianness get_host_endianness(void);
inline bool host_has_little_endianness(void);
inline bool has_host_endianness(tlm_endianness endianness);

} // namespace tlm
```

7.18.3 Rules

a) The function **get_host_endianness** shall return the endianness of the host.

b) The function **host_has_little_endianness** shall return the value true if and only if the host is little-endian.

c) The function **has_host_endianness** shall return the value true if and only if the endianness of the host is the same as that indicated by the argument.

d) If the host is neither little- nor big-endian, the value returned from the above three functions shall be undefined.

7.19 Helper functions for endianness conversion

7.19.1 Introduction

The rules governing the organization of the generic payload data array are well-defined, and in many simple cases, writing host-independent C++ code to create and interpret the data array is a straightforward task. However, the rules do depend on the relationship between the endianness of the modeled component and host endianness, so creating host-independent code can become quite complex in cases involving non-aligned addressing and data word widths that differ from the socket width. A set of helper functions is provided to assist with this task.

With respect to endianness, interoperability depends only on the endianness rules being followed. Use of the helper functions is not necessary for interoperability.

The motivation behind the endianness conversion functions is to permit the C++ code that creates a generic payload transaction for an initiator to be written once with little regard for host endianness, and then to have the transaction converted to match host endianness with a single function call. Each conversion function takes an existing generic payload transaction and modifies that transaction in-place. The conversion functions are organised in pairs, a *to_hostendian* function and a *from_hostendian* function, which should always be used together. The *to_hostendian* function should be called by an initiator before sending a transaction through a transport interface, and *from_hostendian* on receiving back the response.

Four pairs of functions are provided, the *_generic* pair being the most general and powerful, and the *_word*, *_aligned* and *_single* functions being variants that can only handle restricted cases. The transformation performed by the *_generic* functions is relatively computationally expensive, so the other functions should be preferred for efficiency wherever possible.

The conversion functions provide sufficient flexibility to handle many common cases, including both *arithmetic mode* and *byte order mode*. *Arithmetic mode* is where a component stores data words in host-endian format for efficiency when performing arithmetic operations, regardless of the endianness of the component being modeled. *Byte order mode* is where a component stores bytes in an array in ascending address order, disregarding host endianness. The use of arithmetic mode is recommended for simulation speed. Byte order mode may necessitate byte swapping when copying data to and from the generic payload data array.

The conversion functions use the concept of a *data word*. The data word is independent of both the TLM-2.0 socket width and the word width of the generic payload data array. The data word is intended to represent a register that stores bytes in host-endian order within the component model (regardless of the endianness of the component being modeled). If the data word width is different to the socket width, the *hostendian* functions may have to perform an endianness conversion. If the data word is just one byte wide, the *hostendian* functions will effectively perform a conversion from and to *byte order mode*.

In summary, the approach to be taken with the *hostendian* conversion functions is to write the initiator code *as if* the endianness of the host computer matched the endianness of the component being modeled, while keeping the bytes within each data word in actual host-endian order. For data words wider than the host machine word length, use an array in host-endian order. Then if host endianness differs from modeled endianness, simply call the *hostendian* conversion functions.

7.19.2 Definition

namespace tlm {

template<class DATAWORD>
inline void **tlm_to_hostendian_generic**(tlm_generic_payload *, unsigned int);
template<class DATAWORD>
inline void **tlm_from_hostendian_generic**(tlm_generic_payload *, unsigned int);

template<class DATAWORD>
inline void **tlm_to_hostendian_word**(tlm_generic_payload *, unsigned int);
template<class DATAWORD>
inline void **tlm_from_hostendian_word**(tlm_generic_payload *, unsigned int);

template<class DATAWORD>
inline void **tlm_to_hostendian_aligned**(tlm_generic_payload *, unsigned int);
template<class DATAWORD>
inline void **tlm_from_hostendian_aligned**(tlm_generic_payload *, unsigned int);

template<class DATAWORD>
inline void **tlm_to_hostendian_single**(tlm_generic_payload *, unsigned int);
template<class DATAWORD>
inline void **tlm_from_hostendian_single**(tlm_generic_payload *, unsigned int);

inline void **tlm_from_hostendian**(tlm_generic_payload *);

} // namespace tlm

7.19.3 Rules

a) The first argument to a function of the form *to_hostendian* should be a pointer to a generic payload transaction object that would be valid if it were sent through a transport interface. The function should only be called after constructing and initializing the transaction object and before passing it to an interface method call.

b) The first argument to a function of the form *from_hostendian* shall be a pointer to a generic payload transaction object previously passed to *to_hostendian*. The function should only be called when the initiator receives a response for the given transaction or the transaction is complete. Since the function may modify the transaction and its arrays, it should only be called at the end of the lifetime of the transaction object.

c) If a *to_hostendian* function is called for a given transaction, the corresponding *from_hostendian* function should also be called with the same template and function arguments. Alternatively, the function **tlm_from_hostendian**(tlm_generic_payload *) can be called for the given transaction. This function uses additional context information stored with the transaction object (as an ignorable extension) to recover the template and function argument values, but is marginally slower in execution.

d) The second argument to a *hostendian* function should be the width of the local socket through which the transaction is passed, expressed in bytes. This is equivalent to the word length of the generic payload data array with respect to the local socket. This shall be a power of 2.

e) The template argument to a *hostendian* function should be a type representing the internal initiator data word for the endianness conversion. The expression sizeof(DATAWORD) is used to determine the width of the data word in bytes, and the assignment operator of type DATAWORD is used during copying. sizeof(DATAWORD) shall be a power of 2.

f) The implementation of *to_hostendian* adds an extension to the generic payload transaction object to store context information. This means that *to_hostendian* can only be called once before calling *from_hostendian*.

g) The following constraints are common to every pair of *hostendian* functions. The term *integer multiple* means 1 x , 2 x , 3 x , ... and so forth:

Socket width shall be a power of 2

Data word width shall be a power of 2

The streaming width attribute shall be an integer multiple of the data word width

The data length attribute shall be an integer multiple of the streaming width attribute

h) The *hostendian_generic* functions are not subject to any further specific constraints. In particular, they support byte enables, streaming, and non-aligned addresses and word widths.

i) The remaining pairs of functions, namely *hostendian_word*, *hostendian_aligned*, and *hostendian_single*, all share the following additional constraints:

Data word width shall be no greater than socket width, and as a consequence, socket width shall be a power-of-2 multiple of data word width.

The streaming width attribute shall equal the data length attribute. That is, streaming is not supported.

Byte enable granularity shall be no finer than data word width. That is, the bytes in a given data word shall be either all enabled or all disabled.

If byte enables are present, the byte enable length attribute shall equal the data length attribute.

j) The *hostendian_aligned* functions alone are subject to the following additional constraints:

The address attribute shall be an integer multiple of the socket width.

The data length attribute shall be an integer multiple of the socket width.

k) The *hostendian_single* functions alone are subject to the following additional constraints:

The data length attribute shall equal the data word width.

The data array shall not cross a data word boundary, and as a consequence, shall not cross a socket boundary.

7.20 Generic payload extensions

7.20.1 Introduction

The extension mechanism is an integral part of the generic payload, and is not intended to be used separately from the generic payload. Its purpose is to permit attributes to be added to the generic payload. Extensions can be ignorable or non-ignorable, mandatory or non-mandatory.

7.20.1.1 Ignorable extensions

Being *ignorable* means that any component other than the component that added the extension is permitted to behave as if the extension were absent. As a consequence, the component that added the ignorable extension cannot rely on any other component reacting in any way to the presence of the extension, and a component receiving an ignorable extension cannot rely on other components having recognized that extension. This definition applies to generic payload extensions and to extended phases alike.

A component shall not fail and shall not generate an error response because of the absence of an ignorable extension. In other words, ignorable extensions are also non-mandatory extensions. A component may fail or generate an error response because of the presence of an ignorable extension, but also has the choice of ignoring the extension.

In general, an ignorable extension can be thought of as one for which there exists an obvious and safe default value such that any interconnect component or target can behave normally in the absence of the given extension by assuming the default value. An example might be the privilege level associated with a transaction, where the default is the lowest level.

Ignorable extensions may be used to transport auxiliary, side-band, or simulation-related information or meta-data. For example, a unique transaction identifier, the wall-time when the transaction was created, or a diagnostic filename.

Ignorable extensions are permitted by the base protocol.

7.20.1.2 Non-ignorable and mandatory extensions

A non-ignorable extension is an extension that every component receiving the transaction is obliged to act upon if present. A mandatory extension is an extension that is required to be present. Non-ignorable and mandatory extensions may be used when specializing the generic payload to model the details of a specific protocol. Non-ignorable and mandatory extensions require the definition of a new protocol traits class.

7.20.2 Rationale

The rationale behind the extension mechanism is twofold. Firstly, to permit TLM-2.0 sockets that carry variations on the core attribute set of the generic payload to be specialized with the same protocol traits class, thus allowing them to be bound together directly with no need for adaption or bridging. Secondly, to permit easy adaption between different protocols where both are based on the same generic payload and extension mechanism. Without the extension mechanism, the addition of any new attribute to the generic payload would require the definition of a new transaction class, leading to the need for multiple adapters. The extension

mechanism allows variations to be introduced into the generic payload, thus reducing the amount of coding work that needs to be done to traverse sockets that carry different protocols.

7.20.3 Extension pointers, objects and transaction bridges

An extension is an object of a type derived from the class **tlm_extension**. The generic payload contains an array of pointers to extension objects. Every generic payload object is capable of carrying a single instance of every type of extension.

The array-of-pointers to extensions has a slot for every registered extension. The **set_extension** method simply overwrites a pointer, and in principle can be called from an initiator, interconnect component, or target. This provides a very a flexible low-level mechanism, but is open to misuse. The ownership and deletion of extension objects has to be well-understood and carefully considered by the user.

When creating a transaction bridge between two separate generic payload transactions, it is the responsibility of the bridge to copy any extensions, if required, from the incoming transaction object to the outgoing transaction object, and to own and manage the outgoing transaction and its extensions. The same holds for the data array and byte enable array. The methods **deep_copy_from** and **update_original_from** are provided so that a transaction bridge can perform a deep copy of a transaction object, including the data and byte enable arrays and the extension objects. If the bridge adds further extensions to the outgoing transaction, those extensions would be owned by the bridge.

The management of extensions is described more fully in clause 7.5 Generic payload memory management.

7.20.4 Rules

a) An extension can be added by an initiator, interconnect or target component. In particular, the creation of extensions is not restricted to initiators.

b) Any number of extensions may be added to each instance of the generic payload.

c) In the case of an ignorable extension, any component (excepting the component that added the extension) is allowed to ignore the extension, and ignorable extensions are not mandatory extensions. Having a component fail because of either the absence of an ignorable extension or the absence of a response to an ignorable extension would destroy interoperability.

d) There is no built-in mechanism to enforce the presence of a given extension, nor is there a mechanism to ensure that an extension is ignorable.

e) The semantics of each extension shall be application-defined. There are no pre-defined extensions.

f) An extension shall be created by deriving a user-defined class from the class **tlm_extension**, passing the name of the user-defined class itself as a template argument to **tlm_extension**, then creating an object of that class. The user-defined extension class may include members which represent extended attributes of the generic payload.

g) The virtual method **free** of the class **tlm_extension_base** shall delete the extension object. This method may be overridden to implement user-defined memory management of extension, but this is not necessary.

h) The pure virtual function **clone** of class **tlm_extension** shall be defined in the user-defined extension class to clone the extension object, including any extended attributes. This **clone** method is intended for

use in conjunction with generic payload memory management. It shall create a copy of any extension object such that the copy can survive the destruction of the original object with no visible side-effects.

i)   The pure virtual function **copy_from** of class **tlm_extension** shall be defined in the user-defined extension class to modify the current extension object by copying the attributes of another extension object.

j)   The act of instantiating the class template **tlm_extension** shall cause the public data member **ID** to be initialized, and this shall have the effect of registering the given extension with the generic payload object and assigning a unique ID to the extension. The ID shall be unique across the whole executing program. That is, each instantiation of the class template **tlm_extension** shall have a distinct **ID**, whereas all extension objects of a given type shall share the same **ID**.

k)   The generic payload shall behave as if it stored pointers to the extensions in a re-sizable array, where the **ID** of the extension gives the index of the extension pointer in the array. Registering the extension with the generic payload shall reserve an array index for that extension. Each generic payload object shall contain an array capable of storing pointers to every extension registered in the currently executing program.

l)   The pointers in the extension array shall be null when the transaction is constructed.

m)   Each generic payload object can store a pointer to at most one object of any given extension type (but to many objects of different extensions types). (There exists a utility class **instance_specific_extension**, which enables a generic payload object to reference several extension objects of the same type. See 9.4 Instance-specific extensions.)

n)   The function **max_num_extensions** shall return the number of extension types, that is, the size of the extension array. As a consequence, the extension types shall be numbered from **0** to **max_num_extensions()-1**.

o)   The methods **set_extension**, **set_auto_extension,** **get_extension**, **clear_extension,** and **release_extension** are provided in several forms, each of which identify the extension to be accessed in different ways: using a function template, using an extension pointer argument, or using an ID argument. The functions with an ID argument are intended for specialist programming tasks such as when cloning a generic payload object, and not for general use in applications.

p)   The method **set_extension(T*)** shall replace the pointer to the extension object of type T in the array-of-pointers with the value of the argument. The argument shall be a pointer to a registered extension. The return value of the function shall be the previous value of the pointer in the generic payload that was replaced by this call, which may be a null pointer. The method **set_auto_extension(T*)** shall behave similarly, except that the extension shall be marked for automatic deletion.

q)   The method **set_extension(unsigned int, tlm_extension_base*)** shall replace the pointer to the extension object in the array-of-pointers at the array index given by the first argument with the value of the second argument. The given index shall have been registered as an extension ID, otherwise the behavior of the function is undefined. The return value of the function shall be the previous value of the pointer at the given array index, which may be a null pointer. The method **set_auto_extension(unsigned int, tlm_extension_base*)** shall behave similarly, except that the extension shall be marked for automatic deletion

r) In the presence of a memory manager, a call to **set_auto_extension** for a given extension is equivalent to a call to **set_extension** immediatedly followed by a call to **release_extension** for that same extension. In the absence of a memory manager, a call to **set_auto_extension** will cause a run-time error.

s) If an extension is marked for automatic deletion, the given extension object should be deleted or pooled by the implementation of the method **free** of a user-defined memory manager. Method **free** is called by method **release** of class **tlm_generic_payload** when the reference count of the transaction object reaches 0. The extension object may be deleted by calling method **reset** of class **tlm_generic_payload** or by calling method **free** of the extension object itself.

t) If the generic payload object already contained a non-null pointer to an extension of the type being set, then the old pointer is overwritten.

u) The method functions **get_extension(T*&)** and **T* get_extension()** shall each return a pointer to the extension object of the given type, if it exists, or a null pointer if it does not exist. The type **T** shall be a pointer to an object of a type derived from **tlm_extension**. It is not an error to attempt to retrieve a non-existent extension using this function template.

v) The method **get_extension(unsigned int)** shall return a pointer to the extension object with the ID given by the argument. The given index shall have been registered as an extension ID, otherwise the behavior of the function is undefined. If the pointer at the given index does not point to an extension object, the function shall return a null pointer.

w) The methods **clear_extension(const T*)** and **clear_extension()** shall remove the given extension from the generic payload object, that is, shall set the corresponding pointer in the extension array to null. The extension may be specified either by passing a pointer to an extension object as an argument, or by using the function template parameter type, for example **clear_extension<ext_type>()**. If present, the argument shall be a pointer to an object of a type derived from **tlm_extension**. Method **clear_extension** shall not delete the extension object.

x) The methods **release_extension(T*)** and **release_extension()** shall mark the extension for automatic deletion if the transaction object has a memory manager, or otherwise shall delete the given extension by calling the method **free** of the extension object and setting the corresponding pointer in the extension array to null. The extension may be specified either by passing a pointer to an extension object as an argument, or by using the function template parameter type, for example **release_extension<ext_type>()**. If present, the argument shall be a pointer to an object of a type derived from **tlm_extension**.

y) Note that the behavior of method **release_extension** depends upon whether or not the transaction object has a memory manager. With a memory manager, the extension is merely marked for automatic deletion, and continues to be accessible. In the absence of a memory manager, not only is the extension pointer cleared but also the extension object itself is deleted. Care should be taken not to release a non-existent extension object, because doing so will result in a run-time error.

z) The methods **clear_extension** and **release_extension** shall not be called for extensions marked for automatic deletion, for example, an extension set using **set_auto_extension** or already released using **release_extension**. Doing so may result in a run-time error.

aa) Each generic payload transaction should allocate sufficient space to store pointers to every registered extension. This can be achieved in one of two ways, either by constructing the transaction object *after* C++ static initialization, or by calling the method **resize_extensions** *after* static initialization but *before*

using the transaction object for the first time. In the former case, it is the responsibility of the generic payload constructor to set the size of the extension array. In the latter case, it is the responsibility of the application to call **resize_extensions** before accessing the extensions for the first time.

bb) The method **resize_extensions** shall increase the size of the extensions array in the generic payload to accommodate every registered extension.

Example

```
// Showing an ignorable extension

// User-defined extension class
struct ID_extension: tlm::tlm_extension<ID_extension>
{
    ID_extension() : transaction_id(0) {}

    virtual tlm_extension_base* clone() const {          // Must override pure virtual clone method
        ID_extension* t = new ID_extension;
        t->transaction_id = this->transaction_id;
        return t;
    }
                                                          // Must override pure virtual copy_from method
    virtual void copy_from(tlm_extension_base const &ext) {
        transaction_id = static_cast<ID_extension const &>(ext).transaction_id;
    }
    unsigned int transaction_id;
};

// The initiator
struct Initiator: sc_module
{   ...
    void thread() {
        tlm::tlm_generic_payload trans;

        ...
        ID_extension* id_extension = new ID_extension;
        trans.set_extension( id_extension );              // Add the extension to the transaction

        for (int i = 0; i < RUN_LENGTH; i += 4) {
            ...
            ++ id_extension->transaction_id;              // Increment the id for each new transaction
            ...
            socket->b_transport(trans, delay);
            ...

// The target
virtual void b_transport( tlm::tlm_generic_payload& trans, sc_core::sc_time& t)
{   ...
```

```
    ID_extension*    id_extension;
    trans.get_extension( id_extension );                  // Retrieve the extension
    if (id_extension) {                                   // Extension is not mandatory
        char txt[80];
        sprintf(txt, "Received transaction id %d", id_extension->transaction_id);
        SC_REPORT_INFO("TLM-2.0", txt);
     }
    ...
```

// **Showing a new protocol traits class with a mandatory extension**

```
struct cmd_extension: tlm::tlm_extension<cmd_extension>
{                                                     // User-defined mandatory extension class
    cmd_extension(): increment(false) {}
    virtual tlm_extension_base* clone() const {
        cmd_extension* t = new cmd_extension;
        t->increment = this->increment;
        return t;
    }
    virtual void copy_from(tlm_extension_base const &ext) {
        increment = static_cast<cmd_extension const &>(ext).increment;
    }
    bool increment;
};

struct my_protocol_types                              // User-defined protocol traits class
{
 typedef  tlm::tlm_generic_payload    tlm_payload_type;
 typedef  tlm::tlm_phase              tlm_phase_type;
};

struct Initiator: sc_module
{
    tlm_utils::simple_initiator_socket<Initiator, 32, my_protocol_types> socket;
    ...
    void thread() {
        tlm::tlm_generic_payload trans;
        cmd_extension* extension = new cmd_extension;
        trans.set_extension( extension );                 // Add the extension to the transaction
        ...
        trans.set_command(tlm::TLM_WRITE_COMMAND);  // Execute a write command
        socket->b_transport(trans, delay);
        ...
        trans.set_command(tlm::TLM_IGNORE_COMMAND);
        extension->increment = true;                      // Execute an increment command
        socket->b_transport(trans, delay);
```

```
       ...
    ...

// The target
tlm_utils::simple_target_socket<Memory, 32, my_protocol_types> socket;

virtual void b_transport( tlm::tlm_generic_payload& trans, sc_core::sc_time& t)
{
   tlm::tlm_command cmd = trans.get_command();
   ...
   cmd_extension* extension;
   trans.get_extension( extension );                       // Retrieve the command extension
   ...
   if (!extension) {                                       // Check the extension exists
      trans.set_response_status( tlm::TLM_GENERIC_ERROR_RESPONSE );
      return;
   }
   if (extension->increment) {
      if (cmd != tlm::TLM_IGNORE_COMMAND) {        // Detect clash with read or write
         trans.set_response_status( tlm::TLM_GENERIC_ERROR_RESPONSE );
         return;
      }
      ++ m_storage[adr];                               // Execute an increment command
       memcpy(ptr, &m_storage[adr], len);
   }
   ...
```

8 Base protocol and phases

8.1 Phases

8.1.1 Introduction

Class **tlm_phase** is the default phase type used by the non-blocking transport interface class templates and the base protocol. A **tlm_phase** object represents the phase with an **unsigned int** value. Class **tlm_phase** is assignment compatible with type **unsigned int** and with an enumeration having values corresponding to the four phases of the base protocol, namely BEGIN_REQ, END_REQ, BEGIN_RESP, and END_RESP. Because type **tlm_phase** is a class rather than an enumeration, it is able to support an overloaded stream operator to display the value of the phase as ASCII text.

The set of four phases provided by **tlm_phase_enum** can be extended using the macro DECLARE_EXTENDED_PHASE. This macro creates a singleton class derived from **tlm_phase** with a method **get_phase** that returns the corresponding object. That object can be used as a new phase.

For maximal interoperability, an application should only use the four phases of **tlm_phase_enum**. If further phases are required in order to model the details of a specific protocol, the intent is that DECLARE_EXTENDED_PHASE should be used, since this retains assignment compatibility with type **tlm_phase**.

The principle of ignorable versus non-ignorable or mandatory extensions applies to phases in the same way as to generic payload extensions. In other words, ignorable phases are permitted by the base protocol. An ignorable phase has to be ignorable by the recipient in the sense that the recipient can simply act as if it had not received the phase transition, and consequently the sender has to be able to continue in the absence of any response from the recipient. If a phase is not ignorable in this sense, a new protocol traits class should be defined. See 7.2.2 Define a new protocol traits class containing a typedef for tlm_generic_payload.

8.1.2 Class definition

```
namespace tlm {

enum tlm_phase_enum {
    UNINITIALIZED_PHASE=0, BEGIN_REQ=1, END_REQ, BEGIN_RESP, END_RESP };

class tlm_phase{
public:
    tlm_phase();
    tlm_phase( unsigned int );
    tlm_phase( const tlm_phase_enum& );
    tlm_phase& operator= ( const tlm_phase_enum& );
    operator unsigned int() const;
};

inline std::ostream& operator<< ( std::ostream& , const tlm_phase& );
```

```
#define DEFINE_EXTENDED_PHASE(name_arg) \
class tlm_phase_##name_arg : public tlm::tlm_phase{ \
public:\
    static const tlm_phase_##name_arg& get_phase();\
    implementation-defined \
}; \
static const tlm_phase_##name_arg& name_arg=tlm_phase_##name_arg::get_phase()

} // namespace tlm
```

8.1.3   Rules

a)   The default constructor **tlm_phase** shall set the value of the phase to 0, corresponding to the enumeration literal UNINITIALIZED_PHASE.

b)   The methods **tlm_phase(unsigned int)**, **operator=** and **operator unsigned int** shall get or set the value of the phase using the corresponding unsigned int or enum.

c)   The function **operator<<** shall write a character string corresponding to the name of the phase to the given output stream. For example "BEGIN_REQ".

d)   The macro DECLARE_EXTENDED_PHASE(*name_arg*) shall create a new singleton class named **tlm_phase_*name_arg***, derived from **tlm_phase**, and having a public method **get_phase** that returns a reference to the static object so created. The macro argument shall be used as the character string written by **operator<<** to denote the corresponding phase.

e)   The intent is that the object denoted by the static const **name_arg** represents the extended phase that may be passed as a phase argument to *nb_transport*.

Example

```
DECLARE_EXTENDED_PHASE(ignore_me);          // Declare two extended phases
DECLARE_EXTENDED_PHASE(internal_ph);        // Only used within target

struct Initiator: sc_module
{   ...
    {         ...
        phase = tlm::BEGIN_REQ;
        delay = sc_time(10, SC_NS);
        socket->nb_transport_fw( trans, phase, delay );     // Send phase BEGIN_REQ to target

        phase = ignore_me;                                  // Set phase variable to the extended phase
        delay = sc_time(12, SC_NS);
        socket->nb_transport_fw( trans, phase, delay );     // Send the extended phase 2ns later
        ...

struct Target: sc_module
```

```
    {
        ...
        SC_CTOR(Target)
        : m_peq("m_peq", this, &Target::peq_cb) {}              // Register callback with PEQ

        virtual tlm::tlm_sync_enum nb_transport_fw( tlm::tlm_generic_payload& trans,
                tlm::tlm_phase& phase, sc_time& delay ) {
            cout << "Phase = " << phase << endl;               // use overloaded operator<< to print phase
            m_peq.notify(trans, phase, delay);                 // Move transaction to internal queue
            return tlm::TLM_ACCEPTED;
        }

        void peq_cb(tlm::tlm_generic_payload& trans, const tlm::tlm_phase& phase)
        {                                                      // PEQ callback
            sc_time delay;
            tlm::tlm_phase phase_out;
            if (phase == tlm::BEGIN_REQ) {                     // Received BEGIN_REQ from initiator
                phase_out = tlm::END_REQ;
                delay = sc_time(10, SC_NS);
                socket->nb_transport_bw(trans, phase_out, delay);   // Send END_REQ back to initiator

                phase_out = internal_ph;                       // Use extended phase to signal internal event
                delay = sc_time(15, SC_NS);
                m_peq.notify(trans, phase_out, delay);         // Put internal event into PEQ
            }
            else if (phase == internal_ph)                     // Received internal event
            {
                phase_out = tlm::BEGIN_RESP;
                delay = sc_time(10, SC_NS);
                socket->nb_transport_bw(trans, phase_out, delay);   // Send BEGIN_RESP back to initiator
            }
        }                                                      // Ignore phase ignore_me from initiator

        tlm_utils::peq_with_cb_and_phase<Target, tlm::tlm_base_protocol_types> m_peq;
    };
```

8.2   Base protocol

8.2.1   Introduction

The base protocol consists of a set of rules to ensure maximal interoperability between transaction level models of components that interface to memory-mapped buses. The base protocol requires the use of the classes of the TLM-2.0 interoperability layer listed here, together with the rules defined in this clause:

a)   The TLM-2.0 core transport, direct memory and debug transport interfaces. See 4 TLM-2.0 Core Interfaces

b) The socket classes **tlm_initiator_socket** and **tlm_target_socket** (or classes derived from these). See 6.2 Initiator and target sockets

c) The generic payload class **tlm_generic_payload.** See 7 Generic payload.

d) The phase class **tlm_phase**

The base protocol rules permit extensions to the generic payload and to the phases only if those extensions are ignorable. Non-ignorable extensions require the definition of a new protocol traits class. See 7.2.1 Use the generic payload directly, with ignorable extensions

The base protocol is represented by the pre-defined class **tlm_base_protocol_types**. However, this class contains nothing but two type definitions. All components that use this class (as template argument to a socket) are obliged by convention to respect the rules of the base protocol.

In cases where it is necessary to define a new protocol traits class (e.g. because the features of the base protocol are insufficient to model a particular protocol), the rules associated with the new protocol traits class override those of the base protocol. However, for consistency and interoperability it is recommended that the rules and coding style associated with any new protocol traits class should follow those of the base protocol as far as possible. See 7.2.2 Define a new protocol traits class containing a typedef for tlm_generic_payload

This section of the standard specifically concerns the base protocol, but nonetheless may be used as a guide when modeling other protocols. Specific protocols represented by other protocol traits classes may include additional phases and may adopt their own rules for timing annotation, transaction ordering, and so forth. In doing so, they may cease to be compatible with the base protocol.

8.2.2   Class definition

```
namespace tlm {

struct tlm_base_protocol_types
{
 typedef  tlm_generic_payload     tlm_payload_type;
 typede   tlm_phase               tlm_phase_type;
};

} // namespace tlm
```

8.2.3   Base protocol phase sequences

a) The base protocol permits the use of the blocking transport interface, the non-blocking transport interface, or both together. The blocking transport interface does not carry phase information. When used with the base protocol, the constraints governing the order of calls to *nb_transport* are stronger than those governing the order of calls to **b_transport**. Hence *nb_transport* is more for the approximately-timed coding style, and **b_transport** for the loosely-timed coding style

b) The full sequence of phase transitions for a given transaction through a given socket is:

BEGIN_REQ → END_REQ → BEGIN_RESP → END_RESP

c) BEGIN_REQ and END_RESP shall be sent through initiator sockets only, END_REQ and BEGIN_RESP through target sockets only.

d) In the case of the blocking transport interface, a transaction instance is associated with a single call to and return from **b_transport**. The correspondence between the call to **b_transport** and BEGIN_REQ, and the return from **b_transport** and BEGIN_RESP, is purely notional; **b_transport** has no associated phases.

e) For the base protocol, each call to *nb_transport* and each return from *nb_transport* with a value of TLM_UPDATED shall cause a phase transition. In other words, two consecutive calls to *nb_transport* for the same transaction shall have different values for the phase argument. Ignorable phase extensions are permitted, in which case the insertion of an extended phase shall count as a phase transition for the purposes of this rule, even if the phase is ignored.

f) The phase sequence can be cut short by having *nb_transport* return a value of TLM_COMPLETED, but only in one of the following ways. An interconnect component or target may return TLM_COMPLETED when it receives BEGIN_REQ or END_RESP on the forward path. An interconnect component or initiator may return TLM_COMPLETED when it receives BEGIN_RESP on the backward path. A return value of TLM_COMPLETED indicates the end of the transaction with respect to a particular hop, in which case the phase argument should be ignored by the caller (see 4.1.2.7 The tlm_sync_enum return value). TLM_COMPLETED does not imply successful completion, so the initiator should check the response status of the transaction for success or failure.

Examples of early completion

Figure 13
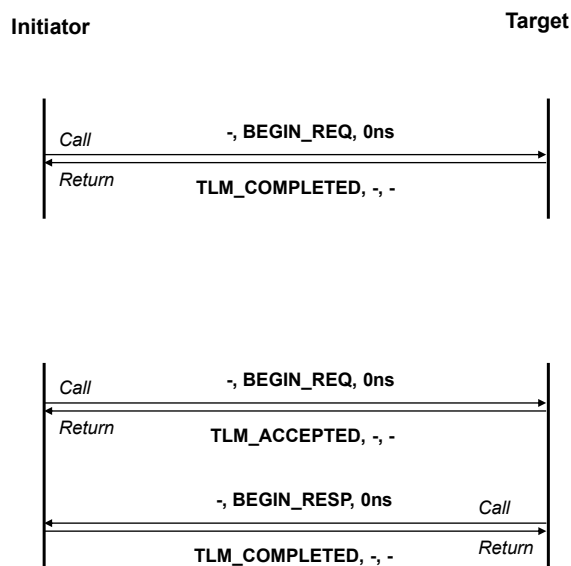
g) A transition to the phase END_RESP shall also indicate the end of the transaction with respect to a particular hop, in which case the callee is not obliged to return a value of TLM_COMPLETED.

h) When TLM_COMPLETED is returned in an upstream direction after having received BEGIN_REQ, this carries with it an implicit END_REQ and an implicit BEGIN_RESP. Hence the initiator should check the response status of the generic payload, and may send BEGIN_REQ for the next transaction immediately.

i) Since TLM_COMPLETED returned after having received BEGIN_REQ carries with it an implicit BEGIN_RESP, this situation is forbidden by the response exclusion rule if there is already a response in progress through a given socket. In this situation the callee should have returned TLM_ACCEPTED instead of TLM_COMPLETED and should wait for END_RESP before sending the next response upstream.

j) Since TLM_COMPLETED returned after having received BEGIN_REQ indicates the end of the transaction, an interconnect component or initiator is forbidden from then sending END_RESP for that same transaction through that same socket.

k) When TLM_COMPLETED is returned in a downstream direction by a component after having received BEGIN_RESP, this carries with it an implicit END_RESP.

l) If a component receives a BEGIN_RESP from a downstream component without having first received an END_REQ for that same transaction, the initiator shall assume an implicit END_REQ immediately preceding the BEGIN_RESP. This is only the case for the same transaction; a BEGIN_RESP does not imply an END_REQ for any other transaction, and a target that receives a BEGIN_REQ cannot infer an END_RESP for the previous transaction.

m) The above points hold regardless of the value of the timing annotation argument to *nb_transport*.

n) A base protocol transaction is complete (with respect to a particular hop) when TLM_COMPLETED is returned on either path, or when END_RESP is sent on the forward path or the return path.

o) In the case where END_RESP is sent on the forward path, the callee may return TLM_ACCEPTED or TLM_COMPLETED. The transaction is complete in either case.

p) A given transaction may complete at different times on different hops. A transaction object passed to *nb_transport* is obliged to have a memory manager, and the lifetime of the transaction object ends when the reference count of the generic payload reaches zero. Any component that calls the **acquire** method of a generic payload transaction object should also call the **release** method at or before the completion of the transaction. See 7.5 Generic payload memory management

q) If a component receives an illegal or out-of-order phase transition, this is an error on the part of the sender. The behavior of the recipient is undefined, meaning that a run-time error may be caused.

8.2.4   Permitted phase transitions

Taking all of the rules in the previous clause into account, the set of permitted phase transitions over a given hop for the base protocol is shown in the following table.

Previous state		Calling path	Phase argument on call	Phase argument on return	Status on return	Response valid	End-of-life		Next state
//rsp		Forward	BEGIN_REQ	-	Accepted				req
//rsp		Forward	BEGIN_REQ	END_REQ	Updated				//req
//rsp		Forward	BEGIN_REQ	BEGIN_RESP	Updated	✓			rsp
//rsp		Forward	BEGIN_REQ	-	Completed	✓	✓		//rsp
req		Backward	END_REQ	-	Accepted				//req
req		Backward	BEGIN_RESP	-	Accepted	✓			rsp
req		Backward	BEGIN_RESP	END_RESP	Updated	✓	✓		//rsp
req		Backward	BEGIN_RESP	-	Completed	✓	✓		//rsp
//req		Backward	BEGIN_RESP	-	Accepted	✓			rsp
//req		Backward	BEGIN_RESP	END_RESP	Updated	✓	✓		//rsp
//req		Backward	BEGIN_RESP	-	Completed	✓	✓		//rsp
rsp		Forward	END_RESP	-	Accepted	✓	✓		//rsp
rsp		Forward	END_RESP	-	Completed	✓	✓		//rsp

a)   req, //req, rsp, //rsp stand for BEGIN_REQ, END_REQ, BEGIN_RESP, and END_RESP respectively.

b)   These phase state transitions are independent of the value of the **sc_time** argument to *nb_transport* (the timing annotation). In other words, a call to *nb_transport* will cause the state transition shown in the table regardless of the value of the timing annotation. (The timing annotation may have the effect of delaying the subsequent execution of the transaction.)

c)   The **previous state** column shows the state of a given hop before a call to *nb_transport*.

d)   The **calling path** column indicates whether the corresponding method is called on the forward path (**nb_transport_fw**) or backward path (**nb_transport_bw**).

e)   The **phase argument on call** column gives the value of the phase argument on the call to *nb_transport*. This will be the phase presented to the callee.

f) The **phase argument on return** column gives the value of the phase argument on return from *nb_transport*. The phase argument is only valid if the method returns TLM_UPDATED.

g) The **status on return** column gives the return value of the *nb_transport* method, Accepted (TLM_ACCEPTED), Updated (TLM_UPDATED), or Completed (TLM_COMPLETED).

h) The **response valid column** is checked if the response status attribute of the transaction is valid on return from the *nb_transport* method.

i) The **end-of-life** column is checked if the transaction has reached the end of its lifetime with respect to this hop, that is, if no further *nb_transport* calls are permitted for the given transaction over the given hop.

j) The **next state** column shows the state of a given hop after return from *nb_transport*.

k) A phase transition can be caused either by the caller (indicated by a '-' in the phase argument on return column) or by the callee.

l) Ignorable phase extensions may be inserted at any point between BEGIN_REQ and END_RESP

m) A valid response does not indicate successful completion. The transaction may or may not have been successful.

n) Figure 14 on the next page presents, in a graphical format, the *nb_transport* call sequences permitted by the base protocol over a given hop. A traversal of the graph from Start to End gives a permitted call sequence for a single transaction instance. The rectangles show calls to *nb_transport* together with the value of the phase argument, the rounded rectangles show the status and where appropriate the value of the phase argument on return. The larger shaded rectangles show phase transitions.

nb_transport Call Sequence for each Base Protocol Transaction   Figure 14
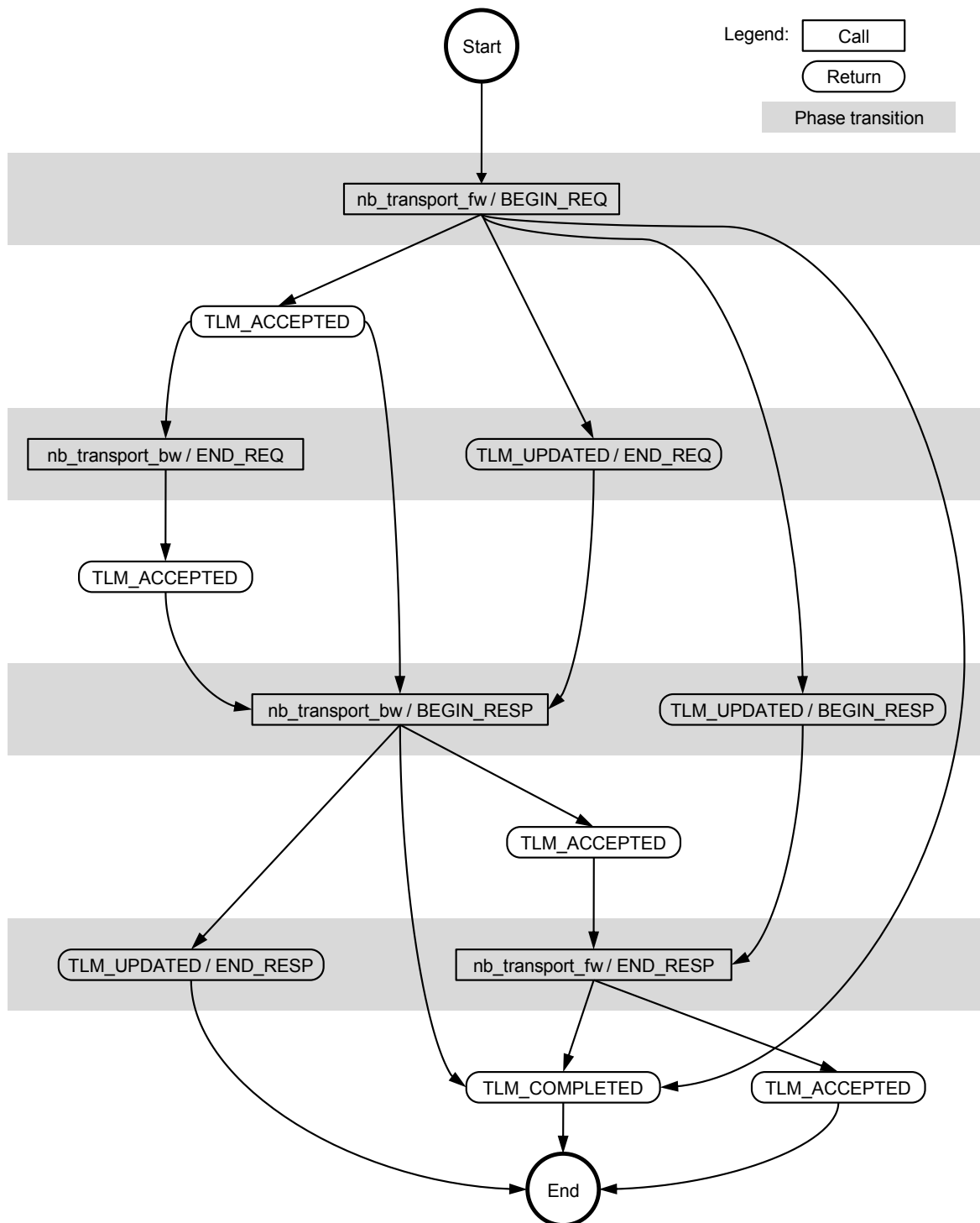
8.2.5   Ignorable phases

Extended phases may be used with the base protocol provided that they are *ignorable phases*. An ignorable phase may be ignored by its recipient.

a)   In general, the recommended way to add extended phases to the four phases of the base protocol is to define a new protocol traits class. See 7.2.2 Define a new protocol traits class containing a typedef for tlm_generic_payload. Ignorable phases are a special and restricted case of extended phases. The main purpose of ignorable phases is to permit extra timing points to be added to the base protocol in order to increase the timing accuracy of the model. For example, an ignorable phase could mark the time of the start of the data transfer from initiator to target.

b)   In the case of a call to *nb_transport*, if it is the callee that is ignoring the phase it shall return a value of TLM_ACCEPTED. In the case that the callee returns TLM_UPDATED, the caller may ignore the phase being passed on the return path but is not obliged to take any specific action to indicate that the phase is being ignored.

c)   The *nb_transport* interface does not provide any way for the caller of *nb_transport* to distinguish between the case where the callee is ignoring the phase, and the case where the callee will respond later on the opposite path. The callee shall return TLM_ACCEPTED in either case.

d)   The presence of an ignorable phase shall not change the order or the semantics of the four phases BEGIN_REQ, END_REQ, BEGIN_RESP, and END_RESP of the base protocol, and is not permitted to result in any of the rules of the base protocol being broken.

e)   An ignorable phase shall not occur before BEGIN_REQ or after END_RESP for a given transaction through a given socket. The presence of an ignorable phase before BEGIN_REQ or after END_RESP would violate the base protocol, and is an error.

f)   The presence of an ignorable phase shall not change the rules concerning the validity of the generic payload attributes or the rules for modifying those attributes. For example, on receipt of an ignorable phase, an interconnect component is only permitted to modify the address attribute, DMI allowed attribute, and extensions. See 7.7 Default values and modifiability of attributes.

g)   With the exception of transparent components as defined below, if the recipient of an ignorable phase does not recognize that phase (that is, the phase is being ignored), the recipient shall not propagate that phase on the forward, the backward or the return path. In other words, a component is only permitted to pass a phase as an argument to an *nb_transport* call if it fully understands the semantics of that phase.

h)   If the recipient of an ignorable phase does recognize that phase, provided that the base protocol is not violated, the behavior of that component is otherwise outside the scope of the base protocol and is undefined by the base protocol. The recipient should obey the semantics of the extended protocol to which the phase belongs.

i)   By definition, a component that sends an ignorable phase cannot require or demand any kind of response from the components to which that phase is sent other than the minimal response of *nb_transport* returning a value of TLM_ACCEPTED. A phase that demands a response is not ignorable, by definition, in which case the recommended approach is to define a new protocol traits class rather than using extensions to the base protocol. This prevents the binding of sockets that represent incompatible protocols.

j) On the other hand, a base-protocol-compliant component that does recognize an incoming extended phase may respond by sending another extended phase on the opposite path according to the rules of some extended protocol agreed in advance. This possibility is permitted by the TLM-2.0 standard, provided that the rules of the base protocol are not broken. For example, such an extended protocol could make use of generic payload extensions.

k) It is possible to create so-called *transparent* interconnect components, which immediately and directly pass through any TLM-2.0 interface method calls between a target socket and an initiator socket contained within the same component. The sole intent of recognizing transparent components in this standard is to allow for checkers and monitors, which typically have one target socket, one initiator socket, and pass through all transactions in both directions without modification.

l) Within a transparent component, the implementation of any TLM-2.0 core interface method shall not consume any simulation time, insert any delay, or call **wait**, but shall immediately make the identical interface method call through the opposing socket (initiator socket to target socket or target socket to initiator socket), passing through all its arguments. Such an interface method shall not modify the value of any argument, including the transaction object, the phase and the delay, with the one exception of generic payload extensions. The routing through such transparent components shall be fixed, and not depend on transaction attributes or phases.

m) As a consequence of the above rules, a transparent component would pass through any extended phase or ignorable phase in either direction.

Example

Ignorable Phases

Figure 15
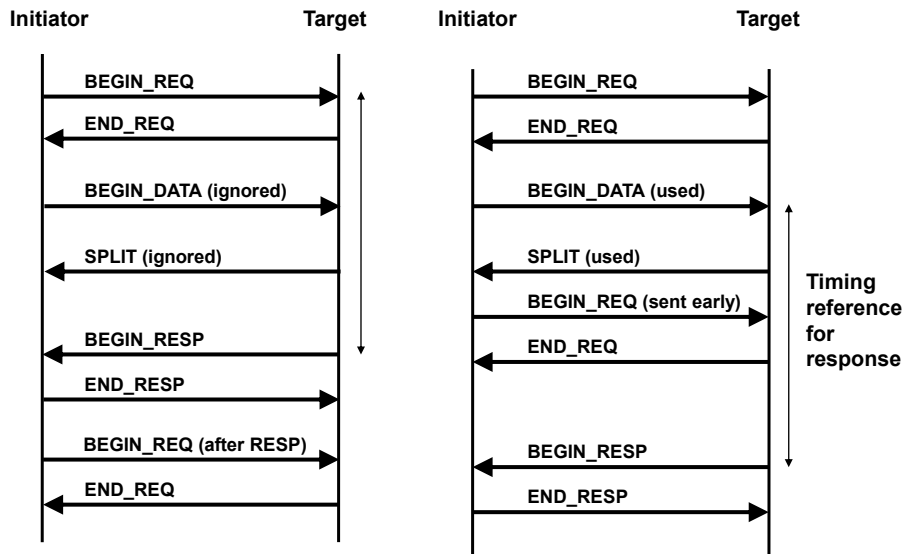
An example of an ignorable phase generated by an initiator would be a phase to mark the first beat of the data transfer from the initiator in the case of a write command. An interconnect component or target that recognized this phase could distinguish between the time at which the command and address become

available and the start of the data transfer. A target that ignored this phase would have to use the BEGIN_REQ phase as its single timing reference for the availability of the command, address and data.

An example of an ignorable phase generated by a target would be a phase to mark a split transaction. An initiator that recognized this phase could send the next request immediately upon receiving the split phase, knowing that the target would be ready to process it. An initiator that ignored the split phase might wait until it had received a response to the first request before sending the second request.

8.2.6   Base protocol timing parameters and flow control

a)   With four phases, it is possible to model the request accept delay (or minimum initiation interval between sending successive transactions), the latency of the target, and the response accept delay. This kind of timing granularity is appropriate for the approximately-timed coding style.

Approximately-timed timing parameters

Figure 16
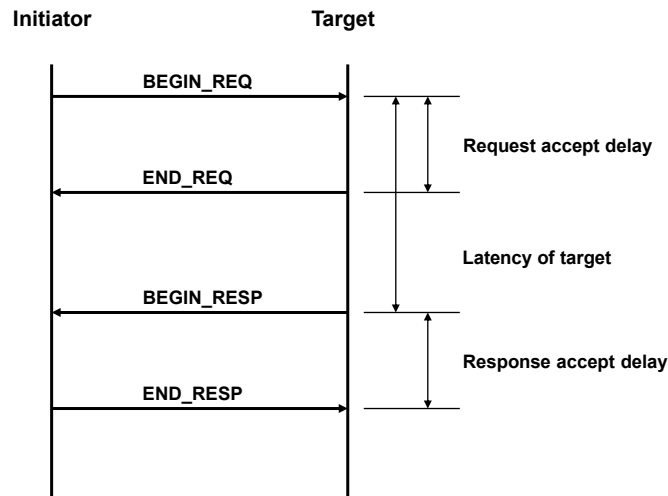
b)   For a write command, the BEGIN_REQ phase marks the time when the data becomes available for transfer from initiator through interconnect component to target. Notionally, the transition to the BEGIN_REQ phase corresponds to the start of the first beat of the data transfer. It is the responsibility of the downstream component to calculate the transfer time, and to send END_REQ back upstream when it is ready to receive the next transfer. It would be natural for the downstream component to delay the END_REQ until the end of the final beat of the data transfer, but it is not obliged to do so.

c)   For a read command, the BEGIN_RESP phase marks the time when the data becomes available for transfer from target through interconnect component to initiator. Notionally, the transition to the BEGIN_RESP phase corresponds to the start of the first beat of the data transfer. It is the responsibility of the upstream component to calculate the transfer time, and to send END_RESP back downstream

when it is ready to receive the next transfer. It would be natural for the upstream component to delay the END_RESP until the end of the final beat of the data transfer, but it is not obliged to do so.

d) For a read command, if a downstream component completes a transaction early by returning TLM_COMPLETED from **nb_transport_fw**, it is the responsibility of the upstream component to account for the data transfer time in some other way, if it wishes to do so (since it is not permitted to send END_RESP).

e) For the base protocol, an initiator or interconnect component shall not send a new transaction through a given socket with phase BEGIN_REQ until it has received END_REQ or BEGIN_RESP from the downstream component for the immediately preceding transaction or until the downstream component has completed the previous transaction by returning the value TLM_COMPLETED from **nb_transport_fw**. This is known as the *request exclusion rule*.

f) For the base protocol, a target or interconnect component shall not respond to a new transaction through a given socket with phase BEGIN_RESP until it has received END_RESP from the upstream component for the immediately preceding transaction or until a component has completed the previous transaction over that hop by returning TLM_COMPLETED. This is known as the *response exclusion rule*.

g) All the rules governing phase transitions, including the request and response exclusion rules, shall be based on method call order alone, and shall not be affected by the value of the time argument (the timing annotation).

h) Successive transactions sent through a given socket using the non-blocking transport interface can be pipelined. By responding to each BEGIN_REQ (or BEGIN_RESP) with an END_REQ (or END_RESP), an interconnect component can permit any number of transaction objects to be *in flight* at the same time. By not responding immediately with END_REQ (or END_RESP), an interconnect component can exercise flow control over the stream of transaction objects coming from an initiator (or target).

i) This rule excluding the possibility of two outstanding requests or responses through a given socket shall only apply to the non-blocking transport interface, and shall have no direct effect on calls to **b_transport**. (The rule may have an indirect effect on a call to **b_transport** in the case that **b_transport** itself calls **nb_transport_fw**.)

j) For a given transaction, BEGIN_REQ shall always start from an initiator and be propagated through zero or more interconnect components until it is received by a target. For a given transaction, an interconnect component is not permitted to send BEGIN_REQ to a downstream component before having received BEGIN_REQ from an upstream component.

k) For a given transaction, BEGIN_RESP shall always start from a target and be propagated through zero or more interconnect components until it is received by an initiator. For a given transaction, an interconnect component is not permitted to send BEGIN_RESP to an upstream component before having received BEGIN_RESP from a downstream component. This applies whether BEGIN_RESP is explicit or is implied by TLM_COMPLETED.

l) For a given transaction, an interconnect component may send END_REQ to an upstream component before having received END_REQ from a downstream component. Similarly, an interconnect component may send END_RESP to a downstream component before having received END_RESP from an upstream component. This applies whether END_REQ and END_RESP are explicit or implicit.

m) END_REQ and END_RESP are primarily for flow control between adjacent components. These two phases do not signal the validity of any standard generic payload attributes. Because these two phases are not propagated causally from end-to-end, they cannot reliably be used to signal the validity of extensions from initiator-to-target or target-to-initiator, but they can be used to signal the validity of extensions between two adjacent components.

n) Whether or not an interconnect component is permitted to send an extended phase before having received the corresponding phase depends on the rules associated with the extended phase in question.

Causality with b_transport

Figure 17
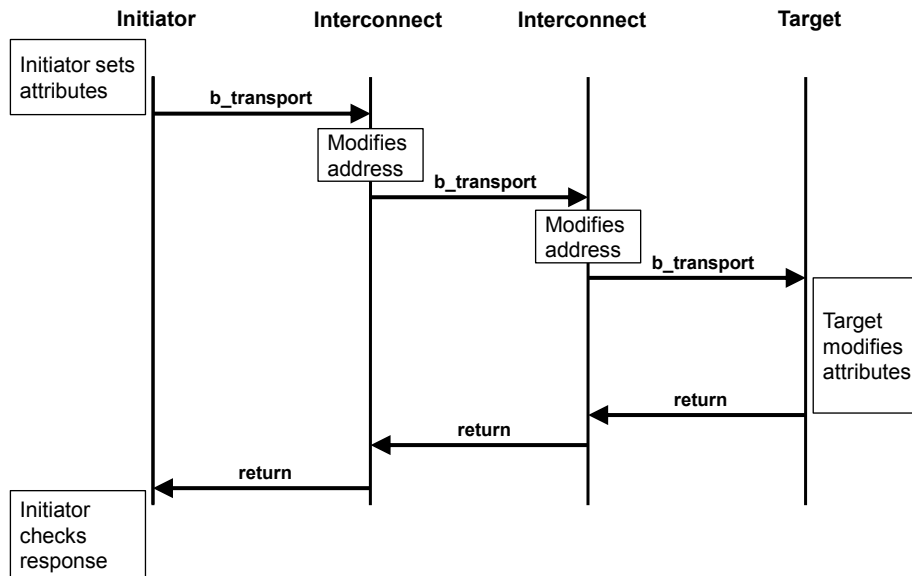
Causality with nb_transport

Figure 18
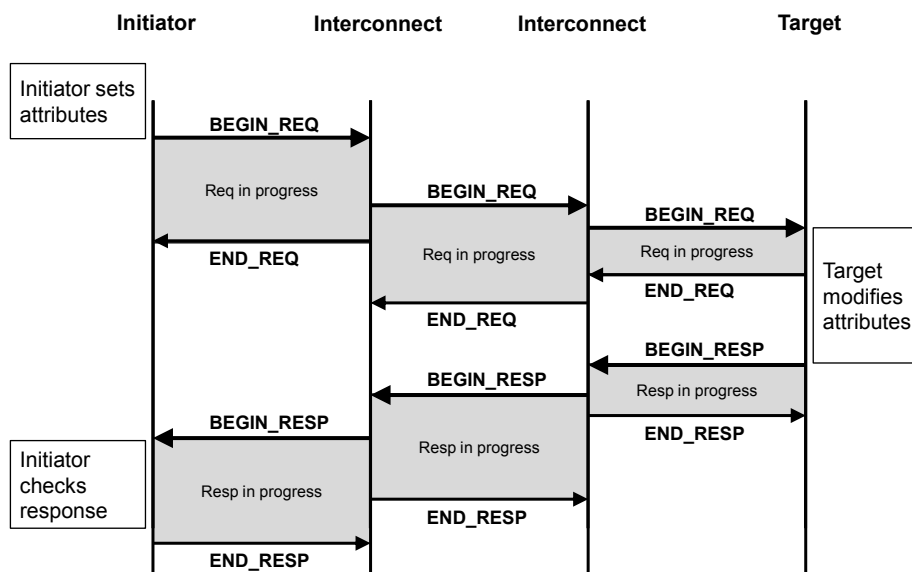
Example

The following pseudo-code illustrates the interaction between timing annotation and the request and response exclusion rules:

```
void initiator_1_thread_process()
{

 // The initiator sends a request to be executed at +1000ns
 phase = BEGIN_REQ;  delay = sc_time(1000, SC_NS);
 status = socket->nb_transport_fw (T1, phase,  delay);
 assert( status == TLM_UPDATED && phase == END_REQ && delay == sc_time(1010, SC_NS) );
 // END_REQ is returned immediately to be executed at +1010ns

 // Note that this is not a recommended coding style
 // With loosely-timed, the initiator would have called b_transport
 // With approximately-timed, the downstream component would have returned TLM_ACCEPTED
 // in order to synchronize, and the initiator would have been forced to wait for END_REQ

 // The initiator is allowed to send the next request immediately, to be executed at +1050ns
 phase = BEGIN_REQ; delay = sc_time(1050, SC_NS);
 status = socket->nb_transport_fw (T2, phase, delay);
 assert( status == TLM_UPDATED && phase == END_REQ && delay == sc_time(1060, SC_NS) );

 // The initiator is technically allowed to send the next request at an earlier local time of +500ns,
 // although the decreased timing annotation is not a recommended coding style
 phase = BEGIN_REQ; delay = sc_time(500, SC_NS);
 status = socket->nb_transport_fw (T3, phase, delay);
 assert( status == TLM_UPDATED && phase == END_REQ && delay == sc_time(510, SC_NS) );

  // The initiator now yields control, allowing other initiators to resume and simulation time to advance
  wait(…);
}

void initiator_2_thread_process()
{
 // Assume the calls below are appended to the transaction stream sent from the first initiator above

 // The second initiator sends a request to be executed at +10ns
 // The timing annotation as seen downstream has decreased from +510ns to +10ns
 // This is typical behavior for loosely-timed initiators
 phase = BEGIN_REQ; delay = sc_time(10, SC_NS);
 status = socket->nb_transport_fw (T4, phase, delay);
 assert( status == TLM_UPDATED && phase == END_REQ && delay == sc_time(30, SC_NS) );
 // END_REQ is returned immediately to be executed at +30ns
```

```
    // The initiator sends the next request to be executed at +20ns, which overlaps with the previous request
    // This is technically allowed because the current phase of the hop is END_REQ, but is not recommended
    phase = BEGIN_REQ; delay = sc_time(20, SC_NS);
    status = socket->nb_transport_fw (T5, phase, delay);
    assert( status == TLM_UPDATED && phase == END_REQ && delay == sc_time(40, SC_NS) );
    // END_REQ is returned immediately to be executed at +40ns

    // The initiator sends the next request to be executed at +0ns, which is before the previous two requests
    // This is technically allowed because the current phase of the hop is END_REQ, but is not recommended
    phase = BEGIN_REQ; delay = sc_time(0, SC_NS);
    status = socket->nb_transport_fw (T6, phase, delay);
    assert( status == TLM_UPDATED && phase == END_REQ && delay == sc_time(60, SC_NS) );
    // END_REQ is returned immediately to be executed at +60ns, overlapping the two previous requests
    // This is technically allowed, but is not recommended
    wait(…);
  }
```

8.2.7   Base protocol rules concerning timing annotation

a)   These rules should be read in conjunction with 4.1.3 Timing annotation with the transport interfaces

b)   There are constraints on the way in which the implementations of **b_transport** and *nb_transport* are permitted to modify the time argument **t** such that the effective local time **sc_time_stamp() + t** is non-decreasing between function call and return. See 4.1.3.1 The sc_time argument

c)   For successive calls to and returns from *nb_transport* through a given socket for a given transaction, the sequence of effective local times shall be non-decreasing. The effective local time is given by the expression **sc_time_stamp() + t**, where **t** is the time argument to *nb_transport*. For this purpose, both calls to and returns from the function shall be considered as part of a single sequence. This applies on the forward and backward paths alike. The intent is that time should not run backwards for a given transaction.

d)   The preceding rule also applies between the call to and the return from **b_transport**. Again, see 4.1.3.1 The sc_time argument

e)   Moreover, for a given transaction object, as requests are propagated from initiator towards target and responses are propagated from target back towards initiator, the sequence of effective local times given by each successive transport method call and return shall be non-decreasing. Request propagation in this sense includes calls to **b_transport** and the BEGIN_REQ phase. Response propagation includes returns from **b_transport**, the BEGIN_RESP phase, and TLM_COMPLETED.

f)   The effective local time may be increased by increasing the value of the timing annotation (the time argument), by advancing SystemC simulation time (**b_transport** only), or both.

g)   For *different* transaction objects, there is no obligation that the effective local times of calls to **b_transport** and *nb_transport* shall be non-decreasing. Nonetheless, each initiator process is generally recommended to call **b_transport** and/or *nb_transport* in non-decreasing effective local time order. Otherwise, downstream components would infer that the out-of-order transactions had originated from

separate initiators and would be free to choose the order in which those particular transactions were executed. However, transactions with out-of-order effective local times may arise wherever streams of transactions from different loosely-timed initiators converge.

h)   For a given socket, an initiator is allowed to pass the same transaction object at different times through the blocking and non-blocking transport interfaces, the direct memory interface, and the transport debug interface, Also, an initiator is permitted to re-use the same trans*action* object fo*r different* transaction instances, all subject to the memory management rules of the generic payload. See 7.5 Generic payload memory management.

8.2.8   Base protocol rules concerning b_transport

a)   **b_transport** calls are re-entrant. The implementation of **b_transport** can call **wait**, and meanwhile another call to **b_transport** can be made for a different transaction object from a different initiator with no constraint on the timing annotation.

b)   In the case that there are multiple processes within the same initiator module, each process shall be regarded as being a separate initiator with respect to the transaction ordering rules. Specifically, there are no constraints on the ordering of b_**transport** calls made from different threads in the same module, regardless of whether those calls are made through the same initiator socket or through different sockets.

c)   An interconnect or target can always deduce that multiple concurrent **b_transport** calls come from different initiator threads and from a different process to any concurrent **nb_transport_fw** calls, and therefore that there are no constraints on the mutual order of those calls. With **b_transport**, one request is allowed to overtake another.

d)   It is forbidden to make a re-entrant call to **b_transport** for the same transaction object through the same socket.

Example

The following pseudo-code fragments show a re-entrant b_transport call:

```
// Two initiator thread processes
void thread1()
{
   socket->b_transport( T1, sc_time(100, SC_NS) );
}

void thread2()
{
   wait( 10, SC_NS );
   socket->b_transport( T2, sc_time(50, SC_NS) );  // T2 overtakes T1
}

// Implementation of b_transport in the target
void b_transport( TRANS& trans, sc_time& t )
{
   wait( t );
```

```
  execute( trans );      // T1 executed at 100ns, T2 executed at 60ns
  t = SC_ZERO_TIME;
}
```

8.2.9   Base protocol rules concerning request and response ordering

The intent of the following rules is to ensure that when an initiator sends a series of pipelined requests to a particular target, those requests will be executed at the target in the same order as they were sent from the initiator. Because the generic payload transaction stores neither the identity of the initiator nor the identity of the target, the initiator can only be inferred from the identity of the incoming socket, and the target can only be inferred from the values of the address and command attributes. The execution order of requests sent to non-overlapping addresses is not guaranteed.

a) The base protocol permits incoming requests or responses arriving through different sockets to be mutually delayed, interleaved, or executed in any order. For example, an interconnect component may assign a higher priority to requests arriving through a particular target socket or having a particular data length, allowing them to overtake lower priority requests. As another example, an interconnect component may re-order responses arriving back through different initiator sockets to match the order in which the corresponding requests were originally received.

b) Request routing shall be deterministic and shall depend only on the address and command attributes of the transaction object. (These are the only attributes common to the transport, DMI and debug transport interfaces.) The address map shall not be modified while there are transactions in progress.

c) If an initiator or interconnect component sends multiple concurrent requests with overlapping addresses on the forward path, those requests shall be routed through the same initiator socket. *Multiple concurrent requests* means requests for which the corresponding responses have not yet been received from the target. *Overlapping addresses* means that at least one byte in the data arrays of the transaction objects shares the same address. Read and write requests to the same address may be routed through different sockets provided they are not concurrent.

d) If an interconnect component (or a target) receives multiple concurrent requests with overlapping addresses through the same target socket by means of incoming calls to **nb_transport_fw**, those requests shall be sent forward (or executed, respectively) in the same order as they were received. *The same order* means the same interface method call order. (Note that if the interface method call order and the effective local time order of a set of transactions were to differ, any component receiving those transactions would be permitted to execute them in any order, regardless of the address. Also note that this rule holds even if the requests in question originate from different initiators.)

e) Note that the preceding rule does not apply for incoming **b_transport** calls, for which there are no constraints on the order of multiple concurrent requests. On the other hand, if incoming **nb_transport_fw** calls are converted to outgoing **b_transport** calls, the **b_transport** calls must be serialized to enforce the ordering rules of the *nb_transport* calls.

f) On the other hand, an interconnect component or target is permitted to re-order multiple concurrent requests that were received through different target sockets, or are sent through different initiator sockets, or whose addresses do not overlap, or for incoming **b_transport** calls.

119

g) Responses may be re-ordered. There is no guarantee that responses will arrive back at an initiator in the same order as the corresponding requests were sent.

h) Note that it would be technically possible with the base protocol to use an ignorable extension that allowed an interconnect component to re-order multiple concurrent requests, in which case the initiator that added the extension must be able to tolerate out-of-order execution at the target. On the other hand, an extension that forced responses to arrive back in the same order as requests were sent would not be an ignorable extension, and hence would not be permitted by the base protocol.

8.2.10 Base protocol rules for switching between b_transport and nb_transport

a) Each thread within an initiator or an interconnect component is permitted to switch between calling **b_transport** and **nb_transport_fw** for different transaction objects. The intent is to permit an initiator to make occasional switches between the loosely-timed and approximately-timed coding styles. An initiator that interleaves calls to **b_transport** and **nb_transport_fw** should have low expectations with regard to timing accuracy.

b) Every interconnect component and target is obliged to support both the blocking and non-blocking transport interfaces, and to maintain any internal state information such that it is accessible from both interfaces. This applies to incoming interface method calls received through the same socket or through different sockets.

c) A thread within an initiator or an interconnect component shall not call both **b_transport** and **nb_transport_fw** for the same transaction instance. Note that a thread may call both **b_transport** and **nb_transport_fw** for the same transaction object provided that object represents a different transaction instance on each occasion.

d) It is recommended that a thread within an initiator or interconnect component should not call **b_transport** if there is still a transaction in progress from a previous **nb_transport_fw** call from that same thread, that is, when there is a previous transaction with a non-zero reference count. Otherwise, a downstream component could wrongly deduce that the two transactions had come from separate initiators.

e) The convenience socket **simple_target_socket** provides an example of how a base protocol target can support both the blocking and the non-blocking transport interfaces while only being required to implement one of **b_transport** and **nb_transport_fw**. See 9.1.2 Simple sockets.

8.2.11 Other base protocol rules

a) A given transaction object shall not be sent through multiple parallel sockets or along multiple parallel paths simultaneously. Each transaction instance shall take a unique well-defined path through a set of components and sockets which shall remain fixed for the lifetime of the transaction instance and is common to the transport, direct memory and debug transport interfaces. Of course, different transactions sent through a given socket may take different paths, that is, they may be routed differently. Also, note that a component may choose dynamically whether to act as an interconnect component or as a target.

b) An upstream component should not know and should not need to know whether it is connected to an interconnect component or directly to a target. Similarly, a downstream component should not know and should not need to know whether it is connected to an interconnect component or directly to an initiator.

c) For a write transaction (TLM_WRITE_COMMAND), a response status of TLM_OK_RESPONSE shall indicate that the write command has completed successfully at the target. The target is obliged to set the response status before returning from **b_transport**, before sending BEGIN_RESP along the backward or return path, or before returning TLM_COMPLETED. In other words, an interconnect component is not permitted to signal the completion of a write transaction without having had confirmation of successful completion from the target. The intent of this rule is to guarantee the coherency of the storage within the target simulation model.

d) For a read transaction (TLM_READ_COMMAND), a response status of TLM_OK_RESPONSE shall indicate that the read command has completed and the generic payload data array has been modified by the target. The target is obliged to set the response status before returning from **b_transport**, before sending BEGIN_RESP along the backward or return path, or before returning TLM_COMPLETED.

8.2.12 Summary of base protocol transaction ordering rules

The following table gives a summary of the transaction ordering rules for the base protocol. For a full description of the rules, refer to the clauses above.

The base protocol ordering rules are a union of the rules from three categories: rules of the core transport interfaces concerning timing annotation, rules specific to the base protocol concerning causality and phases, and rules specific to the base protocol that ensure that pipelined requests are executed at the target in the order expected by the initiator.

Circumstance	Ordering rule
Effective local time order different from interface method call order	Recipient may execute or route transactions in any order. Takes precedence over all other rules
Successive transport method calls and returns for the same transaction through the same socket	Effective local time order shall be non-decreasing
Successive transport method calls from the same initiator process	Effective local time order is recommended to be non-decreasing
Successive transport method calls from the same initiator process	Recommended not to call b_transport if previous nb_transport transaction is still alive
Successive transport method calls for different transactions through the same socket	No obligation on effective local time order, but recommended to be non-decreasing if incoming transaction stream was non-decreasing
With nb_transport only, two requests or two responses outstanding through the same socket	Forbidden
Transactions incoming through different sockets	No obligations on the order in which they are executed or routed on
Multiple concurrent requests with overlapping addresses	If routed forward, shall be sent through the same socket

Multiple concurrent requests with overlapping addresses incoming through the same socket using nb_transport	Shall be executed or routed forward in the same order as they were received
Multiple concurrent requests incoming using b_transport	No obligations on the order in which they are executed or routed forward
Multiple concurrent responses	No obligations on the order in which they are executed or routed back

8.2.13 Guidelines for creating base-protocol-compliant components

This clause contains a set of guidelines for creating base protocol components. This is just a brief restatement of some of the rules presented more fully elsewhere in this document, and is provided for convenience.

8.2.13.1 Guidelines for creating a base protocol initiator

a) Instantiate one initiator socket of class **tlm_initiator_socket** (or a derived class) for each connection to a memory-mapped bus.

b) Allow the **tlm_initiator_socket** to take the default value **tlm_base_protocol_types** for the template TYPES argument.

c) Implement the methods **nb_transport_bw** and **invalidate_direct_mem_ptr**. (An initiator can avoid the need to implement these methods explicitly by instantiating the convenience socket **simple_initiator_socket**.)

d) Set every attribute of each generic payload transaction object before passing it as an argument to **b_transport** or **nb_transport_fw**, remembering in particular to reset the response status and DMI hint attributes before the call. (The byte enable length attribute need not be set in the case where the byte enable pointer attribute is 0, and extensions need not be used.)

e) When using the generic payload extension mechanism, ensure that any extensions are ignorable by the target and any interconnect component.

f) Obey the base protocol rules concerning phase sequencing, flow control, timing annotation, and transaction ordering.

g) On completion of the transaction (or after receiving BEGIN_RESP), check the value of the response status attribute.

8.2.13.2 Guidelines for creating an initiator that calls *nb_transport*

a) Before passing a transaction as an argument to **nb_transport_fw**, set a memory manager for the transaction object and call the **acquire** method of the transaction. Call the **release** method when the transaction is complete.

b) When calling **nb_transport_fw**, set the phase argument to BEGIN_REQ or END_RESP according to state of the transaction. Do not send BEGIN_REQ before having received (or inferred) the END_REQ from the previous transaction

c) When making a series of calls to **nb_transport_fw** for a given transaction, ensure that the effective local times (simulation time + timing annotation) form a non-decreasing sequence of values.

d) Respond appropriately to the incoming phase values END_REQ and BEGIN_RESP whether received on the backward path (a call to **nb_transport_bw**), the return path (TLM_UPDATED returned from **nb_transport_fw**), or implicitly (for example, TLM_COMPLETED returned from **nb_transport_fw**). Incoming phase values of BEGIN_REQ and END_RESP would be illegal. Treat all other incoming phase values as being ignorable.

8.2.13.3 Guidelines for creating a base protocol target

a) Instantiate one target socket of class **tlm_target_socket** (or a derived class) for each connection to a memory-mapped bus.

b) Allow the **tlm_target_socket** to take the default value **tlm_base_protocol_types** for the template TYPES argument.

c) Implement the methods **b_transport**, **nb_transport_fw**, **get_direct_mem_ptr**, and **transport_dbg**. (A target can avoid the need to implement every method explicitly by using the convenience socket **simple_target_socket**.)

d) In the implementations of the methods **b_transport** and **nb_transport_fw**, inspect and act upon the value of every attribute of the generic payload with the exception of the response status, the DMI hint, and any extensions. Rather than implementing the full functionality of the generic payload, a target may choose to respond to a given attribute by generating an error response. Set the value of the response status attribute to indicate the success or failure of the transaction.

e) Obey the base protocol rules concerning phase sequencing, flow control, timing annotation, and transaction ordering.

f) In the implementation of **get_direct_mem_ptr**, either return the value **false**, or inspect and act upon the values of the command and address attributes of the generic payload and set the return value and all the attributes of the DMI descriptor appropriately (class **tlm_dmi**).

g) In the implementation of **transport_dbg**, either return the value 0, or inspect and act upon the values of the command, address, data length, and data pointer attributes of the generic payload.

h) For each interface, the target may inspect and act upon any ignorable extensions in the generic payload, but is not obliged to do so.

8.2.13.4 Guidelines for creating a target that calls *nb_transport*

a) When calling **nb_transport_bw**, set the phase argument to END_REQ or BEGIN_RESP according to state of the transaction. Do not send BEGIN_RESP before having received (or inferred) END_RESP from the previous transaction.

b) When making a series of calls to **nb_transport_bw** for a given transaction, ensure that the effective local times (simulation time + timing annotation) form a non-decreasing sequence of values.

c) Respond appropriately to the incoming phase values BEGIN_REQ and END_RESP, whether received on the forward path (a call to **nb_transport_fw**), the return path (TLM_UPDATED returned from **nb_transport_bw**), or implicitly (for example, TLM_COMPLETED returned from **nb_transport_bw**).

Incoming phase values of END_REQ and BEGIN_RESP would be illegal. Treat all other incoming phase values as being ignorable.

d) In the implementation of **nb_transport_fw**, when needing to keep a pointer or reference to a transaction object beyond the return from the method, call the **acquire** method of the transaction. Call the **release** method when the transaction object is finished with.

8.2.13.5 Guidelines for creating a base protocol interconnect component

a) Instantiate one initiator or target socket of class **tlm_initiator_socket** or **tlm_target_socket** (or derived classes) for each connection to a memory-mapped bus.

b) Allow each socket to take the default value **tlm_base_protocol_types** for the template TYPES argument.

c) Implement the methods **nb_transport_bw** and **invalidate_direct_mem_ptr** for each initiator socket, and the methods **b_transport**, **nb_transport_fw**, **get_direct_mem_ptr**, and **transport_dbg** for each target socket. (The need to implement every method explicitly can be avoided by using the convenience sockets.)

d) Pass on incoming interface method calls as appropriate on both the forward and backward paths, honoring the request and response exclusion rules, the transaction ordering rules, and the rule that no further calls are allowed following TLM_COMPLETED. Do not pass on ignorable phases. The implementations of the **get_direct_mem_ptr** and **transport_dbg** methods may return the values **false** and 0 respectively without forwarding the transaction object.

e) In the implementation of the transport interfaces, the only generic payload attributes modifiable by an interconnect component are the address, DMI hint, and extensions. Do not modify any other attributes. A component needing to modify any other attributes should construct a new transaction object, and thereby become an initiator in its own right.

f) Decode the generic payload address attribute on the forward path and modify the address attribute if necessary according to the location of the target in the system memory map. This applies to the transport, direct memory, and debug transport interfaces.

g) In the implementation of the transport interfaces, obey the base protocol rules concerning phase sequencing, flow control, timing annotation, and transaction ordering.

h) In the implementation of **get_direct_mem_ptr**, do not modify the DMI descriptor attributes on the forward path. Do modify the DMI pointer, DMI start address and end address, and DMI access attributes appropriately on the return path.

i) In the implementation of **invalidate_direct_mem_ptr**, modify the address range arguments before passing the call along the backward path.

j) In the implementation of **nb_transport_fw**, when needing to keep a pointer or reference to a transaction object beyond the return from the function, call the **acquire** method of the transaction. Call the **release** method when the transaction object is finished with.

k) For each interface, the interconnect component may inspect and act upon any ignorable extensions in the generic payload, but is not obliged to do so. If the transaction needs to be extended further, make sure any extensions are ignorable by the other components. Honor the generic payload memory management rules for extensions.

9 Utilities

The utilities comprise a set of classes that are provided for convenience and to help ensure a consistent coding style. The utilities do not belong to the interoperability layer, so use of the utilities is not a requirement for interoperability.

9.1 Convenience sockets

9.1.1 Introduction

There is a family of convenience sockets, each socket implementing some additional functionality to make component models easier to write. The convenience sockets are derived from the classes **tlm_initiator_socket** and **tlm_target_socket**. They are not part of the TLM-2.0 interoperability layer, but are to be found in the namespace **tlm_utils**.

9.1.1.1 Summary of standard and convenience socket types

The convenience sockets are summarized in the following table.

Register callbacks? The socket provides methods to register callbacks for incoming interface method calls, rather than having the socket be bound to an object that implements the corresponding interfaces.

Multi-ports? The socket class template provides number-of-bindings and binding policy template arguments such that a single initiator socket can be bound to multiple target sockets and vice versa.

b – nb conversion? The target socket is able to convert incoming calls to **b_transport** into **nb_transport_fw** calls, and vice versa. A '-' indicates an initiator socket.

Tagged? Incoming interface method calls are tagged with an id to indicate the socket through which they arrived

Class	Register callbacks?	Multi-ports?	b / nb conversion?	Tagged?
tlm_initiator_socket	no	yes	-	no
tlm_target_socket	no	yes	no	no
simple_initiator_socket	yes	no	-	no
simple_initiator_socket_tagged	yes	no	-	yes
simple_target_socket	yes	no	yes	no
simple_target_socket_tagged	yes	no	yes	yes
passthrough_target_socket	yes	no	no	no
passthrough_target_socket_tagged	yes	no	no	yes
multi_passthrough_initiator_socket	yes	yes	-	yes
multi_passthrough_target_socket	yes	yes	no	yes

9.1.1.2 Socket binding table

The bindings permitted between the standard and convenience socket types are summarized in the following table. Each binding is of the form From(To) or From.bind(To), where From and To are sockets of the given type.

To From	tlm-init	simple-init	multi-init		tlm-targ	simple-targ	multi-targ
tlm-init	1				1	1	N:1
simple-init	1				1	1	N:1
multi-init			1		1:M	1:M	N:M
tlm-targ	1*	1*			1	1	
simple-targ	1*	1*					
multi-targ							1

The above table is organized into four quarters as follows:

Hierarchical child-to-parent binding	Initiator-to-target binding
Reverse binding operators	Hierarchical parent-to-child binding

Key	
tlm-init	tlm_initiator_socket
simple-init	simple_initiator_socket or passthrough_initiator_socket
multi-init	multi_passthrough_initiator_socket
tlm-targ	tlm_target_socket
simple-targ	simple_target_socket or simple_target_socket_tagged or passthrough_target_socket or passthrough_target_socket_tagged
multi-targ	multi_passthrough_target_socket
1*	The binding is from inititiator to target, despite the method call being in the direction target.bind(initiator)

9.1.2   Simple sockets

9.1.2.1   Introduction

The *simple sockets* are so-called because they are intended to be simple to use. They are derived from the interoperability layer sockets **tlm_initiator_socket** and **tlm_target_socket**, so can be bound directly to sockets of those types.

Instead of having to bind a socket to an object that implements the corresponding interface, each simple socket provides methods for registering callback methods. Those callbacks are in turn called whenever an incoming interface method call arrives. Callback methods may be registered for each of the interfaces supported by the socket.

The user of a simple socket may register a callback for every interface method, but is not obliged to do so. In particular, for the simple target socket, the user need only register one of **b_transport** and **nb_transport_fw**, in which case incoming calls to the unregistered method will be converted automatically to calls to the registered method. This conversion process is non-trivial, and is dependent upon the rules of the base protocol being respected by the initiator and target.. The **passthrough_target_socket** is a variant of the **simple_target_socket** that does not support conversion between blocking and non-blocking calls.

The current implementation of simple sockets makes use of dynamic processes. Hence, when compiling applications that use simple sockets with current released versions of the OSCI proof-of-concept simulator, it is necessary to define the macro SC_INCLUDE_DYNAMIC_PROCESSES before including the SystemC header file.

9.1.2.2   Class definition

```
namespace tlm_utils {

template <
  typename MODULE,
  unsigned int BUSWIDTH = 32,
  typename TYPES = tlm::tlm_base_protocol_types
>
class simple_initiator_socket :  public tlm::tlm_initiator_socket<BUSWIDTH, TYPES>
{
public:
  typedef typename TYPES::tlm_payload_type            transaction_type;
  typedef typename TYPES::tlm_phase_type              phase_type;
  typedef tlm::tlm_sync_enum                          sync_enum_type;

  simple_initiator_socket();
  explicit simple_initiator_socket( const char* n );

  void register_nb_transport_bw(
    MODULE* mod,
    sync_enum_type (MODULE::*cb)(transaction_type&, phase_type&, sc_core::sc_time&));
```

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


template <
   typename MODULE,
   unsigned int BUSWIDTH = 32,
   typename TYPES = tlm::tlm_base_protocol_types
>
class simple_target_socket : public tlm::tlm_target_socket<BUSWIDTH, TYPES>
{
public:
   typedef typename TYPES::tlm_payload_type   transaction_type;
   typedef typename TYPES::tlm_phase_type     phase_type;
   typedef tlm::tlm_sync_enum                 sync_enum_type;

   simple_target_socket();
   explicit simple_target_socket( const char* n );

   tlm::tlm_bw_transport_if<TYPES> * operator ->();

   void register_nb_transport_fw(
      MODULE* mod,
      sync_enum_type (MODULE::*cb)(transaction_type&, phase_type&, sc_core::sc_time&));

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

   void register_transport_dbg(
      MODULE* mod,
      unsigned int (MODULE::*cb)(transaction_type&));

   void register_get_direct_mem_ptr(
      MODULE* mod,
      bool (MODULE::*cb)(transaction_type&, tlm::tlm_dmi&));
};


template <
   typename MODULE,
   unsigned int BUSWIDTH = 32,
   typename TYPES = tlm::tlm_base_protocol_types
>
```

```
class passthrough_target_socket :   public tlm::tlm_target_socket<BUSWIDTH, TYPES>
{
public:
   typedef typename TYPES::tlm_payload_type   transaction_type;
   typedef typename TYPES::tlm_phase_type     phase_type;
   typedef tlm::tlm_sync_enum                 sync_enum_type;

   passthrough_target_socket();
   explicit passthrough_target_socket( const char* n );

   void register_nb_transport_fw(
      MODULE* mod,
      sync_enum_type (MODULE::*cb)(transaction_type&, phase_type&, sc_core::sc_time&));

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

   void register_transport_dbg(
      MODULE* mod,
      unsigned int (MODULE::*cb)(transaction_type&));

   void register_get_direct_mem_ptr(
      MODULE* mod,
      bool (MODULE::*cb)(transaction_type&, tlm::tlm_dmi&));
};

} // namespace tlm_utils
```

9.1.2.3   Header file

The class definitions for the simple sockets shall be in the header files **tlm_utils/simple_initiator_socket.h**, **tlm_utils/simple_target_socket.h,** and **tlm_utils/passthrough_target_socket.h**.

9.1.2.4   Rules

a) Each constructor shall call the constructor of the corresponding base class passing through the **char*** argument, if there is one. In the case of the default constructors, the **char*** argument of the base class constructor shall be set to sc_gen_unique_name ("simple_initiator_socket"), sc_gen_unique_name ("simple_target_socket"), or sc_gen_unique_name ("passthrough_target_socket") respectively.

b) A **simple_initiator_socket** can be bound to a **simple_target_socket** or a **passthrough_target_socket** by calling the **bind** method or **operator()** of either socket, with precisely the same effect. In either case, the forward path lies in the direction from the initiator socket to the target socket.

c) A **simple_initiator_socket** can be bound to a **tlm_target_socket**, and a **tlm_initiator_socket** can be bound to a **simple_target_socket** or to a **passthrough_target_socket**.

d) A **simple_initiator_socket**, **simple_target_socket** or **passthrough_target_socket** can only implement incoming interface method calls by registering callbacks, not by being bound hierarchically to another socket on a child module. On the other hand, a **simple_initiator_socket** of a child module can be bound hierarchically to a **tlm_initiator_socket** of a parent module, and a **tlm_target_socket** of a parent module can be bound hierarchically to a **simple_target_socket** or **passthrough_target_socket** of a child module.

e) A target is not obliged to register a **b_transport** callback with a simple target socket provided it has registered an **nb_transport_fw** callback, in which case an incoming **b_transport** call will automatically cause the target to call the method registered for **nb_transport_fw**. In this case, the method registered for **nb_transport_fw** shall implement with the rules of the base protocol. See 9.1.2.5 Simple target socket b/nb conversion

f) A target is not obliged to register an **nb_transport_fw** callback with a simple target socket provided it has registered a **b_transport** callback, in which case an incoming **nb_transport_fw** call will automatically cause the target to call the method registered for **b_transport** and subsequently to call **nb_transport_bw** on the backward path.

g) If a target does not register either a **b_transport** or an **nb_transport_fw** callback with a simple target socket, this will result in a run-time error if and only if the corresponding method is called

h) A target should register **b_transport** and **nb_transport_fw** callbacks with a passthrough target socket. Not doing so will result in a run-time error if and only if the corresponding method is called.

i) A target is not obliged to register a **transport_dbg** callback with a simple target socket or a passthrough target socket, in which case an incoming **transport_dbg** call shall return with a value of 0.

j) A target is not obliged to register a **get_direct_mem_ptr** callback with a simple target socket or a passthrough target socket, in which case an incoming **get_direct_mem_ptr** call shall return with a value of false.

k) An initiator should register an **nb_transport_bw** callback with a simple initiator socket. Not doing so will result in a run-time error if and only if the **nb_transport_bw** method is called.

l) An initiator is not obliged to register an **invalidate_direct_mem_ptr** callback with a simple initiator socket, in which case an incoming **invalidate_direct_mem_ptr** call shall be ignored.

9.1.2.5 Simple target socket b/nb conversion

a) In the case that a **b_transport** or **nb_transport_fw** method is called through a socket of class **simple_target_socket** but no corresponding callback is registered, the simple target socket will act as an adapter between the two interfaces.

b) When the simple target socket acts as an adapter, it shall honor the rules of the base protocol both from the point of view of the initiator and from the point of view of the implementation of the **b_transport** or **nb_transport_fw** methods in the target. See 8.2 Base protocol

c) The socket shall pass through the given transaction object without modification and shall not construct a new transaction object.

d) In the case that only the **nb_transport_fw** callback has been registered by the target, the initiator is not permitted to call **nb_transport_fw** while there is an earlier **b_transport** call from the initiator still in progress. This is a limitation of the current implementation of the simple target socket.

Simple target socket nb/b adapter

Figure 19
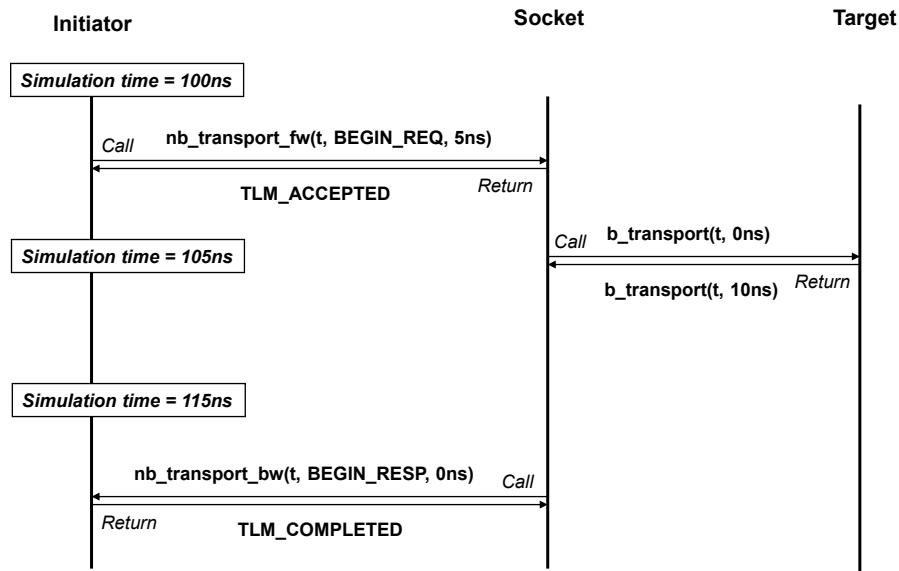
e) Figure 19 shows the case where an initiator calls **nb_transport_fw**, but the target only registers a **b_transport** callback with the simple target socket. The initiator sends BEGIN_REQ, to which the socket returns TLM_ACCEPTED. The socket then calls **b_transport**, and on return sends BEGIN_RESP back to the initiator, to which the initiator returns TLM_COMPLETED. Since it is not permissible in SystemC to call a blocking method directly from a non-blocking method, the socket is obliged to call **b_transport** from a separate internal thread process, not directly from **nb_transport_fw**.

f) Figure 19 shows just one possible scenario. On the final transition, the initiator could have returned TLM_ACCEPTED, in which case the socket would expect to receive a subsequent END_RESP from the initiator. Also, the target could have called **wait** from within **b_transport**.

g) Figure 20 shows the case where an initiator calls **b_transport**, but the target only registers an **nb_transport_fw** callback with the simple target socket. The initiator calls **b_transport**, then the socket and the target handshake using *nb_transport* and obeying the rules of the base protocol. The target may or may not send the END_REQ phase; it may jump straight to the BEGIN_RESP phase. The socket returns TLM_COMPLETED from the call to **nb_transport_bw** for the BEGIN_RESP phase.

Figure 20

Simple target socket b/nb adapter
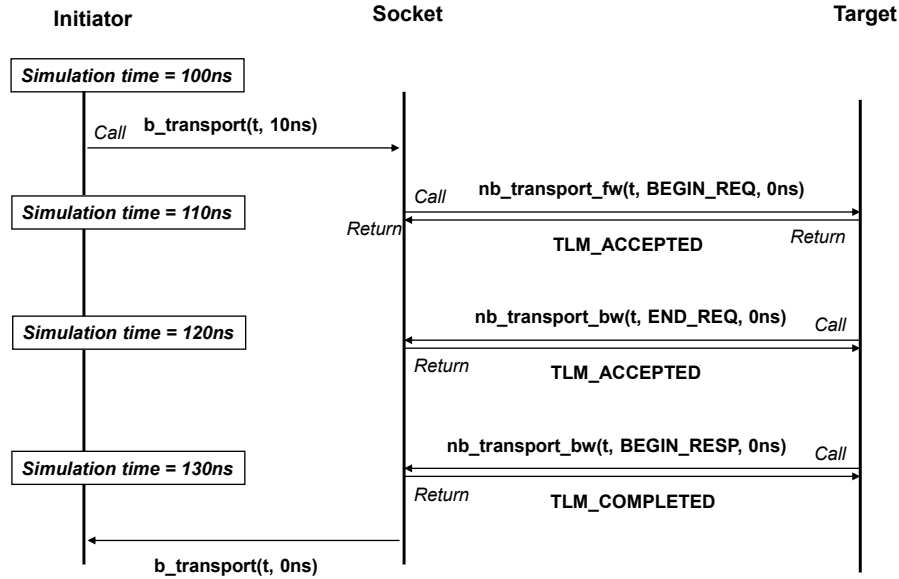
Example

```
#define SC_INCLUDE_DYNAMIC_PROCESSES
#include "tlm.h"
#include "tlm_utils/simple_initiator_socket.h"          // Header files from utilities
#include "tlm_utils/simple_target_socket.h"

struct Initiator: sc_module
{
   tlm_utils::simple_initiator_socket<Initiator, 32, tlm::tlm_base_protocol_types> socket;

   SC_CTOR(Initiator)
   : socket("socket")                                    // Construct and name simple socket
   {                                                      // Register callbacks with simple socket
      socket.register_nb_transport_bw(         this, &Initiator::nb_transport_bw );
      socket.register_invalidate_direct_mem_ptr(   this, &Initiator::invalidate_direct_mem_ptr );
   }
```

```
    virtual tlm::tlm_sync_enum nb_transport_bw(
        tlm::tlm_generic_payload& trans, tlm::tlm_phase& phase, sc_time& delay ) {
            return tlm::TLM_COMPLETED;                    // Dummy implementation
    }

    virtual void invalidate_direct_mem_ptr(sc_dt::uint64 start_range, sc_dt::uint64 end_range)
    { }                                                   // Dummy implementation
};


struct Target: sc_module                                  // Target component
{
    tlm_utils::simple_target_socket<Target, 32, tlm::tlm_base_protocol_types> socket;

    SC_CTOR(Target)
    : socket("socket")                                    // Construct and name simple socket
    {                                                     // Register callbacks with simple socket
        socket.register_nb_transport_fw(      this, &Target::nb_transport_fw );
        socket.register_b_transport(          this, &Target::b_transport );
        socket.register_get_direct_mem_ptr(   this, &Target::get_direct_mem_ptr );
        socket.register_transport_dbg(        this, &Target::transport_dbg );
    }

    virtual void b_transport( tlm::tlm_generic_payload& trans, sc_time& delay )
    { }                                                   // Dummy implementation

    virtual tlm::tlm_sync_enum nb_transport_fw(
        tlm::tlm_generic_payload& trans, tlm::tlm_phase& phase, sc_time& delay ) {
            return tlm::TLM_ACCEPTED;                     // Dummy implementation
    }

    virtual bool get_direct_mem_ptr( tlm::tlm_generic_payload& trans, tlm::tlm_dmi&  dmi_data)
    { return false; }                                     // Dummy implementation

    virtual unsigned int transport_dbg(tlm::tlm_generic_payload& r)
    { return 0; }                                         // Dummy implementation
};

SC_MODULE(Top)
{
    Initiator *initiator;
    Target    *target;
    SC_CTOR(Top) {
        initiator  = new Initiator("initiator");
        target     = new Target("target");
        initiator->socket.bind( target->socket );         // Bind initiator socket to target socket
```

```
      }
  };
```

9.1.3   Tagged simple sockets

9.1.3.1   Introduction

The tagged simple sockets are a variation on the simple sockets that tag incoming interface method calls with an integer id that allows the callback to identify through which socket the incoming call arrived. This is useful in the case where the same callback method is registered with multiple initiator sockets or multiple target sockets. The id is specified when the callback is registered, and gets inserted as an extra first argument to the callback method.

9.1.3.2   Header file

The class definitions for the tagged simple sockets shall be in the same header files as the corresponding simple sockets, that is **tlm_utils/simple_initiator_socket.h**, **tlm_utils/simple_target_socket.h,** and **tlm_utils/passthrough_target_socket.h**.

9.1.3.3   Class definition

```
  namespace tlm_utils {

  template <
    typename MODULE,
    unsigned int BUSWIDTH = 32,
    typename TYPES = tlm::tlm_base_protocol_types
  >
  class simple_initiator_socket_tagged : public tlm::tlm_initiator_socket<BUSWIDTH, TYPES>
  {
  public:
    typedef typename TYPES::tlm_payload_type  transaction_type;
    typedef typename TYPES::tlm_phase_type    phase_type;
    typedef tlm::tlm_sync_enum                sync_enum_type;

    simple_initiator_socket_tagged();
    explicit simple_initiator_socket_tagged( const char* n );

    void register_nb_transport_bw(
      MODULE* mod,
      sync_enum_type (MODULE::*cb)(int, transaction_type&,  phase_type&,   sc_core::sc_time&),
      int id);

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

```
      int id);
};


template <
   typename MODULE,
   unsigned int BUSWIDTH = 32,
   typename TYPES = tlm::tlm_base_protocol_types
>
class simple_target_socket_tagged : public tlm::tlm_target_socket<BUSWIDTH, TYPES>
{
public:
   typedef typename TYPES::tlm_payload_type         transaction_type;
   typedef typename TYPES::tlm_phase_type           phase_type;
   typedef tlm::tlm_sync_enum                       sync_enum_type;
   typedef tlm::tlm_fw_transport_if<TYPES>          fw_interface_type;
   typedef tlm::tlm_bw_transport_if<TYPES>          bw_interface_type;
   typedef tlm::tlm_target_socket<BUSWIDTH, TYPES>  base_type;

   simple_target_socket_tagged();
   explicit simple_target_socket_tagged( const char* n );

   tlm::tlm_bw_transport_if<TYPES> * operator ->();

   void register_nb_transport_fw(
      MODULE* mod,
      sync_enum_type (MODULE::*cb)(int id, transaction_type&, phase_type&, sc_core::sc_time&),
      int id);

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

   void register_transport_dbg(
      MODULE* mod,
      unsigned int (MODULE::*cb)(int id, transaction_type&),
      int id);

   void register_get_direct_mem_ptr(
      MODULE* mod,
      bool (MODULE::*cb)(int id, transaction_type&, tlm::tlm_dmi&),
      int id);
};
```

```
    template <
      typename MODULE,
      unsigned int BUSWIDTH = 32,
      typename TYPES = tlm::tlm_base_protocol_types
    >
    class passthrough_target_socket_tagged : public tlm::tlm_target_socket<BUSWIDTH, TYPES>
    {
    public:
      typedef typename TYPES::tlm_payload_type  transaction_type;
      typedef typename TYPES::tlm_phase_type    phase_type;
      typedef tlm::tlm_sync_enum                sync_enum_type;

      passthrough_target_socket_tagged();
      explicit passthrough_target_socket_tagged( const char* n );

      void register_nb_transport_fw(
        MODULE* mod,
        sync_enum_type (MODULE::*cb)(int id, transaction_type&, phase_type&, sc_core::sc_time&),
        int id);

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

      void register_transport_dbg(
        MODULE* mod,
        unsigned int (MODULE::*cb)(int id, transaction_type&),
        int id);

      void register_get_direct_mem_ptr(
        MODULE* mod,
        bool (MODULE::*cb)(int id, transaction_type&, tlm::tlm_dmi&),
        int id);
    };

    } // namespace tlm_utils
```

9.1.3.4   Rules

a) Each constructor shall call the constructor of the corresponding base class passing through the **char***
    argument, if there is one. In the case of the default constructors, the **char*** argument of the base class
    constructor   shall   be   set   to   sc_gen_unique_name   ("simple_initiator_socket_tagged"   ),
    sc_gen_unique_name   ("simple_target_socket_tagged"   ),   or   sc_gen_unique_name   (
    "passthrough_target_socket_tagged") respectively.

b) Apart from the int id tag, the tagged simple sockets behave in the same way as the untagged simple sockets.

c) A given callback method can be registered with multiple sockets instances using different tags. This is the purpose of the tagged sockets.

d) The int id tag is specified as the final argument of the methods used to register the callbacks. The socket shall prepend this tag as the first argument of the corresponding callback method.

e) A tagged simple socket is not a multi-socket. A tagged simple socket cannot be bound to multiple sockets on other components. See 9.1.4 Multi-sockets.

9.1.4 Multi-sockets

9.1.4.1 Introduction

The multi-sockets are a variation on the tagged simple sockets that permit a single socket to be bound to multiple sockets on other components. In contrast to the tagged simple sockets, which identify through which socket an incoming call arrives, a multi-socket callback is able to identify from which socket on another component an incoming interface method call arrives, using the multi-port index number as the tag. Unlike the other convenience sockets, the multi-sockets also support hierarchical child-to-parent socket binding on both the initiator and target side.

9.1.4.2 Header file

The class definitions for the multi-sockets shall be in the header files **tlm_utils/multi_passthrough_initiator_socket.h** and **tlm_utils/multi_passthrough_target_socket.h**.

9.1.4.3 Class definition

```
namespace tlm_utils {

template <
    typename MODULE,
    unsigned int BUSWIDTH = 32,
    typename TYPES = tlm::tlm_base_protocol_types,
    unsigned int N=0,
    sc_core::sc_port_policy POL = sc_core::SC_ONE_OR_MORE_BOUND
>
class multi_passthrough_initiator_socket : public multi_init_base< BUSWIDTH, TYPES, N, POL>
{
public:
    typedef typename TYPES::tlm_payload_type              transaction_type;
    typedef typename TYPES::tlm_phase_type               phase_type;
    typedef tlm::tlm_sync_enum                          sync_enum_type;
    typedef multi_init_base<BUSWIDTH, TYPES, N, POL>     base_type;
    typedef typename base_type::base_target_socket_type  base_target_socket_type;
```

```
    multi_passthrough_initiator_socket();
    multi_passthrough_initiator_socket(const char* name);
    ~multi_passthrough_initiator_socket();

    void register_nb_transport_bw(
        MODULE* mod,
        sync_enum_type (MODULE::*cb)(int, transaction_type&, phase_type&, sc_core::sc_time&));

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

    // Override virtual functions of the tlm_initiator_socket:
    virtual tlm::tlm_bw_transport_if<TYPES>& get_base_interface();
    virtual sc_core::sc_export<tlm::tlm_bw_transport_if<TYPES> >& get_base_export();

    void bind(base_target_socket_type& s);
    void operator() (base_target_socket_type& s);

    // SystemC standard callback
    //  multi_passthrough_initiator_socket::before_end_of_elaboration must be called from
    // any derived class
    void before_end_of_elaboration();

    // Bind multi initiator socket to multi initiator socket (hierarchical bind)
    void bind(base_type& s);
    void operator() (base_type& s);

    tlm::tlm_fw_transport_if<TYPES>* operator[](int i);
    unsigned int size();
};


template <
    typename MODULE,
    unsigned int BUSWIDTH = 32,
    typename TYPES = tlm::tlm_base_protocol_types,
    unsigned int N=0,
    sc_core::sc_port_policy POL = sc_core::SC_ONE_OR_MORE_BOUND
>
class multi_passthrough_target_socket : public multi_target_base< BUSWIDTH, TYPES, N, POL>
{
public:
    typedef typename TYPES::tlm_payload_type    transaction_type;
    typedef typename TYPES::tlm_phase_type      phase_type;
    typedef tlm::tlm_sync_enum                  sync_enum_type;
```

```
    typedef sync_enum_type
                        (MODULE::*nb_cb)(int, transaction_type&, phase_type&, sc_core::sc_time&);
    typedef void        (MODULE::*b_cb)(int, transaction_type&, sc_core::sc_time&);
    typedef unsigned int  (MODULE::*dbg_cb)(int, transaction_type& txn);
    typedef bool        (MODULE::*dmi_cb)(int, transaction_type& txn, tlm::tlm_dmi& dmi);

    typedef multi_target_base<BUSWIDTH, TYPES, N, POL> base_type;
    typedef typename base_type::base_initiator_socket_type     base_initiator_socket_type;
    typedef typename base_type::initiator_socket_type          initiator_socket_type;

    multi_passthrough_target_socket();
    multi_passthrough_target_socket(const char* name);
    ~multi_passthrough_target_socket();

    void register_nb_transport_fw     (MODULE* mod, nb_cb cb);
    void register_b_transport         (MODULE* mod, b_cb cb);
    void register_transport_dbg       (MODULE* mod, dbg_cb cb);
    void register_get_direct_mem_ptr(MODULE* mod, dmi_cb cb);

    // Override virtual functions of the tlm_target_socket:
    virtual tlm::tlm_fw_transport_if<TYPES>& get_base_interface();
    virtual sc_core::sc_export<tlm::tlm_fw_transport_if<TYPES> >& get_base_export();

    // SystemC standard callback
    //  multi_passthrough_target_socket::end_of_elaboration must be called from any derived class
    void end_of_elaboration();

    void bind(base_type& s);
    void operator() (base_type& s);

    tlm::tlm_bw_transport_if<TYPES>* operator[] (int i);
    unsigned int size();
};

} // namespace tlm_utils
```

9.1.4.4   Rules

a) The base classes *multi_init_base* and *multi_target_base* are implementation-defined, and should not be used directly by applications.

b) Each constructor shall call the constructor of the corresponding base class passing through the **char*** argument, if there is one. In the case of the default constructors, the **char*** argument of the base class constructor shall be set to sc_gen_unique_name ("multi_passthrough_initiator_socket"), or sc_gen_unique_name ("multi_passthrough_target_socket") respectively.

c) Class **multi_passthrough_initiator_socket** and class **multi_passthrough_target_socket** each act as multi-sockets, that is, a single initiator socket can be bound to multiple target sockets, and a single target socket can be bound to multiple initiator sockets. The two class templates have template parameters specifying the number of bindings and the port binding policy, which are used within the class implementation to parameterize the associated **sc_port** template instantiation.

d) A single **multi_passthrough_initiator_socket** can be bound to many **tlm_target_sockets** and/or many **simple_target_sockets** and/or many **passthrough_target_sockets** and/or many **multi_passthrough_target_sockets.** Many **tlm_initiator_sockets** and/or **simple_initiator_sockets** and/or **multi_passthrough_initiators_sockets** can be bound to a single **multi_passthrough_target_socket.**

e) A **multi_passthrough_initiator_socket** can be bound hierarchically to exactly one other **multi_passthrough_initiator_socket.** A **multi_passthrough_target_socket** can be bound hierarchically to exactly one other **multi_passthrough_target_socket.** Other than these two specific cases, a multi-socket cannot be bound hierarchically to another socket. The multiple binding capabilities of multi-sockets do not apply to hierarchical binding, but only apply when binding one or more initiator sockets to one or more target sockets.

f) The binding operators can only be used in the direction initiator-socket-to-target-socket. In other words, unlike classes **tlm_target_socket** and **simple_target_socket**, class **multi_passthrough_target_socket** does not have operators to bind a target socket to an initiator socket.

g) In the case of hierarchical binding, an initiator multi-socket of a child module shall be bound to an initiator multi-socket of a parent module, and a target multi-socket of a parent module bound to a target multi-socket of a child module. This is consistent with the initiator-to-target binding direction rule given above.

h) If an object of class **multi_passthrough_initiator_socket** or **multi_passthrough_target_socket** is bound multiple times, then the method **operator[]** can be used to address the corresponding object to which the socket is bound. The index value is determined by the order in which the methods **bind** or **operator()** were called to bind the sockets. This same index value is used to determine the id tag passed to a callback.

i) For example, consider a **multi_passthrough_initiator_socket** bound to two separate targets. The calls **socket[0]->nb_transport_fw(...)** and **socket[1]->nb_transport_fw()** would address the two targets, and incoming **nb_transport_bw()** method calls from those two targets would carry the tags 0 and 1 respectively.

j) The method **size** shall return the number of socket instances to which the current multi-socket has been bound. As for SystemC multi-ports, if **size** is called during elaboration and before the callback **end_of_elaboration**, the value returned is implementation-defined because the time at which port binding is completed is implementation-defined.

k) In the absence of hierarchical binding to a multi-socket on a child module, a target should register **b_transport** and **nb_transport_fw** callbacks with a target multi-socket. Not doing so will result in a run-time error if and only if the corresponding method is called.

l) In the absence of hierarchical binding to a multi-socket on a child module, a target is not obliged to register a **transport_dbg** callback with a target multi-socket, in which case an incoming **transport_dbg** call shall return with a value of 0.

m) In the absence of hierarchical binding to a multi-socket on a child module, a target is not obliged to register a **get_direct_mem_ptr** callback with a target multi-socket, in which case an incoming **get_direct_mem_ptr** call shall return with a value of false.

n) In the absence of hierarchical binding to a multi-socket on a child module, an initiator should register an **nb_transport_bw** callback with an initiator multi-socket. Not doing so will result in a run-time error if and only if the **nb_transport_bw** method is called.

o) In the absence of hierarchical binding to a multi-socket on a child module, an initiator is not obliged to register an **invalidate_direct_mem_ptr** callback with an initiator multi-socket, in which case an incoming **invalidate_direct_mem_ptr** call shall be ignored.

Example

```
// Initiator component with a multi-socket
struct Initiator: sc_module
{
    tlm_utils::multi_passthrough_initiator_socket<Initiator> socket;

    SC_CTOR(Initiator) : socket("socket") {
        // Register callback methods with socket
        socket.register_nb_transport_bw(this, &Initiator::nb_transport_bw);
        socket.register_invalidate_direct_mem_ptr(this, &Initiator::invalidate_direct_mem_ptr);
        ...
};

struct Initiator_parent: sc_module
{
    tlm_utils::multi_passthrough_initiator_socket<Initiator_parent> socket;
    Initiator  *initiator;

    SC_CTOR(Initiator_parent) : socket("socket") {
        initiator = new Initiator("initiator");
        // Hierarchical binding of initiator socket on child to initiator socket on parent
        initiator->socket.bind( socket );
    }
};
```

Figure 21

Hierarchical Binding of Multi-sockets
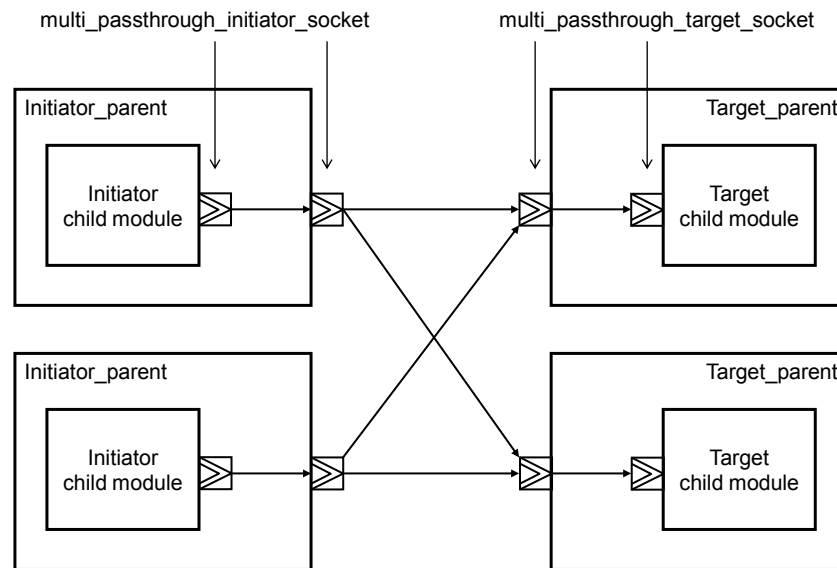

```
struct Target: sc_module
{
    tlm_utils::multi_passthrough_target_socket<Target> socket;

    SC_CTOR(Target) : socket("socket") {
        // Register callback methods with socket
        socket.register_nb_transport_fw(   this, &Target::nb_transport_fw);
        socket.register_b_transport(       this, &Target::b_transport);
        socket.register_get_direct_mem_ptr(this, &Target::get_direct_mem_ptr);
        socket.register_transport_dbg(     this, &Target::transport_dbg);
        ...
};

// Target component with a multi-socket
struct Target_parent: sc_module
{
    tlm_utils::multi_passthrough_target_socket<Target_parent> socket;
    Target  *target;

    SC_CTOR(Target_parent) : socket("socket") {
        target = new Target("target");
        // Hierarchical binding of target socket on parent to target socket on child
```

```
            socket.bind( target->socket );
        }
    };

    SC_MODULE(Top)
    {
        Initiator_parent    *initiator1;
        Initiator_parent    *initiator2;
        Target_parent       *target1;
        Target_parent       *target2;

        SC_CTOR(Top)
        {
            // Instantiate two initiator and two target components
            initiator1   = new Initiator_parent("initiator1");
            initiator2   = new Initiator_parent("initiator2");
            target1      = new Target_parent   ("target1");
            target2      = new Target_parent   ("target2");

            // Bind two initiator multi-sockets to two target multi-sockets
            initiator1->socket.bind( target1->socket );
            initiator1->socket.bind( target2->socket );
            initiator2->socket.bind( target1->socket );
            initiator2->socket.bind( target2->socket );
        }
    };
```

9.2   Quantum keeper

9.2.1   Introduction

Temporal decoupling permits SystemC processes to run ahead of simulation time for an amount of time known as the time quantum. See 5 Global quantum

The utility class **tlm_quantumkeeper** provides a set of methods for managing and interacting with the time quantum. When using temporal decoupling, use of the quantum keeper is recommended in order to maintain a consistent coding style. However, it is straightforward in principle to implement temporal decoupling directly in SystemC. Whether or not the utility class **tlm_quantumkeeper** is used, all temporally decoupled models should reference the global quantum maintained by the class **tlm_global_quantum**.

Class **tlm_quantumkeeper** is in namespace **tlm_utils**.

For a general description of temporal decoupling, see 3.3.2 Loosely-timed coding style and temporal decoupling

For a description of timing annotation, see 4.1.3 Timing annotation with the transport interfaces

9.2.2   Header file

The class definitions for the quantum keeper shall be in the header file **tlm_utils/tlm_quantumkeeper.h**.

9.2.3   Class definition

```
namespace tlm_utils {

class tlm_quantumkeeper
{
public:
    static void set_global_quantum( const sc_core::sc_time& );
    static const sc_core::sc_time& get_global_quantum();

    tlm_quantumkeeper();
    virtual ~tlm_quantumkeeper();

    virtual void inc( const sc_core::sc_time& );
    virtual void set( const sc_core::sc_time& );
    virtual sc_core::sc_time get_current_time() const;
    virtual sc_core::sc_time get_local_time();

    virtual bool need_sync() const;
    virtual void sync();

    void set_and_sync(const sc_core::sc_time& t)
    {
     set(t);
```

```
    if (need_sync())
      sync();
  }

  virtual void reset();

protected:
  virtual sc_core::sc_time compute_local_quantum();
};

} // namespace tlm_utils
```

9.2.4   General guidelines for processes using temporal decoupling

a) For maximum simulation speed, all initiators should use temporal decoupling, and the number of other runnable SystemC processes should be zero or minimized.

b) In an ideal scenario, the only runnable SystemC processes will belong to temporally decoupled initiators, and each process will run ahead to the end of its time quantum before yielding to the SystemC kernel.

c) A temporally decoupled initiator is not obliged to use a time quantum if communication with other processes is explicitly synchronized. Where a time quantum is used, it should be chosen to be less than the typical communication interval between initiators, otherwise important process interactions may be lost, and the model broken.

d) *Yield* means call **wait** in the case of a thread process, or return from the function in the case of a method process.

e) Temporal decoupling runs in the context of the standard SystemC simulation kernel, so events can be scheduled, processes suspended and resumed, and loosely-timed models can be mixed with other coding styles.

f) There is no obligation for every initiator to use temporal decoupling. Processes with and without temporal decoupling can be mixed. However, any process that is not temporally decoupled is likely to become a simulation speed bottleneck.

g) Each temporally decoupled initiator may accumulate any local processing delays and communication delays in a local variable, referred to in this clause as the *local time offset*. It is recommended that the quantum keeper should be used to maintain the local time offset.

h) Calls to the **sc_time_stamp** method will return the simulation time as it was at or near the start of the current time quantum.

Figure 22

Quantum Keeper Terminology
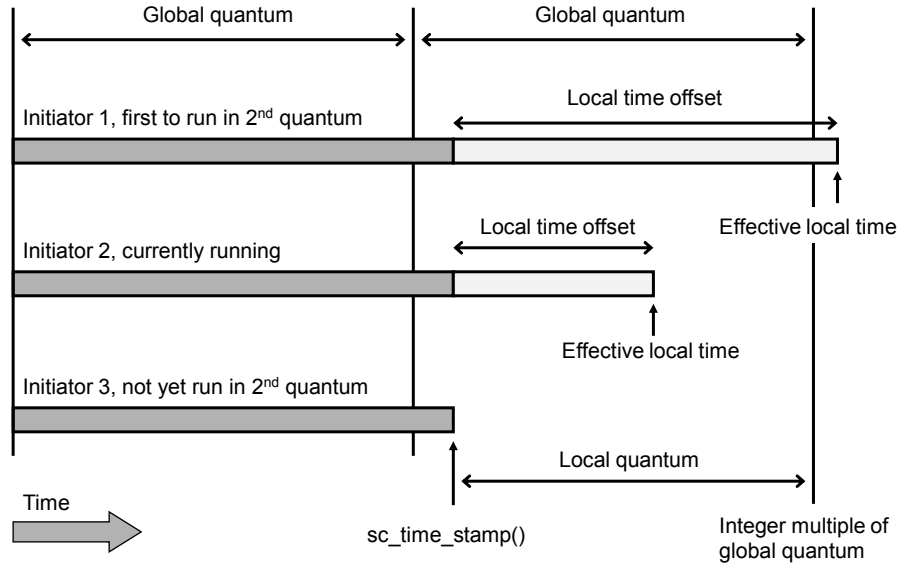
i) The local time offset is unknown to the SystemC scheduler. When using the transport interfaces, the local time offset should be passed as an argument to the **b_transport** or *nb_transport* methods.

j) Use of the *nb_transport* method with temporal decoupling and the quantum keeper is not ruled out, but is not usually advantageous because the speed advantage to be gained from temporal decoupling would be nullified by the high degree of inter-process communication inherent in the approximately-timed coding style.

k) The order in which processes resume within the quantum is under the control of the SystemC scheduler, and by the rules of SystemC, is indeterminate. In the absence of any explicit synchronization mechanism, if a variable is modified by one such process and read by another, the value to be read will be indeterminate. The new value may become available in the current quantum or the next quantum, assuming it only changes relatively infrequently compared to the quantum length, and the application would need to be tolerant of precisely when the new value becomes available. If this is not the case, the application should guard the variable access with an appropriate synchronization mechanism.

l) Any access to a variable or object from a temporally decoupled process will give the value it had at the start of the current time quantum unless it has been modified by the current process or by another temporally decoupled process that has already run in the current quantum. In particular, any **sc_signal** accessed from a temporally decoupled process will have the same value it had at the start of the current time quantum. This is a consequence of the fact that conventional SystemC simulation time (as returned by **sc_time_stamp**) does not advance within the quantum.

9.2.5   Class tlm_quantumkeeper

a)   The constructor shall set the local time offset to SC_ZERO_TIME but shall not call the virtual method **compute_local_quantum**. Because the constructor does not calculate the local quantum, an application should call the method **reset** immediately after constructing a quantum keeper object.

b)   The implementation of class **tlm_quantum_keeper** shall not create a static object of class **sc_time**, but the constructor may create objects of class **sc_time**. This implies that an application may call function **sc_core::sc_set_time_resolution** before, and only before, constructing the first quantum keeper object.

c)   The method **set_global_quantum** shall set the value of the global quantum to the value passed as an argument, but shall not modify the local quantum. The method **get_global_quantum** shall return the current value of the global quantum. After calling **set_global_quantum** it is recommended to call the method **reset** to recalculate the local quantum.

d)   The method **get_local_time** shall return the value of the local time offset.

e)   The method **get_current_time** shall return the value of the effective local time, that is, **sc_time_stamp() + local_time_offset**

f)   The method **inc** shall add the value passed as an argument to the local time offset.

g)   The method **set** shall set the value of the local time offset to the value passed as an argument.

h)   The method **need_sync** shall return the **value** true if and only if the local time offset is greater than the local quantum.

i)   The method **sync** shall call **wait(local_time_offset)** to suspend the process until simulation time equals the effective local time, and shall then call method **reset**..

j)   The method **set_and_sync** is a convenience method to call **set**, **need_sync**, and **sync** in sequence. It should not be overridden.

k)   The method **reset** shall call the method **compute_local_quantum** and shall set the local time offset back to SC_ZERO_TIME.

l)   The method **compute_local_quantum** of class **tlm_quantumkeeper** shall call the method **compute_local_quantum** of class **tlm_global_quantum**, but may be overridden in order to calculate a smaller value for the local quantum.

m)   The class **tlm_quantumkeeper** should be considered the default implementation for the quantum keeper. Applications may derive their own quantum keeper from class **tlm_quantumkeeper** and override the method **compute_local_quantum**, but this is unusual.

n)   When the local time offset is greater than or equal to the local quantum, the process should yield to the kernel. It is strongly recommended that the process does this by calling the **sync** method.

o)   There is no mechanism to enforce synchronization at the end of the time quantum. It is the responsibility of the initiator to check **need_sync** and call **sync** as needed.

p)   The **b_transport** method may itself yield such that the value of **sc_time_stamp** can be different before and after the call. The value of the local time offset and any timing annotations are always expressed relative to the current value of **sc_time_stamp**. On return from **b_transport** or **nb_transport_fw**, it is

the responsibility of the initiator to set the local time offset of the quantum keeper by calling the **set** method, then check for synchronization by calling the **need_sync** method.

q) If an initiator needs to synchronize before the end of the time quantum, that is, if an initiator needs to suspend execution so that simulation time can catch up with the local time, it may do so by calling the **sync** method or by explicitly waiting on an event. This gives any other processes the chance to execute, and is known as synchronization-on-demand.

r) Making frequent calls to **sync** will reduce the effectiveness of temporal decoupling.

Example

```
struct Initiator: sc_module                           // Loosely-timed initiator
{
    tlm_utils::simple_initiator_socket<Initiator> init_socket;

    tlm_utils::tlm_quantumkeeper m_qk;                // The quantum keeper

    SC_CTOR(Initiator) : init_socket("init_socket") {
        SC_THREAD(thread);                            // The initiator process
        ...
        m_qk.set_global_quantum( sc_time(1, SC_US) ); // Replace the global quantum
        m_qk.reset();                                 // Re-calculate the local quantum
    }

    void thread() {
        tlm::tlm_generic_payload trans;
        sc_time delay;
        trans.set_command(tlm::TLM_WRITE_COMMAND);
        trans.set_data_length(4);

        for (int i = 0; i < RUN_LENGTH; i += 4) {
            int word = i;
            trans.set_address(i);
            trans.set_data_ptr( (unsigned char*)(&word) );

            delay = m_qk.get_local_time();            // Annotate b_transport with local time
            init_socket->b_transport(trans, delay );
            qk.set( delay );                          // Update qk with time consumed by target

            m_qk.inc( sc_time(100, SC_NS) );          // Further time consumed by initiator
            if ( m_qk.need_sync() ) m_qk.sync();      // Check local time against quantum
        }
    }
    ...
};
```

9.3   Payload event queue

9.3.1   Introduction

A payload event queue (PEQ) is a class that maintains a queue of SystemC event notifications, where each notification carries an associated transaction object. Each transaction is written into the PEQ annotated with a delay, and each transaction emerges from the back of the PEQ at a time calculated from the current simulation time plus the annotated delay.

Two payload event queues are provided as utilities. As well as being useful in their own right, the PEQ is of conceptual relevance in understanding the semantics of timing annotation with the approximately-timed coding style. However, it is possible to implement approximately-timed models without using the specific payload event queues given here. In an approximately-timed model, it is often appropriate for the recipient of a transaction passed using *nb_transport* to put the transaction into a PEQ with the annotated delay. The PEQ will schedule the timing point associated with the *nb_transport* call to occur at the correct simulation time.

Transactions are inserted into a PEQ by calling the **notify** method of the PEQ, passing a delay as an argument. There is also a **notify** method that schedules an immediate notification. The delay is added to the current simulation time (**sc_time_stamp**) to calculate the time at which the transaction will emerge from the back end of the PEQ. The scheduling of the events is managed internally using a SystemC timed event notification, exploiting the property of class **sc_event** that if the **notify** method is called whilst there is a notification pending, the notification with the earliest simulation time will remain while the other notification gets cancelled.

Transactions emerge in different ways from the two PEQ variants. In the case of **peq_with_get**, the method **get_event** returns an event that is notified whenever a transaction is ready to be retrieved. The method **get_next_transaction** should be called repeatedly to retrieve any available transactions one at a time.

In the case of **peq_with_cb_and_phase**, a callback method is registered as a constructor argument, and that method is called as each transaction emerges. This particular PEQ carries both a transaction object and a phase object with each notification, and both are passed as arguments to the callback method.

For an example, see 8.1 Phases

The current implementation of **peq_with_cb_and_phase** makes use of dynamic processes. Hence, when compiling applications that use **peq_with_cb_and_phase** with current released versions of the OSCI proof-of-concept simulator, it is necessary to define the macro SC_INCLUDE_DYNAMIC_PROCESSES before including the SystemC header file.

9.3.2   Header file

The class definitions for the two payload event queues shall be in the header files **tlm_utils/peq_with_get.h** and **tlm_utils/peq_with_cb_and_phase.h**.

9.3.3   Class definition

    namespace tlm_utils {

    template <class PAYLOAD>

```
    class peq_with_get : public sc_core::sc_object
    {
    public:
        typedef PAYLOAD transaction_type;

        peq_with_get(const char* name);

        void notify( transaction_type& trans, const sc_core::sc_time& t );
        void notify( transaction_type& trans );

        transaction_type* get_next_transaction();
        sc_core::sc_event& get_event();
        void cancel_all();
    };

    template<typename OWNER, typename TYPES=tlm::tlm_base_protocol_types>
    class peq_with_cb_and_phase : public sc_core::sc_object
    {
    public:
        typedef typename TYPES::tlm_payload_type          tlm_payload_type;
        typedef typename TYPES::tlm_phase_type            tlm_phase_type;
        typedef void (OWNER::*cb)(tlm_payload_type&, const tlm_phase_type&);

        peq_with_cb_and_phase(OWNER* , cb );
        peq_with_cb_and_phase(const char* , OWNER* , cb);
        ~peq_with_cb_and_phase();

        void notify ( tlm_payload_type& , const tlm_phase_type& , const sc_core::sc_time& );
        void notify ( tlm_payload_type& , const tlm_phase_type& );
        void cancel_all();
    };

    } // namespace tlm_utils
```

9.3.4   Rules

a)   The **notify** method shall insert a transaction into the PEQ. The transaction shall emerge from the PEQ at time **t1 + t2**, where **t1** is the value returned from **sc_time_stamp()** at the time **notify** is called, and **t2** is the value of the **sc_time** argument to **notify**. In the case of immediate notification, the transaction shall emerge in the current evaluation phase of the SystemC scheduler.

b)   Transactions may be queued in any order and emerge in the order given by the previous rule. Transactions do not necessarily emerge in the order in which they were inserted.

c)   There is no limit to the number of transactions that may be in the PEQ at any given time.

d) If several transactions are queued to emerge at the same time, they shall all emerge in the same evaluation phase (that is, the same delta cycle) in the order in which they were inserted.

e) The **cancel_all** method shall immediately remove all queued transactions from the PEQ, effectively restoring the PEQ to the state it had immediately after construction. This is the only way to remove transactions from a PEQ.

f) The **PAYLOAD** template argument to class **peq_with_get** shall be the name of the transaction type used by the PEQ.

g) The **get_event** method shall return a reference to an event that is notified when the next transaction is ready to emerge from the PEQ. If more than one transaction is ready to emerge in the same evaluation phase (that is, in the same delta cycle), the event is notified once only.

h) The **get_next_transaction** method shall return a pointer to a transaction object that is ready to emerge from the PEQ, and shall remove the transaction object from the PEQ. If a transaction is not retrieved from the PEQ in the evaluation phase in which the corresponding event notification occurs, it will still be available for retrieval on a subsequent call to **get_next_transaction** at the current time or at a later time.

i) If there are no more transactions to be retrieved in the current evaluation phase, **get_next_transaction** shall return a null pointer.

j) The **TYPES** template argument to class **peq_with_cb_and_phase** shall be the name of the protocol traits class containing the transaction and phase types used by the PEQ.

k) The **OWNER** template argument to class **peq_with_cb_and_phase** shall be the type of the class of which the PEQ callback method is a member. This will usually be the parent module of the PEQ instance.

l) The **OWNER*** argument to the constructor **peq_with_cb_and_phase** shall be a pointer to the object of which the PEQ callback method is a member. This will usually be the parent module of the PEQ instance.

m) The **cb** argument to the constructor **peq_with_cb_and_phase** shall be the name of the PEQ callback method, which shall be a member function.

n) The implementation of class **peq_with_cb_and_phase** shall call the PEQ callback method whenever a transaction object is ready to emerge from the PEQ. The first argument of the callback is a reference to the transaction object and the second argument a reference to the phase object, as passed to the corresponding **notify** method.

o) The implementation shall call the PEQ callback method from a SystemC method process, so the callback method shall be non-blocking.

p) The implementation shall only call the PEQ callback method once for each transaction. After calling the PEQ callback method, the implementation shall remove the transaction object from the PEQ. The PEQ callback method may be called multiple times in the same evaluation phase.

9.4   Instance-specific extensions

9.4.1   Introduction

The generic payload contains an array of pointers to extension objects such that each transaction object can contain at most one instance of each extension type. This mechanism alone does not directly permit multiple instances of the same extension to be added to a given transaction object. This clause describes a set of utilities that provide instance-specific extensions, that is, multiple extensions of the same type added to a single transaction object.

An instance-specific extension type is created using a class template **instance_specific_extension**, used in a similar manner to class **tlm_extension**. Unlike **tlm_extension**, applications are not required or permitted to implement virtual **clone** and **copy_from** methods. The access methods are restricted to **set_extension**, **get_extension**, **clear_extension** and **resize_extensions**. Automatic deletion of instance-specific extensions is not supported, so a component calling **set_extension** should also call **clear_extension**. As for class **tlm_extension**, method **resize_extensions** need only be called if a transaction object is constructed during static initialization.

An instance-specific extension is accessed using an object of type **instance_specific_extension_accessor**. This class provides a single method **operator()** which returns a proxy object through which the access methods can be called. Each object of type **instance_specific_extension_accessor** gives access to a distinct set of extension objects, even when used with the same transaction object.

In the class definition below, terms in *italics* are implementation-defined names that should not be used directly by an application..

9.4.2   Header file

The   class   definitions   for   the   instance-specific   extensions   shall   be   in   the   header   file **tlm_utils/instance_specific_extensions.h**

9.4.3   Class definition

```
namespace tlm_utils {

template <typename T>
class instance_specific_extension : public implementation-defined {
public:
    virtual ~instance_specific_extension();
};

template<typename U>
class proxy {
public:
    template <typename T> T* set_extension( T* );
    template <typename T> void get_extension( T*& ) const;
    template <typename T> void clear_extension( const T* );
```

```
        void resize_extensions();
};


class instance_specific_extension_accessor {
public:
    instance_specific_extension_accessor();

    template<typename T> proxy< implementation-defined >& operator() ( T& );
};


} // namespace tlm_utils
```

Example

```
struct my_extn : tlm_utils::instance_specific_extension<my_extn> {
 int num;                                               // User-defined extension attribute
};

struct Interconnect: sc_module
{
    tlm_utils::simple_target_socket<Interconnect>    targ_socket;
    tlm_utils::simple_initiator_socket<Interconnect> init_socket;
    ...
    tlm_utils::instance_specific_extension_accessor  accessor;
    static int count;

    virtual tlm::tlm_sync_enum nb_transport_fw(
        tlm::tlm_generic_payload& trans, tlm::tlm_phase& phase,  sc_time& delay )
    {
        my_extn* extn;
        accessor(trans).get_extension(extn);           // Get existing extension
        if (extn) {
            accessor(trans).clear_extension(extn);     // Delete existing extension
        } else {
            extn = new my_extn;
            extn->num = count++;
            accessor(trans).set_extension(extn);       // Add new extension
        }
        return init_socket->nb_transport_fw( trans, phase, delay );
    } ...
};

... SC_CTOR(Top) {
    // Transaction object passes through two instances of Interconnect
    interconnect1 = new Interconnect("interconnect1");
```

```
interconnect2 = new Interconnect("interconnect2");
interconnect1->init_socket.bind( interconnect2->targ_socket );
...
```

10   TLM-1 and analysis ports

The following TLM-1 core interfaces together with the **tlm_fifo** channel, the analysis interface, and the analysis ports are still current OSCI standards and are shipped with the TLM-2.0 software distribution. However, they are separate from the main body of the TLM-2.0 standard, and not documented in detail here.

10.1   TLM-1 core interfaces

The transport method with the signature **transport(const REQ& , RSP&)** was not part of TLM-1, but has been added in TLM-2.0.

```
namespace tlm {

// Bidirectional blocking interfaces
template < typename REQ , typename RSP >
class tlm_transport_if : public virtual sc_core::sc_interface
{
public:
  virtual RSP transport( const REQ& ) = 0;
  virtual void transport( const REQ& req , RSP& rsp ) { rsp = transport( req ); }
};

// Uni-directional blocking interfaces
template < typename T >
class tlm_blocking_get_if : public virtual sc_core::sc_interface
{
public:
  virtual T get( tlm_tag<T> *t = 0 ) = 0;
  virtual void get( T &t ) { t = get(); }
};

template < typename T >
class tlm_blocking_put_if : public virtual sc_core::sc_interface
{
public:
  virtual void put( const T &t ) = 0;
};

// Uni-directional non blocking interfaces
template < typename T >
class tlm_nonblocking_get_if : public virtual sc_core::sc_interface
{
public:
  virtual bool nb_get( T &t ) = 0;
  virtual bool nb_can_get( tlm_tag<T> *t = 0 ) const = 0;
```

```
  virtual const sc_core::sc_event &ok_to_get( tlm_tag<T> *t = 0 ) const = 0;
};


template < typename T >
class tlm_nonblocking_put_if : public virtual sc_core::sc_interface
{
public:
  virtual bool nb_put( const T &t ) = 0;
  virtual bool nb_can_put( tlm_tag<T> *t = 0 ) const = 0;
  virtual const sc_core::sc_event &ok_to_put( tlm_tag<T> *t = 0 ) const = 0;
};


// Combined uni-directional blocking and non blocking
template < typename T >
class tlm_get_if :
  public virtual tlm_blocking_get_if< T > ,
  public virtual tlm_nonblocking_get_if< T > {};


template < typename T >
class tlm_put_if :
  public virtual tlm_blocking_put_if< T > ,
  public virtual tlm_nonblocking_put_if< T > {};


// Peek interfaces
template < typename T >
class tlm_blocking_peek_if : public virtual sc_core::sc_interface
{
public:
  virtual T peek( tlm_tag<T> *t = 0 ) const = 0;
  virtual void peek( T &t ) const { t = peek(); }
};


template < typename T >
class tlm_nonblocking_peek_if : public virtual sc_core::sc_interface
{
public:
  virtual bool nb_peek( T &t ) const = 0;
  virtual bool nb_can_peek( tlm_tag<T> *t = 0 ) const = 0;
  virtual const sc_core::sc_event &ok_to_peek( tlm_tag<T> *t = 0 ) const = 0;
};


template < typename T >
class tlm_peek_if :
  public virtual tlm_blocking_peek_if< T > ,
  public virtual tlm_nonblocking_peek_if< T > {};
```

```
// Get_peek interfaces
template < typename T >
class tlm_blocking_get_peek_if :
  public virtual tlm_blocking_get_if<T> ,
  public virtual tlm_blocking_peek_if<T> {};

template < typename T >
class tlm_nonblocking_get_peek_if :
  public virtual tlm_nonblocking_get_if<T> ,
  public virtual tlm_nonblocking_peek_if<T> {};

template < typename T >
class tlm_get_peek_if :
  public virtual tlm_get_if<T> ,
  public virtual tlm_peek_if<T> ,
  public virtual tlm_blocking_get_peek_if<T> ,
  public virtual tlm_nonblocking_get_peek_if<T>
  {};

} // namespace tlm
```

10.2 TLM-1 fifo interfaces

```
namespace tlm {

// Fifo debug interface
template< typename T >
class tlm_fifo_debug_if : public virtual sc_core::sc_interface
{
public:
  virtual int used() const = 0;
  virtual int size() const = 0;
  virtual void debug() const = 0;

  // non blocking peek and poke - no notification. n is index of data :
  // 0 <= n < size(), where 0 is most recently written, and size() − 1 is oldest ie the one about to be read.
  virtual bool nb_peek( T & , int n ) const = 0;
  virtual bool nb_poke( const T & , int n = 0 ) = 0;
};

// Fifo interfaces
template < typename T >
class tlm_fifo_put_if :
  public virtual tlm_put_if<T> ,
```

```
    public virtual tlm_fifo_debug_if<T> {};

  template < typename T >
  class tlm_fifo_get_if :
   public virtual tlm_get_peek_if<T> ,
   public virtual tlm_fifo_debug_if<T> {};

  } // namespace tlm
```

10.3 tlm_fifo

```
  namespace tlm {

  template <typename T>
  class tlm_fifo :
   public virtual tlm_fifo_get_if<T>,
   public virtual tlm_fifo_put_if<T>,
   public sc_core::sc_prim_channel
  {
  public:
     explicit tlm_fifo( int size_ = 1 );
     explicit tlm_fifo( const char* name_, int size_ = 1 );
     virtual ~tlm_fifo();

     T get( tlm_tag<T> *t = 0 );
     bool nb_get( T& );
     bool nb_can_get( tlm_tag<T> *t = 0 ) const;
     const sc_core::sc_event &ok_to_get( tlm_tag<T> *t = 0 ) const;

     T peek( tlm_tag<T> *t = 0 ) const;
     bool nb_peek( T& ) const;
     bool nb_can_peek( tlm_tag<T> *t = 0 ) const;
     const sc_core::sc_event &ok_to_peek( tlm_tag<T> *t = 0 ) const;

     void put( const T& );
     bool nb_put( const T& );
     bool nb_can_put( tlm_tag<T> *t = 0 ) const;
     const sc_core::sc_event& ok_to_put( tlm_tag<T> *t = 0 ) const;

     void nb_expand( unsigned int n = 1 );
     void nb_unbound( unsigned int n = 16 );
     bool nb_reduce( unsigned int n = 1 );
     bool nb_bound( unsigned int n );
```

```
    bool nb_peek( T & , int n ) const;
    bool nb_poke( const T & , int n = 0 );

    int used() const;
    int size() const;
    void debug() const;

    static const char* const kind_string;
    const char* kind() const;
};
```

10.4 Analysis interface and analysis ports

Analysis ports are intended to support the distribution of transactions to multiple components for analysis, meaning tasks such as checking for functional correctness or collecting functional coverage statistics. The key feature of analysis ports is that a single port can be bound to multiple channels or *subscribers* such that the port itself replicates each call to the interface method **write** with each subscriber. An analysis port can be bound to zero or more subscribers or other analysis ports, and can be unbound.

Each subscriber implements the **write** method of the **tlm_analysis_if**. The method is passed a **const** reference to a transaction, which a subscriber may process immediately. Otherwise, if the subscriber wishes to extend the lifetime of the transaction, it is obliged to take a deep copy of the transaction object, at which point the subscriber effectively becomes the initiator of a new transaction and is thus responsible for the memory management of the copy.

Analysis ports should not be used in the main operational pathways of a model, but only where data is tapped off and passed to the side for analysis. Interface **tlm_analysis_if** is derived from **tlm_write_if**. The latter interface is not specific to analysis, and may be used for other purposes. For example, see 9.3 Payload event queue.

The **tlm_analysis_fifo** is simply an infinite **tlm_fifo** that implements the **tlm_analysis_if** to write a transaction to the fifo. The **tlm_fifo** also supports the **tlm_analysis_triple**, which consists of a transaction together with explicit start and end times.

10.4.1 Class definition

```
namespace tlm {

// Write interface
template <typename T>
class tlm_write_if : public virtual sc_core::sc_interface {
public:
   virtual void write( const T& ) = 0;
};

template <typename T>
class tlm_delayed_write_if : public virtual sc_core::sc_interface {
public:
   virtual void write( const T& , const sc_core::sc_time& ) = 0;
};

// Analysis interface
template < typename T >
class tlm_analysis_if : public virtual tlm_write_if<T>
{
};

template < typename T >
class tlm_delayed_analysis_if : public virtual tlm_delayed_write_if<T>
```

```
    {
    };

    // Analysis port
    template < typename T>
    class tlm_analysis_port : public sc_core::sc_object , public virtual tlm_analysis_if< T >
    {
    public:
       tlm_analysis_port();
       tlm_analysis_port( const char * );

       // bind and () work for both interfaces and analysis ports, since analysis ports implement the analysis
    interface
       void bind( tlm_analysis_if<T> &  );
       void operator() ( tlm_analysis_if<T> & );
       bool unbind( tlm_analysis_if<T> & );

       void write( const T & );
    };

    // Analysis triple
    template< typename T>
    struct tlm_analysis_triple {

       sc_core::sc_time start_time;
       T transaction;
       sc_core::sc_time end_time;

       // Constructors
       tlm_analysis_triple();
       tlm_analysis_triple( const tlm_analysis_triple &triple );
       tlm_analysis_triple( const T &t );

       operator T() { return transaction; }
       operator const T& () const { return transaction; }
    };

    // Analysis fifo - an unbounded tlm_fifo
    template< typename T >
    class tlm_analysis_fifo :
       public tlm_fifo< T > ,
       public virtual tlm_analysis_if< T > ,
       public virtual tlm_analysis_if< tlm_analysis_triple< T > > {

    public:
       tlm_analysis_fifo( const char *nm ) : tlm_fifo<T>( nm, -16 ) {}
```

```
    tlm_analysis_fifo() : tlm_fifo<T>( -16 ) {}

    void write( const tlm_analysis_triple<T> &t ) { nb_put( t ); }
    void write( const T &t ) { nb_put( t ); }
};

} // namespace tlm
```

10.4.2 Rules

a) **tlm_write_if** and **tlm_analysis_if** (and their delayed variants) are unidirectional, non-negotiated, non-blocking transaction-level interfaces, meaning that the callee has no choice but to immediately accept the transaction passed as an argument.

b) The constructor shall pass any character string argument to the constructor belonging to the base class **sc_object** to set the string name of the instance in the module hierarchy.

c) The **bind** method shall register the subscriber passed as an argument with the analysis port instance so that any call to the **write** method shall be passed on to the registered subscriber. Multiple subscribers may be registered with a single analysis port instance.

d) The **operator()** shall be equivalent to the **bind** method.

e) There may be zero subscribers registered with any given analysis port instance, in which case calls to the **write** method shall not be propagated.

f) The **unbind** method shall reverse the effect of the **bind** method, that is, the subscriber passed as an argument shall be removed from the list of subscribers to that analysis port instance.

g) The **write** method of class **tlm_analysis_port** shall call the **write** method of every subscriber registered with that analysis port instance, passing on the argument as a **const** reference.

h) The **write** method is non-blocking. It shall not call **wait**.

i) The **write** method shall not modify the transaction object passed as a **const** reference argument, nor shall it modify any data associated with the transaction object (such as the data and byte enable arrays of the generic payload).

j) If the implementation of the **write** method in a subscriber is unable to process the transaction before returning control to the caller, the subscriber shall be responsible for taking a deep copy of the transaction object and for managing any memory associated with that copy thereafter.

k) The constructors of class **tlm_analysis_fifo** shall each construct an unbounded **tlm_fifo**.

l) The **write** methods of class **tlm_analysis_fifo** shall call the **nb_put** method of the base class **tlm_fifo**, passing on their argument to **nb_put**.

Example

```
struct Trans   // Analysis transaction class
{
   int i;
};

struct Subscriber: sc_object, tlm::tlm_analysis_if<Trans>
{
   Subscriber(const char* n) : sc_object(n) {}

   virtual void write(const Trans& t)
   {
      cout << "Hello, got " << t.i << "\n";   // Implementation of the write method
   }
};

SC_MODULE(Child)
{
   tlm::tlm_analysis_port<Trans> ap;

   SC_CTOR(Child) : ap("ap")
   {
      SC_THREAD(thread);
   }
   void thread()
   {
      Trans t = {999};
      ap.write(t);                  // Interface method call to the write method of the analysis port
   }
};

SC_MODULE(Parent)
{
   tlm::tlm_analysis_port<Trans> ap;

   Child* child;

   SC_CTOR(Parent) : ap("ap")
   {
      child = new Child("child");
      child->ap.bind(ap);           // Bind analysis port of child to analysis port of parent
   }
};
```

```
SC_MODULE(Top)
{
  Parent* parent;
  Subscriber* subscriber1;
  Subscriber* subscriber2;

  SC_CTOR(Top)
  {
    parent      = new Parent("parent");
    subscriber1 = new Subscriber("subscriber1");
    subscriber2 = new Subscriber("subscriber2");

    parent->ap.bind( *subscriber1 );   // Bind analysis port to two separate subscribers
    parent->ap.bind( *subscriber2 );   // This is the key feature of analysis ports
  }
};
```

11 Glossary

This glossary contains brief, informal descriptions for a number of terms and phrases used in this standard. Where appropriate, the complete, formal definition of each term or phrase is given in the main body of the standard. Each glossary entry contains either the clause number of the definition in the main body of the standard or an indication that the term is defined in ISO/IEC 14882:2003 or IEEE Std 1666™-2005.

adapter: A module that connects a transaction level interface to a pin level interface (in the general sense of the word interface) or that connects together two transaction level interfaces, often at different abstraction levels. An adapter may be used to convert between two sockets specialized with different protocol types. See *bridge, transactor*.

approximately timed: A modeling style for which there exists a one-to-one mapping between the externally observable states of the model and the states of some corresponding detailed reference model such that the mapping preserves the sequence of state transitions but not their precise timing. The degree of timing accuracy is undefined. See *cycle approximate*.

attribute (of a transaction): Data that is part of and carried with the transaction and is implemented as a member of the transaction object. These may include attributes inherent in the bus or protocol being modeled, and attributes that are artefacts of the simulation model (a timestamp, for example).

automatic deletion: A generic payload extension marked for automatic deletion will be deleted at the end of the transaction lifetime, that is, when the transaction reference count reaches 0.

backward path: The calling path by which a target or interconnect component makes interface method calls back in the direction of another interconnect component or the initiator.

base protocol: A protocol traits class consisting of the generic payload and tlm_phase types, together with an associated set of protocol rules which together ensure maximal interoperability between transaction-level models

bidirectional interface: A TLM-1 transaction level interface in which a pair of transaction objects, the request and the response, are passed in opposite directions, each being passed according to the rules of the unidirectional interface. For each transaction object, the transaction attributes are strictly readonly in the period between the first timing point and the end of the transaction lifetime.

blocking: Permitted to call the **wait** method. A blocking function may consume simulation time or perform a context switch, and therefore shall not be called from a method process. A blocking interface defines only blocking functions.

blocking transport interface: A blocking interface of the TLM-2.0 standard which contains a single method **b_transport**. Beware that there still exists a blocking transport method named **transport**, part of TLM-1.

bridge: A component connecting two segments of a communication network together. A bus bridge is a device that connects two similar or dissimilar memory-mapped buses together. See *adapter, transaction bridge, transactor*.

caller: In a function call, the sequence of statements from which the given function is called. The referent of the term may be a function, a process, or a module. This term is used in preference to *initiator* to refer to the caller of a function as opposed to the initiator of a transaction.

callee: In a function call, the function that is called by the caller. This term is used in preference to *target* to refer to the function body as opposed to the target of a transaction.

channel: A class that implements one or more interfaces or an instance of such a class. A channel may be a hierarchical channel or a primitive channel or, if neither of these, it is strongly recommended that a channel at least be derived from class **sc_object**. Channels serve to encapsulate the definition of a communication mechanism or protocol. (SystemC term)

child: An instance that is within a given module. Module A is a *child* of module B if module A is *within* module B. (SystemC Term)

combined interfaces: Pre-defined groups of core interfaces used to parameterize the socket classes. There are four combined interfaces: the blocking and non-blocking forward and backward interfaces.

component: An instance of a SystemC module. This standard recognizes three kinds of component; the initiator, interconnect component, and target.

convenience socket: A socket class, derived from **tlm_initiator_socket** or **tlm_target_socket**, that implements some additional functionality and is provided for convenience. Several convenience sockets are provided as utilities.

core interface: One of the specific transaction level interfaces defined in this standard, including the blocking and non-blocking transport interface, the direct memory interface, and the debug transport interface. Each core interface is an *interface proper*. The core interfaces are distinct from the generic payload API.

cycle accurate: A modeling style in which it is possible to predict the state of the model in any given cycle at the external boundary of the model and thus to establish a one-to-one correspondence between the states of the model and the externally observable states of a corresponding RTL model in each cycle, but which is not required to explicitly re-evaluate the state of the entire model in every cycle or to explicitly represent the state of every boundary pin or internal register. This term is only applicable to models that have a notion of cycles.

cycle approximate: A model for which there exists a one-to-one mapping between the externally observable states of the model and the states of some corresponding cycle accurate model such that the mapping preserves the sequence of state transitions but not their precise timing. The degree of timing accuracy is undefined. This term is only applicable to models that have a notion of cycles.

cycle count accurate, cycle count accurate at transaction boundaries: A modeling style in which it is possible to establish a one-to-one correspondence between the states of the model and the externally observable states of a corresponding RTL model as sampled at the timing points marking the boundaries of a transaction. A cycle count accurate model is not required to be cycle accurate in every cycle, but is required to accurately predict both the functional state and the number of cycles at certain key timing points as defined by the boundaries of the transactions through which the model communicates with other models.

declaration: A C++ language construct that introduces a name into a C++ program and specifies how the C++ compiler is to interpret that name. Not all declarations are definitions. For example, a class declaration specifies the name of the class but not the class members, while a function declaration specifies the function parameters but not the function body. (See definition.) (C++ term)

definition: The complete specification of a variable, function, type, or template. For example, a class definition specifies the class name and the class members, and a function definition specifies the function parameters and the function body. (See declaration.) (C++ term)

effective local time: The current time within a temporally decoupled initiator. effective_local_time = sc_time_stamp() + local_time_offset

exclusion rule: A rule of the base protocol that prevents a request or a response being sent through a socket if there is already a request or a response (respectively) in progress through that socket. The base protocol has two exclusion rules, the request exclusion rule and the response exclusion rule, which act independently of one another.

extension: A user-defined object added to and carried around with a generic payload transaction object, or a user-defined class that extends the set of values that are assignment compatible with the tlm_phase type. An ignorable extension may be used with the base protocol, but a non-ignorable or mandatory extension requires the definition of a new protocol traits class.

forward path: The calling path by which an initiator or interconnect component makes interface method calls forward in the direction of another interconnect component or the target.

generic payload: A specific set of transaction attributes and their semantics together defining a transaction payload which may be used to achieve a degree of interoperability between loosely timed and approximately timed models for components communicating over a memory-mapped bus. The same transaction class is used for all modeling styles.

global quantum: The default time quantum used by every quantum keeper and temporally decoupled initiator. The intent is that all temporally decoupled initiators should typically synchronize on integer multiples of the global quantum, or more frequently on demand.

hierarchical binding: Binding a socket on a child module to a socket on a parent module, or a socket on a parent module to a socket on a child module, passing transactions up or down the module hierarchy.

hop: The interface method call path between two adjacent components en route from initiator to target. A hop consists of one initiator socket bound to one target socket. In order to be transported from initiator to target, a transaction may need to pass over multiple hops. The number of hops between an initiator and a target is always one greater than the number of interconnect components.

ignorable extension: A generic payload extension that may be ignored by any component other than the component that set the extension. An ignorable extension is not required to be present. Ignorable extensions are permitted by the base protocol.

ignorable phase: A phase, created by the macro DECLARE_EXTENDED PHASE, that may be ignored by any component that receives the phase and that cannot demand a response of any kind. Ignorable phases are permitted by the base protocol.

initiator: A module that can initiate transactions. The initiator is responsible for initializing the state of the transaction object, and for deleting or reusing the transaction object at the end of the transaction's lifetime. An initiator is usually a master and a master is usually an initiator, but the term *initiator* means that a component can initiate transactions, whereas the term *master* means that a component can take control of a bus. In the case of the TLM-1 interfaces, the term *initiator* as defined here may not be strictly applicable, so the terms *caller* and *callee* may be used instead for clarity.

initiator socket: A class containing a port for interface method calls on the forward path and an export for interface method calls on the backward path. A socket overloads the SystemC binding operators to bind both the port and the export.

interconnect component: A module that accesses a transaction object, but does not act as an initiator or a target with respect to that transaction. An interconnect component may or may not be permitted to modify the attributes of the transaction object, depending on the rules of the payload. An arbiter or a router would typically be modeled as an interconnect component, the alternative being to model it as a target for one transaction and an initiator for a separate transaction.

interface: A class derived from class **sc_interface**. An interface proper is an interface, and in the object-oriented sense a channel is also an interface. However, a channel is not an interface proper. (SystemC term)

Interface Method Call (IMC): A call to an interface method. An interface method is a member function declared within an interface. The IMC paradigm provides a level of indirection between a method call and the implementation of the method within a channel such that one channel can be substituted with another without affecting the caller. (SystemC term)

interface proper: An abstract class derived from class **sc_interface** but not derived from class **sc_object**. An interface proper declares the set of methods to be implemented within a channel and to be called through a port. An interface proper contains pure virtual function declarations, but typically contains no function definitions and no data members. (SystemC term)

interoperability: The ability of two or more transaction level models from diverse sources to exchange information using the interfaces defined in this standard. The intent is that models that implement common memory-mapped bus protocols in the programmers view use case should be interoperable without the need for explicit adapters. Furthermore, the intent is to reduce the amount of engineering effort needed to achieve interoperability for models of divergent protocols or use cases, although it is expected that adapters will be required in general.

interoperability layer: The subset of classes in this standard that are necessary for interoperability. The interoperability layer comprises the TLM-2.0 core interfaces, the initiator and target sockets, the generic payload, **tlm_global_quantum** and **tlm_phase**. Closely related to the base protocol.

lifetime (of an object): The lifetime of an object starts when storage is allocated and the constructor call has completed, if any. The lifetime of an object ends when storage is released or immediately before the destructor is called, if any. (C++ term)

lifetime (of a transaction): The period of time that starts when the transaction becomes valid and ends when the transaction becomes invalid. Because it is possible to pool or re-use transaction objects, the lifetime of a transaction object may be longer than the lifetime of the corresponding transaction. For example, a transaction object could be a stack variable passed as an argument to multiple *put* calls of the TLM-1 interface.

local quantum: The amount of simulation time remaining before the initiator is required to synchronize. Typically, the local quantum equals the current simulation time subtracted from the next largest integer multiple of the global quantum, but this calculation can be overridden for a given quantum keeper.

local time offset: Time as measured relative to the most recent quantum boundary in a temporally decoupled initiator. The timing annotation arguments to the **b_transport** and *nb_transport* methods are local time offsets. effective_local_time = sc_time_stamp() + local_time_offset

loosely timed: A modeling style that represents minimal timing information sufficient only to support features necessary to boot an operating system and to manage multiple threads in the absence of explicit synchronization between those threads. A loosely timed model may include timer models and a notional arbitration interval or execution slot length. Some users adopt the practice of inserting random delays into loosely timed descriptions in order to test the robustness of their protocols, but this practice does not change the basic characteristics of the modeling style.

master: This term has no precise technical definition in this standard, but is used to mean a module or port that can take control of a memory-mapped bus in order to initiate bus traffic, or a component that can execute an autonomous software thread and thus initiate other system activity. Generally, a bus master would be an initiator.

memory manager: A user-defined class that performs memory management for a generic payload transaction object. A memory manager must provide a **free** method, called when the reference count of the transaction reaches 0.

method: A function that implements the behavior of a class. This term is synonymous with the C++ term *member function*. In SystemC, the term *method* is used in the context of an *interface method call*. Throughout this standard, the term *member function* is used when defining C++ classes (for conformance to the C++ standard), and the term *method* is used in more informal contexts and when discussing interface method calls. (SystemC term)

multi-socket: One of a family of convenience sockets that can be bound to multiple sockets belonging to other components. An initiator multi-socket can be bound to more than one target socket, and more than one initiator socket can be bound to a single target multi-socket. When calling interface methods through multi-sockets, the destinations are distinguished using the subscript operator.

nb_transport: The **nb_transport_fw** and **nb_transport_bw** methods. In this document, the italicized term *nb_transport* is used to describe both methods in situations where there is no need to distinguish between them.

non-blocking: Not permitted to call the **wait** method. A non-blocking function is guaranteed to return without consuming simulation time or performing a context switch, and therefore may be called from a thread process or from a method process. A non-blocking interface defines only non-blocking functions.

non-blocking transport interface: A non-blocking interface of the TLM-2.0 standard. There a two such interfaces, containing methods named **nb_transport_fw** and **nb_transport_bw**.

object: A region of storage. Every object has a type and a lifetime. An object created by a definition has a name, whereas an object created by a new expression is anonymous. (C++ term)

opposite path: The path in the opposite direction to a given path. For the forward path, the opposite path is the forward return path or the backward path. For the backward path, the opposite path is the forward path or the backward return path.

parent: The inverse relationship to *child*. Module A is the *parent* of module B if module B is a *child* of module A. (SystemC term)

payload event queue (PEQ): A class that maintains a queue of SystemC event notifications, where each notification carries an associated transaction object. Transactions are put into the queue annotated with a delay, and each transaction pops out of the back of queue at the time it was put in plus the given delay. Useful when combining the non-blocking interface with the approximately-timed coding style.

phase: A period in the lifetime of a transaction. The phase is passed as an argument to the non-blocking transport method. Each phase transition is associated with a timing point. The timing point may be delayed by an amount given by the time argument to *nb_transport*.

phase transition: A change to the value of the phase argument of the non-blocking transport method as marked by each call to *nb_transport* and each return from *nb_transport* with a value of TLM_UPDATED.

programmers view (PV): The use case of the software programmer who requires a functionally accurate, loosely timed model of the hardware platform for booting an operating system and running application software.

protocol traits class: A class containing a **typedef** for the type of the transaction object and the phase type, which is used to parameterize the combined interfaces, and effectively defines a unique type for a protocol.

quantum: In temporal decoupling, the amount a process is permitted to run ahead of the current simulation time.

quantum keeper: A utility class used to store the local time offset from the current simulation time, which it checks against a local quantum.

request: For the base protocol, the stage during the lifetime of a transaction when information is passed from the initiator to the target. In effect, the request transports generic payload attributes from the initiator to the target, including the command, the address, and for a write command, the data array. (The transaction is actually passed by reference and the data array by pointer.)

response: For the base protocol, the stage during the lifetime of a transaction when information is passed from the target back to the initiator. In effect, the response transports generic payload attributes from the target back to the initiator, including the response status, and for a read command, the data array. (The transaction is actually passed by reference and the data array by pointer.)

return path: The control path by which the call stack of a set of interface method calls is unwound along either the forward path or the backward path. The return path for the forward path can carry information from target to initiator, and the return path for the backward path can carry information from initiator to target.

simple socket: One of a family of convenience sockets that are simple to use because they allows callback methods to be registered directly with the socket object rather than the socket having to be bound to another object that implements the required interfaces. The simple target socket avoids the need for a target to implement both blocking and non-blocking transport interfaces by providing automatic conversion between the two.

slave: This term has no precise technical definition in this standard, but is used to mean a reactive module or port on a memory-mapped bus that is able to respond to commands from bus masters, but is not able itself to initiate bus traffic. Generally, a slave would be modeled as a target.

socket: See initiator socket and target socket

standard error response: The behavior prescribed by this standard for a generic payload target that is unable to execute a transaction successfully. A target should either a) execute the transaction successfully or b) set the response status attribute to an error response or c) call the SystemC report handler.

sticky extension: A generic payload extension object that is not deleted (either automatically or explicitly) at the end of life of the transaction object, and thus remains with the transaction object when it is pooled. Sticky extensions are not deleted by the memory manager.

synchronize: To yield such that other processes may run, or when using temporal decoupling, to yield and wait until the end of the current time quantum.

synchronization-on-demand: The action of a temporally decoupled process when it yields control back to the SystemC scheduler so that simulation time may advance and other processes run in addition to the synchronization points that may occur routinely at the end of each quantum.

tagged socket: One of a family of convenience sockets that add an int id tag to every incoming interface method call in order to identify the socket (or element of a multi-socket) through which the transaction arrived.

target: A module that represents the final destination of a transaction, able to respond to transactions generated by an initiator, but not itself able to initiate new transactions. For a write operation, data is copied from the initiator to one or more targets. For a read operation, data is copied from one target to the initiator. A target may read or modify the state of the transaction object. In the case of the TLM-1 interfaces, the term *target* as defined here may not be strictly applicable, so the terms *caller* and *callee* may be used instead for clarity.

target socket: A class containing a port for interface method calls on the backward path and an export for interface method calls on the forward path. A socket also overloads the SystemC binding operators to bind both port and export.

temporal decoupling: The ability to allow one or more initiators to run ahead of the current simulation time in order to reduce context switching and thus increase simulation speed.

timing annotation: The **sc_time** argument to the **b_transport** and *nb_transport* methods. A timing annotation is a local time offset. The recipient of a transaction is required to behave as if it had received the transaction at effective_local_time = sc_time_stamp() + local_time_offset.

timing point: A significant time within the lifetime of a transaction. A loosely-timed transaction has two timing points corresponding to the call to and return from **b_transport**. An approximately-timed base protocol transaction has four timing points, each corresponding to a phase transition.

TLM-1: The first major version of the OSCI Transaction Level Modeling standard. TLM-1 was released in 2005.

TLM-2.0: The second major version of the OSCI Transaction Level Modeling standard. This document describes TLM-2.0.

traits class: In C++ programming, a class that contains definitions such as typedefs that are used to specialize the behavior of a primary class, typically by having the traits class passed as a template argument to the primary class. The default template parameter provides the default traits for the primary class.

transaction: An abstraction for an interaction or communication between two or more concurrent processes. A transaction carries a set of attributes and is bounded in time, meaning that the attributes are only valid within a specific time window. The timing associated with the transaction is limited to a specific set of timing points, depending on the type of the transaction. Processes may be permitted to read or modify attributes of the transaction, depending on the protocol.

transaction bridge: A component that acts as the target for an incoming transaction and as the initiator for an outgoing transaction, usually for the purpose of modeling a bus bridge. See *bridge*

transaction instance: A unique instance of a transaction. A transaction instance is represented by one transaction object, but the same transaction object may be re-used for several transaction instances.

transaction object: The object that stores the attributes associated with a transaction. The type of the transaction object is passed as a template argument to the core interfaces.

transaction level (TL): The abstraction level at which communication between concurrent processes is abstracted away from pin wiggling to transactions. This term does not imply any particular level of granularity with respect to the abstraction of time, structure, or behavior.

transaction level model, transaction level modeling (TLM): A model at the transaction level and the act of creating such a model, respectively. Transaction level models typically communicate using function calls, as opposed to the style of setting events on individual pins or nets as used by RTL models.

transactor: A module that connects a transaction level interface to a pin level interface (in the general sense of the word interface) or that connects together two or more transaction level interfaces, often at different abstraction levels. In the typical case, the first transaction level interface represents a memory-mapped bus or other protocol, the second interface represents the implementation of that protocol at a lower abstraction level. However, a single transactor may have multiple transaction level or pin level interfaces. See *adapter, bridge*.

transparent component: A interconnect component with the property that all incoming interface method calls are propagated immediately through the component without delay and without modification to the arguments or to the transaction object (extensions excepted). The intent of a transparent component is to allow checkers and monitors to pass ignorable phases.

transport interface: The one and only bidirectional core interface in TLM-1. The transport interface passes a request transaction object from caller to callee, and returns a response transaction object from callee to caller. TLM-2.0 adds separate blocking and non-blocking transport interfaces.

unidirectional interface: A TLM-1.0 transaction level interface in which the attributes of the transaction object are strictly readonly in the period between the first timing point and the end of the transaction lifetime. Effectively, the information represented by the transaction object is strictly passed in one direction either from caller to callee or from callee to caller. In the case of **void put(const T& t)**, the first timing point is marked by the function call. In the case of **void get(T& t)**, the first timing point is marked by the return from the function. In the case of **T get()**, strictly speaking there are two separate transaction objects, and the return from the function marks the degenerate end-of-life of the first object and the first timing point of the second.

untimed: A modeling style in which there is no explicit mention of time or cycles, but which includes concurrency and sequencing of operations. In the absence of any explicit notion of time as such, the sequencing of operations across multiple concurrent threads must be accomplished using synchronization primitives such as events, mutexes and blocking FIFOs. Some users adopt the practice of inserting random delays into untimed descriptions in order to test the robustness of their protocols, but this practice does not change the basic characteristics of the modeling style.

utilities: A set of classes of the TLM-2.0 standard that are provided for convenience only, and are not strictly necessary to achieve interoperability between transaction-level models.

valid: The state of an object returned from a function by pointer or by reference, during any period in which the object is not deleted and its value or behavior remains accessible to the application. (SystemC term)

within: The relationship that exists between an instance and a module if the constructor of the instance is called from the constructor of the module, and also provided that the instance is not *within* a nested module. (SystemC term)

yield: Return control to the SystemC scheduler. For a thread process, to yield is to call **wait**. For a method process, to yield is to return from the function.

12 Index