November 2007

OSCI TLM2 USER MANUAL

Software version: TLM 2.0 Draft 2

Document version: 1.0.0

Contributors

The TLM2.0-draft-2 standard was created under the leadership of the following individuals:

Trevor Wieman, Intel, TLM-WG Chair
Stuart Swan, Cadence, TLM-WG Vice-Chair

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 the release of TLM2.0-draft-2:

Tom Aernoudt, CoWare
James Aldis, OCP-IP
John Aynsley, Doulos
Guillaume Audeon, ARM
Bill Bunton, ESLX
Mark Burton, GreenSocs
Jack Donovan, ESLX
Othman Fathy, Mentor Graphics
Alan Fitch, Doulos
Karthick Gururaj, NXP
Atsushi Kasuya, Jeda
Tim Kogel, CoWare
Laurent Maillet-Contoz, ST Microelectronics
Marcelo Montoreano, Synopsys
Rishiyur Nikhil, Bluespec
Victor Reyes, NXP
Adam Rose, Mentor Graphics
Olaf Scheufen, Synopsys
Alan Su, Springsoft
Stuart Swan, Cadence
Bart Vanthournout, CoWare
Yossi Veller, Mentor Graphics
Trevor Wieman, Intel
Charles Wilson, ESLX

The following people have also contributed to 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
Adam Erickson, Cadence
Frank Ghenassia, ST Microelectronics
Mark Glasser, Mentor Graphics
Andrew Goodrich, Forte Design
Serge Goosens, CoWare
Thorsten Groetker, Synopsys
Robert Guenzel, GreenSocs
Kamal Hashmi, SpiraTech
Holger Keding, Synopsys
Devon Kehoe, Mentor Graphics

Wolfgang Klingauf, GreenSocs
David Long, Doulos
Kiyoshi Makino, Mentor
Mike Meredith, Forte Design
David Pena, Cadence
Nizar Romdhane, ARM
Stefan Schmermbeck, Chipvision
Shiri Shem-Tov, Freescale
Jean-Philippe Strassen, ST Microelectronics
Tsutomu Takei, STARC
Jos Verhaegh, NXP
Maurizio Vitale, Philips Semiconductors
Vincent Viteau, Summit Design
Thomas Wilde, Infineon
Hiroyuki Yagi, STARC
Eugene Zhang, Jeda

Contents

1 OVERVIEW 1

1.1 Scope 2

1.2 Source code and documentation 2

2 REFERENCES 3

2.1 Bibliography 3

3 INTRODUCTION 4

3.1 Background 4

3.2 Transaction-level modeling, use cases and abstraction 4

3.3 Coding styles 5
3.3.1 Untimed coding style 5
3.3.2 Loosely-timed coding style and temporal decoupling 6
3.3.3 Untimed versus loosely-timed modeling 7
3.3.4 Approximately-timed coding style 8
3.3.5 Characterization of untimed, loosely-timed and approximately-timed coding styles 8
3.3.6 Switching between loosely-timed and approximately-timed modeling 8
3.3.7 Cycle-accurate modeling 9
3.3.8 Blocking versus non-blocking transport interfaces 9
3.3.9 Use cases and coding styles 10

3.4 Initiators, targets, sockets, and bridges 10

3.5 DMI and debug transaction interfaces 12

3.6 Namespaces 12

3.7 Header files 12

4 CORE TLM2 INTERFACES 13

4.1 Blocking transport interface 13
4.1.1 Introduction 13
4.1.2 Migration path from TLM1 13
4.1.3 Class definition 13
4.1.4 The TRANS template argument 14

4.1.5 Rules ... 14

4.2 Non-blocking transport interface ... 14
4.2.1 Introduction ... 14
4.2.2 Class definition ... 15
4.2.3 The TRANS template argument ... 15
4.2.4 The PHASE template argument ... 15
4.2.5 The nb_transport call ... 16
4.2.6 The trans argument ... 16
4.2.7 The phase argument ... 17
4.2.8 The sc_time argument ... 17
4.2.9 The tlm_sync_enum return value ... 19
4.2.10 Coding styles and tlm_sync_enum ... 20
4.2.11 Coding styles and tlm_phase ... 20
4.2.12 Phase sequences for loosely-timed and approximately-timed coding styles ... 21
4.2.13 Message sequence charts for nb_transport ... 22
4.2.13.1 Loosely-timed with timing annotation ... 22
4.2.13.2 Loosely-timed with sync ... 23
4.2.13.3 Loosely-timed with temporal decoupling ... 24
4.2.13.4 Loosely-timed with temporal decoupling and synchronization-on-demand ... 25
4.2.13.5 Loosely-timed with temporal decoupling and quantum ... 26
4.2.13.6 Approximately-timed timing parameters ... 27
4.2.13.7 Approximately-timed using backward path ... 28
4.2.13.8 Approximately-timed with timing annotation ... 29
4.2.13.9 Approximately-timed with polling ... 30
4.2.13.10 Loosely- to approximately-timed adapter ... 31
4.2.13.11 Approximately-timed initiator to loosely-timed target ... 32
4.2.14 Transaction lifetime example ... 32

4.3 Direct memory interface ... 33
4.3.1 Introduction ... 33
4.3.2 Class definition ... 33
4.3.3 get_direct_mem_ptr method ... 34
4.3.4 DMI_MODE template argument and tlm_dmi_mode class ... 35
4.3.5 tlm_dmi class ... 36
4.3.6 invalidate_direct_mem_ptr method ... 37
4.3.7 Optimization using a DMI Hint ... 38

4.4 Debug transaction interface ... 38
4.4.1 Introduction ... 38
4.4.2 Class definition ... 38
4.4.3 Rules ... 39

5 SOCKETS AND COMBINED INTERFACES ... 41
5.1.1 Introduction ... 41
5.1.2 Class definition ... 42

5.1.3 Rules ... 45

6 ANALYSIS INTERFACE AND ANALYSIS PORTS ... 49

6.1 Class definition ... 49

6.2 Rules ... 51

7 QUANTUM KEEPER ... 54

7.1 Introduction ... 54

7.2 Class definition ... 54

7.3 Rules for processes using temporal decoupling ... 55

8 PAYLOAD EVENT QUEUE ... 58

8.1 Introduction ... 58

8.2 Class definition ... 58

9 GENERIC PAYLOAD ... 61

9.1 Introduction ... 61

9.2 Extensions and interoperability ... 61
9.2.1 Use the generic payload directly, with ignorable extensions ... 62
9.2.2 Define a new protocol types class containing a typedef for tlm_generic_payload ... 63
9.2.3 Define a new protocol types class and a new transaction type ... 64

9.3 Generic payload attributes and methods ... 64

9.4 Class definition ... 64

9.5 Generic payload memory management ... 66

9.6 Constructors, assignment, and destructor ... 67

9.7 Default values and modifiability of attributes ... 68

9.8 Command attribute ... 68

9.9 Address attribute ... 69

9.10 Data pointer attribute ... 69

9.11 Data length attribute 70

9.12 Byte enable pointer attribute 71

9.13 Byte enable length attribute 72

9.14 Streaming width attribute 72

9.15 Endianness 73

9.16 Atomic Transactions 75

9.17 DMI allowed attribute 75

9.18 Response status attribute 75
9.18.1 The standard error response 77

9.19 Extension mechanism 81
9.19.1 Introduction 81
9.19.2 Rationale 81
9.19.3 Management of extensions, rules for safe use, and bridges 81
9.19.4 Rules 82

10 TLM1 LEGACY 87

10.1 TLM1 core interfaces 87

10.2 TLM1 fifo interfaces 89

10.3 tlm_fifo 90

11 GLOSSARY 93

1 Overview

This document is the User Manual for the OSCI Transaction Level Modeling standard, version 2.0-draft-2. This version of the standard supersedes version 2.0-draft-1, and is not generally compatible with 2.0-draft-1. This version of the standard includes the core interfaces from TLM 1.0.

TLM2 consists of a set of core interfaces, analysis ports, initiator and target sockets, and the generic payload. The core interfaces include the core interfaces from TLM1 together with the blocking and non-blocking transport interfaces, the direct memory interface (DMI), the debug transaction interface, the write interface, and the analysis interface. The core interfaces support untimed, loosely-timed and approximately-timed coding styles. 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 quantum keeper and payload event queue (PEQ) are utility classes for use with the non-blocking transport interface.

The TLM2 classes are layered on top of the SystemC class library as shown in the diagram below. The core interfaces can be used alone without the higher layers. The core interfaces and the initiator and target sockets can be used together without the generic payload. It is technically possible to use the generic payload directly with the core interfaces without using the initiator and target sockets, although this approach is not recommended. For maximum interoperability, it is recommended that the core TLM2 interfaces, sockets and generic payload be used together in concert.

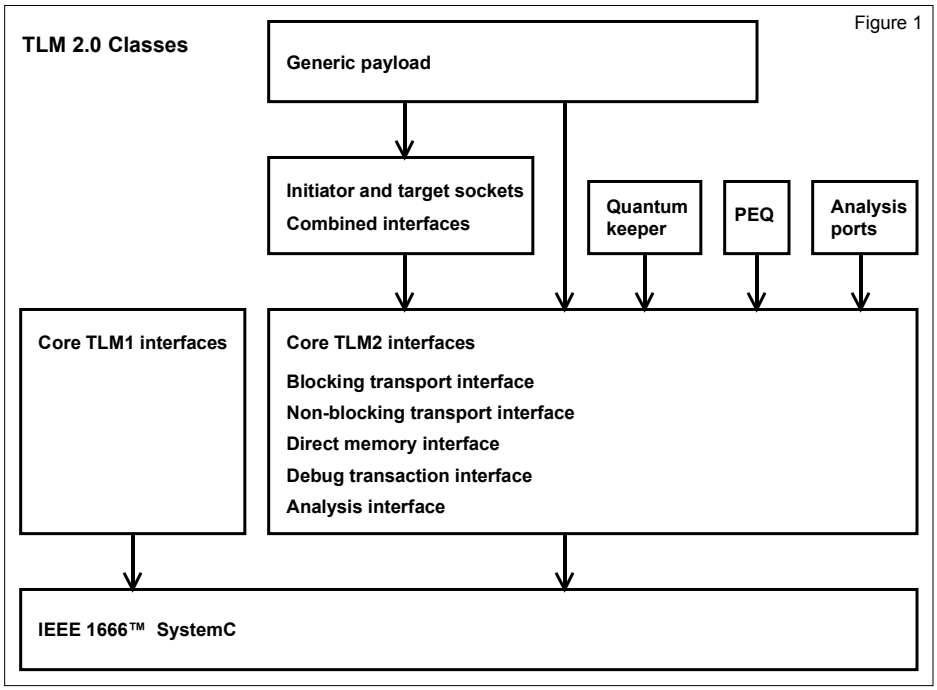


Figure 1

1.1 Scope

This document describes the contents of the TLM2.0 standard. The main focus of this document is the key concepts and semantics of the TLM2 core interfaces and classes. It does not describe all the supporting code, examples, and unit test. It lists the core TLM1 interfaces, but does not define their semantics. This document is not a definitive language reference manual. It is the intention that this document will be extended over time to add more practical guidelines on how to use TLM2.0

1.2 Source code and documentation

The TLM2.0 draft 2 release has a hierarchical directory structure as follows:

docs Documentation, including User Manual, white papers, and Doxygen
examples A set of application-oriented examples with their own documentation
include The C++ source code, including readme files and release notes
unit_tests 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 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 TLM1 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 busses and other on-chip communication networks:

a) TLM1 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. TLM2 addresses this shortcoming with the generic payload.

b) TLM1 has no support for timing annotation, so no standard way of communicating timing information between models. TLM1 models would typically implement delays by calling **wait**, which slows down simulation. TLM2 addresses this shortcoming with the non-blocking transport interface.

c) The TLM1 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. TLM2 addresses this shortcoming with transaction objects whose lifetime extends across several transport calls, supported by a new transport interface.

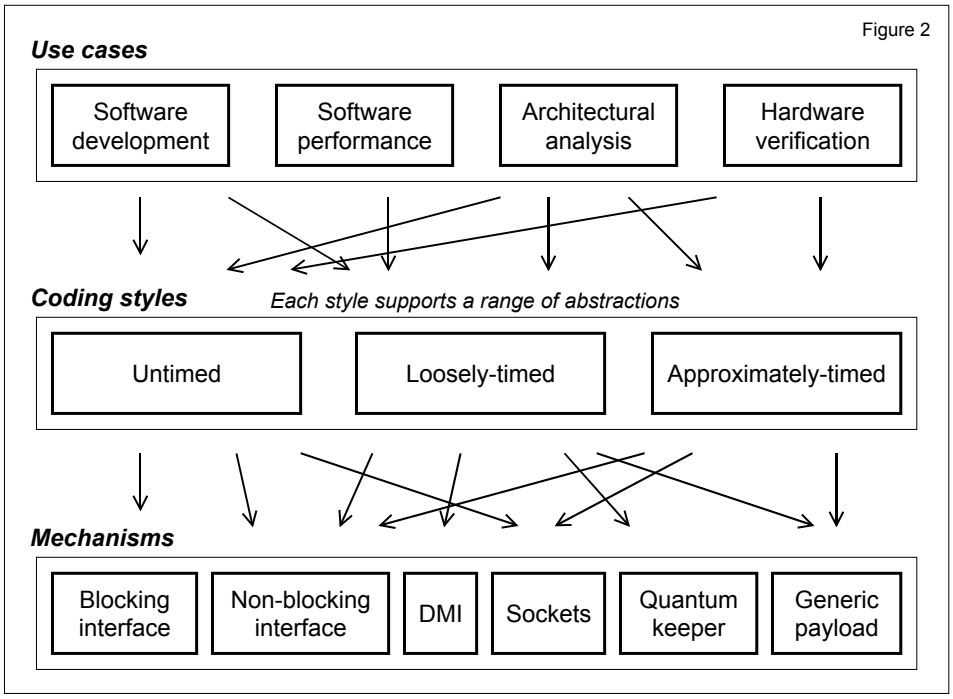
3.2 Transaction-level modeling, use cases and abstraction

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 TLM2 activity explicitly recognizes the existence of a variety of use cases for transaction-level modeling (see the Requirements Specification for TLM2.0), but rather than defining an abstraction level around each use case, TLM2 takes the approach of distinguishing between interfaces (APIs) on the one hand, and coding styles on the other. The TLM2 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. Each coding style can support a range of abstraction across functionality, timing and communication.

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 preempted by any other process. Process scheduling can be modeled in one of two ways, either by having the user introduce explicit communication

and synchronization points into their application, or by having a coding style or underlying modeling framework that forces each process to yield control at certain points. The former style is typified by the CSP (Communicating Sequential Processes) and KPN (Kahn Process Network) formalisms, and is easily implemented in SystemC using FIFOs, semaphores, or other synchronization primitives. This allows a completely untimed modeling style where in principle simulation can run without advancing simulation time, and all synchronization points are determined in advance and explicitly coded. The latter style is suitable when modeling multiple software threads running in an environment where the threads are not running in lockstep, but communicate from time-to-time, possibly at points in time that cannot be determined completely in advance. In this standard, such a coding style is called *loosely-timed*.



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. TLM2 recognizes several coding styles, which should be used as a guide to model writing. For simplicity and clarity, this document restricts itself to three specific named coding styles; *untimed*, *loosely-timed* and *approximately-timed*. In principle, it would be possible to define other coding styles based on the TLM2 mechanisms.

3.3.1 Untimed coding style

In TLM2, the untimed coding style makes use of the blocking transport interface. This interface does not make any provision for annotating timing information, but since the **transport** method is blocking, it could in

principle call **wait** to model a latency or to introduce a random delay into the process scheduling. The application is expected to implement synchronization between processes by calling explicit synchronization statements. For example, a blocking transport call could **wait** on an event notified by a separate process, or could return a flag to instruct the caller to **wait**, giving other processes a chance to execute.

3.3.2 Loosely-timed coding style and temporal decoupling

The loosely-timed coding style makes use of the non-blocking transport interface. This interface provides for timing annotation and for multiple phases and timing points during the lifetime of a transaction. In the case of the generic payload, a loosely-timed transactions has exactly two timing points marking the beginning of the request and the beginning of the response (or, equivalently, the end of the transaction). In principle these two timing points could occur at the same simulation time, but more usually they would be skewed.

The loosely-timed coding style is appropriate for the use case of software development in an MPSoC environment, where multiple software threads are running on a virtual model of a hardware platform, possibly under the control of an operating system. The loosely-timed coding styles supports the modeling of timers and course-grained process scheduling, sufficient to model the booting and running of an operating system. It also supports temporal decoupling, where each software thread is permitted to run ahead in a local “time warp” until it reaches the point when it needs to synchronize with the rest of the system. Temporal decoupling can result in very fast simulation for certain systems.

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 would be to allow processes to run ahead of the current simulation time, or to *warp* time.

In general, a process can be allowed to run ahead of simulation time until it encounters a dependency on a variable updated by another process, or needs to interact with another process. At that point, there is a decision to be made. In terms of general simulation techniques, outside the context of SystemC, there are several options. The process can be allowed to continue running, but the scheduler may need to backtrack later if it is discovered that the process assumed the wrong value. The scheduler may launch several alternative runs of the same process in parallel, assuming a different value for the external variable in each run, then pick the correct run after-the-fact when the value becomes known. This may make sense with dedicated hardware support. A third option is to return control to the simulation kernel when an external dependency is encountered, only resuming the process later in simulation time when the external value becomes known.

In TLM2, running ahead of simulation time is called *temporal decoupling*. Temporal decoupling does *not* permit backtracking, and each individual 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 accept 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 a very significant assumption concerning communication and timing in the modeled system. It assumes that no damage will be done by sampling the value too early, and that any subsequent change to the value will be picked up in a subsequent process execution. This assumption is valid either if there is some explicit handshaking associated with the value, or if the value only changes relatively infrequently.

Temporal decoupling, and in particular allowing the time of process interactions to slip in time, 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 is avoided by the TLM2 quantum keeper, which imposes an upper limit on the time a process is allowed to run ahead, known as the *quantum*. 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 introduces large timing inaccuracies, possibly to the point where important events are missed and the model 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 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. Depending on the detail of the models, this could give a simulation speed improvement of up to 1000X.

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 it likely to become a simulation speed bottleneck.

In TLM2, temporal decoupling is supported by the **tlm_quantum_keeper** class and the timing annotation of the non-blocking transport interface.

3.3.3 Untimed versus loosely-timed modeling

Explicit temporal decoupling is not required in an untimed model, because an untimed model does not rely on the notion of simulation time for process synchronization and scheduling. An untimed initiator can implicitly run ahead of simulation time until it reaches the next explicit synchronization point. An untimed model relies on the presence of explicit synchronization points in the system model in order to pass control between initiators at predetermined points during the 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 complete absence of explicit synchronization points, 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. Synchronization-on-demand in a loosely-timed model is equivalent to explicit synchronization in an untimed model.

The non-blocking transport interface can support both the untimed and the loosely-timed coding styles, as well as the approximately-timed coding style. However, the blocking transport interface is simpler to use if neither timing annotation nor temporal decoupling are required.

3.3.4 Approximately-timed coding style

The non-blocking transport interface also supports the approximately-timed coding style, which is appropriate for the use cases of architectural analysis and performance analysis. For approximately-timed modeling, a transaction is broken down into multiple phases, with an explicit timing point marking the transition between phases. Because the loosely-timed and approximately-timed styles both use the same non-blocking transport interface, a degree of interoperability between them is possible.

In the case of the generic payload 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.

The approximately-timed coding style does not use temporal decoupling, but rather each process 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 two kinds of delay: the latency of the target, and the initiation interval or accept delay of the target. The annotated delays are implemented by making calls to the SystemC scheduler, that is, **wait**(delay) or **notify**(delay).

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

The three coding styles can be characterized in terms of how running processes use simulation time and when they yield to the SystemC scheduler.

Untimed. The notion of simulation time is not required, and each process runs up to the next explicit synchronization point before yielding.

Loosely-timed. Simulation time is used, but processes are temporally decoupled from simulation time. Each process keeps a tally of the time it consumes, and may yield because it reaches an explicit synchronization point or because it has consumed its time quantum.

Approximately-timed. Processes run in lock-step with SystemC simulation time. Delays annotated onto process interactions are implemented using timeouts (wait) or timed event notifications.

3.3.6 Switching between loosely-timed and approximately-timed modeling[1]

A model may switch from the loosely-timed to the 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.

[1] The details of LT/AT switching are to be added before the final release of TLM2.0

3.3.7 Cycle-accurate modeling

Cycle-accurate modeling is beyond the scope of TLM2 at present. It is possible to create cycle-accurate models using SystemC and TLM1 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, the approximately-timed coding style can be extended to encompass cycle-accurate modeling by defining an appropriate set of phases together with rules concerning which attributes of the transaction can be modified and read in which phase, and by whom. In the limit, each phase represents a single cycle. The TLM2.0 Draft 2 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 separate interfaces that exist in TLM2 to support distinct coding styles. Applications should choose an interface depending on the coding style used (untimed, loosely-timed or approximately-timed), which in turn will depend on the use case. It is possible to adapt between the two interfaces or even to combine both interfaces in a single socket. However, mixing interface types and mixing coding styles does incur some cost, both in terms of simulation speed and coding complexity, so should be avoided where possible.

The blocking transport interface is particularly appropriate for models that include explicit, strong synchronization between processes such that the sequence of synchronization points is deterministic and independent of implementation detail. It is also appropriate where an untimed transaction-level model is to be transformed directly to an RTL implementation without any intermediate timed SystemC model.

The non-blocking transport interface is particulary appropriate for models that include timing information, or that are to be refined from a loosely-timed to an approximately-timed model, or where the exact sequence of synchronization points between concurrent processes is indeterminate or dependent upon hardware implementation detail or an operating system scheduler.

The strength of the blocking transport interface is that it allows a simplified coding style for models that do not require specific timing detail. The strength of the non-blocking transport interface is that it supports both temporal decoupling in a loosely-timed coding style and timing annotation for approximately-timed modeling.

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	Untimed or 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 bridges

The TLM2 transport 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 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 act as an initiator or a target for that transaction, typical examples being arbiters and routers.

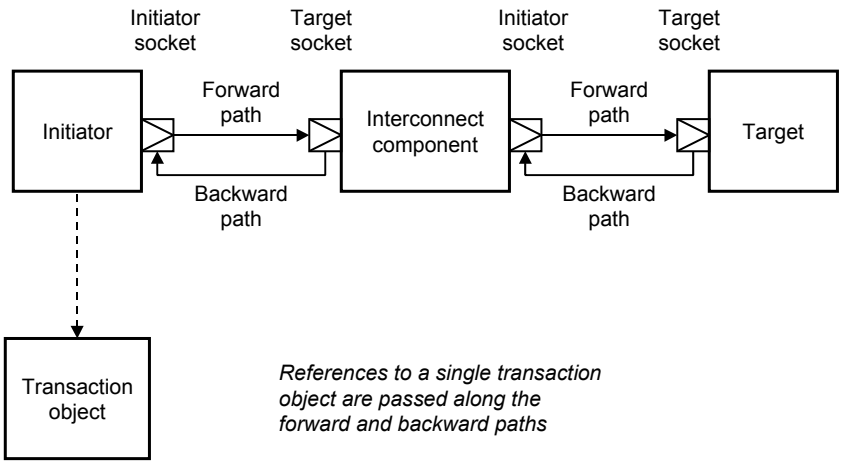
In order to illustrate the idea, this paragraph will describe the lifetime of a typical transaction object. The transaction object is created by an initiator and passed as an argument of a method of the transport interface (blocking or non-blocking). That method is implemented by an interconnect component 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, or passed by making explicit transport method calls in the opposite direction from target back to initiator, known as the *backward path*. This choice is determined by the return value from the **nb_transport** method.

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. 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 contains 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. The initiator and target socket classes overload the SystemC port binding operator to implicitly bind both forward and backward paths.

Figure 3



As well as the transport interfaces, the sockets also encapsulate the DMI and debug transaction 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. Such a component would be a *bridge* between TLM2 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 bridge between two separate TLM2 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.

It is also possible for components to use SystemC ports and exports directly with the core TLM2 interfaces, without consistent using sockets. Sockets are provided for convenience, and are recommended for maximal interoperability and a consistent coding style.

3.5 DMI and debug transaction interfaces

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

The transport interfaces are combined with the DMI and debug transaction interfaces in the standard initiator and target sockets. All three interfaces, transport, DMI, and debug, can be used in parallel to access a target. The DMI has both forward (initiator-to-target) and backward (target-to-initiator) interfaces, whereas debug only has a forward interface.

3.6 Namespaces

All of the TLM2 classes are declared in a single top-level C++ namespace **tlm**. Particular implementations of the TLM2 classes may choose to nest further namespaces within namespace **tlm**, but such nested namespaces shall not be used in applications.

3.7 Header files

User code should only #include the header file **tlm.h** from the top-level include directory of the release.

4 Core TLM2 interfaces

In addition to the core interfaces from TLM1, TLM2 adds blocking and non-blocking transport interfaces, a direct memory interface (DMI), and a debug transaction interface.

4.1 Blocking transport interface

4.1.1 Introduction

The new TLM2 blocking transport interface supports an untimed modeling style. This interface has deliberate similarities with the transport interface from TLM1, which is still part of the TLM2 standard, but the TLM1 transport interface and the TLM2 blocking transport interface are not identical. In particular, the new **b_transport** method takes a single argument passed by non-const reference, whereas the TLM1 **transport** method takes a request as a single const reference argument and returns a response by value. TLM1 assumes separate request and response objects passed by value (or const reference), whereas TLM2 assumes a single transaction object passed by reference, whether using the blocking or the non-blocking TLM2 interfaces.

4.1.2 Migration path from TLM1

The old TLM1 and the new TLM2 interfaces are both part of the TLM2 standard. The TLM1 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 TLM1 interfaces and the new TLM2 interfaces.

4.1.3 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) = 0;
};

} // namespace tlm
```


4.1.4 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**. 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.

4.1.5 Rules

a) The caller is responsible for allocating storage and for constructing the transaction object. The **b_transport** method shall be called with a fully initialized transaction object.

b) In the case of the generic payload, the call to **b_transport** shall mark the first timing point of the transaction. The return from **b_transport** shall mark the final timing point of the transaction.

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

d) The caller is responsible for deleting or pooling the transaction object after the call.

e) The callee shall assume that the caller will invalidate the transaction object upon return from **b_transport**.

f) The **b_transport** method does not support timing annotation or temporal decoupling.

g) It is recommended that the transaction object should not contain timing information. Timing should be annotated using the non-blocking transport interface.

h) **b_transport** may make one or several calls to **nb_transport**. It is straightforward to create an adapter between an untimed initiator and a loosely-timed or approximately-timed target.

4.2 Non-blocking transport interface

4.2.1 Introduction

The new non-blocking transport interface exists to support the loosely-timed and approximately-timed coding styles. This interface is the central innovation of the TLM2 standard. It uses a similar argument-passing mechanism to the new blocking transport interface in that the **nb_transport** method passes a non-const reference to the transaction object, but there the similarity ends. The **nb_transport** method also passes a phase to indicate the state of the transaction, and a time to annotate the delay of the next phase transition. Together, these arguments support multiple phases within the lifetime of a single transaction object, timing annotation, and temporal decoupling for fast simulation.

The non-blocking transport interface also supports the untimed coding style, but for untimed modeling the blocking transport interface may be preferred for its simplicity. The non-blocking transport interface is particularly suited for modeling pipelined transactions, which would be awkward using blocking transport.

The 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 non-blocking transport interface can be used separately from the generic payload to model specific protocols. Both the transaction type and the phase type are template parameters.

4.2.2 Class definition

```
namespace tlm {

enum tlm_phase { BEGIN_REQ, END_REQ, BEGIN_RESP, END_RESP };
enum tlm_sync_enum { TLM_REJECTED = 0, TLM_ACCEPTED = 1,
                     TLM_UPDATED = 2, TLM_COMPLETED = 3 };

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

} // namespace tlm
```

4.2.3 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**. 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.

4.2.4 The PHASE template argument

The intent is that this core interface may be used for transactions with any number of phases and timing points. A specific type **tlm_phase** is provided for use with the generic payload.

For maximum interoperability, applications should use the default arguments **tlm_phase** and **tlm_generic_payload** with the loosely-timed or approximately-timed coding styles. Applications are free to use the type **tlm_phase** with other transaction types, or to use the generic payload with other phase types, but doing either will restrict interoperability.

4.2.5 The nb_transport call

a) The **nb_transport** method shall not call **wait**, directly or indirectly.

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

c) Exceptionally, if both the blocking and non-blocking transport interfaces are being used together, **nb_transport** may need to call **b_transport**. This is only technically possible if it can be guaranteed that the **b_transport** method does not call **wait**, directly or indirectly, but is in any case bad practice. Otherwise, the solution is to call to **b_transport** from a separate thread process, spawned or notified by the original **nb_transport** call.

d) With both the loosely-timed and approximately-timed coding styles, **nb_transport** is called along both the forward and backward paths between initiator and target. In the case of the generic payload, the two paths should pass through exactly the same sequence of components and sockets, obviously in reverse order.

e) The initiator is responsible for deleting or pooling the transaction object after the final timing point. The final timing point may be marked by a call to or a return from **nb_transport** either on the forward path or the backward path. If the final call to **nb_transport** is on the forward path, the initiator may delete the transaction object on return from **nb_transport**. If the final call to **nb_transport** is on the backward path, the initiator may delete the transaction object the next time the initiator process resumes, in which case the transaction object shall remain valid and accessible by the target until the target yields to the SystemC scheduler. In other words, if the final call to **nb_transport** is from target to initiator, the target has a chance to inspect the state of the transaction object before it yields control. After that, any interconnect component or target must assume that the transaction object is invalid. See clause 4.2.12 Phase sequences for loosely-timed and approximately-timed coding styles

f) If an interconnect component or a target needs to access the state of the transaction after the final call to **nb_transport** for a particular transaction instance, it must make a copy of the transaction object.

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

4.2.6 The trans argument

a) The initiator is responsible for allocating storage and constructing the transaction object passed as the first argument. The **nb_transport** method shall be called with a fully initialized transaction object.

b) 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 and targets.

c) The caller of **nb_transport** need not be the initiator. In particular, the target may call **nb_transport** to signal a timing point back to the initiator.

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 clause 9.7 Default values and modifiability of attributes

e) The initiator is responsible for deleting or pooling the transaction object after the final timing point. The initiator, and only the initiator, may delete the transaction object.

4.2.7 The phase argument

a) Each call to **nb_transport** passes a reference to a phase object. A transition from one phase to another marks a timing point. Successive calls to **nb_transport** with the same phase shall not mark timing points. A timing annotation using the **sc_time** argument shall delay the phase transition, if there is one.

b) The attributes of a transaction are notionally stable during each phase, only changing at the timing points that mark phase transitions. Any change to the transaction object occurring in the middle of a phase should only become visible to other components at the next timing point.

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

d) Any change to the state of the transaction should be accompanied by a change to the phase argument such that either caller or callee can detect the change by comparing the value of the phase argument from one call to the next.

e) The value of the phase argument represents the current state of the protocol state machine for the communication between caller and callee. Where a single transaction object is passed between more than two components (initiator, interconnect, target), each caller/callee connection requires (notionally, at least) a separate protocol state machine.

f) Whereas the transaction object has a lifetime and a scope that may extend beyond any single call to **nb_transport**, the phase is normally local to the caller. Each **nb_transport** call for a given transaction may have a separate phase object.

g) For the default phase argument **tlm_phase**, the final timing point depends on the coding style adopted. For the loosely-timed coding style, the transition to the phase BEGIN_RESP marks the final timing point. For the approximately-timed coding style, the transition to the phase END_RESP marks the final timing point.

h) The enum **tlm_phase** is specific to the generic payload. Other protocols may use this same phase type or may substitute their own phase type (with a corresponding loss of interoperability).

4.2.8 The sc_time argument

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

b) The time argument shall be non-negative, and is always relative to the current simulation time.

c) When using the loosely-timed coding style, the caller may pass a positive value for the time argument to indicate to the callee that it should behave as if the transaction were received at time **sc_time_stamp() + t**. This is the local time with respect to temporal decoupling, in other words, an offset from the start time of the current time quantum.

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

e) For a description of the quantum, see clause 7.3 Rules for processes using temporal decoupling

f) If the callee is not able to determine how to process the transaction on the basis of predicting any necessary state information as it will be at a future point in time (**sc_time_stamp() + t**), the callee should return a value of TLM_ACCEPTED. It is still the responsibility of the callee to behave as if the transaction were received at the given future time, and in order to do this, the callee would typically create a timed notification that would cause a process to be resumed at the given future time. As an example, the TLM2 kit includes a payload event queue class **tlm_peq** which may be used for this purpose. See clause 8 Payload event queue

g) With a positive value for the time argument, the callee is being invited to become temporally decoupled from the simulation time, or to "live in a time warp". If the target is a simple slave that only serves one master, that may be 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 (synchronization-on-demand) before it can predict the future state of the transaction with certainty.

h) Having the caller pass a positive value for the time argument to **nb_transport** is normally associated with the loosely-timed coding style, but is still technically possible in an approximately-timed model. When using the approximately-timed coding style, the only reasonable behavior for the callee would be to create a timed notification with the given delay (perhaps using the payload event queue), and return a value of TLM_ACCEPTED. (Having the callee return a positive value for the time argument is usual for both coding styles.)

i) If **nb_transport** is called consecutively at simulation times and with values for the time argument such that the second call is effectively to be processed before the first call, in other words the timing points would occur in reverse order when taking into account the time of the call and the time argument, the callee has two options. Either put the transaction into a payload event queue (or similar) to delay the arrival of the transaction until the correct time (synchronization-on-demand), or process the transactions in the order of the **nb_transport** calls and assume that the system design can tolerate out-of-order execution (because of the existence of some explicit mechanism in the system to enforce the correct causal chain of events). The former option is characteristic of the approximately-timed coding style, and the latter option of the loosely-timed coding style.

j) On return from **nb_transport**, it is the responsibility of the caller to behave as if it had received notification that the transaction will change state at time **sc_time_stamp() + t**, where **t** is the **sc_time** argument to **nb_transport**. In other words, the time argument is used to annotate a latency to the **nb_transport** call, and it is the caller's responsibility to realize that latency.

k) On return from **nb_transport**, the caller has three options for implementing an annotated latency. It can run in temporally decoupled mode, it can put the transaction into a payload event queue (or similar), or it can call **wait(t)** (assuming the caller is a thread process).

l) This non-blocking transport interface is explicitly intended to support pipelined transactions. For example, several successive calls from the same caller at the same time to **nb_transport** could each create notifications (these could be, but are not necessarily, SystemC event notifications) of transaction timing points at distinct future times. It is the responsibility of the caller and callee to keep track of these actual or effective notifications.

m) The callee may increase the value of the time argument, but shall not decrease the value. This rule is consistent with time not running backward in a SystemC simulation.

4.2.9 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 clause 7.3 Rules for processes using temporal decoupling.

b) In principle, synchronization can always 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 when using the loosely-timed coding style, a process should synchronize by calling the **sync** method of class **tlm_quantum_keeper**.

c) When it is stated in a rule that a process should synchronize, the process may execute further statements before actually yielding control.

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

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

f) **TLM_REJECTED**. The callee has rejected the transaction. The callee shall not have modified the state of the transaction object, the phase, or the time argument during the call. The caller should synchronize before attempting to call **nb_transport** again for the same transaction object. This is similar to a return value of false from the methods such as **nb_put** of the TLM1 non-blocking interface, and can be used to code a polling-style interface. TLM_REJECTED does not imply an error.

g) **TLM_ACCEPTED**. The callee has accepted the transaction. The callee shall not have modified the state of the transaction object, the phase, or the time argument during the call. The caller should ignore the values of the arguments following the call, since they shall not have changed. The callee would typically register the fact that the transaction has been accepted by changing some internal state or by making a further call to **nb_transport**. The component containing the caller should expect a response from the component containing the callee through the backward path between them, and should take no further action with respect to this transaction until it receives that response. Following the call to **nb_transport**, the caller should synchronize. This is known as synchronization-on-demand.

h) **TLM_UPDATED**. The callee has accepted and updated the transaction. The callee shall have modified the state of the transaction object and the phase argument during the call. In other words, the callee shall have advanced the state of the protocol state machine associated with the transaction. The callee may have increased the value of the time argument. Following the call to **nb_transport**, the caller should inspect the phase argument and transaction object and take the appropriate action. The caller should behave as if the phase transition occurred at time **sc_time_stamp()+t**, where **t** is the time argument. Depending on the protocol, there may or may not be a subsequent response on the opposing path. Following the call to **nb_transport**, the caller should synchronize (synchronization-on-demand).

i) **TLM_COMPLETED**. The callee has accepted the transaction, and the transaction has been completed. Following the call to **nb_transport**, the caller should inspect the transaction object and take the appropriate action. The caller should behave as if the transaction completed at time **sc_time_stamp()+t**, where **t** is the time argument. The callee is not obliged to have updated the phase argument and the caller should ignore the phase argument, since a transition to the final phase is implicit in the return value TLM_COMPLETED. There shall be no further **nb_transport** calls associated with this transaction

instance 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. Depending on the specific details of the protocol and the modeling style, the caller may or may not need to synchronize following the call.

4.2.10 Coding styles and tlm_sync_enum

The recommended usage of the **tlm_sync_enum** return value across the coding styles is as follows.

LT = Loosely-timed, AT = Approximately-timed, CA = Cycle accurate.

tlm_sync_enum	**Coding style**	**Transaction and phase arguments**	**Timing annotation**
TLM_REJECTED	AT, CA	Unmodified	No
TLM_ACCEPTED	LT, AT	Unmodified	No
TLM_UPDATED	AT	Updated	Yes
TLM_COMPLETED	LT, AT	Updated, but caller may ignore phase	Yes

4.2.11 Coding styles and tlm_phase

The following table summarizes the usage of the **tlm_phase** values across the coding styles.

Forward means that **nb_transport** is called on the forward path, and *backward* that **nb_transport** is called on the backward path.

Call means that the phase transition is indicated by the call to **nb_transport**, and *return* means that the phase transition is indicated by the return from **nb_transport** using TLM_UPDATED or TLM_COMPLETED.

tlm_phase	**Coding style**	**Path**
BEGIN_REQ	Loosely-timed and approximately-timed	Forward (call)
END_REQ	Approximately-timed only	Forward (return) Backward (call)
BEGIN_RESP	Loosely-timed and approximately-timed	Forward (return) Backward (call)
END_RESP	Approximately-timed only	Forward (call) Backward (return)

4.2.12 Phase sequences for loosely-timed and approximately-timed coding styles

This clause is specific to the phase type **tlm_phase** as used by the generic payload, but may be used as a guide when using the non-blocking transport interface to model other protocols. In order to model other protocols it may be necessary to define other phases, but doing so may result in a loss of interoperability with the generic payload.

The full sequence of phase transitions for the loosely-timed coding style is:

BEGIN_REQ → BEGIN_RESP

The full sequence of phase transitions for the approximately-timed coding style is:

BEGIN_REQ → END_REQ → BEGIN_RESP → END_RESP

However, at the level of the non-blocking transport interface itself there is no sharply-drawn distinction between the two coding styles, and both of these sequences can be pre-empted by **nb_transport** returning a value of TLM_COMPLETED. Hence it is possible to directly connect a loosely-timed initiator to an approximately-timed target, and vice versa, although the results may or may not be meaningful from a modeling perspective.

In the case that the sequence of phase transitions is cut short (that is, does not run through to BEGIN_RESP for loosely-timed or END_RESP for approximately-timed), the process responsible shall return a value of TLM_COMPLETED.

A return value of TLM_COMPLETED indicates the end of the transaction, in which case the callee is not obliged to update the phase argument. Completed does not necessarily mean successful, so the initiator should check the response status in the transaction for success or failure. A transition to the phase END_RESP shall also indicate the end of the transaction, in which case the callee is not obliged to return a value of TLM_COMPLETED. The return value could be TLM_ACCEPTED, TLM_UPDATED, or TLM_COMPLETED, depending whether END_RESP is sent on the forward or backward path.

If an approximately-timed initiator receives a BEGIN_RESP from a target without having first received an END_REQ, the initiator shall assume an implicit END_REQ immediately preceding the BEGIN_RESP.

Taking all the previous rules into account, the set of permitted phase transition sequences is as follows, where implicit phase transitions are shown in parenthesis. In each case the transaction may or may not have been successful.

BEGIN_REQ (→ BEGIN_RESP)

BEGIN_REQ → BEGIN_RESP

BEGIN_REQ → END_REQ (→ BEGIN_RESP → END_RESP)

BEGIN_REQ → END_REQ → BEGIN_RESP (→ END_RESP)

BEGIN_REQ (→ END_REQ) → BEGIN_RESP → END_RESP

BEGIN_REQ → END_REQ → BEGIN_RESP → END_RESP

4.2.13 Message sequence charts for nb_transport

The permitted sequence of timing points and **nb_transport** calls for the loosely-timed and approximately-timed coding styles are illustrated below with a series of message sequence charts. 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.

4.2.13.1 Loosely-timed with timing annotation

The loosely-timed coding style is restricted to two phases BEGIN_REQ and BEGIN_RESP, in that order. BEGIN_REQ is sent from initiator to target, and BEGIN_RESP from target back to initiator. If the target can immediately calculate the time of the response and the next state of the transaction, it may return with the value TLM_COMPLETED, the updated transaction, and annotate the delay of the transition to the BEGIN_RESP phase, as shown below.

When **nb_transport** returns the value TLM_COMPLETED the callee is not actually obliged to update the phase argument, so in the chart below the call could have returned with the phase still set to BEGIN_REQ.

Transactions may be pipelined. The initiator could call **nb_transport** to send another transaction to the target before the delay returned from the first call had elapsed. It is the responsibility of the initiator to account for the delays as it wishes.

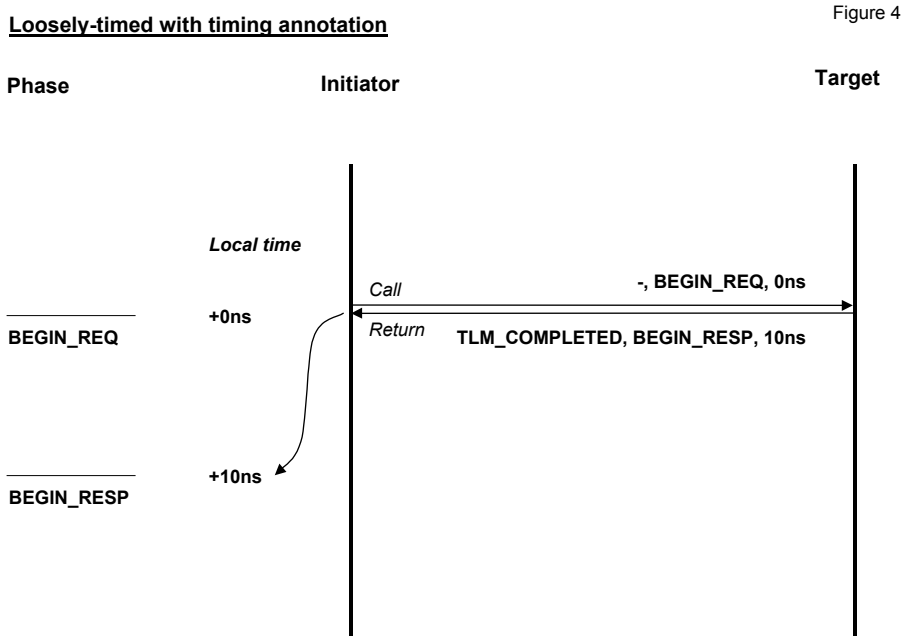


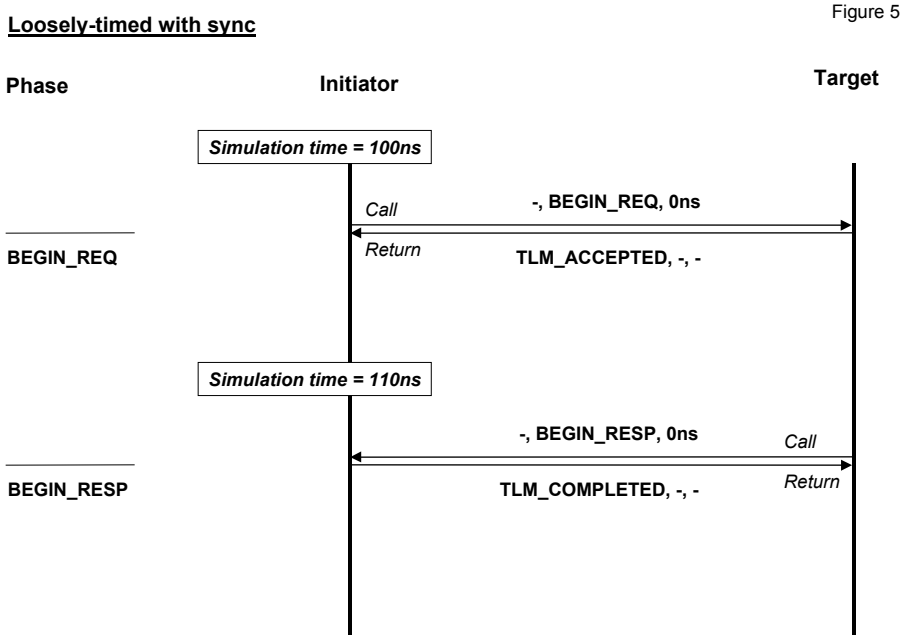
Figure 4

4.2.13.2 Loosely-timed with sync

If the target is unable to calculate the time of the response or the next state of the transaction, it should return with the value TLM_ACCEPTED, as shown below. This tells the initiator to expect a response on the backward path later, but only after the initiator has yielded control to the SystemC scheduler. The target subsequently calls **nb_transport** on the backward path with the phase set to BEGIN_RESP.

The final timing point of this transaction is marked by the call to **nb_transport** by the target. On return from **nb_transport**, the transaction object will remain valid only until the target yields control back to the SystemC scheduler (by calling **wait**, **sync**, or by returning from a method process). At that point the initiator process may resume and delete the transaction object.

Figure 5



4.2.13.3 Loosely-timed with 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 **nb_transport**, as shown below. The target may further advance the local time by increasing the value of the time argument, as may subsequent calls to **nb_transport**. Adding the time returned from the call to the simulation time gives the notional time at which the transaction completes, but simulation time itself cannot advance until the initiator yields.

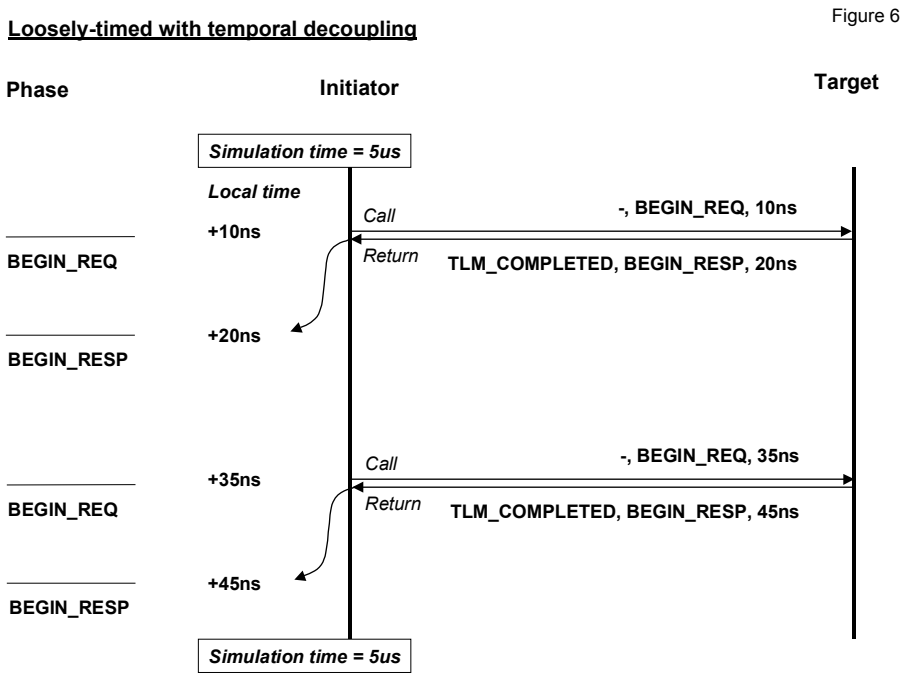
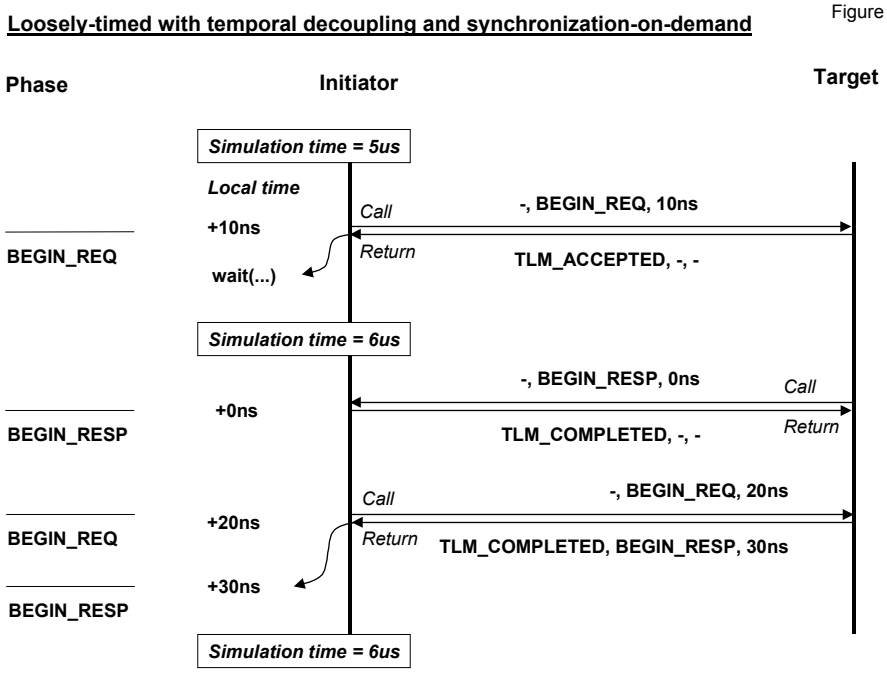


Figure 6

4.2.13.4 Loosely-timed with temporal decoupling and synchronization-on-demand

When a temporally decoupled initiator runs ahead of simulation time and passes a transaction to a target, it is effectively asking the target to predict the future, or run "in a time warp". If the target is unable to calculate the next state of the transaction based on the information it has at the current simulation time, then instead of returning a value of TLM_COMPLETED, the target may refuse to complete the transaction at this time by returning a value of TLM_ACCEPTED. This is "synchronization-on-demand". The initiator should yield control to the scheduler at some point, although it may execute further statements before doing so. Subsequently, after simulation time has advanced, the target will have sufficient information to calculate the next state of the transaction and can call **nb_transport** on the backward path.

Figure 7



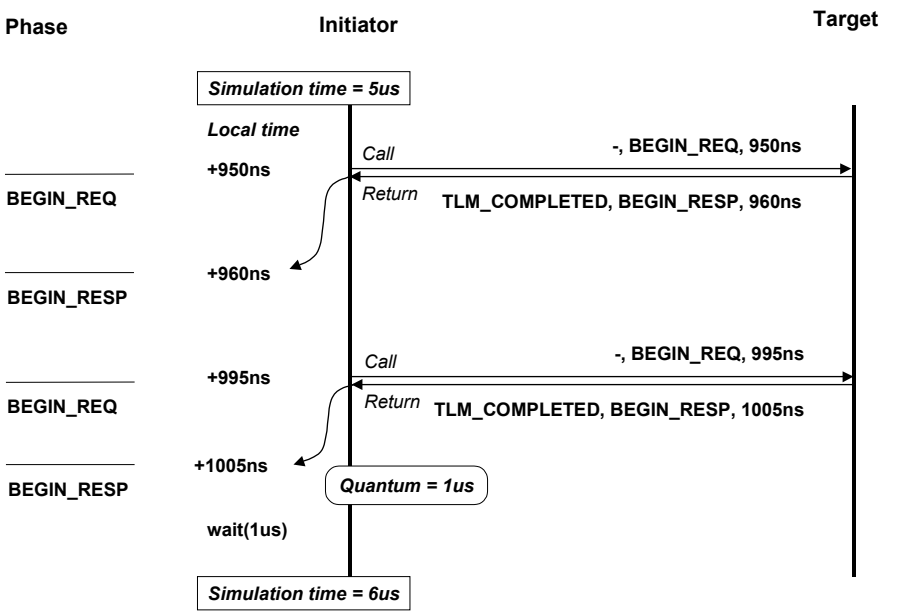
4.2.13.5 Loosely-timed with temporal decoupling and quantum

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 until the next quantum boundary. 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.

Figure 8



4.2.13.6 Approximately-timed timing parameters

a) The approximately-timed coding style with the generic payload uses four phases, having the BEGIN_REQ and BEGIN_RESP phases in common with loosely-timed, but also adding the END_REQ and END_RESP phases. BEGIN_REQ and END_RESP are sent from initiator to target, END_REQ and BEGIN_RESP from target back to initiator.

b) Note that a loosely-timed target responds to a BEGIN_REQ with a BEGIN_RESP, whereas an approximately-timed target may respond to a BEGIN_REQ with either an END_REQ or a BEGIN_RESP. In the latter case, the END_REQ is implicit.

c) 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.

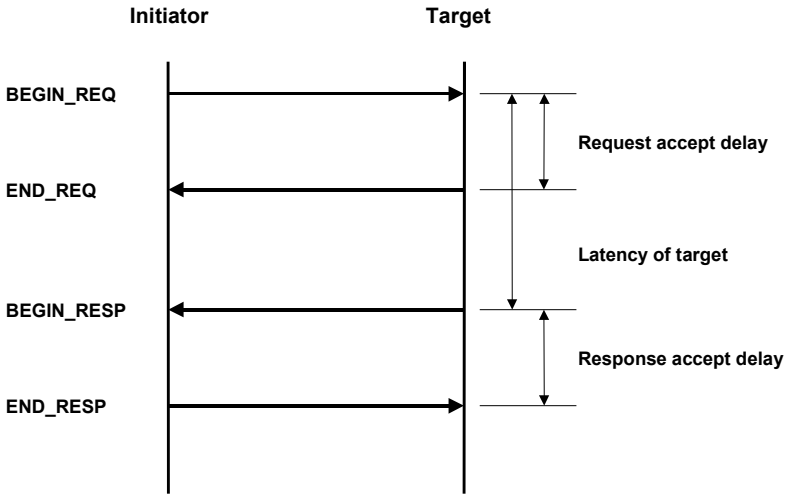
d) Successive transactions can be pipelined.

e) An initiator shall not start a new transaction with phase BEGIN_REQ until it has received END_REQ from the target for the previous transaction or until the target has completed the previous transaction by returning value TLM_COMPLETED.

f) A target shall not respond to a new transaction with phase BEGIN_RESP until it has received END_RESP from the initiator for the previous transaction or until the initiator has completed the previous transaction by returning value TLM_COMPLETED.

Figure 9

Approximately-timed timing parameters



4.2.13.7 Approximately-timed using backward path

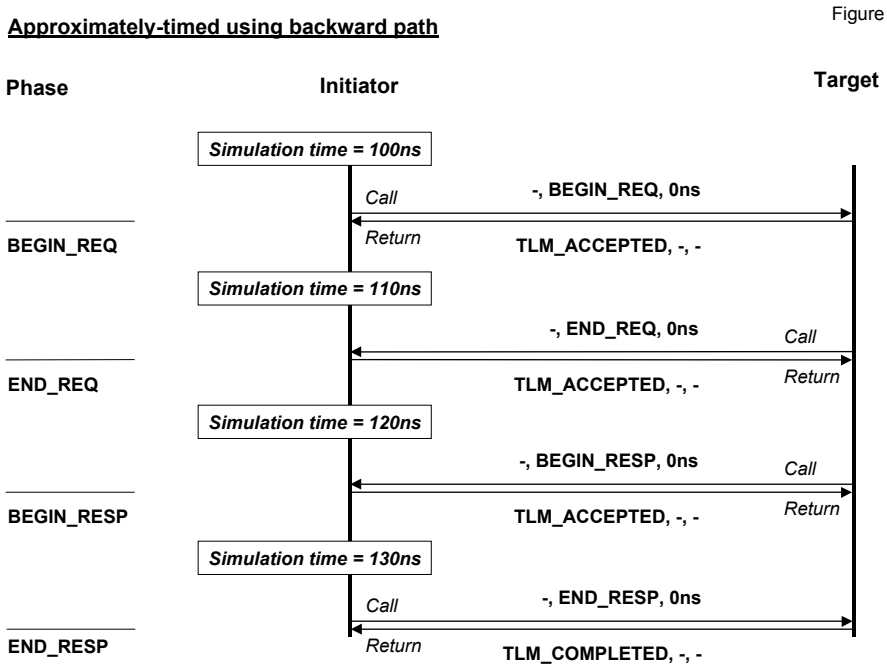
With the approximately-timed coding style for the generic payload, a transaction is passed back-and-forth twice between initiator and target. (For other protocols, this number may be smaller or larger.) As with loosely-timed, a target may or may not be able to respond immediately to a transaction, and this is reflected in the value returned from the **nb_transport** function call.

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 phase transition, it should return a value of TLM_ACCEPTED, the same as for loosely-timed. 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.

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.

A callee can return TLM_COMPLETED at any stage 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.

Figure 10

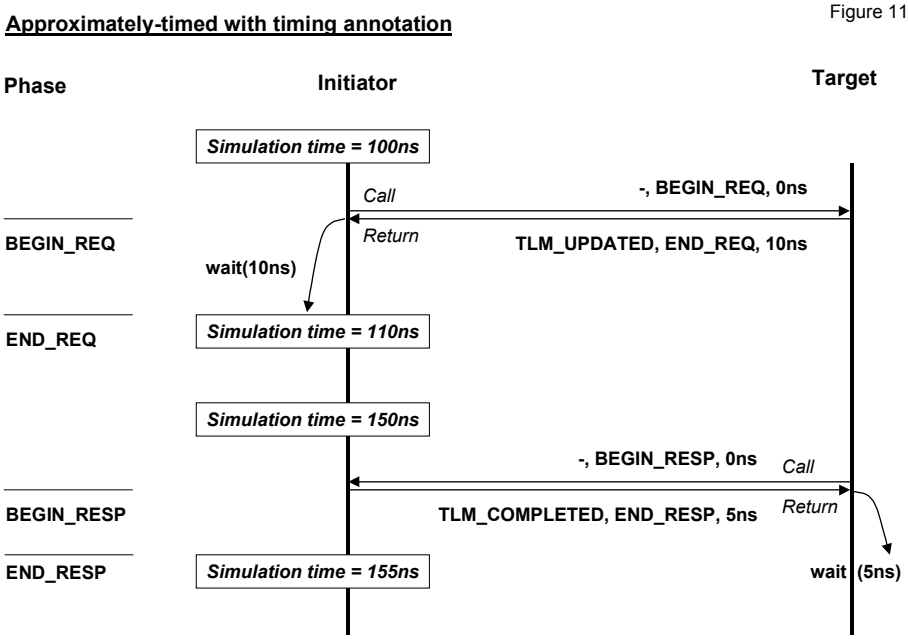


4.2.13.8 Approximately-timed with timing annotation

If the recipient of an **nb_transport** call can immediately calculate the next state of the transaction and the time and the delay to the next phase transition, it may return the new state on return from **nb_transport** rather than using the opposite path. For the loosely-timed coding style this was done by returning TLM_COMPLETED. This is also possible for the approximately-timed coding style when the transaction is complete, but the return value TLM_UPDATED is provided for the case where this timing point does *not* mark the end of the transaction.

With TLM_UPDATED, the callee should update the transaction and the phase and annotate the delay to the phase transition.

Figure 11

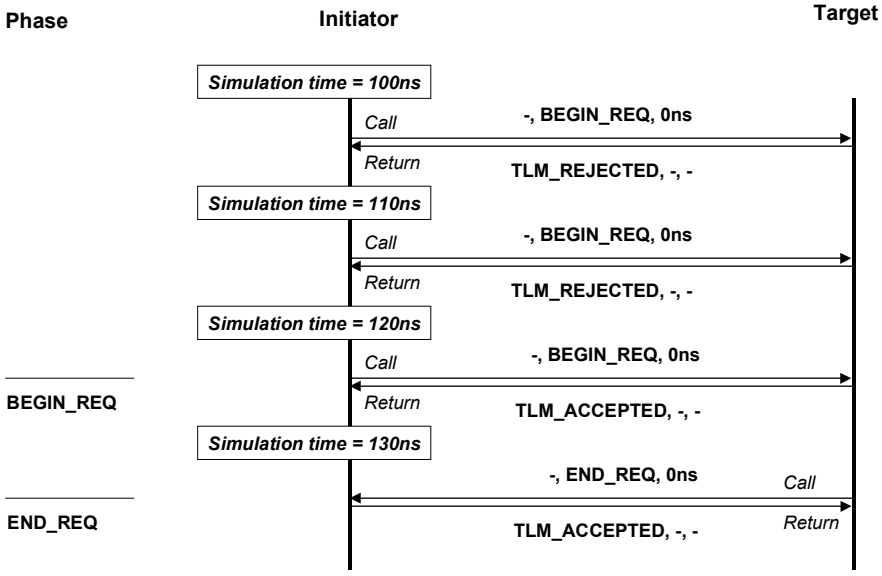


4.2.13.9 Approximately-timed with polling

A target is permitted to bounce an incoming transaction by returning a value of TLM_REJECTED from **nb_transport**. TLM_REJECTED means that the target refused to even register the existence of the given transaction, leaving the initiator no alternative but to yielding control and trying again later. TLM_REJECTED is quite unlike TLM_ACCEPTED, where the target accepts the transaction even though it cannot provide an immediate response.

The return value of TLM_REJECTED can be used to model a polling style of communication.

Figure 12



4.2.13.10 Loosely- to approximately-timed adapter

A loosely-timed initiator can work with an approximately-timed target in one of two ways, either by having the initiator respond to the full set of approximately-timed phase transitions, or by inserting an adapter between them. The downside is that any simulation speed advantage that would be gained from temporal decoupling in the initiator is wiped out by the need to synchronize with the target. The message sequence chart for communication using an adapter is shown below.

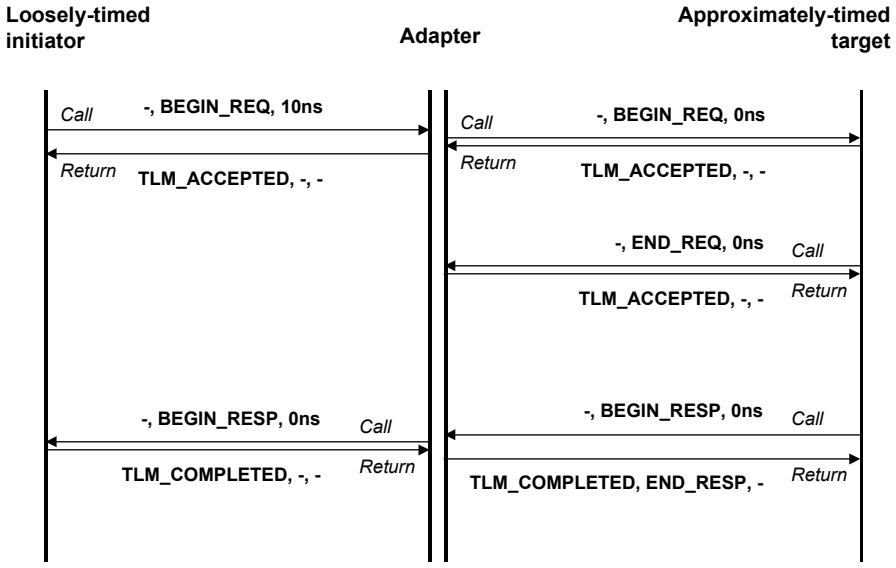
Notice that the loosely-timed initiator is temporally decoupled, and passes the local time on the first call to **nb_transport**. The target ignores the local time, and immediately forces the initiator to synchronize. The phase transition to END_REQ is not required by the initiator. The BEGIN_RESP is passed back to the initiator, which returns with TLM_COMPLETED and regards the transaction as being complete. The adapter can generate the transition to the END_RESP phase and pass it back to the target on return from the function call. The approximately-timed target may call **wait** to advance simulation time between **nb_transport** calls.

If the adapter is omitted and the initiator connected directly to the target, the loosely-timed initiator must be able to ignore the END_REQ call from the target. All of this assumes that the generic payload is being used. Other protocols with additional phases would require their own rules.

The loosely-timed initiator is free to delete or pool the transaction object just as soon as it resumes execution following the final call to **nb_transport** from adapter to initiator on the backward path. If adapter or target needs to retain the state of the transaction after this point, it must take a copy of the transaction object.

Figure 13

Loosely- to approximately-timed adapter

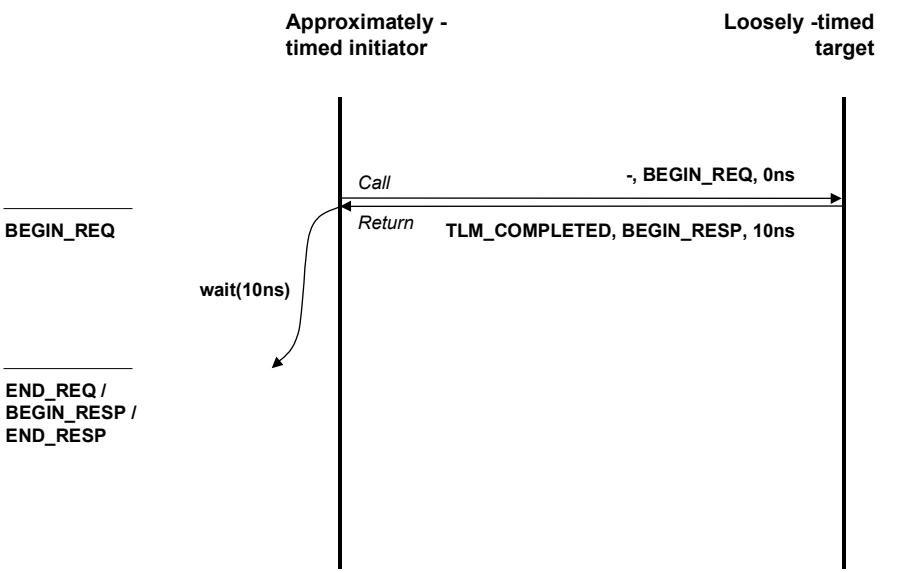


4.2.13.11 Approximately-timed initiator to loosely-timed target

The non-blocking transport interface used with **tlm_phase** permits an approximately-timed initiator to be connected to a loosely-timed target, directly or using an adapter. Whether this will actually be useful depends on the details of the initiator model, but the mechanism works in principle.

The loosely-timed target may complete the transaction on return from **nb_transport**, possibly annotating a delay for the transition from the BEGIN_REQ to the BEGIN_RESP phase. On return from **nb_transport**, the initiator should suspend for the given delay, and assume an implicit END_REQ phase coinciding with the transition to BEGIN_RESP. Since the target has indicated that the transaction is complete, the initiator must not call **nb_transport** again to send END_RESP to the target, so the transition to END_RESP is also implicit. The initiator must be robust enough to handle the three timing points being coincident. Because the target is only loosely-timed, there is no way for the initiator to distinguish between the accept delay and the latency of the target, leading to possible timing inaccuracies.

Figure 14



4.2.14 Transaction lifetime example

A typical transaction lifetime might run as follows. A transaction is created by an initiator and is passed as an **nb_transport** argument to an interconnect component representing a bus bridge, which returns with a value of TLM_ACCEPTED. That bridge component in turn passes the transaction as an **nb_transport** argument to a target, which also returns with a value of TLM_ACCEPTED. Some time later the target is able to complete

the transaction, and passes the transaction back to the bridge as an **nb_transport** argument through the backward path between target and bridge, **nb_transport** returning the value of TLM_COMPLETED. The bridge passes the transaction back to the initiator as an **nb_transport** argument through the backward path between bridge and initiator, also getting a return value of TLM_COMPLETED. Finally, the initiator maintains the transaction object in a memory pool for subsequent reuse.

4.3 Direct memory interface

4.3.1 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 **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.

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 transaction 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.

4.3.2 Class definition

```
namespace tlm {

// Defined in tlm_helpers.h
enum tlm_endianness {TLM_UNKNOWN_ENDIAN, TLM_LITTLE_ENDIAN, TLM_BIG_ENDIAN};[2]

class tlm_dmi_mode
{
public:
   enum Type { READ = 0x1, WRITE = 0x2, READ_WRITE = READ|WRITE };
```

[2] The generic payload helper functions are usable, but are still under development. The helpers have not yet been tuned to the proposed model for endianness in the generic payload.


```
    Type type;
};

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

    void init();

    unsigned char* dmi_ptr;

    sc_dt::uint64 dmi_start_address;
    sc_dt::uint64 dmi_end_address;

    sc_core::sc_time  read_latency;
    sc_core::sc_time  write_latency;

    tlm_endianness endianness;
};

template <typename DMI_MODE = tlm_dmi_mode>
class tlm_fw_direct_mem_if : public virtual sc_core::sc_interface
{
public:
  virtual bool get_direct_mem_ptr(     const sc_dt::uint64& address,
      DMI_MODE& dmi_mode,
      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
```

4.3.3 get_direct_mem_ptr method

a) The **get_direct_mem_ptr** method shall only be called by an initiator or by an interconnect component.

b) The **address** argument shall be the address for which direct memory access is requested.

c) The **dmi_mode** argument shall pass a reference to a DMI mode object constructed by the initiator. The DMI mode object shall indicate the mode of DMI access being requested. The default type for this

argument is the class **tlm_dmi_mode**, which includes a member to indicate whether the initiator is requesting read access or write access to the given address.

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

e) Any interconnect components should pass on 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 the initiator socket, decoding and where necessary modifying the address exactly as they would for the corresponding transport interface. 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.

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

g) Any interconnect components shall pass on the **dmi_mode** and **dmi_data** arguments without modification in the forward direction, although they may modify the DMI mode or DMI descriptor upon return from the **get_direct_mem_ptr** method, that is, when unwinding the function calls from target back to initiator.

h) 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**.

i) If the target is not able to support DMI access to the given address, it shall 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**.

4.3.4 DMI_MODE template argument and tlm_dmi_mode class

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

b) The intent of the DMI mode class is to indicate to the target the mode of DMI access being requested, typically read access or write access.

c) The initiator shall be responsible for managing the DMI mode object, and may re-use a DMI mode object from one call to the next

d) The default value of the DMI_MODE template argument shall be the class **tlm_dmi_mode**, which contains a single data member indicating whether the initiator is requesting read access or write access.

e) Member **type** shall be set by the initiator to indicate the kind of DMI access requested, READ for readonly access, or WRITE for writeonly access. The target may modify the value of member **type** in order to promote READ or WRITE access to READ_WRITE access. The initiator shall not set the member **type** to READ_WRITE. An interconnect component is permitted to restrict the kind of access by overwriting a value of READ_WRITE with READ or WRITE on the backward path only.

f) An application may substitute another class in place of **tlm_dmi_mode** in order to add further attributes used to by the target to determine the kind of DMI access being requested. For example, the DMI mode might include a CPU ID to allow the target to service DMI requests differently depending on the kind of CPU making the request.

g) If an application needs to substitute another DMI mode class but still needs to distinguish between DMI read and write access, it is recommended that the new DMI mode class should be derived from **tlm_dmi_mode**. The precise semantics of such a DMI mode class are application-specific.

h) The initiator is responsible for only using those modes of DMI access for which it has been granted access by the target using the DMI mode object.

4.3.5 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. An initiator may re-use a DMI descriptor from one call to the next, in which case the initiator shall call the **init** method to re-initialize the object between calls to **get_direct_mem_ptr**.

b) 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.

c) Method **init** shall initialize the members as described below.

d) Member **dmi_ptr** shall be set to point to the storage location corresponding to the value of member **dmi_start_address**. 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.

e) The storage in the DMI region is represented with type **unsigned char***. For a full description of how memory organization and endianness are handled in TLM2, see clause 9.10 Data pointer attribute

f) Members **dmi_start_address** and **dmi_end_address** shall be set to the addresses of the first and the last bytes in the DMI region, where the meaning of the DMI region is determined by the value returned from the **get_direct_mem_ptr** method (**true** or **false**).

g) If **get_direct_mem_ptr** return the value **true**, the DMI region indicated by **dmi_start_address** and **dmi_end_address** 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.

h) 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.

i) 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.

j) 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.

k) Any interconnect components that pass on the **get_direct_mem_ptr** call are obliged to transform these addresses as they do the address **argument**. Any transformations on the addresses in the DMI descriptor shall occur as the descriptor is passed along the backward path. For example, the target may set **dmi_start_address** 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.

l) A target may disallow DMI access to the entire address space (**dmi_start_address** = 0, **dmi_end_address** = 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 mislead into thinking that DMI was disallowed across the entire system address space.

m) Members **read_latency** and **write_latency** shall be set to the latencies of read and write memory transactions, respectively. 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 member **type**.

n) 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.

o) Since DMI gives direct access to a memory region in the target, the memory organization will have the endianness of the internal data representation within the target, regardless of the endianness of the bus interface being modeled.

p) Member **endianness** shall be set by the target to indicate the endianness of the target for the purpose of DMI.

4.3.6 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) 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.

g) 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.

h) 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**.

i) An implementation of **invalidate_direct_mem_ptr** shall not call **get_direct_mem_ptr**, directly or indirectly.

4.3.7 Optimization using a DMI Hint

a) The DMI hint 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 hint of a normal transaction passed through the transport interface.

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

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

d) For an initiator wishing to use DMI, 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 hint of the transaction

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

4.4 Debug transaction interface

4.4.1 Introduction

The debug transaction 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 transaction interface is non-intrusive. Because the debug transaction interface follows the same path as the transport interface, the implementation of the debug transaction interface can perform the same address translation as for regular transactions.

For example, the debug transaction 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 transaction 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.

4.4.2 Class definition

```
namespace tlm {
```



```
class tlm_debug_payload
{
public:
   sc_dt::uint64 address;
   bool do_read;
   unsigned int num_bytes;
   unsigned char* data;
};

class tlm_transport_dbg_if : public virtual sc_core::sc_interface
{
public:
  virtual unsigned int transport_dbg(tlm_debug_payload& r) = 0;
};

} // namespace tlm
```

4.4.3 Rules

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

b) The initiator shall construct the debug payload object, and shall set each of the members of the object before passing it as an argument to **transport_dbg**.

c) Member **address** shall be set to the first address in the region to be read or written. Member **address** may be modified by any interconnect component that is performing address decoding or translation. 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.

d) Member **address** 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 member **address** may have been overwritten.

e) Member **do_read** shall be set to **true** to read from (or peek) the target, or **false** to write to (or poke) the target.

f) Member **num_bytes** shall be set to the number of bytes to be read or written. This may be 0, in which case the target shall not read or write any bytes.

g) Member **data** shall be set 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). 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 **num_bytes**.

h) 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.

i) The **data** array shall always have the endianness of the host machine. The implementation of **transport_dbg** shall be responsible for converting between the endianness used by the target and the

endianness of the host machine such that the debug transaction interface as it appears to the initiator is independent of target endianness.

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

k) **transport_dbg** shall not call wait, shall not create any event notifications, and shall not have any side effects on the target or any interconnect component.

5 Sockets and combined interfaces

5.1.1 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.

The combined interfaces create groupings of the core TLM2 interfaces for use by the initiator and target sockets. It would be technically possible to define new socket classes using the combined interfaces, but this is not recommended for interoperability. Note that these groupings include the transport, DMI and debug transaction interfaces, but do not include any TLM1 core interfaces.

The groups are categorized along two dimensions: forward and backward interfaces, and blocking and non-blocking transport interfaces. The forward interfaces provide method calls on the forward path from initiator socket to target socket, and the backward interfaces on the backward path from target socket to initiator socket. Neither the blocking transport interface nor the debug transaction interface require a backward calling path.

Each of the four combined interfaces is parameterized with a single *protocol types class* that defines the types of the transaction object, the phase, and the DMI mode. This class effectively identifies the protocol type. Sockets can only be bound together if they have the identical protocol type. The default protocol type is the class **tlm_generic_payload_types**. If an application defines a new protocol it should parameterize the combined interfaces with a new protocol types class, whether the new protocol is based on the generic payload or is entirely new.

As well as the initiator and target socket classes parameterized with the combined interfaces, there are four convenience socket classes parameterized with the protocol types class.

The initiator and target sockets provide the following benefits:

a) They group the transport, direct memory and debug transaction 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.

5.1.2 Class definition

```
namespace tlm {

// The default protocol types class:
struct tlm_generic_payload_types
{
   typedef  tlm_generic_payload  tlm_payload_type;
   typedef  tlm_phase     tlm_phase_type;
   typedef  tlm_dmi_mode   tlm_dmi_mode_type;
};

// The forward non-blocking interface:
template< typename TYPES = tlm_generic_payload_types >
class tlm_fw_nb_transport_if
   : public virtual tlm_nonblocking_transport_if<typename TYPES::tlm_payload_type ,
                                                 typename TYPES::tlm_phase_type >
   , public virtual tlm_fw_direct_mem_if<typename TYPES::tlm_dmi_mode_type>
   , public virtual tlm_transport_dbg_if
{};

// The backward non-blocking interface:
template < typename TYPES = tlm_generic_payload_types >
class tlm_bw_nb_transport_if
  : public virtual tlm_nonblocking_transport_if<  typename TYPES::tlm_payload_type ,
                                                  typename TYPES::tlm_phase_type >
  , public virtual tlm_bw_direct_mem_if
{};

// The forward blocking interface:
template < typename TYPES = tlm_generic_payload_types >
class tlm_fw_b_transport_if
  : public virtual tlm_blocking_transport_if<   typename TYPES::tlm_payload_type >
  , public virtual tlm_fw_direct_mem_if<  typename TYPES::tlm_dmi_mode_type >
  , public virtual tlm_transport_dbg_if
{};

// The backward blocking interface:
class tlm_bw_b_transport_if
  : public virtual tlm_bw_direct_mem_if
{};
```



```
// Initiator socket
template <
   unsigned int BUSWIDTH = 32,
   typename FW_IF = tlm_fw_nb_transport_if<>,
   typename BW_IF = tlm_bw_nb_transport_if<>
>
class tlm_initiator_socket : public sc_core::sc_port<FW_IF>
{
public:
   typedef FW_IF fw_interface_type;
   typedef BW_IF bw_interface_type;
   typedef sc_core::sc_port<fw_interface_type> port_type;
   typedef sc_core::sc_export<bw_interface_type> export_type;
   typedef tlm_ target_socket<BUSWIDTH, fw_interface_type, bw_interface_type> target_socket_type;

   tlm_initiator_socket( const char* );

   unsigned int get_bus_width() const;

   void bind( target_socket_type& );
   void operator() ( target_socket_type& );
   void bind( tlm_initiator_socket& );
   void operator() ( tlm_initiator_socket& );
   void bind( bw_interface_type& );
   void operator() ( bw_interface_type& );

protected:
  export_type mExport;
};

// Target socket
template <
   unsigned int BUSWIDTH = 32,
   typename FW_IF = tlm_fw_nb_transport_if<>,
   typename BW_IF = tlm_bw_nb_transport_if<>
>
class tlm_target_socket : 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> port_type;
   typedef sc_core::sc_export<fw_interface_type> export_type;
   typedef tlm_initiator_socket<BUSWIDTH, fw_interface_type, bw_interface_type>
      initiator_socket_type;
```



```
    tlm_target_socket( const char* );

    unsigned int get_bus_width() const;

    void bind( initiator_socket_type& );
    void operator() ( initiator_socket_type& );
    void bind( tlm_target_socket& );
    void operator() ( tlm_target_socket& );
    void bind( fw_interface_type& );
    void operator() ( fw_interface_type& );

    bw_interface_type* operator-> ();

  protected:
    port_type mPort;
  };

  // Convenience sockets using blocking- and non-blocking transport interfaces
  template < unsigned int BUSWIDTH = 32, typename TYPES = tlm_generic_payload_types >
  class tlm_nb_initiator_socket
    : public tlm_initiator_socket<    BUSWIDTH,
                                      tlm_fw_nb_transport_if<TYPES>,
                                      tlm_bw_nb_transport_if<TYPES> >
  {
  public:
    tlm_nb_initiator_socket(const char* name) : tlm_initiator_socket(name) {}
  }

  template < unsigned int BUSWIDTH = 32, typename TYPES = tlm_generic_payload_types >
  class tlm_b_initiator_socket
    : public tlm_initiator_socket<    BUSWIDTH,
                                      tlm_fw_b_transport_if<TYPES>,
                                      tlm_bw_b_transport_if<TYPES> >
  {
  public:
    tlm_b_initiator_socket(const char* name) : tlm_initiator_socket(name) {}
  }

  template < unsigned int BUSWIDTH = 32, typename TYPES = tlm_generic_payload_types >
  class tlm_nb_target_socket
    : public tlm_target_socket<   BUSWIDTH,
                                  tlm_fw_nb_transport_if<TYPES>,
                                  tlm_bw_nb_transport_if<TYPES> >
  {
  public:
    tlm_nb_target_socket(const char* name) : tlm_target_socket(name) {}
```



```
}

template < unsigned int BUSWIDTH = 32, typename TYPES = tlm_generic_payload_types >
class tlm_b_target_socket
   : public tlm_target_socket<  BUSWIDTH,
                                tlm_fw_b_transport_if<TYPES>,
                                tlm_bw_b_transport_if<TYPES> >
{
public:
   tlm_b_target_socket(const char* name) : tlm_target_socket(name) {}
}

} // namespace tlm
```

5.1.3 Rules

a) For class **tlm_initiator_socket**, the constructor 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, calling **sc_gen_unique_name** to avoid name clashes and adding the suffix **"_export"**. For example, the call **tlm_initiator_socket("foo")** would set the port name to **"foo"** and the export name to **"foo_export"**.

b) For class **tlm_target_socket**, the constructor 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, calling **sc_gen_unique_name** to avoid name clashes and adding the suffix **"_port"**. For example, the call **tlm_target_socket("foo")** would set the export name to **"foo"** and the port name to **"foo_port"**.

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

d) 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 clause 9.11 Data length attribute.

e) 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.

f) 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.

g) 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.)

h) 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.

i) 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.

j) 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 child socket to parent socket, or parent socket to child socket, passing transactions up or down the module hierarchy.

k) 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_initiator_socket** is derived from **sc_port**, and **tlm_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.

l) In order for two sockets to be bound together, they must share the same forward and backward interface types and bus widths, and hence must share the same protocol types class (default **tlm_generic_payload_types**). Strong type checking between sockets can be achieved by defining a new protocol types class for each distinct protocol, whether or not that protocol is based on the generic payload.

m) 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.

n) The convenience sockets **tlm_nb_initiator_socket**, **tlm_b_initiator_socket**, **tlm_nb_target_socket**, and **tlm_b_target_socket** should generally be used in preference to **tlm_initiator_socket** and **tlm_target_socket**.

Example

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

struct Initiator: sc_module, tlm::tlm_bw_nb_transport_if<> // Initiator implements the bw interface
{
    tlm::tlm_nb_initiator_socket<32> init_socket;          // Protocol types default to generic payload

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



```
    void T() {                                          // Process generates one dummy transaction
        tlm::tlm_generic_payload t;
        tlm::tlm_phase phase = tlm::BEGIN_REQ;
        sc_time delay = SC_ZERO_TIME;
        init_socket->nb_transport(t, phase, delay);
    }

    virtual tlm::tlm_sync_enum nb_transport(
        tlm::tlm_generic_payload& trans, tlm::tlm_phase& phase, sc_core::sc_time& t) {
        return tlm::TLM_REJECTED;                       // 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_nb_transport_if<>   // Target implements the fw interface
{
    tlm::tlm_nb_target_socket<32> targ_socket;          // Protocol types default to generic payload

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

    virtual bool get_direct_mem_ptr(
        const sc_dt::uint64& address, tlm::tlm_dmi_mode& dmi_mode, tlm::tlm_dmi&  dmi_data)
    { return false; }                                   // Dummy implementation

    virtual unsigned int transport_dbg(tlm::tlm_debug_payload& r)
    { return 0; }                                       // Dummy implementation
};

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

    SC_CTOR(Top) {
        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_nb_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_nb_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(Top) {
        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
    }
};
```


6 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 clause 8 Payload event queue.

The TLM2 kit includes the **tlm_analysis_fifo**, which 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.

6.1 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
```

6.2 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(T);
   }
   void T()
   {
      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
  }
};
```


7 Quantum keeper

7.1 Introduction

Temporal decoupling permits SystemC processes to run ahead of simulation time for an amount of time known as the time quantum. When using the loosely-timed coding style, the delays annotated to the **nb_transport** method should be interpreted as local times defined relative to the start of the current time quantum (which is always the current simulation time as returned by **sc_time_stamp()**).

Temporal decoupling permits a significant simulation speed improvement by reducing the number of context switches and events.

The 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 strongly recommended in order to maintain a consistent coding style. However, it is straightforward in principle to implement temporal decoupling directly in SystemC.

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

For a description of timing annotation and the non-blocking transport interfaces, see clause 4.2.8 The sc_time argument

7.2 Class definition

```
namespace tlm {

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();

  void inc( const sc_core::sc_time& );
  sc_core::sc_time get_current_time() const;
  const sc_core::sc_time& get_local_time() const;
  sc_core::sc_time& get_local_time();

  bool need_sync() const;
  void reset();
  void sync();
```



```
protected:
  virtual sc_core::sc_time compute_local_quantum();

};

} // namespace tlm
```

7.3 Rules for processes using temporal decoupling

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

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 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) For maximum simulation speed, all initiators should use temporal decoupling, and the number of other runnable SystemC processes should be zero or minimized.

d) 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.

e) *Yield* means call **wait** in the case of a thread process, or return from the function in the case of a method process.

f) 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.

g) 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.

h) The process should accumulate any local processing delays and communication delays in a local variable, referred to in this clause as the local time offset. It is strongly recommended that the quantum keeper should be used to maintain the local time offset..

i) Calls to the **sc_time_stamp** method will return the simulation time as it was at the start of the current time quantum.

j) The local time offset is unknown to the SystemC scheduler. When using the non-blocking transport interface, the local time offset should be passed as an argument to the **nb_transport** method.

k) Any access to a non-local object will give the value as it was at the start of the time quantum, unless it has been modified by this or another temporally decoupled process. In particular, any **sc_signal** cannot have been updated.

l) The constructor shall set the local time offset to SC_TIME_ZERO and shall call the method **compute_local_quantum**.

m) 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 **reset** to recalculate the local quantum.

n) The method **get_local_time** shall return the current value of the local time offset.

o) The method **get_current_time** shall return the current value of the effective local time, which shall be equal to **sc_time_stamp() + local_time_offset**

p) The method **inc** shall add the value passed as an argument to the local time offset.

q) The method **need_sync** shall return the **value** true if and only if the local time offset is greater than the local quantum.

r) The local quantum is the amount of simulation time remaining in the current quantum before the initiator is required to synchronize. In other words, the local quantum is the local time offset subtracted from the time quantum being used by the current initiator.

s) 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**..

t) The method **reset** shall call the method **compute_local_quantum** and shall set the local time offset back to SC_ZERO_TIME.

u) The method **compute_local_quantum** shall recalculate and return the value of the local quantum. The default implementation shall calculate the local quantum by subtracting the value of **sc_time_stamp** from the next largest integer multiple of the global quantum. This method may be overridden.

v) 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.

w) 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.

x) 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.

y) 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.

z) The time quantum should be chosen to be less than the typical communication interval between initiators, otherwise important process interactions may be lost, and the model broken.

aa) If both the blocking and non-blocking transport interfaces are being used together, an initiator using temporal decoupling may be in the situation where it directly or indirectly makes a call to **b_transport**, typically from an adapter. If the blocking transport method does indeed call **wait**, then the caller shall be a thread process (otherwise there will be a SystemC error).

bb) An initiator should synchronize before making a call to **b_transport** or to any other blocking method that cannot support temporal decoupling. In other words, you should drop out of time-warp before calling a method that can only live in the present.

cc) An initiator that makes frequent blocking calls cannot make effective use of temporal decoupling because it is obliged to synchronize before every blocking call.

Example

```
struct Initiator1: sc_module, tlm::tlm_bw_nb_transport_if<> // Loosely-timed initiator
{
    tlm::tlm_nb_initiator_socket<32> init_socket;

    tlm::tlm_quantumkeeper m_qk;                          // The quantum keeper

    SC_CTOR(Initiator1) : init_socket("init_socket") {
        SC_THREAD(T);                                     // The initiator process
        init_socket.bind( *this );                        // Initiator socket bound to the initiator itself
        m_qk.set_global_quantum( sc_time(1, SC_US) );     // Replace the global quantum
        m_qk.reset();                                     // Re-calculate the local quantum
    }

    void T() {
        tlm::tlm_generic_payload t;
        tlm::tlm_phase phase;
        t.set_command(tlm::TLM_WRITE_COMMAND);
        t.set_data_length(4);

        for (int i = 0; i < RUN_LENGTH; i += 4) {
            int word = i;
            t.set_address(i);
            t.set_data_ptr( (unsigned char*)(&word) );
            phase = tlm::BEGIN_REQ;
                                                          // Annotate nb_transport with local time
            tlm::tlm_sync_enum status = init_socket->nb_transport(t, phase,  m_qk.get_local_time() );
            switch (status) {
                ...
            }

            m_qk.inc( sc_time(100, SC_NS) );              // Keep a tally of time consumed
            if ( m_qk.need_sync() ) m_qk.sync();          // Check local time against quantum
        }
    }
    ...
};
```


8 Payload event queue

8.1 Introduction

The 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 pops out of the back of the PEQ at a time calculated from the current simulation time plus the annotated delay.

The class **tlm_peq** is not a definitive part of the TLM2 standard[3]. As well as being useful in its own right, the PEQ is included in this document because of its conceptual relevance to understanding the semantics of the non-blocking transport interface. It is particularly relevant when combining the non-blocking interface with the approximately-timed coding style. However, it is possible to implement **nb_transport** without using the specific payload event queue given here.

Transactions are inserted into the PEQ by calling the **write** method of **delayed_write_export**, passing the delay associated with the transaction as an argument. 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 from the PEQ through calls to the **write** method of the **write_port**. If the intent is to have a process resume each time a transaction emerges, a simple solution is to bind the **write_port** to a **tlm_fifo**.

Transactions emerge at scheduled times as calculated from the delay argument, not in the order in which they were inserted. If several transactions are scheduled to emerge at the same time, they will all emerge in a single delta cycle. Transactions cannot be lost or cancelled.

The PEQ uses the interfaces **tlm_write_if** and **tlm_delayed_write_if,** which are definitive parts of the TLM2 standard. The write interface is the base class for **tlm_analysis_if**. See clause 6 Analysis interface and analysis ports

Because the PEQ is included primarily as an example, the full source code is listed below.

8.2 Class definition

```
namespace tlm {

// Write interface
template <typename T>
```

[3] An alternative payload event queue implementation MyPEQ is included as an example in the TLM2 kit. MyPEQ has some advantages over tlm_peq, and may become the preferred implementation. The original tlm_peq implementation is shown in this document.


```
class tlm_write_if[4] : 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;
};

template< typename T>
class tlm_peq :  public sc_core::sc_module ,  public virtual tlm_delayed_write_if< T >
{
public:
   sc_core::sc_export< tlm_delayed_write_if< T > > delayed_write_export;
   sc_core::sc_port< tlm_write_if< T > > write_port;

   SC_HAS_PROCESS( tlm_peq );

   tlm_peq( sc_core::sc_module_name nm ) : sc_core::sc_module( nm ) , write_port("write_port") {
      delayed_write_export( *this );

      SC_METHOD( wake_up_method );
         dont_initialize();
         sensitive << m_wake_up;
   }

   int size() const { return m_map.size(); }

   int posted_before( const sc_core::sc_time &time ) const {
      int i = 0;
      for( typename std::multimap< const sc_core::sc_time , T>::const_iterator iter = m_map.begin();
            iter != m_map.end();
            ++iter ) {
         if( (*iter).first < time ) {i++; }
      }
      return i;
   }

   int posted_at( const sc_core::sc_time &time ) const { return m_map.count( time ); }

   void write( const T &transaction , const sc_core::sc_time &time ) {
```

[4] The write interface is listed here for convenience. For the main listing, see clause 6 Analysis interface and analysis ports


```
    m_map.insert( pair_type( time + sc_core::sc_time_stamp() , transaction ) );
    m_wake_up.notify( time );
  }

private:
  typedef std::pair<const sc_core::sc_time, T> pair_type;
  typedef std::multimap<const sc_core::sc_time, T> map_type;

  void wake_up_method() {
    sc_core::sc_time now = sc_core::sc_time_stamp();

    // must be something there, and it must be scheduled for now
    assert( m_map.size() > 0 );
    assert( (*(m_map.begin())).first == now );

    for( typename std::multimap<const sc_core::sc_time, T>::const_iterator iter = m_map.begin();
      (*iter).first == now; iter = m_map.begin() )
    {
      write_port->write( (*iter).second );
      m_map.erase( m_map.begin() );
      if( m_map.size() == 0 ) { return; }
    }

    m_wake_up.notify( (*m_map.begin()).first - now );
  }

  sc_core::sc_event m_wake_up;
  map_type m_map;
};

} // namespace tlm
```


9 Generic payload

9.1 Introduction

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 busses 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 busses. 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 be used as the basis for modeling protocols other than memory-mapped busses, but memory-mapped busses are its primary purpose.

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 specialised attributes.

For specific protocols, whether standard or proprietary, modeling and interoperability are the responsibility of the protocol owner and are outside the scope of OSCI. It is up to the protocol owner 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.

9.2 Extensions and interoperability

The goal of the generic payload is to enable interoperability between memory-mapped bus models, but all busses 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, a 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 TLM2 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 types class containing a **typedef** for **tlm_generic_payload**.

c) Define a new protocol types 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.

9.2.1 Use the generic payload directly, with ignorable extensions

a) In this case, the transaction type is **tlm_generic_payload**, and the protocol types class for the combined interfaces is **tlm_generic_payload_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 a core transport interface specialized with the generic payload will be interoperable with any other model that uses the same transport interface, provided that those models respect the semantics of the generic payload.

b) In this case, it is strongly recommended that any generic payload extensions should be ignorable. *Ignorable* means that a target or interconnect component shall not fail and shall not generate an error response because of the absence of a given extension, and that the component shall perform its primary function in the same way regardless of whether the given extension is present or absent.

c) If an extension is deemed 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) 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. In the end, the definition of *ignorable* comes down to a matter of judgement.

e) 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.

f) 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.

g) Note that there are two separate transport interfaces, blocking and non-blocking, and that interoperability between those interfaces depends on the coding style chosen and may require adapters.

9.2.2 Define a new protocol types class containing a typedef for tlm_generic_payload

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

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

c) 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, but the user should consider the semantics of the extended generic payload very carefully.

d) There are two recommended patterns of use, outlined below.

e) The first pattern is to use the new protocol types class throughout the initiator, interconnect and target. This pattern supports strong compile-time type checking when binding sockets.

f) The second pattern is to pass the generic payload transaction object through a series of intiator-to-target socket connections where the sockets at the extreme ends of the path use the new protocol types class, but some of the intervening sockets use **tlm_generic_payload_types**.

g) When passing a generic payload transaction between sockets parameterized with different protocol types 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 generic payload components 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.

9.2.3 Define a new protocol types class and a new transaction type

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

b) A new protocol types 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.

9.3 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, return 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.

9.4 Class definition

```
namespace tlm {

class tlm_extension_base
{
public:
   virtual tlm_extension_base* clone() const = 0;
   virtual ~tlm_extension_base() {}
};

template <typename T>
class tlm_extension : public tlm_extension_base
{
public:
   virtual tlm_extension_base* clone() 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
};

class tlm_generic_payload {
public:
    // Constructors, assignment, destructor
    tlm_generic_payload();
    tlm_generic_payload( const tlm_generic_payload& );
    tlm_generic_payload& operator= ( const tlm_generic_payload& );
    tlm_generic_payload* deep_copy() const;
    virtual ~tlm_generic_payload();

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

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

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

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

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

    inline bool* get_byte_enable_ptr() const;
    inline void set_byte_enable_ptr( bool* );
    inline unsigned int get_byte_enable_length() const;
    inline void set_byte_enable_length( const unsigned int );
```



```
  // DMI hint
  void set_dmi_allowed( bool );
  bool get_dmi_allowed() const;

  inline tlm_response_status get_response_status() const;
  inline void set_response_status( const tlm_response_status );
  inline std::string get_response_string();
  inline bool is_response_ok();
  inline 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> void get_extension( T*& ) const;
  tlm_extension_base* get_extension( unsigned int ) const;

  template <typename T> void clear_extension( const T* );
  void clear_extension( unsigned int );
  void resize_extensions();
};

} // namespace tlm
```

9.5 Generic payload memory management

a) The initiator shall be responsible for memory allocation and memory management for the transaction object. This could be static, automatic (stack) or dynamically allocated (new) storage. The initiator may create a new object for each transaction instance, or may implement a memory pooling strategy to reuse transaction objects.

b) The initiator shall not delete the transaction object until the lifetime of the transaction is complete. The initiator may determine when the transaction is complete using the transaction phase, which depends on the coding style used.

c) 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. Similarly, the initiator shall be responsible for constructing any extension objects, and for deleting or pooling those objects when the transaction is complete.

d) The **tlm_generic_payload** class has a shallow copy constructor and shallow assignment operator such that the copy shares the same data array, byte enable array and extensions as the original. The class also has a **deep_copy** method. When taking a deep copy, the caller is obliged to act as the initiator of the copy, meaning that the caller must take responsibility for the memory management of the data and byte enable arrays and extensions copied to the new transaction.

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

f) Also see clause 9.19.3 Management of extensions, rules for safe use, and bridges

9.6 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 copy constructor shall make a shallow copy of the object, that is, it shall copy the data pointer, the byte enable pointer, and the array of pointers to extensions, but not the data, byte enables or extension objects themselves.

c) The assignment operator shall be shallow in the same sense as the copy constructor, that is, after the assignment the attributes of the l-value shall be equal to the attribute of the r-value, including data pointer, byte enable pointer, and the array of pointers to extensions.

d) The **deep_copy** method shall make a complete and deep copy of the object, including the data array, byte enable array, and extension objects, such that the copy can survive the destruction of the original transaction object with no visible side-effects. **deep_copy** shall construct a new object and shall return a pointer to that object. **deep_copy** shall call the clone method of class **tlm_extension** to clone each extension object. The **deep_copy** method may be used to make a new copy of a transaction for analysis beyond the lifetime of the transaction. The **deep_copy** method may also be required when creating a bridge between two different protocol types each based on the generic payload.

e) The implementation of the virtual destructor in class **tlm_generic_payload** shall not delete the data array, the byte enable array, or any extension objects. Typically, the destructor is not expected to be overridden.

9.7 Default values and modifiability of attributes

The default values and modifiability of the generic payload attributes 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 array	-	No	**Yes** (read)
Data length	1	No	No
Byte enable pointer	0	No	No
Byte enable array	-	No	No
Byte enable length	1	No	No
Streaming width	0	No	No
DMI allowed	false	**Yes**	**Yes**
Response status	TLM_INCOMPLETE_RESPONSE	No	**Yes**
Extension pointers	0	**Yes**	**Yes**

9.8 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.

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 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) On receipt of a generic payload transaction with the command attribute equal to TLM_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.

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

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

9.9 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 address of the first byte to be read from or written to the local array in the target. This always correspond to the first byte in the array pointed to by the data pointer attribute.

c) A byte at a given index in the data array shall be transferred to or from the address given by the formula **address_attribute + (array_index % streaming_width)**

d) 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.

e) 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).

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

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

c) 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.

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

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

f) The storage for the data array shall be allocated by the initiator. 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.

g) 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

h) For a read command, the contents of the data array shall be overwritten by the target (honoring the semantics of the byte enable) but by no other component.

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

9.11 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 classes **tlm_initiator_socket** and **tlm_target_socket**, 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 1.

9.12 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) 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.

c) The byte enable mask may be defined by a small pattern applied repeatedly or by a large pattern covering the whole data array. See clause 9.13 Byte enable length attribute

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

e) 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.

f) 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 the contents of the byte enable array shall not be overwritten by any interconnect component or target.

g) 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.

h) 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.

i) 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.

j) 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.

k) 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.

l) 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 protocol types class. See clause 9.2.2 Define a new protocol types class containing a **typedef** for **tlm_generic_payload**

m) In the case of the read command, instead of using byte enables, the target may be able to copy a contiguous block of bytes from the local array to the transaction data array, then mask the data later. This would place the responsibility for implementing the byte enables with the initiator instead of the target, and may or may not be safe because it causes an area of storage owned by the initiator to be entirely overwritten.

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

9.13 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 bytes 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.

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 1.

9.14 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. The streaming width attribute shall determine the width of the stream, that is, the number of bytes transferred on each beat.

d) The address to or from which each byte is being copied in the target shall be reset to the value of the address attribute at the start of each beat.

e) With respect to the interpretation of the data array, a transaction with a non-zero 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 effectively steps down the original data array maintaining the sequence of bytes.

f) A streaming width of 0 shall be interpreted in the same way as a streaming width greater than or equal to the value of the data length attribute, that is, the data transfer is a normal transfer, not a streaming transfer.

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 clause 9.15 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 default value of the streaming width attribute shall be 0.

9.15 Endianness

When using the generic payload to transfer data between initiator and target, both the endianness of the host machine and the endianness of the initiator and target being modeled are relevant. This clause defines rules to ensure interoperability between initiators and targets using the generic payload, so is specifically concerned with the organisation 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) 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 classes **tlm_initiator_socket** and **tlm_target_socket**, the contents of the data and byte enable arrays shall be interpreted using the BUSWIDTH template parameter of the socket. The effective word length shall be calculated as (BUSWIDTH + 7)/8 bytes, and in the following rules is denoted as **W**.

c) The order of the bytes within the data array shall use host endianness. That is, on a little-endian host processor, **data**[0] shall be the least significant byte of a word, and on a big-endian host processor, **data**[0] shall be the most significant byte of a word.

d) In other words, using the notation {a,b,c,d} to list the elements of an array in ascending 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 {LSB_0, ..., MSB_0, LSB_1, ..., MSB_1, LSB_2, ...}, and on a big-endian host {MSB_0, ... LSB_0, MSB_1, ... LSB_1, MSB_2, ...}, where the number of byte in each word is given by **W**, the total number of byte is given by the **data_length** attribute, and the effective number of words is approximated by **data_length / W**.

e) 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”.

f) It is strongly recommended that applications should be independent of host endianness. This may require the use of helper functions or conditional compilation.

g) 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.[5]

h) The data length attribute is not required to be an integer multiple of **W**.

i) The address attribute is not required to be an integer multiple of **W**, that is, addresses do not have to be aligned with word boundaries.

j) Transferring whole words aligned to word boundaries simplifies access to the data array. However, the intent is that the complexities of part-word and non-aligned access will be hidden within helper functions.

k) If the data length attribute is not an integer multiple of **W**, there will be fewer bytes in the final word. These bytes may be at the least significant or the most significant end of the word, depending on host endianness. For example, given a big-endian host, **W** = 4 and **data** = {1,2,3,4,5,6}, then MSB_0 = 1, LSB_0 = 4, MSB_1 = 5, LSB_1 = undefined.

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

m) Part-word and non-aligned transfers can always be expressed using integer multiples of **W** together with byte enables. 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, 1, 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, 1, 1, 1, 1, 0, 0}.

n) 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

[5] The helper functions are still under development.

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** = {1,1,1,1,0,0,0,1}.

o) 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.

p) When the endianness chosen for a width conversion matches the host endianness, the width conversion is effectively free, meaning that the transaction can be forwarded from socket-to-socket without modification, or the incoming and outgoing transactions can share the same data and byte enable arrays.

q) If a width conversion is necessary 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 generally necessitate the construction of a new generic payload transaction.

r) 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.

9.16 Atomic Transactions

(The TLM Working Group are considering the inclusion of a mechanism for atomic transactions which could, for example, be used to implement a read-modify-write operation. The details were not finalized in time for inclusion in this document.)

9.17 DMI allowed attribute

a) The method **set_dmi_allowed** shall set the DMI allowed attribute to the value passed as an argument. The method **get_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 clause 4.3.7 Optimization using a DMI Hint

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

9.18 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 clause 9.18.1 The standard error response

e) The response status attribute shall be set to TLM_INCOMPLETE_RESPONSE by the initiator, and may be overwritten by the target. The response status attribute should not be overwritten by any interconnect component, because the default value TLM_INCOMPLETE_RESPONSE indicates that the transaction was not delivered to the target.

f) The target may set the response status attribute to TLM_OK_RESPONSE to indicate that it was able to execute the command successfully, or to one of the five error responses listed in the table below to indicate an error. The target should choose the appropriate error response depending on the cause of the error.

Error response	**Interpretation**
TLM_ADDRESS_ERROR_RESPONSE	Unable to act upon the address attribute, or address out-of-range
TLM_COMMAND_ERROR_RESPONSE	Unable to execute the command
TLM_BURST_ERROR_RESPONSE	Unable to act upon the data length or streaming width
TLM_BYTE_ENABLE_ERROR_RESPONSE	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) It is recommended that the initiator should always check the response status attribute after calling the method of a transport interface. 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.

9.18.1 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 busses, 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 ignore 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_process() {
    tlm::tlm_generic_payload trans;                    // Construct default generic payload
    tlm::tlm_phase phase;
    sc_time delay = SC_ZERO_TIME;

    trans.set_command(tlm::TLM_WRITE_COMMAND);         // A write command
    trans.set_data_length(4);                          // Write 4 bytes

    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'

        phase = tlm::BEGIN_REQ;
        tlm::tlm_sync_enum status = init_socket->nb_transport(trans, phase, delay);

        if (status == tlm::TLM_COMPLETED)              // Check return value of nb_transport
            if (trans.get_response_status() <= 0)      // Check response status in transaction
                SC_REPORT_ERROR("TLM2", trans.get_response_string().c_str());
        ...
}
...

// The target
virtual tlm::tlm_sync_enum nb_transport(
    tlm::tlm_generic_payload& trans, tlm::tlm_phase& phase, 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();
    bool*             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 tlm::TLM_COMPLETED;
    }
    if (byt) {                                  // Target unable to support byte enable attribute
        trans.set_response_status( tlm::TLM_BYTE_ENABLE_ERROR_RESPONSE );
        return tlm::TLM_COMPLETED;
    }
    if (wid) {                                  // Target unable to support streaming width attribute
        trans.set_response_status( tlm::TLM_BURST_ERROR_RESPONSE );
        return tlm::TLM_COMPLETED;
    }

    switch (phase) {
    case tlm::BEGIN_REQ:                        // Execute command
        if ( cmd == tlm::TLM_WRITE_COMMAND )
            memcpy( &m_storage[adr], ptr, len );
        else if ( cmd == tlm::TLM_READ_COMMAND )
            memcpy( ptr, &m_storage[adr], len );
        break;
    case tlm::END_REQ:
    case tlm::BEGIN_RESP:
    case tlm::END_RESP:
        ;
    }
    trans.set_response_status( tlm::TLM_OK_RESPONSE );  // Successful completion
    return tlm::TLM_COMPLETED;
}
...

// Showing generic payload with byte enables

// The initiator
union Byte_enable {
    int  m_int;
    bool m_bool4[4];
};

void thread_process() {
    tlm::tlm_generic_payload trans;
    tlm::tlm_phase phase;

    Byte_enable byte_enable;
    byte_enable.m_int = 0x00010001;         // Uses host-endianness MSB..LSB
```



```
    trans.set_command(tlm::TLM_WRITE_COMMAND);
    trans.set_data_length(4);
    trans.set_byte_enable_ptr( byte_enable.m_bool4 );
    trans.set_byte_enable_length(4);
    ...
...

// The target
virtual tlm::tlm_sync_enum nb_transport(
        tlm::tlm_generic_payload& trans, tlm::tlm_phase& phase, 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();
    bool*              byt   = trans.get_byte_enable_ptr();
    unsigned int       bel   = trans.get_byte_enable_length();
    unsigned int       wid   = trans.get_streaming_width();

    switch (phase) {
    case tlm::BEGIN_REQ:
        if (cmd == tlm::TLM_WRITE_COMMAND) {
            if (byt) {
                for (unsigned int i = 0; i < len; i++)
                    if ( byt[ i % bel ] )                  // Byte enable applied repeatedly up data array
                        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 tlm::TLM_COMPLETED;
            } else
                memcpy(ptr, &m_storage[adr], len);
        }
        break;
        ...
    ...
```


9.19 Extension mechanism

9.19.1 Introduction

The extension mechanism is an integral part of the generic payload, and cannot 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 mandatory. An ignorable extension is an extension that may be ignored by any or all interconnect components or targets that receive the generic payload transaction. The main intent of ignorable extensions is to model auxiliary information, simulation artefacts, side-band information, or meta-data that do not have a direct effect on the functionality of the downstream components. A mandatory extension is an extension that any interconnect component or target receiving the transaction is obliged to inspect and to act upon. The main intent of mandatory extension is for use when specializing the generic payload to model the specific details of a protocol.

9.19.2 Rationale

The rationale behind the extension mechanism is to permit TLM ports or sockets that carry variations on the core attribute set of the generic payload to be specialized with the same transaction type, thus allowing them to be bound together directly with no need for adaption or bridging. Without the extension mechanism, the addition of any new attribute to the generic payload would require the definition of a new protocol class, leading to a new template specialization of the core interface class, which would be type-incompatible with the generic payload and with any other such specialization. The extension mechanism allows minor variations to be introduced into the generic payload without breaking the type compatibility of TLM ports, thus reducing the amount of coding work that needs to be done to connect ports that carry slightly different information.

9.19.3 Management of extensions, rules for safe use, and 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. The extension objects themselves are usually managed by the initiator of the transaction, which is responsible for creating the extensions, adding the extensions to a transaction, and deleting or pooling the extensions when the transaction is complete. Every generic payload object is capable of carrying 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. Hence, a set of guidelines for safe use are given below. The user should consider the ownership and deletion of extension objects very carefully.

a) The general principle is that whichever process constructs the extension object and sets the pointer in the generic payload should also clear the pointer and delete the extension object.

b) The recommended approach is to have the initiator, and only the initiator, call **set_extension**. If an interconnect component or target wishes to pass a value back to the initiator using an extension, the initiator should construct and add the extension. Moreover, for maximum safety, the initiator should not

assume that an extension pointer is still valid on completion of a transaction, so when pooling transactions, the initiator should call **set_extension** before each transport call.

c) An interconnect component may be justified in calling **set_extension** if the author does not have access to the source code of the initiator. In this case, the interconnect component should not overwrite any existing extension pointer, and should remove the extension before the completion of the transaction.

d) If an interconnect component does indeed overwrite an existing extension pointer (contrary to recommended practice) when passing a transaction forward toward the target, that same interconnect component should restore the original extension pointer before passing the transaction back toward the initiator.

e) When creating a 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.) A bridge may perform a deep copy of the extension objects, or a shallow copy of the extension array. In the latter case, the extension objects are still owned by the initiator of the incoming transaction. If the bridge calls **set_extension** to add further extensions to the outgoing transaction, those extensions would be owned by the bridge.

f) See clause 9.5 Generic payload memory management

9.19.4 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, it is recommended that any interconnect or target component should be free to ignore the given extension, but this cannot and should not be enforced by the implementation. Having an interconnect or target component generate a standard error response because of the absence of an extension is possible, but is not recommended practice.

d) In the case of an ignorable extension, it is recommended that the presence or absence of a given extension should have no effect on the primary functionality of any component, but may, for example, have an effect on diagnostic reporting, debug, or optimization.

e) There is no built-in mechanism to enforce the presence of a given extension.

f) The semantics of each extension are application-defined. There are no pre-defined extensions.

g) 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.

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 by the **deep_copy** method of the generic payload. It shall create a complete and deep copy of any extension object such that the copy can survive the destruction of the original object with no visible side-effects.

i) 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.

j) 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.

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

l) 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).

m) The methods **set_extension**, **get_extension**, and **clear_extension** each have two definitions, one a function template, and the other with an ID argument. The function template determines which extension to set, get or clear using the template argument. The non-templated function determines which extension to set, get, or clear using the ID argument.

n) The function template forms of these three methods should generally be used. The functions with an ID argument are intended for low-level programming such as when cloning a generic payload object.

o) The member function **template<typename T> T* 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.

p) The member function **tlm_extension_base* 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.

q) 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.

r) The member function **template<typename T> void get_extension(T*&)** shall return a pointer to the extension object of the given type, if it exists, or a null pointer if it does not exist. The argument 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.

s) The member function **tlm_extension_base* 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.

t) The member function **template<typename T> void clear_extension(const T*)** shall remove the extension given by the argument from the generic payload object. The argument shall be a pointer to an object of a type derived from **tlm_extension**.

u) The member function **void clear_extension(unsigned int)** shall remove from the generic payload object the extension at the array index given by the argument. The given index shall have been registered as an extension ID, otherwise the behavior of the function is undefined.

v) 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.

w) 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>
{
    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;
    }
    ID_extension() : transaction_id(0) {}
    unsigned int transaction_id;                         // An ignorable attribute
};

// The initiator
struct Initiator: sc_module, tlm::tlm_bw_nb_transport_if<>
{   ...
    void thread_process() {
        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
            ...
            tlm::tlm_sync_enum status = init_socket->nb_transport(trans, phase, delay);
            ...
        ...
```



```
// The target
virtual tlm::tlm_sync_enum nb_transport(
    tlm::tlm_generic_payload& trans, tlm::tlm_phase& phase, 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("TLM2", txt);
    }
    ...

// Showing a new protocol types class based on an extended generic payload

// User-defined extension class
struct  Incr_cmd_extension: tlm::tlm_extension<Incr_cmd_extension>
{
    virtual tlm_extension_base* clone() const {
        Incr_cmd_extension* t = new Incr_cmd_extension;
        t->incr_cmd = this->incr_cmd;
        return t;
    }
    Incr_cmd_extension() : incr_cmd(false) {}
    bool incr_cmd;
};

struct incr_payload_types               // User-defined protocol types class using the generic payload
{
    typedef  tlm::tlm_generic_payload tlm_payload_type;
    typedef  tlm::tlm_phase               tlm_phase_type;
    typedef  tlm::tlm_dmi_mode            tlm_dmi_mode_type;
};

struct  Initiator: sc_module, tlm::tlm_bw_nb_transport_if<incr_payload_types>
{
    tlm::tlm_nb_initiator_socket<32, incr_payload_types > init_socket;
    ...
    void thread_process() {
        tlm::tlm_generic_payload trans;
        Incr_cmd_extension* incr_cmd_extension = new Incr_cmd_extension;
        trans.set_extension( incr_cmd_extension );                  // Add the extension to the transaction
        ...
        trans.set_command( tlm::TLM_WRITE_COMMAND );     // Execute a write command
        tlm::tlm_sync_enum status = init_socket->nb_transport(trans, phase, delay);
        ...
```



```
        trans.set_command( tlm::TLM_IGNORE_COMMAND );
        incr_cmd_extension->incr_cmd = true;                    // Execute an increment command
        tlm::tlm_sync_enum status = init_socket->nb_transport(trans, phase, delay);
        ...
    ...

// The target
tlm::tlm_nb_target_socket<32, incr_payload_types > targ_socket;

 virtual tlm::tlm_sync_enum nb_transport(
    tlm::tlm_generic_payload& trans, tlm::tlm_phase& phase, sc_core::sc_time& t)
{
    tlm::tlm_command cmd = trans.get_command();
    ...
    Incr_cmd_extension* incr_cmd_extension;
    trans.get_extension( incr_cmd_extension );              // Retrieve the extension
    if (incr_cmd_extension->incr_cmd) {                     // Assume the extension exists
        if (cmd != tlm::TLM_IGNORE_COMMAND) {               // Detect clash with read or write
            trans.set_response_status( tlm::TLM_GENERIC_ERROR_RESPONSE );
            return tlm::TLM_COMPLETED;
        }
        ++ m_storage[adr];                                  // Execute an increment command
    }
    ...
```


10 TLM1 legacy

The following core interfaces and **tlm_fifo** channel from TLM1 are still part of the TLM2.0-draft-2 standard, but are not documented in detail here.

10.1 TLM1 core interfaces

The transport method with the signature **transport(const REQ& , RSP&)** was not part of TLM1.0, but has been added in TLM2.0.

```
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 TLM1 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;
};
```


11 Glossary

Blue = taken from the SystemC LRM

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. Typically, an adapter is used to convert between two transaction-level interfaces of different types. See *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).

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.

bidirectional interface: A TLM 1.0 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 TLM2 standard which contains a single method **b_transport**. Confusingly, there is also still a blocking transport interface left as a legacy from TLM1 which contains a method named **transport**.

bridge: A module that connects together two similar or dissimilar transaction-level interfaces, each representing a memory-mapped bus or other protocol, usually at the same abstraction level. A bus bridge is a device that connects two similar or dissimilar buses together. A communication bridge is a device that connects network segments on the data link layer of a network. In TLM2, a bridge is a component that acts as a target for an incoming transaction and an initiator for an outgoing transaction. See *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.

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 transaction 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)

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 level protocol which may be used to achieve a degree of interoperability between untimed, loosely timed and approximately timed models for components communicating over a memory-mapped bus. The same transaction class is used for all modeling styles.

generic payload API: The class interface through which the generic payload is accessed. The generic payload API is distinct from the core interfaces.

global quantum: The default time quantum. The intent is that all temporally decoupled initiators should typically synchronize on integer multiples of the global quantum, or more frequently on demand.

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 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 1.0 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 also overloads the SystemC binding operators to bind both port and export.

interconnect component: A module that accesses a transaction object, but does 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.

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 TLM1 interface.

local quantum: The amount of simulation time remaining in the current quantum before the initiator is required to synchronize. Typically, the local quantum is the local time offset subtracted from the global quantum.

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.

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)

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 TLM2 standard which contains a single method **nb_transport**.

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)

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: The period in the lifetime of a transaction occurring between successive timing points. The phase is passed as an argument to the non-blocking transport method.

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 types class: A class containing a **typedef** for the type of the transaction object, the phase and the DMI mode, used to parameterize the combined interfaces, and effectively defining a unique type for a protocol.

quantum: In temporal decoupling, the amount a process is permitted to run ahead of the current simulation time.

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.

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: An indication from the **nb_transport** method back to its caller that it was unwilling or unable to fulfil a request to effectively execute a transaction at a future time (temporal decoupling), and therefore that the caller must yield control back to the SystemC scheduler so that simulation time may advance and other processes run.

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 1.0 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 point: A point in time at which the processes that are interacting through a transaction either transfer control or are synchronized. Certain timing points are implemented as function calls or returns, others as event notifications. Timing points mark the boundaries between the phases of a transaction. Consecutive timing points could occur in different delta cycles at the same simulation time.

TLM1: The first major version of the OSCI Transaction Level Modeling standard. TLM1.0 was released in 2005.

TLM2: The second major version of the OSCI Transaction Level Modeling standard. This document describes TLM2.0-draft-2.

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 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*.

transport interface: The one and only bidirectional core interface in TLM1. The transport interface passes a request transaction object from caller to callee, and returns a response transaction object from callee to caller. TLM2 adds separate blocking and non-blocking transport interfaces.

unidirectional interface: A TLM 1 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.

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.

