May 2008

OSCI TLM-2 USER MANUAL

Software version: TLM-2.0 Final

Document version: DRAFT_JA13

Contributors

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

Bart Vanthournout, CoWare, TLM-WG Chair
James Aldis, Texas Instruments, 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 TLM-2.0-final:

Rob ??????, Virtutech
Tom Aernoudt, CoWare
James Aldis, Texas Instruments
John Aynsley, Doulos
Guillaume Audeon, ARM
Geoff Barrett, Broadcom
Bill Bunton, ESLX
Mark Burton, GreenSocs
Jerome Cornet, ST Microelectronics
Jack Donovan, ESLX
Jakob Engblom, Virtutech
Robert Guenzel, GreenSocs
Tim Kogel, CoWare
Laurent Maillet-Contoz, ST Microelectronics
Kiyoshi Makino, Mentor Graphics
Marcelo Montoreano, Synopsys
Victor Reyes, NXP
Olaf Scheufen, Synopsys
Bart Vanthournout, CoWare
Kaz Yoshinaga, Starc
Trevor Wieman, Intel
Charles Wilson, ESLX

The following is a list of active technical participants in the OSCI TLM Working Group at the time of the release of TLM-2.0-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 Graphics
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

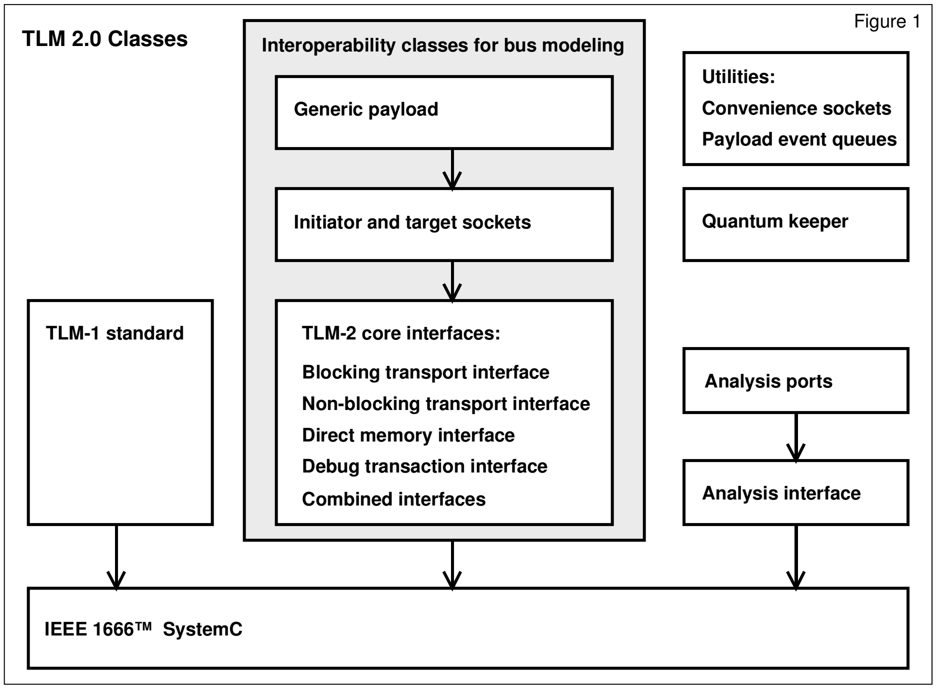
1 Overview

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

TLM-2.0 consists of a set of core interfaces, initiator and target sockets, the generic payload, utilities, the analysis interfaces and ports, and the TLM-1.0 core interfaces. The TLM-2 core interfaces consist of the blocking and non-blocking transport interfaces, the direct memory interface (DMI), and the debug transaction interface. The core interfaces introduced in TLM-2 support 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 TLM-2 classes are layered on top of the SystemC class library as shown in the diagram below. For maximum interoperability, and particularly for memory-mapped bus modeling, it is recommended that the TLM-2 core interfaces, sockets and generic payload be used together in concert. In cases where the generic payload is inappropriate, it is possible for the core interfaces and the initiator and target sockets, or the core interfaces alone, to be used with an alternative transaction type. It is even technically possible for the generic payload to be used directly with the core interfaces without the initiator and target sockets, although this approach is not recommended.

It is not strictly necessary to use the utilities, quantum keeper, analysis interfaces and analysis ports to achieve interoperability between bus models. Nonetheless, these classes are documented and maintained as part of the TLM-2.0 standard.



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

1.1 Scope

This document describes the contents of the TLM-2.0 standard. The main focus of this document is the key concepts and semantics of the TLM-2 core interfaces and classes. It does not describe all the supporting code, examples, and unit test. It lists the TLM-1 core interfaces, but does not define their semantics. 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 TLM-2.0

1.2 Source code and documentation

The TLM-2.0-final 2 release has a hierarchical directory structure as follows:

include	The C++ source code of the TLM-2 standard, with readme files and release notes
include/tlm_h/tlm_req_rsp	The TLM-1 standard
include/tlm_h/tlm_req_rsp/tlm_1_interfaces	TLM-1 core interfaces
include/tlm_h/tlm_req_rsp/tlm_channels	TLM-1 fifo and req-rsp channels
include/tlm_h/tlm_req_rsp/tlm_ports	TLM-1 non-blocking ports with event finders
include/tlm_h/tlm_req_rsp/tlm_adapters	TLM-1 slave-to-transport & transport-to-master adapters
include/tlm_h/tlm_trans	TLM-2 interoperability classes
include/tlm_h/tlm_trans/tlm_2_interfaces	TLM-2 core interfaces
include/tlm_h/tlm_trans/tlm_generic_payload	TLM-2 generic payload
include/tlm_h/tlm_trans/tlm_sockets	TLM-2 sockets
include/tlm_h/tlm_quantum	TLM-2 quantum keeper
include/tlm_h/tlm_analysis	TLM-2 analysis interface and ports
utilities	TLM-2 standard utility classes not essential for interoperability
docs	Documentation, including User Manual, white papers, and Doxygen
examples	A set of application-oriented examples with their own documentation
unit_test	A set of regression tests

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

2 References

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

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

IEEE Std 1666-2005, SystemC Language Reference Manual

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

2.1 Bibliography

The following books provide useful background information:

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

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

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

3 Introduction

3.1 Background

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

a) TLM-1 has no standard transaction class, so each application has to create its own non-standard classes, resulting in very poor interoperability between models from different sources. TLM-2 addresses this shortcoming with the generic payload.

b) TLM-1 has no support for timing annotation, so no standard way of communicating timing information between models. TLM-1 models would typically implement delays by calling **wait**, which slows down simulation. TLM-2 addresses this shortcoming with the addition of timing annotation to the blocking and non-blocking transport interface.

c) The TLM-1 interfaces require all transaction objects and data to be passed by value or const reference, which slows down simulation. Some applications work around this restriction by embedded pointers in transaction objects, but this is non-standard and non-interoperable. TLM-2 addresses this shortcoming with transaction objects whose lifetime extends across several transport calls, supported by a new transport interface.

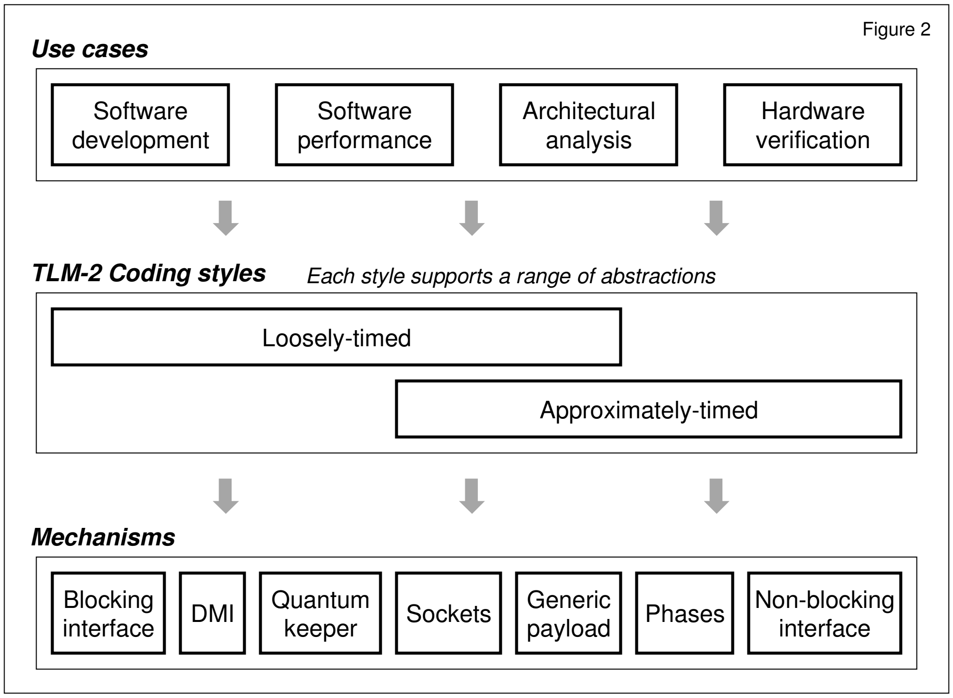
3.2 Transaction-level modeling, use cases and abstraction

There has been a longstanding discussion in the ESL community concerning what is the most appropriate taxonomy of abstraction levels for transaction level modeling. Models have been categorized according to a range of criteria, including granularity of time, frequency of model evaluation, functional abstraction, communication abstraction, and use cases. The TLM-2 activity explicitly recognizes the existence of a variety of use cases for transaction-level modeling (see the Requirements Specification for TLM-2.0), but rather than defining an abstraction level around each use case, TLM-2 takes the approach of distinguishing between interfaces (APIs) on the one hand, and coding styles on the other. The TLM-2 standard defines a set of interfaces which should be thought of as low-level programming mechanisms for implementing transaction-level models, then describes a number of coding styles that are appropriate for, but not locked to, the various use cases.

The definitions of the standard TLM-2 interfaces stand apart from the descriptions of the coding styles. It is the TLM-2 interfaces which form the normative part of the standard and ensure interoperability. Each coding style can support a range of abstraction across functionality, timing and communication. In principle users can create their own coding styles, although this is not encouraged.

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

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



A more detailed transaction-level model may need to associate multiple protocol-specific timing points with each transaction, such as timing points to mark the start and the end of each phase of the protocol. By choosing an appropriate number of timing points, it is possible to model communication to a high degree of timing accuracy without the need to execute the component models on every single clock cycle. In this standard, such a coding style is called *approximately-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. TLM-2 recognizes several coding styles which should be used as a guide to model writing. For simplicity and clarity, this document restricts itself to elaborating two specific named coding styles; *loosely-timed* and *approximately-timed*. By their nature the coding styles are not precisely defined, and the rules governing the TLM-2 core interfaces are defined independently from these coding styles. In principle, it would be possible to define other coding styles based on the TLM-1 and TLM-2 mechanisms.

3.3.1 Untimed coding style

TLM-2 does not make explicit provision for an untimed coding style, because all contemporary bus-based systems require some notion of time in order to model software running on one or more embedded processors. However, untimed modeling is supported by the TLM-1 core interfaces.

3.3.2 Loosely-timed coding style and temporal decoupling

The loosely-timed coding style makes use of the blocking transport interface. This interface allows only two timing points to be associated with each transaction, corresponding to (but not necessarily simultaneous with) the call to and return from the blocking transport function. In the case of the generic payload, these two timing points mark the beginning of the request phase and the beginning of the response phase. 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 coarse-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 TLM-2, 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 TLM-2 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 TLM-2, temporal decoupling is supported by the **tlm_quantum_keeper** class and the timing annotation of the blocking and non-blocking transport interface.

3.3.3 Synchronization in loosely-timed models

An untimed model relies on the presence of explicit synchronization points in order to pass control between initiators at predetermined points during execution. A loosely-timed model can also benefit from explicit synchronization in order to guarantee deterministic execution, but a loosely-timed model is able to make progress even in the 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.

3.3.4 Approximately-timed coding style

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

For approximately-timed modeling, a transaction is broken down into multiple phases, with an explicit timing point marking the transition between phases. In the case of the unextended 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.

It is possible to use the non-blocking transport interface for loosely-timed modeling by restricting the number of phases to two, although the blocking transport interface is generally preferred because of its simplicity.

The approximately-timed coding style cannot generally exploit temporal decoupling because of the need for timing accuracy. Instead, each process typically executes in lock step with the SystemC scheduler. Process interactions are annotated with specific delays. To create an approximately-timed model, it is generally sufficient to annotate 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 loosely-timed and approximately-timed coding styles

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

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

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

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

3.3.6 Switching between loosely-timed and approximately-timed modeling[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.

3.3.7 Cycle-accurate modeling

Cycle-accurate modeling is beyond the scope of TLM-2 at present. It is possible to create cycle-accurate models using SystemC and TLM-1 as it stands, but the requirement for the standardization of a cycle-accurate coding style still remains an open issue, possibly to be addressed by a future OSCI standard.

In principle, 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 TLM-2.0 release includes sufficient machinery for this, but the details have not been worked out.

3.3.8 Blocking versus non-blocking transport interfaces

The blocking and non-blocking transport interfaces are distinct interfaces that exist in TLM-2 to support different levels of timing detail. The blocking transport interface is only able to model the start and end of a transaction, with the transaction being completed within a single function call. The non-blocking transport interface allows a transaction to be broken down into multiple timing points, and typically requires multiple function calls for a single transaction.

For interoperability, the blocking and non-blocking transport interfaces are combined into a single interface. A model that initiates transactions may used the blocking or non-blocking transport interfaces (or both) according to coding style. Any model that provides a TLM-2 transport interface is obliged to provide both the blocking and non-blocking forms for maximal interoperability, although such a model is not obliged to implement both interfaces internally.

TLM-2 provides a mechanism (the so-called *convenience socket*) to automatically convert incoming blocking or non-blocking transport calls to non-blocking or blocking transport calls, respectively. Converting transport call types does incur some cost, particularly converting an incoming non-blocking call to a blocking implementation. However, the cost overhead is mitigated by the fact that any approximately-timed model is likely to dominate simulation time anyway. The existence of even a single approximately-timed model is likely to wipe out the speed benefit to be gained from using exclusively loosely-timed models.

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

The strength of the blocking transport interface is that it allows a simplified coding style for models that are able to complete a transaction in a single function call. The strength of the non-blocking transport interface is

[1] The details of LT/AT switching are to be added before the final release of TLM-2.0

that it supports the association of multiple timing points with a single transaction. In principle, either interface supports temporal decoupling, but the speed benefits of temporal decoupling are likely to be nullified by the presence of multiple timing points for approximately-timed models.

3.3.9 Use cases and coding styles

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

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

3.4 Initiators, targets, sockets, and bridges

The TLM-2 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.

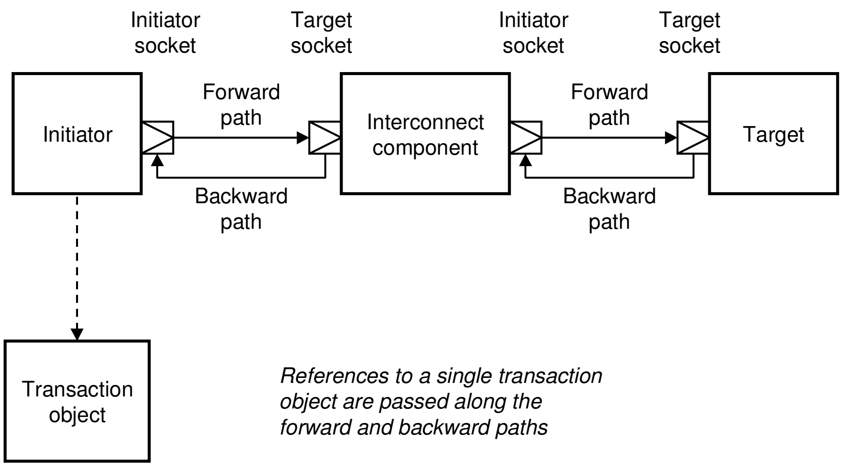


Figure 3

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

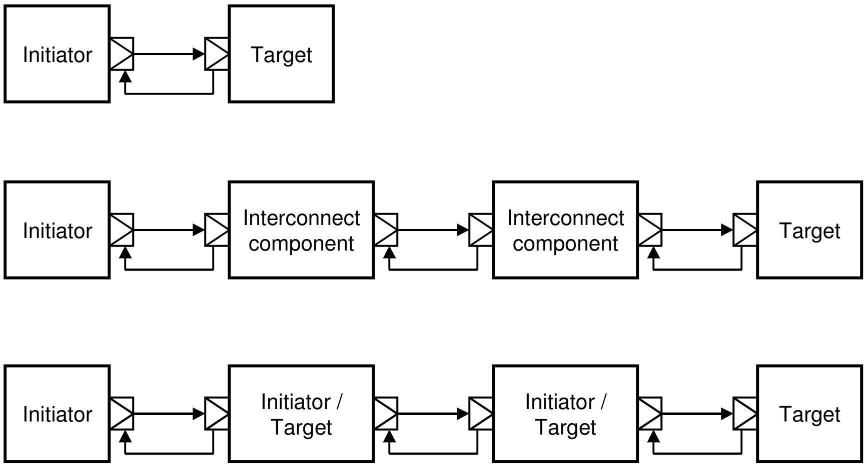


Figure 4

classes overload the SystemC port binding operator to implicitly bind both forward and backward paths.

As well as the transport interfaces, the sockets also encapsulate the DMI and debug 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 TLM-2 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 TLM-2 transactions. An interconnect component would pass on a pointer to a single transaction object, which is the best approach for simulation speed. A transaction bridge would require the transaction object to be copied, which gives much more flexibility because the two transactions could have different attributes.

The use of TLM-2 sockets are recommended for maximal interoperability, convenience, and a consistent coding style. Whilst it is possible for components to use SystemC ports and exports directly with the TLM-2 core interfaces, this is not recommended.

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 a loosely-timed simulation, whereas the debug transaction interface is for debug access free of the delays or side-effects associated with regular transactions.

The DMI has both forward (initiator-to-target) and backward (target-to-initiator) interfaces, whereas debug only has a forward interface.

3.6 Combined interfaces and sockets

The blocking and non-blocking transport interfaces are combined with the DMI and debug transaction interfaces in the standard initiator and target sockets. All four interfaces (the two transport interfaces, DMI, and debug) can be used in parallel to access a given target (subject to the rules described in this standard). These combined interfaces are one of the keys to ensuring interoperability between components using the TLM-2 standard, the other key being the generic payload.

The standard target sockets provide all four interfaces, so each target component must effectively implement the methods of all four interfaces. However, the design of the blocking and non-blocking transport interfaces together with the provision of convenience sockets to convert between the two means that a given target need only implement either the blocking or the non-blocking transport method, not both, according to the speed and accuracy requirements of the model.

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

3.7 Namespaces

With the exception of the utilities, all of the TLM-2 classes are declared in a single top-level C++ namespace **tlm**. Particular implementations of the TLM-2 classes may choose to nest further namespaces within namespace **tlm**, but such nested namespaces shall not be used in applications.

3.8 Header files

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

4 TLM-2 core interfaces

In addition to the core interfaces from TLM-1, TLM-2 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 TLM-2 blocking transport interface is intended to support the loosely-timed coding style. The blocking transport interface is appropriate where an initiator wishes to complete a transaction with a target during the course of a single function call, the only timing points of interest being those that mark the start and the end of the transaction.

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

This blocking transport interface has deliberate similarities with the transport interface from TLM-1, which is still part of the TLM-2 standard, but the TLM-1 transport interface and the TLM-2 blocking transport interface are not identical. In particular, the new **b_transport** method has a single transaction argument passed by non-const reference and a second argument to annotate timing, whereas the TLM-1 **transport** method takes a request as a single const reference request argument, has no timing annotation, and returns a response by value. TLM-1 assumes separate request and response objects passed by value (or const reference), whereas TLM-2 assumes a single transaction object passed by reference, whether using the blocking or the non-blocking TLM-2 interfaces.

The **b_transport** method has an argument to add a timing annotation to the transaction. This argument is used on both the call to and the return from **b_transport** to indicate the time of the start and end of the transaction, respectively, relative to the current simulation time. Although blocking transport does not have an explicit phase argument, the start of the transaction is equivalent to the transition to the BEGIN_REQ phase and the end of the transaction is equivalent to the transition to the BEGIN_RESP phase of the unextended generic payload.

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

4.1.2 Migration path from TLM-1

The old TLM-1 and the new TLM-2 interfaces are both part of the TLM-2 standard. The TLM-1 blocking and non-blocking interfaces are still useful in their own right. For example, a number of vendors have used these interfaces in building functional verification environments for HDL designs.

The intent is that the similarity between the old and new blocking transport interfaces should ease the task of building adapters between legacy models using the TLM-1 interfaces and the new TLM-2 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, sc_core::sc_time& t) = 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 **b_transport** method may call **wait**, directly or indirectly.

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

c) 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.

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

e) 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. In the case of the unextended generic payload, these timing points correspond to the transition to the BEGIN_REQ phase and the transition to the BEGIN_RESP phase, respectively.

f) The timing points may be offset from the simulation time of the function call and return using timing annotation.

g) Timing annotation??????????????????

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

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

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

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

l) **b_transport** may make one or several calls to **nb_transport**. It is straightforward to create an adapter between an initiator that calls **b_transport** and a target that implements **nb_transport**.

4.2 Non-blocking transport interface

4.2.1 Introduction

The new non-blocking transport interface is intended to support the approximately-timed coding style. The non-blocking transport interface is appropriate where it is desired to model the detailed sequence of interactions between initiator and target during the course of a each transaction. In other words, to break down a transaction into multiple phases, where each phase transition marks an explicit timing point.

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

The non-blocking transport interface uses both the forward path from initiator to target and the backward path from target to initiator.

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

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

4.2.2 Class definition

```
namespace tlm {

enum tlm_phase_enum {
```

```
            UNINITIALIZED_PHASE=0,  BEGIN_REQ=1,  END_REQ,  BEGIN_RESP,  END_RESP };

enum tlm_sync_enum { TLM_ACCEPTED,  TLM_UPDATED,  TLM_COMPLETED };

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

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

} // namespace tlm
```

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

4.2.4 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 transaction type **tlm_generic_payload** and the default phase type **tlm_phase**. 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_fw and nb_transport_bw call

a) There are two non-blocking transport methods, **nb_transport_fw** for use on the forward path, and **nb_transport_bw** for use on the backward path. Aside from their names and calling direction these two methods have similar semantics, and are henceforth described as one within this clause using the abbreviated term **nb_transport**.

b) In the case of the unextended generic payload, the forward and backward paths should pass through exactly the same sequence of components and sockets in opposing order.

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

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

e) 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.

f) 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

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

h) 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.

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

j) An **nb_transport_fw** call on the forward path shall under no circumstances directly or indirectly make a call to **nb_transport_bw** on the 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) 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 **Error! Reference source not found. Error! Reference source not found.**

d) 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 **Error! Reference source not found. Error! Reference source not found.**

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 TLM-2 kit includes a payload event queue class **tlm_peq** which may be used for this purpose. See clause **Error! Reference source not found. Error! Reference source not found.**

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 **Error! Reference source not found. Error! Reference source not found.**.

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_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.

g) **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).

h) **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_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

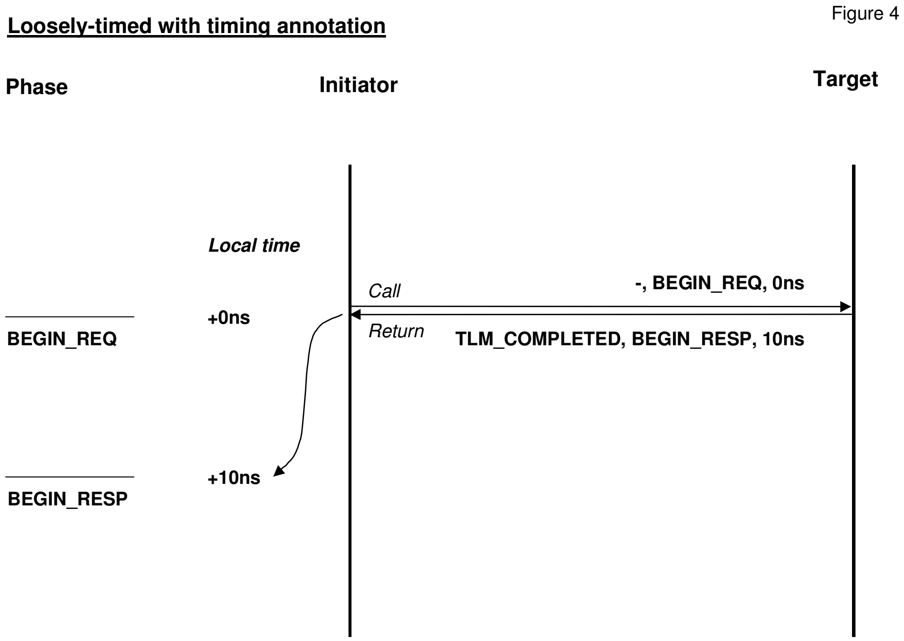
The permitted sequence of timing points and **nb_transport** calls for the loosely-timed and approximately-timed coding styles are illustrated below with a series of message sequence charts. The arguments and return value passed to and from **nb_transport** are shown using the notation return, phase, 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.



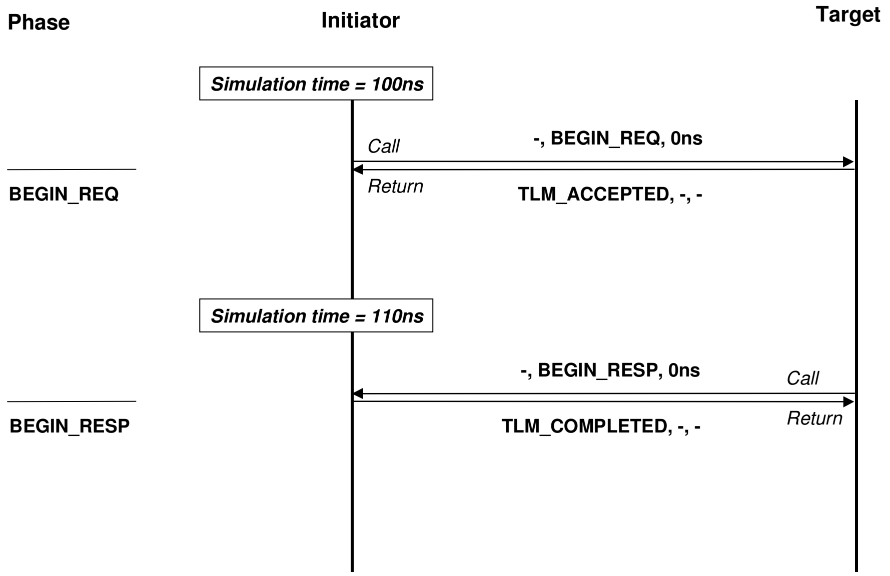
4.2.13.2 Loosely-timed with sync

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

Loosely-timed with sync

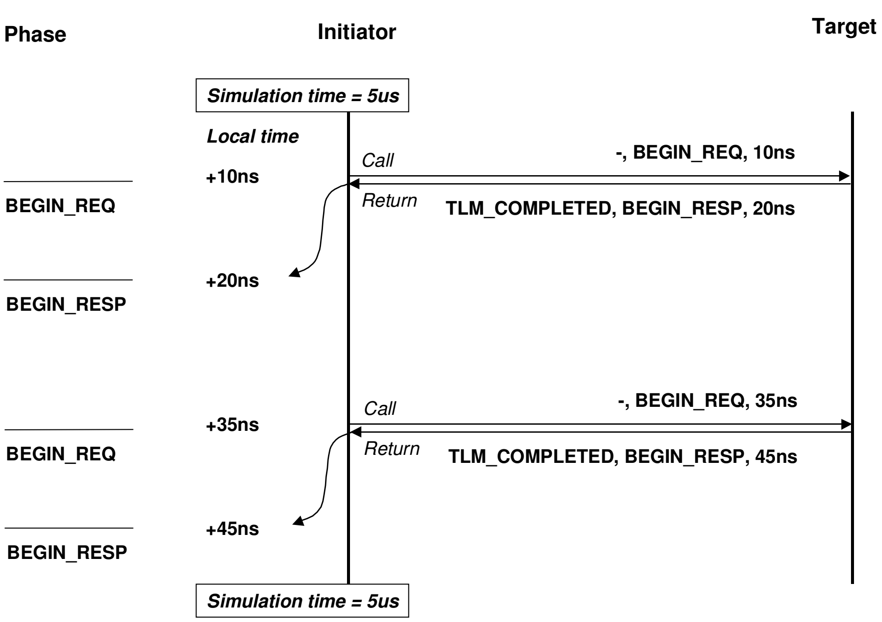


4.2.13.3 Loosely-timed with temporal decoupling

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

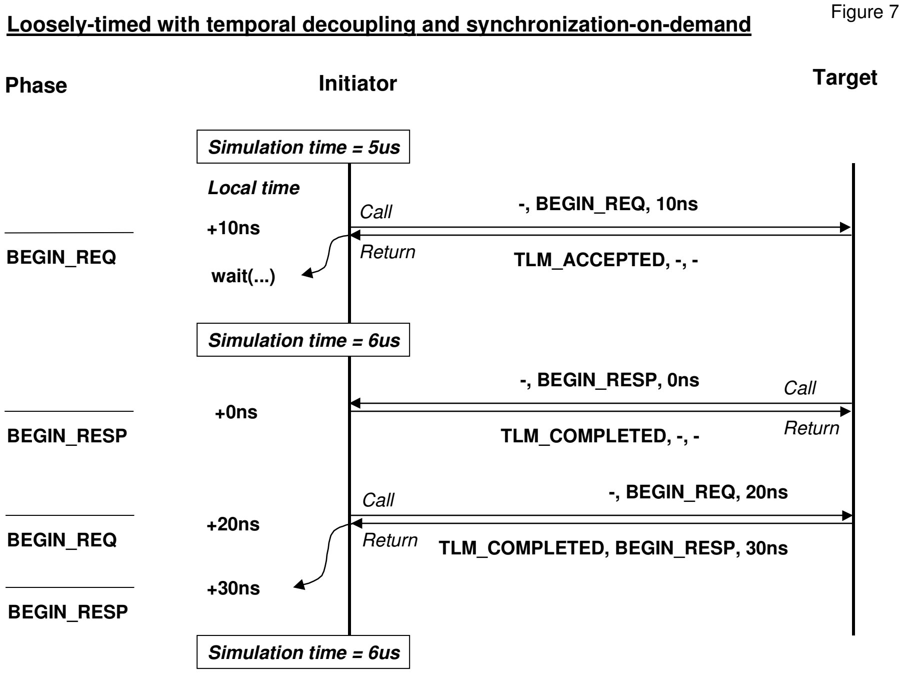
Loosely-timed with temporal decoupling



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



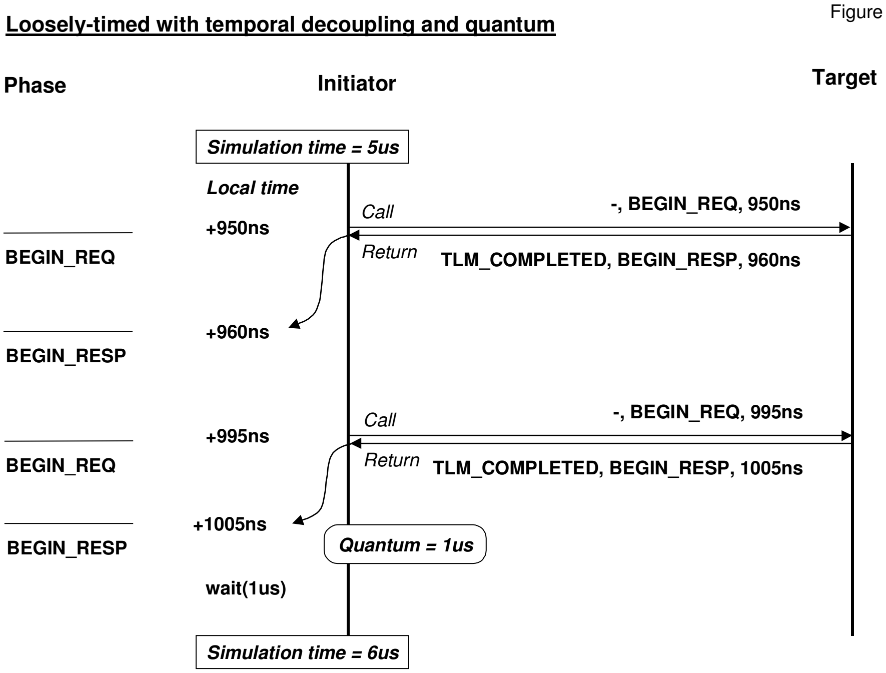
4.2.13.5 Loosely-timed with temporal decoupling and quantum

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.

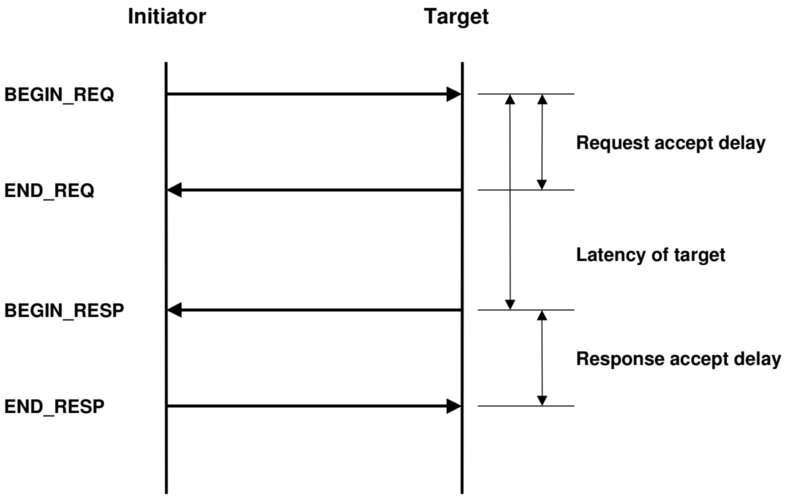
d) Successive transactions can be pipelined.

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.

Approximately-timed timing parameters

Figure 9



4.2.13.7 Approximately-timed using backward path

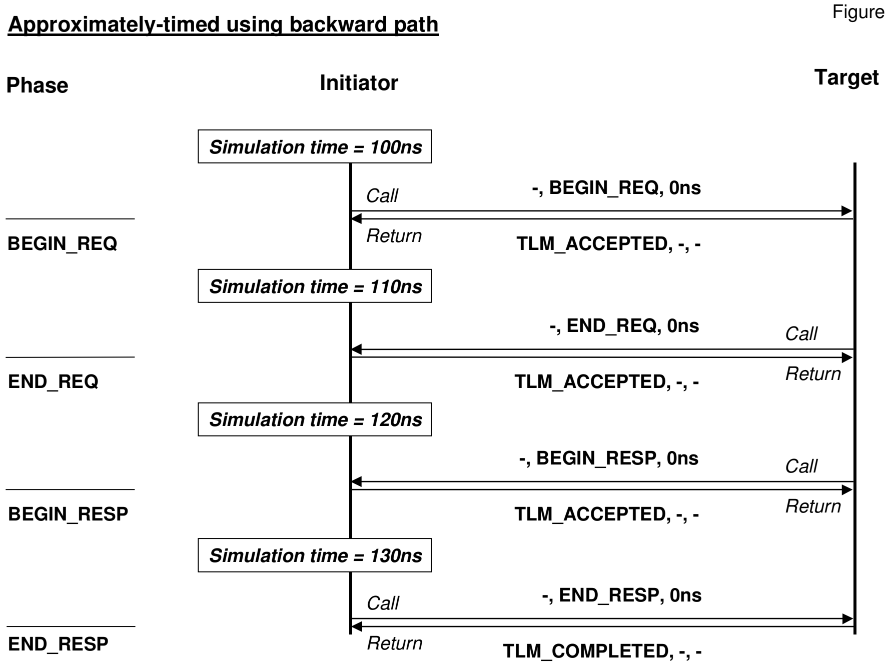
With the approximately-timed coding style for the generic payload, a transaction is passed back-and-forth twice between initiator and target. (For other protocols, this number may be smaller or larger.) As with loosely-timed, a target may or may not be able to respond immediately to a transaction, and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

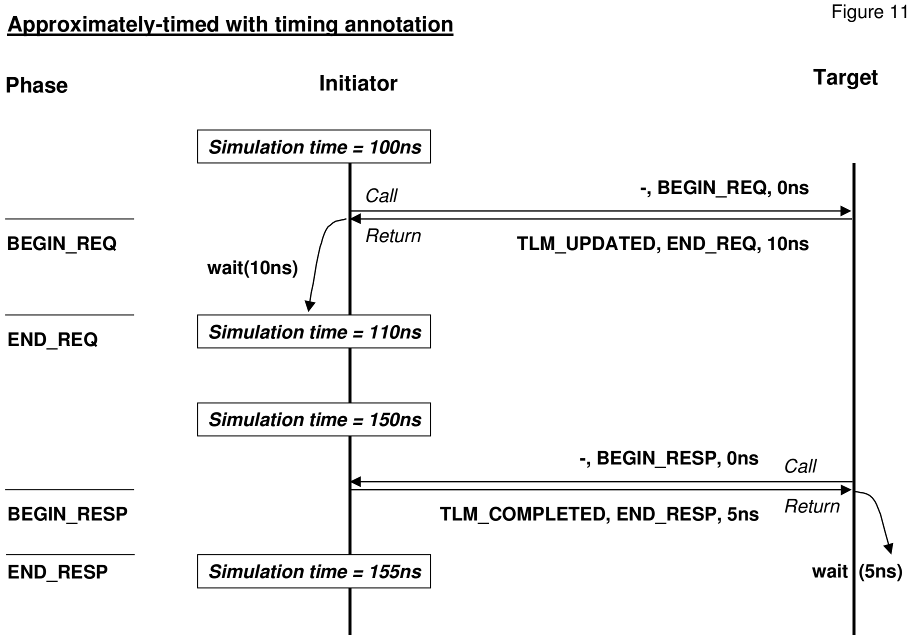


4.2.13.8 Approximately-timed with timing annotation

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 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.

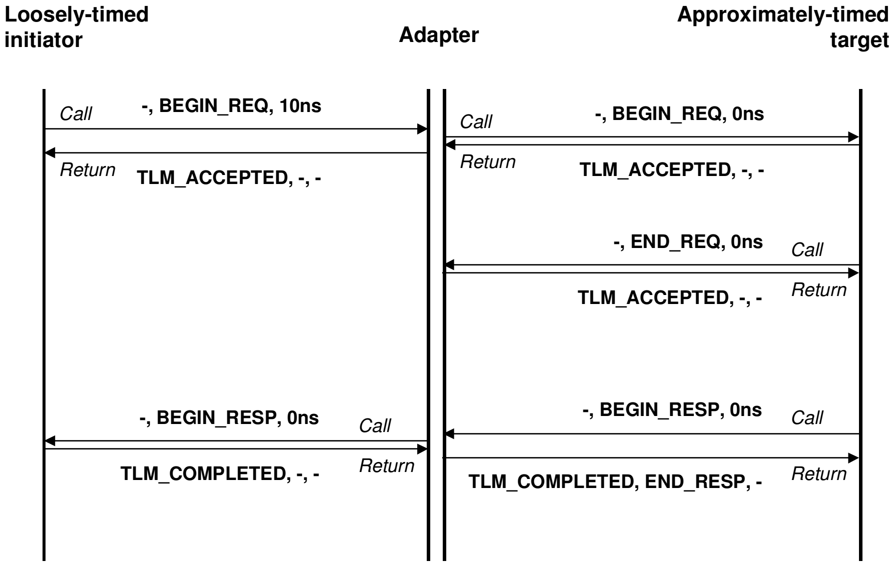
Notice that the loosely-timed initiator is temporally decoupled, and passes the local time on the first call to **nb_transport**. The target ignores the local time, and immediately forces the initiator to synchronize. The phase transition to END_REQ is not required by the initiator. The BEGIN_RESP is 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



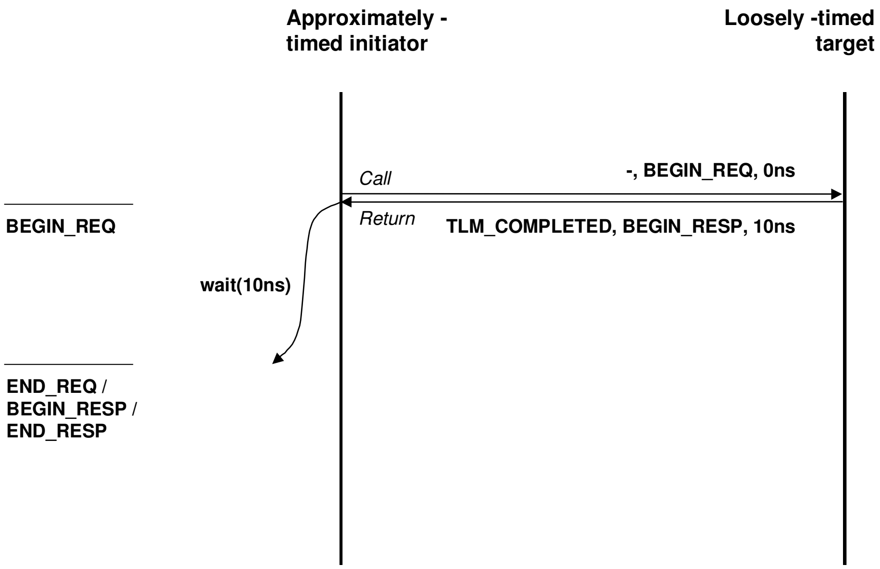
4.2.13.10 Approximately-timed initiator to loosely-timed target

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

Approximately-timed initiator to loosely-timed target


Approximately - timed initiator
Loosely -timed target
Call
-, BEGIN_REQ, 0ns
Return
TLM_COMPLETED, BEGIN_RESP, 10ns
BEGIN_REQ
wait(10ns)
END_REQ / BEGIN_RESP / END_RESP

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, but should be identical to the forward and backward paths for the corresponding transport calls through the same sockets.

A DMI pointer is requested by passing a transaction along the forward path. The default DMI transaction type is **tlm_generic_payload**, where only the command and address attributes of the transaction object are used. DMI follows the same approach to extension as the transport interface, that is, a DMI request may contain ignorable extensions, but any non-ignorable extension requires the definition of a new protocol types class (see clause **Error! Reference source not found. Error! Reference source not found.**).

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 {

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

```
    void init();

    enum Type { READ = 0x1, WRITE = 0x2, READ_WRITE = READ|WRITE };

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

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

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

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

} // namespace tlm
```

4.3.3 get_direct_mem_ptr method

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

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

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

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

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

f) 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**.

g) 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**.

h) Given multiple calls to **get_direct_mem_ptr**, a target may grant DMI access to multiple initiators for the same memory region at the same time.

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

4.3.4 template argument and tlm_generic_payload class

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

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

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

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

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

f) The command attribute of the transaction object shall be set by the initiator to indicate the kind of DMI access being requested, TLM_READ_COMMAND for read access, or TLM_WRITE_COMMAND for write access.

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

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

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

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

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

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

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) A DMI descriptor shall have the following attributes: the DMI pointer attribute, the granted access type attribute, the start address attribute, the end address attribute, the read latency attribute, and the write latency attribute,

c) 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.

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

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

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

g) 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 **Error! Reference source not found. Error! Reference source not found.**

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

i) The methods **allow_read**, **allow_write** and **allow_read_write** shall set the granted access type attribute to the value READ, WRITE or READ_WRITE respectively.

j) The method **is_read_allowed** shall return true if and only if the granted access type attribute has the value READ or READ_WRITE. The method **is_write_allowed** shall return true if and only if the granted access type attribute has the value WRITE or READ_WRITE. The method **is_read_write_allowed** shall return true if and only if the granted access type attribute has the value READ_WRITE.

m) The target shall set the granted access type attribute to the type of access being granted. A target is permitted to respond to a request for read access by granting READ or READ_WRITE access, and to a request for write access by granting WRITE or READ_WRITE access. An interconnect component is permitted to restrict the granted access type by overwriting a value of READ_WRITE with READ or WRITE on the backward path only.

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

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

p) The start and end address attributes shall be set by the target (or modified by the interconnect) to point to the addresses of the first and the last bytes in the DMI region, where the meaning of the DMI region is determined by the value returned from the **get_direct_mem_ptr** method (**true** or **false**).

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

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

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

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

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

v) 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.

w) 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.

x) 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.

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

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

aa) The read and write latency attributes 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 granted access type attribute.

bb) 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.

cc) 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.

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

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

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

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

j) 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.

The default debug transaction type is **tlm_generic_payload**, where only the command, address, data length and data pointer attributes of the transaction object are used. Debug transactions follow the same approach to extension as the transport interface, that is, a debug transaction may contain ignorable extensions, but any non-ignorable extension requires the definition of a new protocol types class (see clause **Error! Reference source not found. Error! Reference source not found.**).

4.4.2 Class definition

```
namespace tlm {

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

} // namespace tlm
```

4.4.3 TRANS template argument and tlm_generic_payload class

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

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

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

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

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

4.4.4 Rules

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

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

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

d) The command attribute of the transaction object shall be set by the initiator to indicate the kind of debug access being requested; TLM_READ_COMMAND for read access to the target, or TLM_WRITE_COMMAND for write access to the target.

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

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

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

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

i) The data length attribute shall be set by the initiator 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.

j) The data pointer attribute shall be set by the initiator to the address of an array from which values are to be copied to the target (for a write), or to which values are to be copied from the target (for a read). This array shall be allocated by the initiator, and shall not be deleted before the return from **transport_dbg**. The size of the array shall be at least equal to the value of the data length attribute.

k) 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.

l) 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.

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

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

o) **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.

p) **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.