TLM 2.0 LRM：TLM-2.0 Language Reference Manual

Transaction Level Modeling 事务级建模

独立于语言的一个标准，常用于系统建模，加速软硬件协同开发。在芯片开发中，常配合SystemC使用来进行系统设计。

TLM关注的重点是通信而不是进程实现的算法。尤其关注对片上地址映射总线进行建模。TLM 2.0具有分层结构，底层灵活通用，上层专用于建模。

## Introduction

### Background

TLM-1标准 -> TLM-2.0标准

TLM-1标准定义了一组用于通过值或常量引用传输事务的核心接口。这组接口在某些应用中有成功地被使用，但在模拟内存映射总线和其他芯片内通信网络方面存在几个缺点：

* + TLM-1没有标准的事务类，因此每个应用程序都必须创建自己的非标准类，导致来自不同来源的模型之间的互操作性非常差。TLM-2.0通过通用负载（generic payload）来解决这个问题。
  + TLM-1不支持时序注释(timing annotation)，因此没有一种标准的方法在模型之间传递时序信息。TLM-1模型通常通过调用wait来实现延迟，这会使仿真变慢。TLM-2.0通过在阻塞和非阻塞传输接口中添加时序注释来解决这个问题。
  + TLM-1接口要求所有事务对象和数据通过值或常量引用传递，这会减慢仿真的速度。一些应用程序通过在事务对象中嵌入指针来绕过这个限制，但这是非标准且不可互操作的。TLM-2.0通过具有跨多个传输调用的生命周期的事务对象，并通过新的传输接口来支持来解决了这个问题。

### Transaction-level modeling，use cases and abstraction

在ESL（Electronic System Level）社区中，关于事务级建模（Transaction Level Modeling）的抽象级别分类的最合适的分类方式一直存在着长期的讨论。模型可以根据**时间的粒度**、**模型评估的频率**、**功能抽象**、**通信抽象**和**使用案例**等一系列标准进行分类。TLM-2.0活动明确承认了事务级建模存在多种使用案例。TLM-2.0明确承认了事务级建模的各种用例的存在（参见TLM-2.0 requirement），但与定义每个用例周围的抽象层级不同，TLM-2.0采用了在接口（API）和编码样式之间区分的方法。TLM-2.0标准定义了一组接口，应将其视为实现事务级模型的低级编程机制，然后描述了一些适合于但不仅限于各种用例的编码样式。

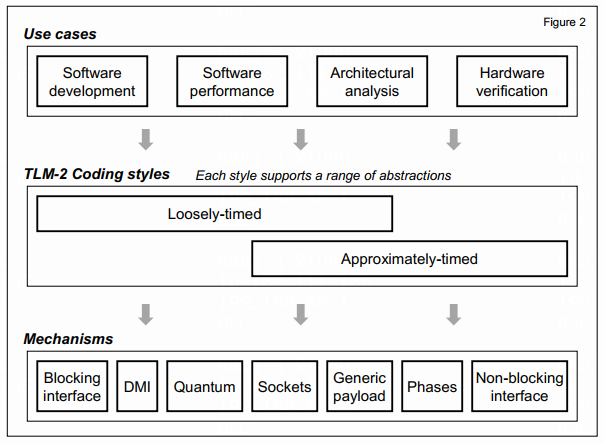
TLM-2.0标准接口的定义与编码样式的描述有所区别。TLM-2.0接口是标准的规范部分，确保了互操作性。每种编码样式可以支持跨功能、时序和通信的一系列抽象。原则上，用户可以创建自己的编码样式。

一个由单个软件线程组成的非时序功能模型可以编写为一个C函数或一个单一的SystemC进程，并且有时被称为**算法模型**。这样的模型本质上不是**事务级模型**，因为根据定义，**事务是通信的抽象**，而单线程模型没有进程间通信。**事务级模型需要多个SystemC进程来模拟并发执行和通信**。

一个包含多个进程（多个软件线程）的抽象事务级别模型需要一些机制，使得这些线程可以相互让出控制权。这是因为SystemC使用协作式多任务模型，执行中的进程不能被任何其他进程抢占。SystemC进程通过调用wait来让出控制权（对于线程进程），或者在方法进程中返回给内核。对wait的调用通常被隐藏在编程接口（API）后面，该接口可能模拟一个特定的抽象或具体协议，该协议可能依赖于时间信息或不依赖于时间信息。

同步(Synchronization)可以是强同步(strong)，即通信事件的序列事先精确确定，或者是弱同步(weak)，即通信事件的序列部分由各个进程的详细时序确定。在SystemC中，可以使用FIFO或信号量来轻松实现强同步，这样可以实现完全非时序的建模风格，原则上模拟可以在不推进模拟时间的情况下运行。在这个意义上，这种非时序的建模超出了TLM-2.0的范围。另一方面，允许多个嵌入式软件线程并行运行的快速虚拟平台模型可以使用强同步或弱同步。在这个标准中，这种模型的适当编码方式被称为松散时序的(loosely-timed)。

一个更详细的事务级模型可能需要将多个与每个事务相关的协议特定的时序点关联起来，例如标记协议各个阶段的开始和结束的时序点。通过选择适当数量的时序点，可以不需要在每个时钟周期上执行组件模型的情况下，对通信进行高度准确的时序建模。在这个标准中，这种编码风格被称为近似时序的(approximately-timed)。



### Coding styles

编码风格是一组良好配合使用的编程语言习惯用法，而不是特定的抽象级别或软件编程接口。为了简单和清晰起见，本文档仅限于详细说明两种特定的命名编码风格：松散时序和近似时序。由于其性质，编码风格并没有精确定义，并且管理TLM-2.0核心接口的规则是独立于这些编码风格定义的。原则上，可以基于TLM-1和TLM-2.0机制定义其他编码风格。

#### Untimed coding style

TLM-2.0并没有明确提供无时序编码风格，因为所有现代基于总线的系统都需要一定的时间概念来模拟在一个或多个嵌入式处理器上运行的软件。然而，TLM-1核心接口支持无时序建模。（术语无时序有时用于指称包含有限的、未指定准确度的时序信息的模型。在TLM-2.0中，这样的模型被称为松散时序。）

#### Loosely-timed coding style and temporal decoupling

松散时序(loosely-timed)编码风格使用阻塞传输接口。该接口只允许与每个事务关联两个时间点，分别对应于对阻塞传输函数的调用和返回。在基本协议的情况下，第一个时间点标志着请求的开始，第二个时间点标志着响应的开始。这两个时间点可以在相同的仿真时间或不同的时间发生。

松散时序编码风格适用于使用MPSoC的虚拟平台模型进行软件开发的用例，其中软件内容可能包括一个或多个操作系统。松散时序编码风格支持建模计时器(timer)和中断(interrupt)，足以启动操作系统并在目标机器上运行任意代码。

松散时序编码风格还支持时间解耦(temporal decoupling)，其中允许单个SystemC进程在本地的“时间扭曲(time warp)”中先行运行，直到它们达到需要与系统的其他部分同步的点，才会实际推进仿真时间。时间解耦可以在某些系统中导致非常快速的仿真，因为它增加了数据和代码的局部性，并减少了仿真器的调度开销。每个进程在切换到下一个进程之前，允许运行一定的时间片或量子(quantum)，或者在达到显式同步点时放弃控制权。

时间解耦：时间解耦是一种软件设计技术，用于将系统组件的时间依赖性降低到最低限度。在传统的实时系统中，不同组件的操作必须按照严格的时间要求进行同步，以确保系统的正确性。时间解耦的目标是将组件之间的时间关系放松，使得它们能够以更灵活的方式进行异步通信和交互。通过时间解耦，系统组件可以不再依赖于精确的时钟同步或预定的时间间隔进行操作。相反，它们可以根据需要和可用性进行通信和处理，而不受严格的时间约束。这种解耦可以提高系统的可扩展性、灵活性和可重用性，使组件能够以更自由、异步的方式进行交互。时间解耦可以通过使用事件驱动的编程模型、消息传递机制、异步通信接口和优化的调度算法等技术实现。它在现代的软件系统中得到了广泛应用，尤其是在分布式系统、多核系统和实时系统中。

就SystemC本身而言，SystemC调度器对仿真时间有严格的控制。调度器将仿真时间推进到下一个事件的时间点，然后运行在该时间点或对该事件敏感的任何进程。SystemC进程只在当前仿真时间（通过调用方法sc\_time\_stamp获得）运行，并且每当SystemC进程读取或写入变量时，它访问该变量在当前仿真时间下的状态。当一个进程运行结束后，它必须将控制权返回给仿真内核。**如果仿真模型以细粒度编写，则事件调度和进程上下文切换的开销成为仿真速度的主导因素。**加快仿真的一种方法是允许进程超前运行当前仿真时间，即时间解耦。

在SystemC中实现时间解耦时，一个进程可以在仿真时间之前运行，直到它需要与另一个进程进行交互，例如读取或更新属于另一个进程的变量。在这一点上，进程可以访问当前值并继续运行（可能会损失一些时间精度），也可以将控制权返回给仿真内核，在仿真时间赶上本地“时间扭曲”时才恢复进程。每个进程都负责确定在不破坏模型功能的情况下是否可以在仿真时间之前运行。当一个进程遇到外部依赖时，有两种选择：强制同步，即让出执行权，允许所有其他进程正常运行，直到仿真时间追上，或者采样或更新当前值并继续。同步选项确保与标准SystemC仿真语义的功能一致性。继续使用当前值依赖于对被建模系统中通信和时序的假设。它假设在太早或太晚采样或更新值时不会造成损害。在虚拟平台仿真的背景下，这个假设通常是有效的，因为软件栈不应依赖于硬件时序的低级细节。

**时间解耦是松散时序编码风格的特征。**

如果一个进程被允许无限制地超越仿真时间运行，SystemC调度器将无法操作，其他进程将永远无法执行。通过参考全局量子(global quantum)，可以避免这种情况，该量子对一个进程在仿真时间之前被允许运行的时间设置了上限。量子由应用程序设置，量子值代表模拟速度和准确性之间的权衡。如果量子太小，进程强制频繁地让步和同步，从而减慢模拟速度。如果量子太大，在系统中可能引入时序不一致性，甚至可能导致系统停止运行。

举个例子，考虑一个系统的仿真，该系统包括一个处理器、一个内存、一个定时器和一些慢速外部外设。运行在处理器上的软件大部分时间都花在从系统内存中取指令和执行指令上，只有在被定时器中断时才与系统的其他部分进行交互，比如每1毫秒一次。模拟处理器的ISS可以被允许在SystemC仿真时间之前以最多1毫秒的量子运行，直接访问内存模型，但只以定时器中断的速率与外设模型同步。关键在于ISS不必像传统的硬件软件协同仿真那样锁定到系统的硬件部分的仿真时间时钟。根据模型的细节，仅仅进行时间解耦就可以将仿真速度提高约10倍，当结合DMI时可以提高到100倍。

在某些情况下，有些进程可能是时间解耦的，而其他进程则不是，并且不同的进程可能使用不同的时间量子值。然而，任何没有时间解耦解耦的进程都可能成为仿真速度的瓶颈。

在TLM-2.0中，通过tlm\_global\_quantum类和阻塞和非阻塞传输接口的时间注释，支持时间解耦。实用类tlm\_quantumkeeper为访问全局时间量子提供了一种方便的方式。

#### Synchronization in loosely-timed models

一个无时序模型依赖于显式同步点的存在（即调用wait或阻塞方法调用），以便在执行过程中在预定的点之间传递控制权给发起者。一个松散时序模型也可以从显式同步中受益，以保证确定性执行，但是松散时序模型能够在没有显式同步点（调用wait）的情况下取得进展，因为每个发起者只会在时间量子结束之前运行到一定程度，然后让出控制权。松散时序模型可以通过使用按需同步来提高其时序精度，即在达到时间量子结束之前将控制权让给调度器。

时间量子(time quantum)机制并不是为了确保系统同步的正确性，而是一种仿真机制，在基于调度器的仿真环境中允许多个系统发起者取得进展(make progress)。时间量子机制并不是设计系统级显式同步方案的替代方案。

#### Approximately-timed coding style

非阻塞传输接口支持近似时序(approximately-timed)编码风格，适用于架构探索和性能分析的用例。非阻塞传输接口提供了时间注解以及事务生命周期中的多个阶段和时间点。

对于近似时序建模，一个事务被分解为多个阶段，其中显式的时间点标记了阶段之间的转换。在基本协议的情况下，有四个时间点分别标记了请求的开始和结束以及响应的开始和结束。特定的协议可能需要添加更多的时间点，这可能导致与通用载荷的直接兼容性丢失。

虽然可以在使用非阻塞传输接口时只使用两个阶段来指示事务的开始和结束，但通常在松散时序建模中更倾向于使用阻塞传输接口。由于需要时序的准确性，近似时序编码风格通常无法充分利用时间上的解耦。相反，每个进程通常与SystemC调度器一起同步执行。进程之间的交互通过特定的延迟进行注解。为创建一个近似时序模型，通常只需注解表示写和读命令的数据传输时间和目标的延迟。对于基本协议，数据传输时间实际上等于两个连续请求或两个连续响应之间的最小启动间隔或接收延迟。这些注解的延迟(annotated delay)通过调用SystemC调度器来实现，即wait(delay)或notify(delay)。

#### Characterization of loosely-timed and approximately-timed coding style

编码风格可以根据时间点和时间解耦来描述

* + 松散时序(Loosely-timed)：每个事务只有两个时序点，标记事务的开始和结束。使用的是仿真时间，但进程可以与仿真时间进行时间上的解耦。每个进程会记录自己相对于仿真时间提前了多久，并且可能因为到达显式的同步点或消耗完时间片而进行中断。
  + 近似时序(Approximately-timed)：每个事物有多个时间点，进程通常需要与SystemC模拟时间同步运行。对进程交互中注释的延迟是通过超时（wait）或定时事件通知来实现的。
  + 无时序(Utimed)：仿真时间的概念是不必要的。进程会在预先确定的显式同步点上进行中断。

#### Switching between loosely-timed and approximately-timed modeling

在仿真过程中，模型可以在松散时序和近似时序编码风格之间进行切换。这个想法是，在松散时序级别上快速运行重置和引导序列，然后一旦仿真达到一个有趣的阶段，就切换到近似时序建模进行更详细的分析。

#### Cycle-accurate modeling

目前，TLM-2.0的范围超出了准确时序建模的范围。使用现有的SystemC和TLM-1，可以创建准确时序的模型，但标准化准确时序编码风格的要求仍然是一个待解决的问题，可能会由未来的OSCI标准来解决。原则上，可以通过定义一组适当的阶段和规则，将近似时序编码风格扩展到包含准确时序建模。TLM-2.0版本包含了足够的机制来实现这一点，但具体细节尚未确定。

#### Blocking versus non-blocking transport interface

阻塞和非阻塞传输接口是TLM-2.0中存在的不同接口，用于支持不同级别的时序详细信息。阻塞传输接口只能对事务的开始和结束进行建模，事务在单个函数调用中完成。非阻塞传输接口允许将事务分解为多个时序点，并且通常需要多个函数调用来完成一个单独的事务。

为了实现互操作性，阻塞和非阻塞传输接口被组合成一个单一的接口。发起事务的模型可以根据编码样式使用阻塞或非阻塞传输接口（或两者兼有）。任何提供TLM-2.0传输接口的模型都有义务提供阻塞和非阻塞形式，以实现最大的互操作性，尽管该模型不一定要在内部实现两个接口。

TLM-2.0提供了一种机制（所谓的便利套接字the so-called convenience socket），用于自动将传入的阻塞或非阻塞传输调用转换为非阻塞或阻塞传输调用。转换传输调用类型确实会产生一些成本，特别是将传入的非阻塞调用转换为阻塞实现。然而，成本开销得到了缓解，尤其是由于近似定时模型的存在很可能会对仿真速度产生负面影响。

C++的静态类型规则强制执行两个接口的实现，但发起方可以动态选择调用阻塞或非阻塞传输方法。不同的发起方可以调用不同的方法，或者给定的发起方可以在运行时在阻塞和非阻塞调用之间切换。该标准包括规定了对同一目标的阻塞和非阻塞传输调用的混合和排序规则。

阻塞传输接口的优势在于它允许对能够在单个函数调用中完成事务的模型进行简化编码。非阻塞传输接口的优势在于它支持将多个时间点与单个事务关联起来。原则上，任一接口都支持时间解耦，但是对于具有多个时间点的近似定时模型来说，时间解耦的速度优势可能会被抵消。

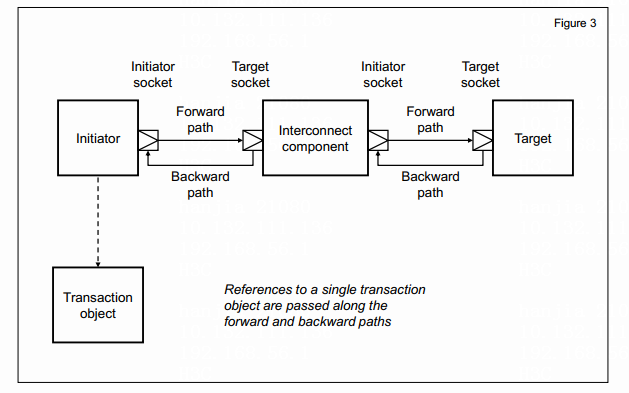
#### Use cases and coding styles

下表总结了事务级建模的用例和编码风格之间的映射：

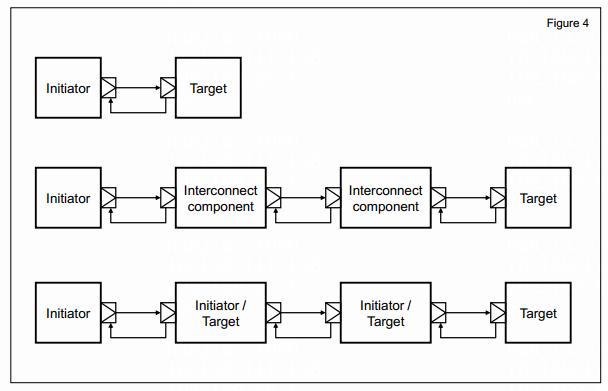
|  |  |
| --- | --- |
| **Use Case** | **Coding style** |
| Software application development | Loosely-timed |
| Software performance analysis | Loosely-timed |
| Hardware architectural analysis | Loosely-timed or approximately-timed |
| Hardware performance verification | Approximately-timed or cycle-accurate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

### Initiators，target，socket，and transaction bridges

TLM-2.0核心接口在initiators和targets之间传递事务。initiator是一个可以发起事务的模块，即创建新的事务对象并通过调用核心接口的方法将其传递给其他模块。target是事务的最终目的地。在写事务的情况下，发起方（如处理器）将数据写入目标方（如内存）。在读事务的情况下，发起方从目标方读取数据。中继组件是访问事务但不作为该事务的发起方或目标方的模块，典型的例子包括仲裁器和路由器。发起方，中继和目标方的角色可以动态变化。例如，某个组件可能对某些事务充当中继，但对其他事务充当目标方。



为了说明这个想法，本段将描述典型事务对象的生命周期。事务对象是由一个发起方创建的，并作为传输接口的一个方法的参数传递（可以是阻塞或非阻塞方法）。该方法由一个中继组件interconnect component（如仲裁器）实现，该中继组件在将事务对象传递给进一步的传输调用之前可以读取事务对象的属性。第二个传输方法由第二个中继组件（如路由器）实现，它再通过第三方传输调用将事务传递到目标方，如内存，作为事务对象的最终目的地。（实际的中继组件数量将因事务而异。有可能一个都没有。）这个方法调用序列被称为前向路径(forward path)。事务在目标方被执行，事务对象可以通过两种方式之一返回给发起方：一种是随着传输方法调用的返回而返回，被称为返回路径；另一种是通过从目标方返回到发起方的相反路径上进行显式传输方法调用而传递，称为后向路径(backward path)。这个选择是由非阻塞传输方法的返回值决定的。（严格来说，前向路径和后向路径各有一个返回路径，但意义通常从上下文中很清楚。）



前向路径是由发起方或中继组件在向另一个中继组件或目标方向前进时进行接口方法调用的调用路径。后向路径是由目标方或中继组件在向另一个中继组件或发起方向后退时进行接口方法调用的调用路径。从发起方到目标方的整个路径由多个跳跃组成，每个跳跃连接两个相邻组件。从发起方到目标方的跳数比该路径上的中继组件数多一个。当使用通用负载时，前向路径和后向路径应始终通过相同的一组组件和socket传递，显然是逆序的。

为了支持前向和后向路径，组件之间的每个连接都需要一个端口(port)和一个导出(export)，两者都必须被绑定。这是通过initiator socket和target socket来实现的。initiator socket为前向路径上的接口方法调用提供了一个端口，并为后向路径上的接口方法调用提供了一个导出。target socket则相反。（更具体地说，initiator socket派生自sc\_port类，具有一个sc\_export，反之亦然。）initiator和target socket类重载了SystemC端口绑定运算符，以隐式绑定前向和后向路径。

除了传输接口之外，socket还封装了DMI和调试传输接口(debug transport interface)

使用sockets时，发起方组件至少有一个initiator socket，目标方组件至少有一个target socket，互连组件至少有一个initiator socket和一个target socket。一个组件可以有多个socket，用于传输不同类型的事务，在这种情况下，单个组件可以为多个独立的事务充当发起方或目标方。

为了模拟总线桥接(bus bridge)，有两种选择。一是将**总线桥接建模为互连组件**，二是将**总线桥接建模为两个单独的TLM-2.0事务之间的事务桥接**。互连组件将传递一个指向单个事务对象的指针，这是提高模拟速度的最佳方法。事务桥接需要复制事务对象，这样可以提供更多的灵活性，因为两个事务可以具有不同的属性。

建议最大程度地使用TLM-2.0 sockets，以实现最大的互操作性、方便性和一致的编码风格。虽然组件可以直接使用SystemC的port和export与TLM-2.0核心接口互连，但这并不推荐做法。

### DMI and debug transport interface

直接内存接口（DMI）和调试传输接口是与传输接口不同的专用接口，提供直接访问和调试访问目标所拥有的内存区域。一旦DMI请求被授予，DMI接口就使发起方能够绕过传输接口使用的互连组件的常规路径。DMI旨在加速松散定时仿真中的常规内存事务，而调试传输接口用于不带延迟或与常规事务相关的副作用的调试访问。

DMI具有正向（发起方到目标方）和反向（目标方到发起方）接口，而调试仅具有正向接口。请参见4.2节“直接内存接口”和4.3节“调试传输接口”。

### Combined interfaces and sockets

在标准的发起方和目标方套接字中，阻塞和非阻塞传输接口与DMI和调试传输接口结合在一起。这四个接口（两个传输接口、DMI和调试接口）可以并行使用来访问给定的目标（遵守本标准所规定的规则）。这些组合接口是确保使用TLM-2.0标准的组件之间互操作性的关键之一，另一个关键是通用有效载荷。请参见6.1节“组合接口”。

标准目标套接字提供了这四个接口，因此每个目标组件必须有效地实现所有四个接口的方法。然而，阻塞和非阻塞传输接口的设计以及提供方便套接字来在两者之间进行转换的方式意味着给定的目标只需要根据模型的速度和准确性要求实现阻塞或非阻塞传输方法之一，而不需要同时实现两者。

根据速度和准确性要求，给定的发起方可以选择通过任何或所有的核心接口来调用方法。上面提到的编码风格有助于指导选择适当的接口功能集。通常，松散时序的发起方会调用阻塞传输、DMI和调试接口，而近似时序的发起方会调用非阻塞传输和调试接口。

### Namespace

TLM-2.0类应在两个顶层C++命名空间中声明，即tlm和tlm\_utils。TLM-2.0类的具体实现可以选择在这两个命名空间中嵌套更多的命名空间，但这些嵌套的命名空间不应在应用程序中使用。

tlm命名空间包含组成内存映射总线建模的互操作性接口的类。

tlm\_utils命名空间包含的是在内存映射总线模型的接口互操作性方面并非严格必需的工具类，但它们仍然是TLM-2.0标准的一部分。

## TLM-2.0 Core Interface

TLM-2.0在TLM-1.0的core interfaces上添加了blocking和non-blocking transport interfaces、DMI(a direct memory interface)、debug transport interface。

### Transport interfaces

传输接口是在发起者、目标和互连组件之间传输事务的主要接口。阻塞和非阻塞传输接口都支持时间注释和时间解耦，但只有非阻塞传输支持事务生命周期内的多个阶段。阻塞传输没有明确的阶段参数，阻塞传输与非阻塞传输接口的阶段之间的任何关联都是纯粹的概念性的。只有非阻塞传输方法返回一个值，指示是否使用了返回路径。

传输接口和通用有效载荷（generic payload）的设计是为了结合使用，以快速、抽象地建模内存映射总线。传输接口模板是使用事务类型进行特化的，使它们可以单独使用而不需要通用有效载荷，尽管这样将失去许多互操作性的好处。

关于事务对象的内存管理、事务排序和允许的函数调用顺序的规则取决于传递给传输接口的特定事务类型作为模板参数，而这又取决于传递给套接字的协议特性类作为模板参数（如果使用套接字）。

#### Blocking transport interface

TLM-2.0的**阻塞传输接口**旨在支持松散定时(loosely-timed)的编码风格。阻塞传输接口适用于当发起方希望在单个函数调用期间与目标完成事务时，只有标记事务开始和结束的时序点才是感兴趣的。

阻塞传输接口只使用从发起方到目标方的正向路径。

TLM-2.0的阻塞传输接口故意与TLM-1的传输接口相似，TLM-1传输接口仍然是TLM-2.0标准的一部分，但TLM-1传输接口和TLM-2.0的阻塞传输接口并不完全相同。特别是，新的b\_transport方法具有由非const引用传递的单个事务参数和一个用于标注时序的第二个参数，而TLM-1传输方法使用一个const引用请求参数作为单个请求，没有时序注释，并且通过值返回一个响应。TLM-1假设通过值（或const引用）传递单独的请求和响应对象，而TLM-2.0假设无论是使用阻塞还是非阻塞的TLM-2.0接口，都使用通过引用传递的单个事务对象。

b\_transport方法具有时序注释参数。该单个参数既用于调用b\_transport时标记事务的开始时间，也用于返回b\_transport时标记事务的结束时间，相对于当前仿真时间。

##### 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 |

##### The TRANS template argument

这个核心接口的目的是用于传输任意类型的事务。为了简化在事务属性细节不太重要的模型之间的互操作性，提供了一个特定的事务类型tlm\_generic\_payload。

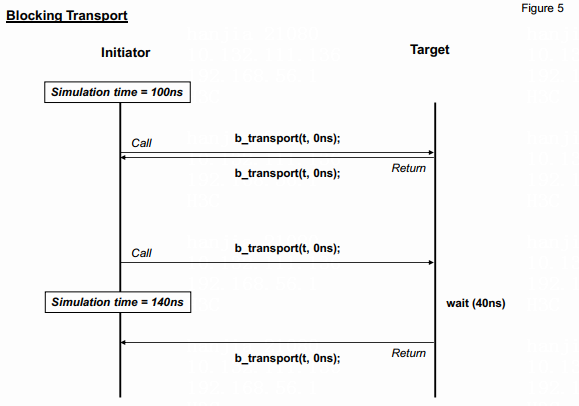
为了最大限度地提高互操作性，应用程序应该使用默认的事务类型tlm\_generic\_payload和基本协议。请参见8.2基本协议。为了建模特定的协议，应用程序可以替换为自己的事务类型。使用不同事务类型专门化的接口的sockets无法绑定在一起，可以提供编译时检查，但限制了互操作性。

##### Rules

* b\_transport方法可以直接或间接调用wait
* b\_transport方法不能在method进程中被调用,在调用b\_transport后，thread会阻塞等wait。
* 在从一个调用到下一个调用以及跨调用传输接口、DMI和调试传输接口时，发起者可以重复使用事务对象。
* 调用b\_transport标记了事务的第一个时序点。从b\_transport返回标记了事务的最后一个时序点。
* 时间注释参数允许将时序点与执行函数调用和返回的仿真时间（由sc\_time\_stamp()返回的值）偏移。
* 被调用者可以修改或更新事务对象，但必须遵守事务类TRANS所施加的任何约束。
* 建议事务对象不应包含时序信息。应该使用sc\_time参数对b\_transport进行时序注释。
* 通常情况下，互连组件应该将b\_transport调用沿着从起始者到目标者的正向路径传递。换句话说，互连组件的目标套接字的b\_transport实现可以调用起始者套接字的b\_transport方法。
* 关于b\_transport的实现是否允许调用nb\_transport\_fw取决于与协议相关的规则。对于基本协议来说，convenience 套接字simple\_target\_socket可以自动进行这种转换。详见9.1.2简单套接字。
* b\_transport的实现不应调用nb\_transport\_bw。

##### Message sequence chart – blocking transport

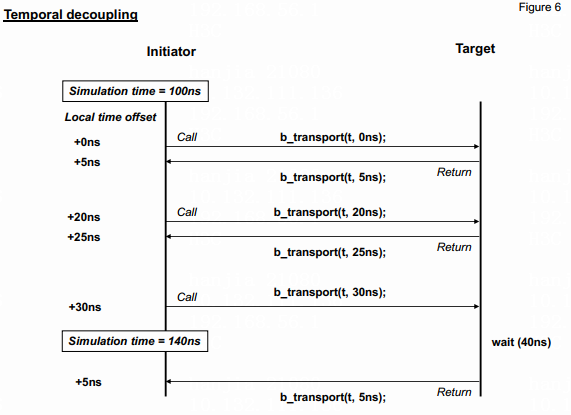
阻塞传输方法可以立即返回（即在当前SystemC评估阶段）或者可以将控制权交给调度器，并且仅在后续的仿真时间点才返回给启动者。虽然启动者线程可能被阻塞，但根据协议，启动者中的另一个线程可能在第一个调用返回之前被允许调用b\_transport。



##### Message sequence chart – temporal decoupling

一个在当前仿真时间之前以虚构的本地时间运行的时间解耦的initiator，应该向b\_transport传递一个非零值的时间参数，如下所示。initiator和target可以通过增加时间参数的值进一步提前本地时间偏移。将从调用返回的时间参数与当前仿真时间相加，可以得到每个事务完成的虚拟时间，但是仿真时间本身在启动者线程未让出之前不能提前。

b\_transport的主体本身可以调用wait，在这种情况下，本地时间偏移应被重置为零。在下面的图中，initiator的最终返回发生在仿真时间140ns，但带有一个标注的延迟为5ns，从而给出了有效的本地时间为145ns。

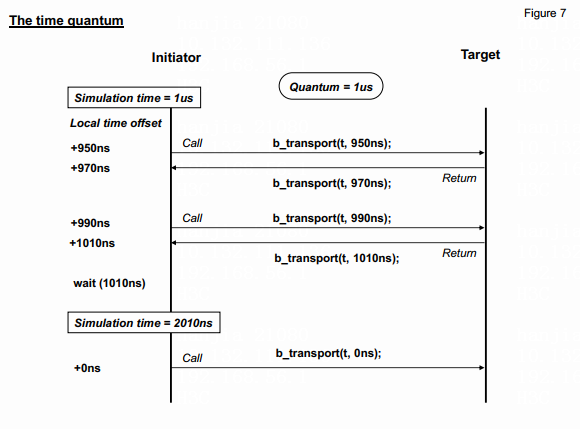


##### Message sequence chart – the time quantum

一个时间解耦的initiator将继续推进本地时间，直到超过时间量子(time quantum)。在那时，initiator有义务通过暂停执行来同步，直接或间接地调用带有本地时间作为参数的wait方法。这允许模型中的其他initiator运行并追赶上来，这实际上意味着initiator依次执行，每个initiator负责通过跟踪自己的本地时间确定何时交回控制。原始initiator只有在仿真时间提前到下个时间量子后才能再次运行。

松散时序编码风格中延迟的主要目的是允许每个启动者确定何时交回控制。最好的做法是模型不依赖于时序的细节以正确运行。

在每个时间量子内，由给定initiator生成的事务按严格的顺序进行，但不会推进仿真时间。本地时间不由SystemC调度器跟踪。



#### Non-blocking transport interface

**非阻塞传输接口**旨在支持近似时序(approximately-timed)编码风格。非阻塞传输接口适用于希望在每个事务过程中建模发起方和目标之间详细交互序列的情况。换句话说，它可以将一个事务分解为多个阶段，其中每个阶段转换与一个时序点相关联。**每次调用和返回非阻塞传输方法都可能对应一个阶段转换**。

通过将时序点的数量限制为两个，可以使用非阻塞传输接口来实现松散时序编码风格，但这通常不被推荐。对于松散时序建模，一般更倾向于使用阻塞传输接口，因为它更简单。非阻塞传输接口特别适用于**建模流水线事务**，使用阻塞传输接口来建模这种情况会很麻烦。

非阻塞传输接口同时使用从发起方到目标方的正向路径和从目标方到发起方的反向路径。有两个不同的接口，即tlm\_fw\_nonblocking\_transport\_if和tlm\_bw\_nonblocking\_transport\_if，用于在相反的路径上使用。

非阻塞传输接口与阻塞传输接口使用类似的参数传递机制，即非阻塞传输方法传递一个非const引用给事务对象和一个时序注释，但相似之处就到此为止了。非阻塞传输方法还传递一个阶段参数来指示事务的状态，并返回一个枚举值来指示函数的返回是否也表示了一个阶段转换。

阻塞传输和非阻塞传输都支持时序注释，但只有非阻塞传输支持在事务的生命周期内有多个阶段。阻塞传输和非阻塞传输接口以及通用负载（generic payload）的设计是为了在快速、抽象建模内存映射总线时一起使用。然而，传输接口可以与通用负载分开使用来建模特定的协议。事务类型和阶段类型都是非阻塞传输接口的模板参数。

##### Class definition

namespace tlm {

enum **tlm\_sync\_enum** { TLM\_ACCEPTED, TLM\_UPDATED, TLM\_COMPLETED };

template <typename TRANS = tlm\_generic\_payload, typename PHASE = tlm\_phase>

class **tlm\_fw\_nonblocking\_transport\_if** : public virtual sc\_core::sc\_interface {

public:

virtual tlm\_sync\_enum **nb\_transport\_fw**(TRANS& trans, PHASE& phase, sc\_core::sc\_time& t) = 0;  
};

template <typename TRANS = tlm\_generic\_payload, typename PHASE = tlm\_phase>

class **tlm\_bw\_nonblocking\_transport\_if** : public virtual sc\_core::sc\_interface {

public:

virtual tlm\_sync\_enum **nb\_transport\_bw**(TRANS& trans, PHASE& phase, sc\_core::sc\_time& t) = 0;

};  
} // namespace tlm

##### The TRANS and PHASE template argument

非阻塞传输接口的目的是可用于传输任何类型的事务，并具有任何数量的阶段和时间点。提供了一个特定的事务类型tlm\_generic\_payload来简化模型之间的互操作性，其中事务属性的精确细节不太重要，并且提供了一个特定类型tlm\_phase来与基础协议一起使用。参见8.2基本协议

为了实现最大的互操作性，应用程序应该使用默认的事务类型tlm\_generic\_payload和默认的阶段类型tlm\_phase来处理基本协议。为了对特定的协议建模，应用程序可以替换它们自己的事务类型和阶段类型。使用专用于不同事务类型的接口的套接字不能绑定在一起，这提供了编译时检查，但限制了互操作性。

##### The nb\_transport\_fw and nb\_transport\_bw calls

1. 有两种non-blocking传输方法，nb\_transport\_fw用作forward path，nb\_transport\_bw用作backward path。除了它们的名字和调用方向，这两个方法有相似的语义。在本文档中，斜体术语nb\_transport用于描述无需区分的情况下的两种方法。
2. 在基本协议的情况下，前向和后向路径应该以相反的顺序通过完全相同的组件和套接字序列。每个组件负责使用目标套接字(通过该套接字第一次接收到事务)将任何返回的事务路由到发起者。
3. nb\_transport\_fw只能被forward path调用，nb\_transport\_bw只能被backward path调用。
4. 前向路径上的nb\_transport\_fw调用在任何情况下都不得直接或间接调用反向路径上的nb\_transport\_bw，反之亦然。
5. *nb\_transport*方法，不能直接或间接调用wait
6. *nb\_transport*方法可以被一个thread进程或method进程调用
7. nb\_transport不允许调用b\_transport。一种解决方案是从一个单独的线程进程中调用b\_transport，该线程进程由原来的nb\_transport\_fw方法产生或通知。对于基础协议，提供了一个方便的套接字simple\_target\_socket，它能够自动进行这种转换。参见9.1.2 Simple sockets
8. 非阻塞传输接口明确支持流水线事务。换句话说，同一个进程对nb\_transport\_fw的几次连续调用都可以启动单独的事务，而不必等待第一个事务完成。
9. 原则上，事务的最终时间点可以通过在正向路径或反向路径上调用nb\_transport或从nb \_ transport返回来标记。

##### The trans argument

1. 给定事务对象的生命周期可以延长到超过从nb\_transport返回的时间，使得对nb\_transport的一系列调用可以在发起者、互连组件和目标之间来回传递单个事务对象。
2. 如果有多个对nb\_transport的调用与给定事务实例相关联，则同一个事务对象将作为参数传递给每个这样的调用。换句话说，给定的事务实例应该由单个事务对象来表示。
3. initiator可以重用给定的事务对象来表示多个事务实例，或者跨越对传输接口、DMI和调试传输接口的调用。
4. 由于事务对象的生命周期可能会延续到对nb\_transport的多次调用，因此调用方或被调用方都可以修改或更新事务对象，这取决于事务类**TRANS**所施加的任何约束。例如，对于generic payload，在读取命令的情况下，目标可以更新事务对象的数据阵列，但是不应该更新命令字段。参见7.7默认值和属性的可修改性

##### The phase argument

1. 对nb\_transport的每个调用都会传递一个对阶段对象的引用。在基本协议的情况下，不允许以相同的相位连续调用nb\_transport。每个阶段转换都有一个相关的时间点。使用sc\_time参数的时序注释应延迟相对于相位转换的时序点。
2. phase参数通过引用传递，caller和callee都可能修改phase
3. phase参数的目的应该是用于通知组件是否以及何时允许它们读取或修改事务的属性。如果协议的规则允许给定组件在特定阶段期间修改事务属性的值，则该组件可以在该阶段期间的任何时间修改该值，并且可以在该阶段期间修改任何次数。协议应该禁止其他组件在该阶段读取该属性的值，只允许在下一个阶段转换后读取该值。
4. phase参数的值表示给定hop下的协议状态机的当前状态。当单个事务对象在两个以上的组件(initiator、interconnect、target)之间传递时，每一跳(至少在概念上)需要一个单独的协议状态机。
5. 尽管事务对象有一个生命周期和一个范围，可以扩展到对nb\_transport的任何单个调用之外，但阶段对象通常是调用方的本地对象。给定事务的每个nb\_transport调用可以具有单独的阶段对象。不同跳上的相应相变可能发生在仿真时间的不同点。
6. 默认phase类型tlm\_phase特定于基础协议。其他协议可能使用或扩展tlm\_phase类型，或者可能替换它们自己的phase类型(相应地丧失了互操作性)。参见8.1 Phase。

##### The tlm\_sync\_enum return value

1. 有几处提到了同步的概念。同步是将控制权让给SystemC调度程序，以便其他进程可以运行，但对于时间上的解耦有额外的含义。这一点在别处有更充分的讨论。参见9.2.4 General guideline for processes using temporal decoupling。
2. 原则上，同步可以通过让步(yielding)来完成 (在thread进程的情况下调用wait，或者在method进程的情况下return到内核)，但是时间上解耦的发起方应该通过调用tlm\_quantum\_keeper类的sync方法来同步。一般来说，为了允许其他SystemC进程运行，启动器需要不时地(time-to-time)进行同步。
3. 以下规则适用于forward和backward path
4. nb\_transport返回值的含义是固定的，不随transaction类型或phase类型而变化。因此，以下规则不限于基本协议，而是适用于用于参数化非阻塞传输接口的每个transaction和phase类型。
5. **TLM\_ACCEPTED**。被调用方不应在调用期间修改事务对象的状态、阶段或时间参数。换句话说，TLM\_ACCPETED指示返回路径没有被使用。调用者可以忽略调用后的nb\_transport参数的值，因为被调用者必须保持它们不变。通常，在包含被调用方的组件能够响应事务之前，调用方必须让步。对于基本协议，忽略可忽略阶段的被调用者应该返回TLM \_ACCEPTED。
6. **TLM\_UPDATED**。被调用方已更新事务对象。被调用方可能已经修改了阶段参数的状态，可能已经修改了事务对象的状态，并且可能已经在调用期间增加了时间参数的值。换句话说，TLM \_UPDATED指示返回路径正在被使用，并且被调用者已经推进了与事务相关联的协议状态机的状态。被调用者是否真的必须修改每个参数取决于协议。在调用nb\_transport之后，调用者应该检查阶段、事务和时间参数，并采取适当的措施。
7. **TLM\_COMPLETED**。被调用方已经更新了事务对象，事务完成。被调用方可能已经修改了事务对象的状态，并且可能已经在调用期间增加了时间参数的值。phase参数的值未定义。换句话说，TLM \_COMPLETED指示返回路径正在被使用，并且关于特定套接字的事务已经完成。在调用nb\_transport之后，调用者应该检查事务对象并采取适当的操作，但是应该忽略phase参数。沿着前向或后向路径，通过当前套接字将不会有与该特定事务相关联的进一步传输调用。这种意义上的完成并不一定意味着成功完成，因此根据事务类型，调用方可能需要检查嵌入在事务对象中的响应状态。
8. 一般来说，没有义务通过nb\_transport返回TLM\_COMPLETED来完成交易。当协议的最后阶段作为参数传递给nb\_transport时，在任何情况下，关于特定套接字的事务都是完成的。(对于基本协议，最终阶段是END\_RESP。)换句话说，TLM\_COMPLETED不是强制的。
9. 对于三个返回值中的任何一个，根据协议，在调用nb\_transport之后，调用者可能需要让步，以便允许包含被调用者的组件生成响应或释放事务对象。

##### tlm\_sync\_enum summary

|  |  |  |  |
| --- | --- | --- | --- |
| **tlm\_sync\_enum** | **Transaction object** | **Phase on return** | **Timing annotation on return** |
| TLM\_ACCEPTED | Unmodified | Unchanged | Unchanged |
| TLM\_UPDATED | Updated | Changed | May be increased |
| TLM\_COMPLETED | Updated | Ignored | May be increased |