



## **OSCI TLM-2.0**

**The Transaction Level Modeling standard of the  
Open SystemC Initiative (OSCI)**



# OSCI TLM-2.0

Software version: TLM-2.0.1

Document version: ja7

This presentation authored by: John Aynsley, Doulos

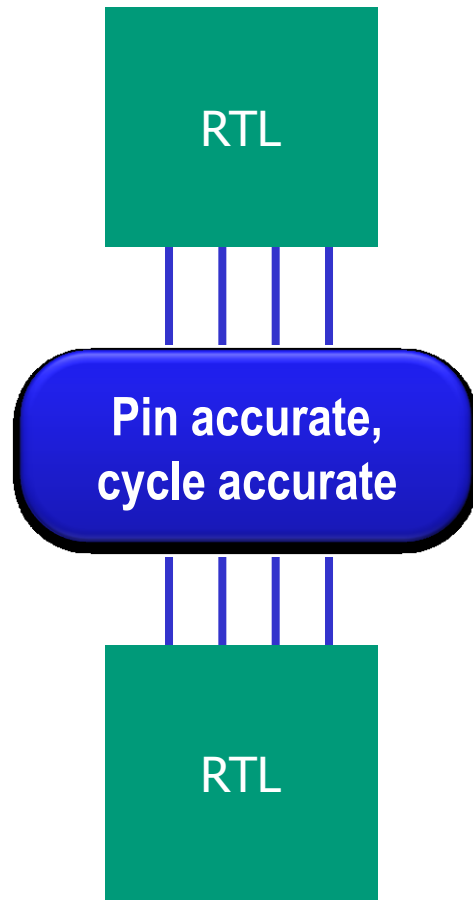
## CONTENTS

- ☐ Introduction
- ☐ Transport Interfaces
- ☐ DMI and Debug Interfaces
- ☐ Sockets
- ☐ The Generic Payload
- ☐ The Base Protocol
- ☐ Analysis Ports

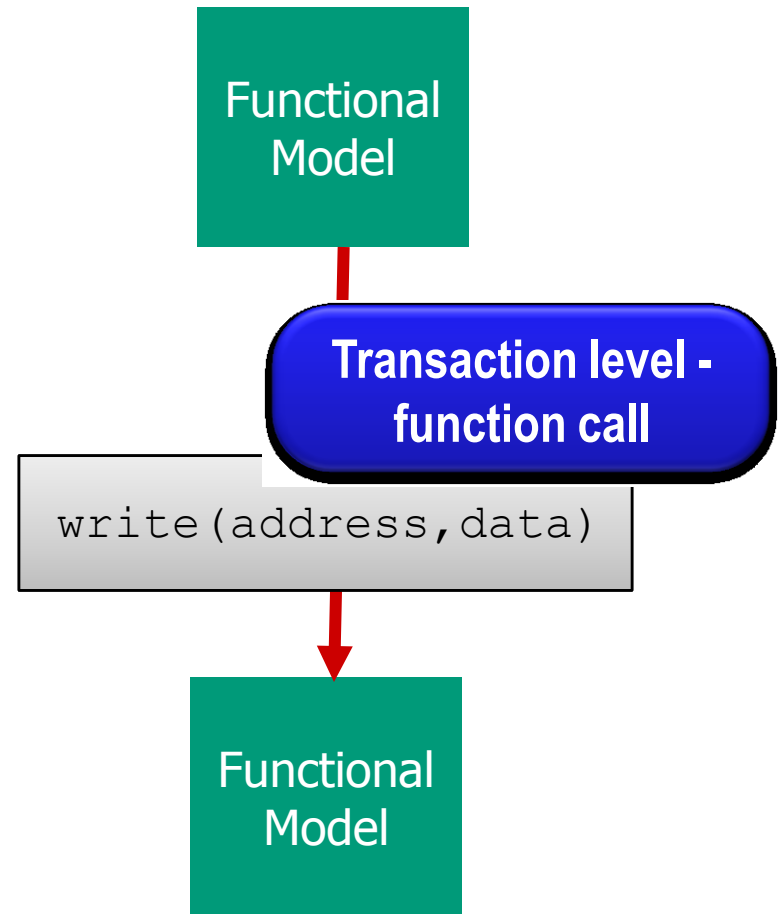
# INTRODUCTION

- ❑ Transaction Level Modeling 101
- ❑ OSCI TLM-1 and TLM-2
- ❑ Coding Styles
- ❑ Structure of the TLM-2.0.1 Kit

# Transaction Level Modeling 101

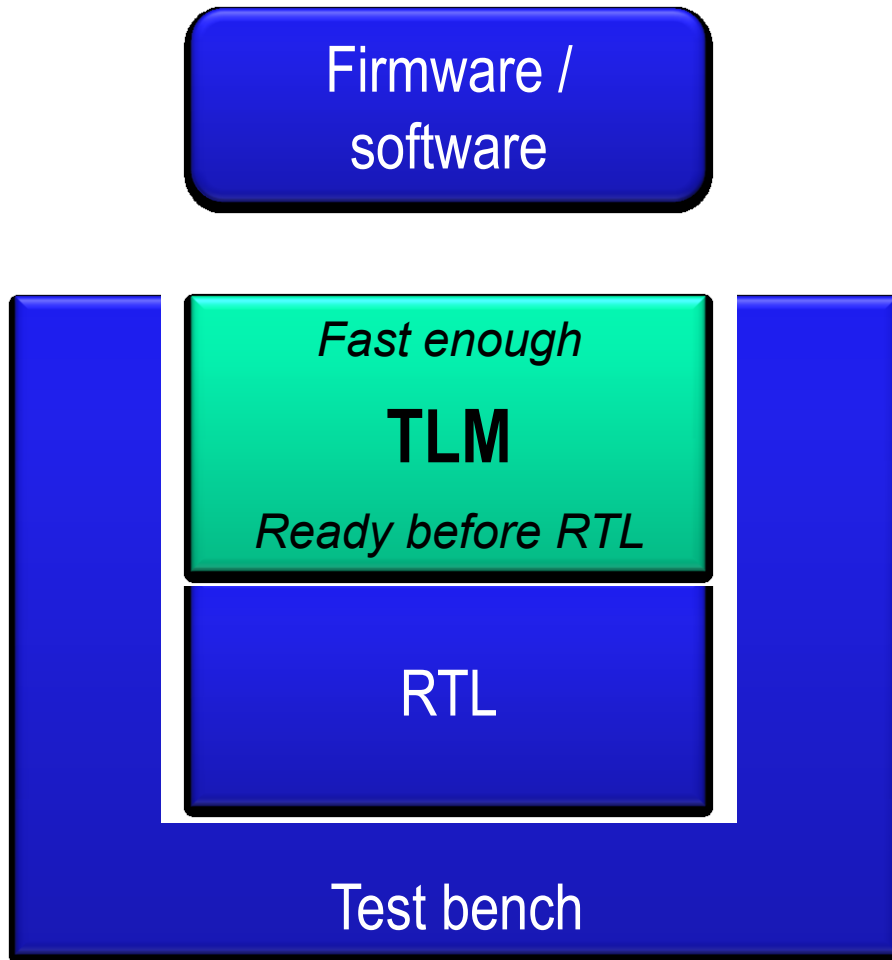


Simulate every event

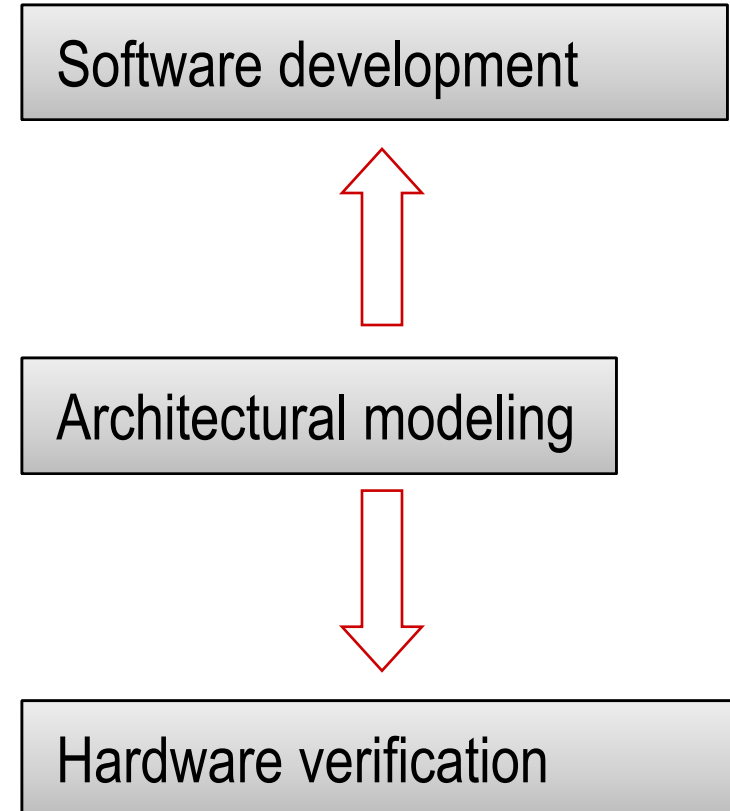


100-10,000 X faster simulation

# Reasons for Using TLM



*Accelerates product release schedule*



*TLM = golden model*

# Typical Use Cases for TLM

- Represents key architectural components of hardware platform
- Architectural exploration, performance modeling
- Software execution on virtual model of hardware platform
- Golden model for hardware functional verification
- Available before RTL
- Simulates much faster than RTL

**Early!**

**Fast!**

# OSCI TLM Development





# TLM-1.0 → TLM-2.0

- TLM-2.0 is the new standard for interoperability between memory mapped bus models
  - Incompatible with TLM-2.0-draft1 and TLM-2.0-draft2
- TLM-1.0 is not deprecated (put, get, nb\_put, nb\_get, transport)
- TLM-1.0 is included within TLM-2.0
  - Migration path from TLM-1.0 to TLM-2.0 (see examples)



# TLM-2 Requirements

- Transaction-level memory-mapped bus modeling
- Register accurate, functionally complete
- Fast enough to boot software O/S in seconds
- Loosely-timed and approximately-timed modeling
- Interoperable API for memory-mapped bus modeling
- Generic payload and extension mechanism
- Avoid adapters where possible

**Speed**

**Interoperability**

See *TLM\_2\_0\_requirements.pdf*

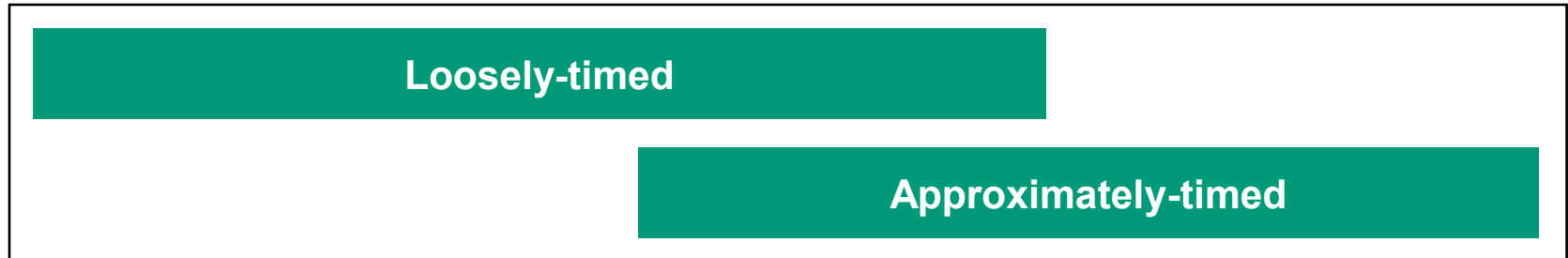


# Use Cases, Coding Styles and Mechanisms

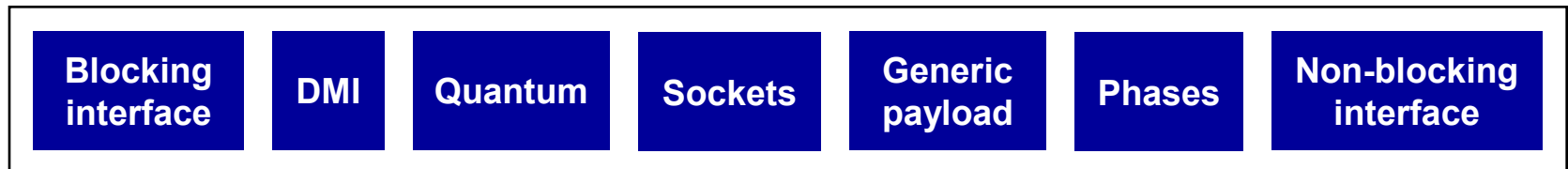
## *Use cases*



## *TLM-2 Coding styles*



## *Mechanisms*



# Coding Styles

## ■ Loosely-timed

- Only sufficient timing detail to boot O/S and run multi-core systems
- Processes can run ahead of simulation time (temporal decoupling)
- Each transaction has 2 timing points: *begin* and *end*
- Uses direct memory interface (DMI)

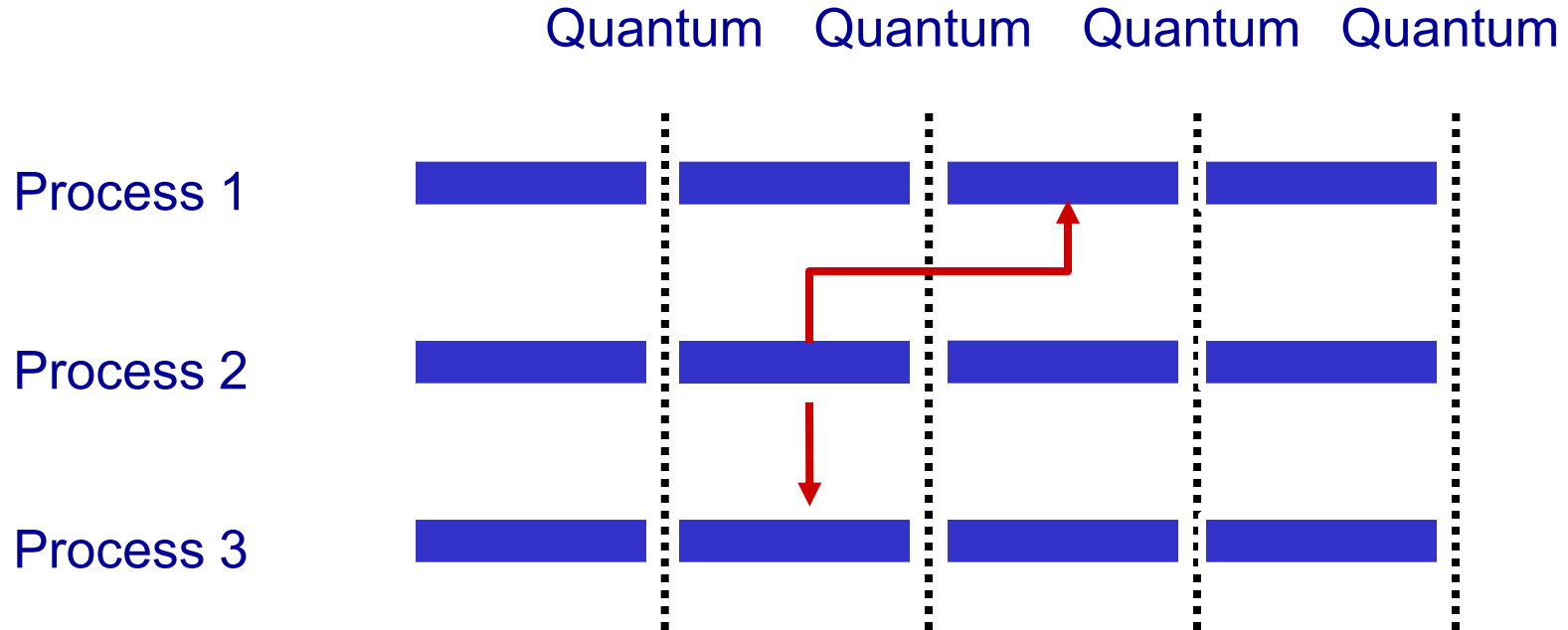
## ■ Approximately-timed

- *aka* cycle-approximate or cycle-count-accurate
- Sufficient for architectural exploration
- Processes run in lock-step with simulation time
- Each transaction has 4 timing points (extensible)

## ■ Guidelines only – not definitive



# Loosely-timed



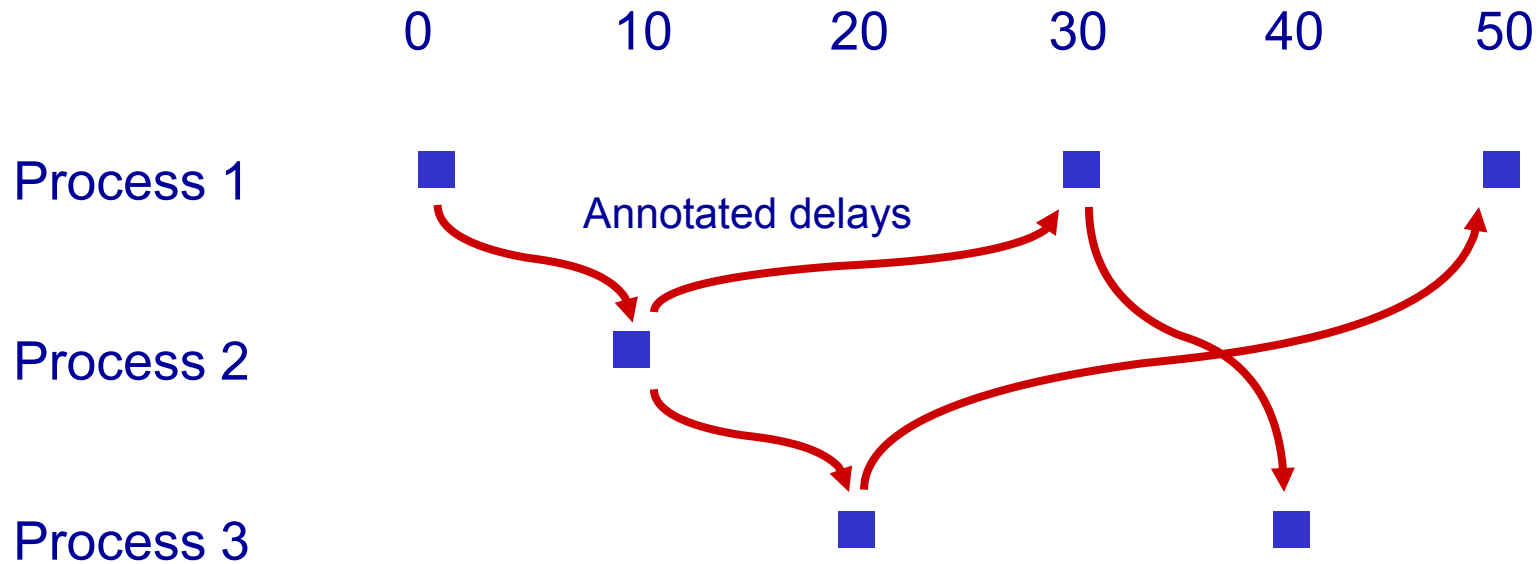
*Each process runs ahead up to quantum boundary*

*sc\_time\_stamp() advances in multiples of the quantum*

*Deterministic communication requires explicit synchronization*



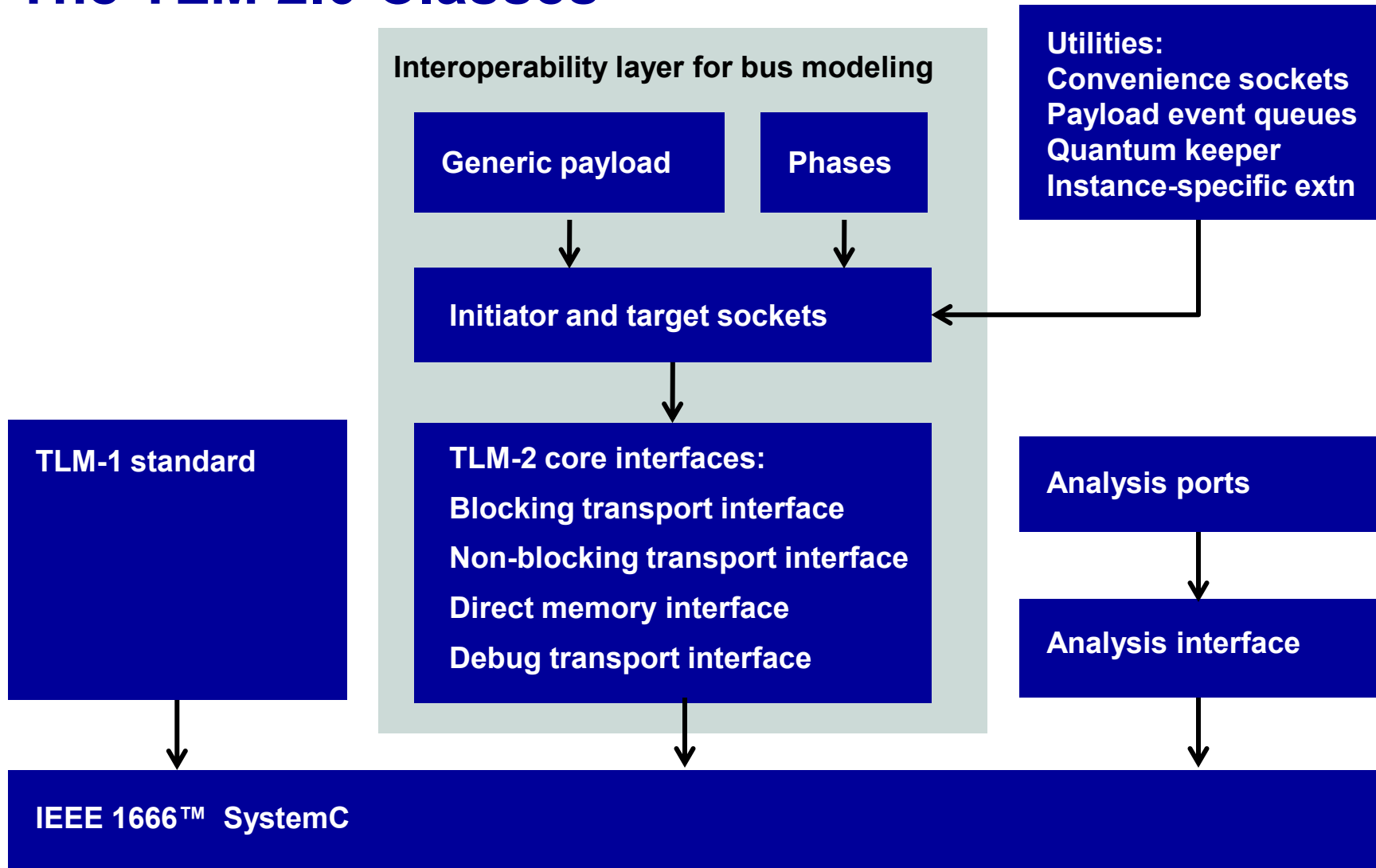
# Approximately-timed



*Each process is synchronized with SystemC scheduler  
Delays can be accurate or approximate*



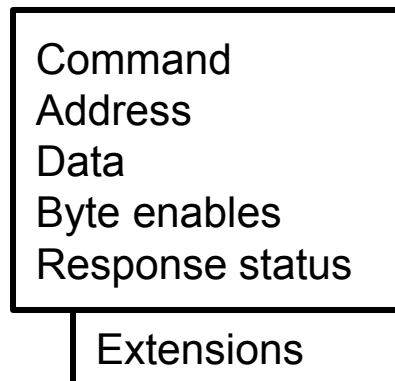
# The TLM 2.0 Classes



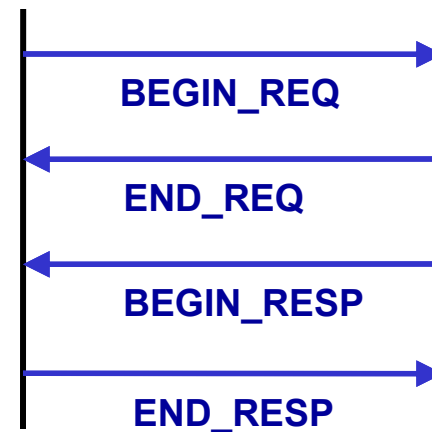
# Interoperability Layer



## 2. Generic payload



## 3. Base protocol



*Maximal interoperability for memory-mapped bus models*





# Utilities

- **tlm\_utils**
  - Convenience sockets
  - Payload event queues
  - Quantum keeper
  - Instance-specific extensions
- **Productivity**
- **Shortened learning curve**
- **Consistent coding style**
- ***Not* part of the interoperability layer – write your own?**



# Directory Structure

include/tlm

tlm\_h

tlm\_2\_interfaces

tlm\_generic\_payload

tlm\_sockets

tlm\_quantum

tlm\_1

tlm\_req\_rsp

tlm\_analysis

tlm\_utils

*TLM-2 interoperability classes*

*TLM-2 core interfaces*

*TLM-2 generic payload*

*TLM-2 initiator and target sockets*

*TLM-2 global quantum*

*TLM-1.0 legacy*

*Analysis interface, port, fifo*

*TLM-2 utilities*

docs

doxygen

examples

unit\_test

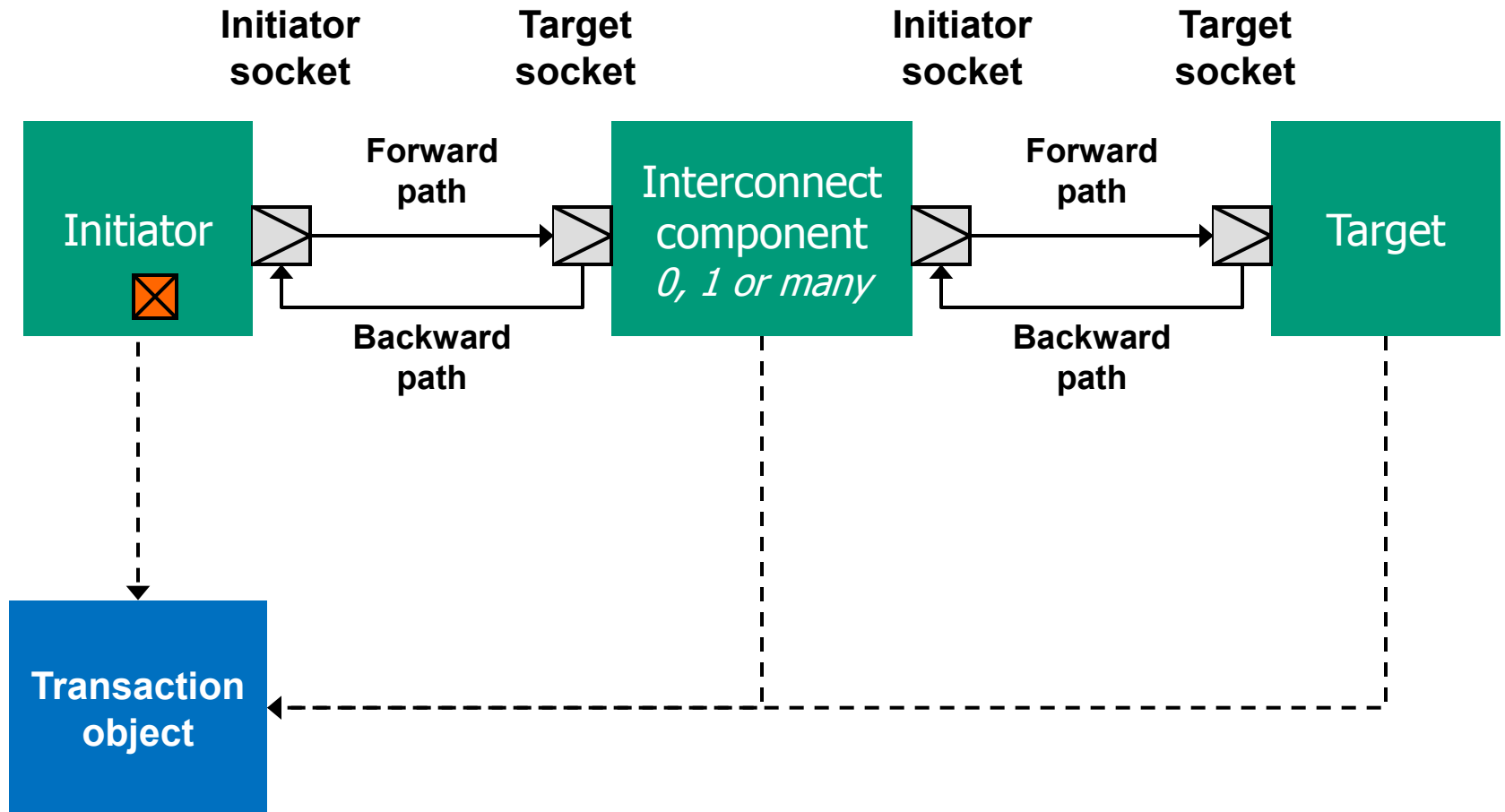


# TRANSPORT INTERFACES

- ☐ Initiators and Targets
- ☐ Blocking Transport Interface
- ☐ Timing Annotation and the Quantum Keeper
- ☐ Non-blocking Transport Interface
- ☐ Timing Annotation and the Payload Event Queue

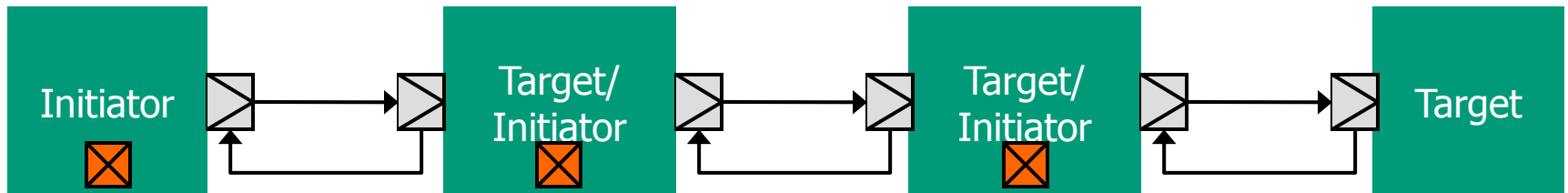
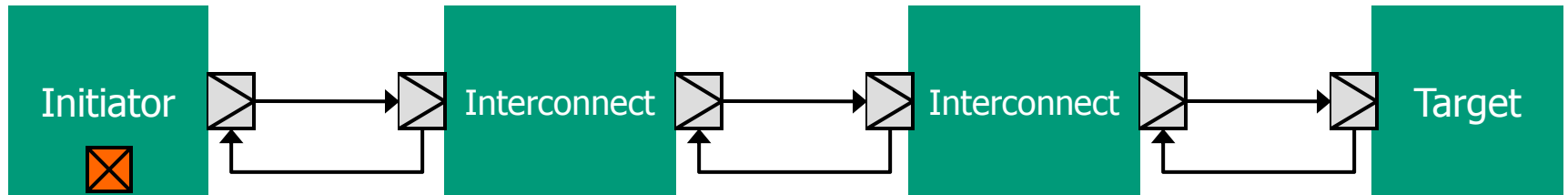
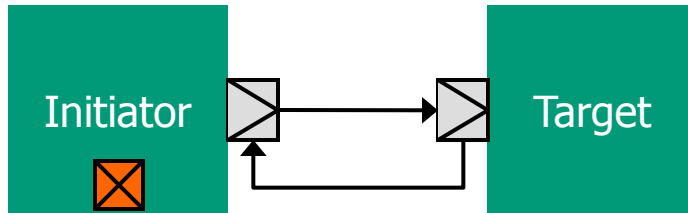


# Initiators and Targets



*References to a single transaction object are passed along the forward and backward paths*

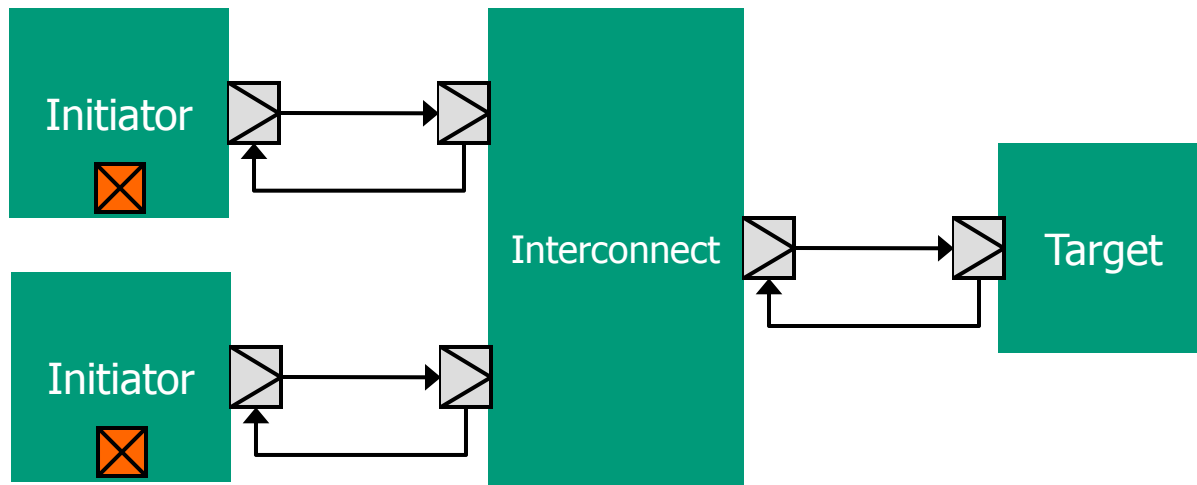
# TLM-2 Connectivity



*Transaction memory management needed*



# Convergent Paths



# Blocking versus Non-blocking Transport

- **Blocking transport interface**
  - Includes timing annotation
  - Typically used with loosely-timed coding style
  - Forward path only
- **Non-blocking transport interface**
  - Includes timing annotation and transaction phases
  - Typically used with approximately-timed coding style
  - Called on forward and backward paths
- **Share the same transaction type for interoperability**
- **Unified interface and sockets – can be mixed**



# TLM-2 Core Interfaces - Transport

**tlm\_blocking\_transport\_if**

```
void b_transport( TRANS& , sc_time& );
```

**tlm\_fw\_nonblocking\_transport\_if**

```
tlm_sync_enum nb_transport_fw( TRANS& , PHASE& , sc_time& );
```

**tlm\_bw\_nonblocking\_transport\_if**

```
tlm_sync_enum nb_transport_bw( TRANS& , PHASE& , sc_time& );
```



# TLM-2 Core Interfaces - DMI and Debug

**tlm\_fw\_direct\_mem\_if**

```
bool get_direct_mem_ptr( TRANS& trans , tlm_dmi& dmi_data );
```

**tlm\_bw\_direct\_mem\_if**

```
void invalidate_direct_mem_ptr( sc_dt::uint64 start_range,  
                                sc_dt::uint64 end_range );
```

**tlm\_transport\_dbg\_if**

```
unsigned int transport_dbg( TRANS& trans );
```

*May all use the generic payload transaction type*



# Blocking Transport

*Transaction type*



```
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;  
};
```

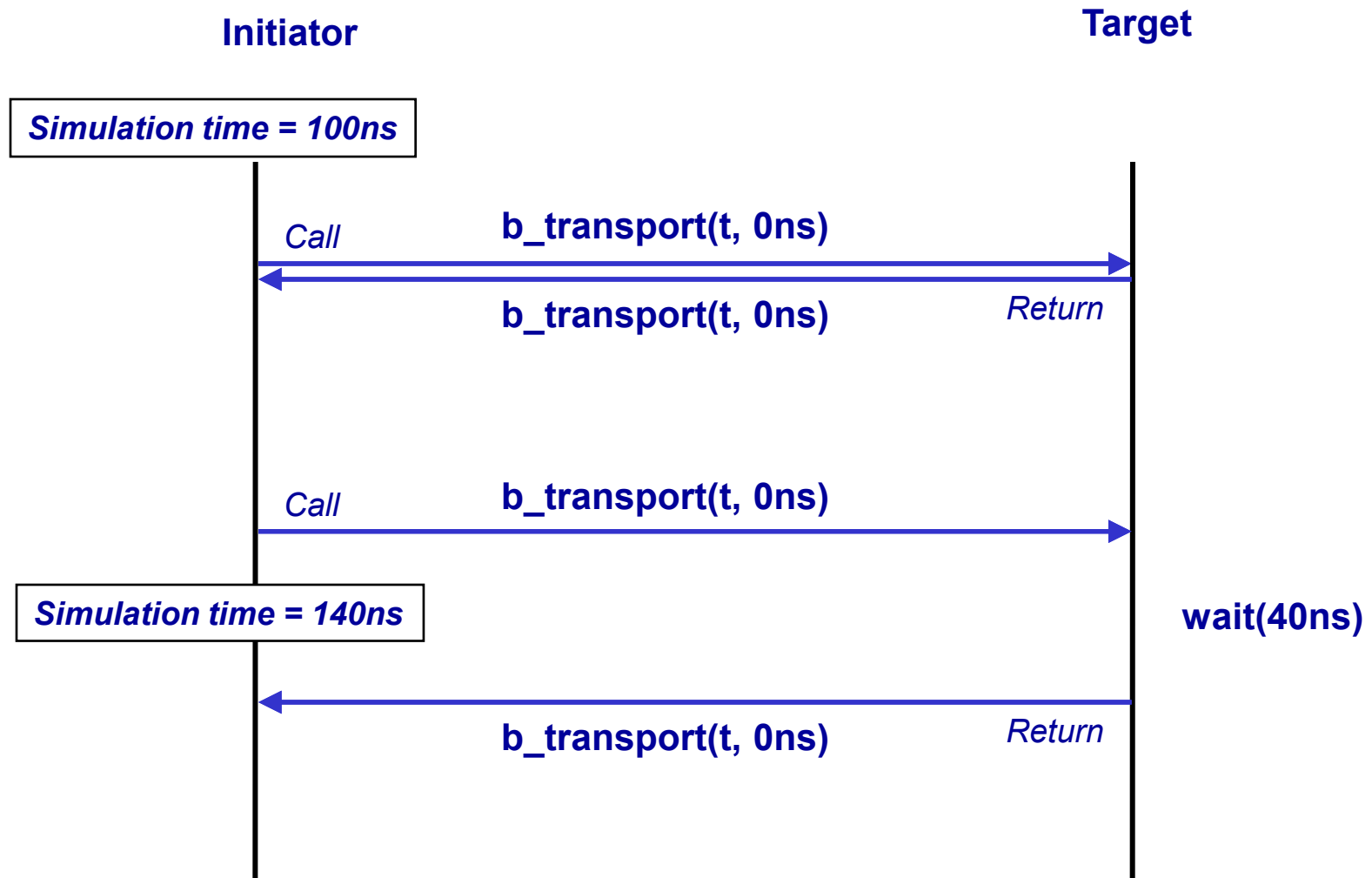


*Transaction object*



*Timing annotation*

# Blocking Transport



*Initiator is blocked until return from `b_transport`*

# Timing Annotation

```
virtual void b_transport ( TRANS& trans , sc_core::sc_time& delay )  
{  
    // Behave as if transaction received at sc_time_stamp() + delay  
    ...  
    delay = delay + latency;  
}
```

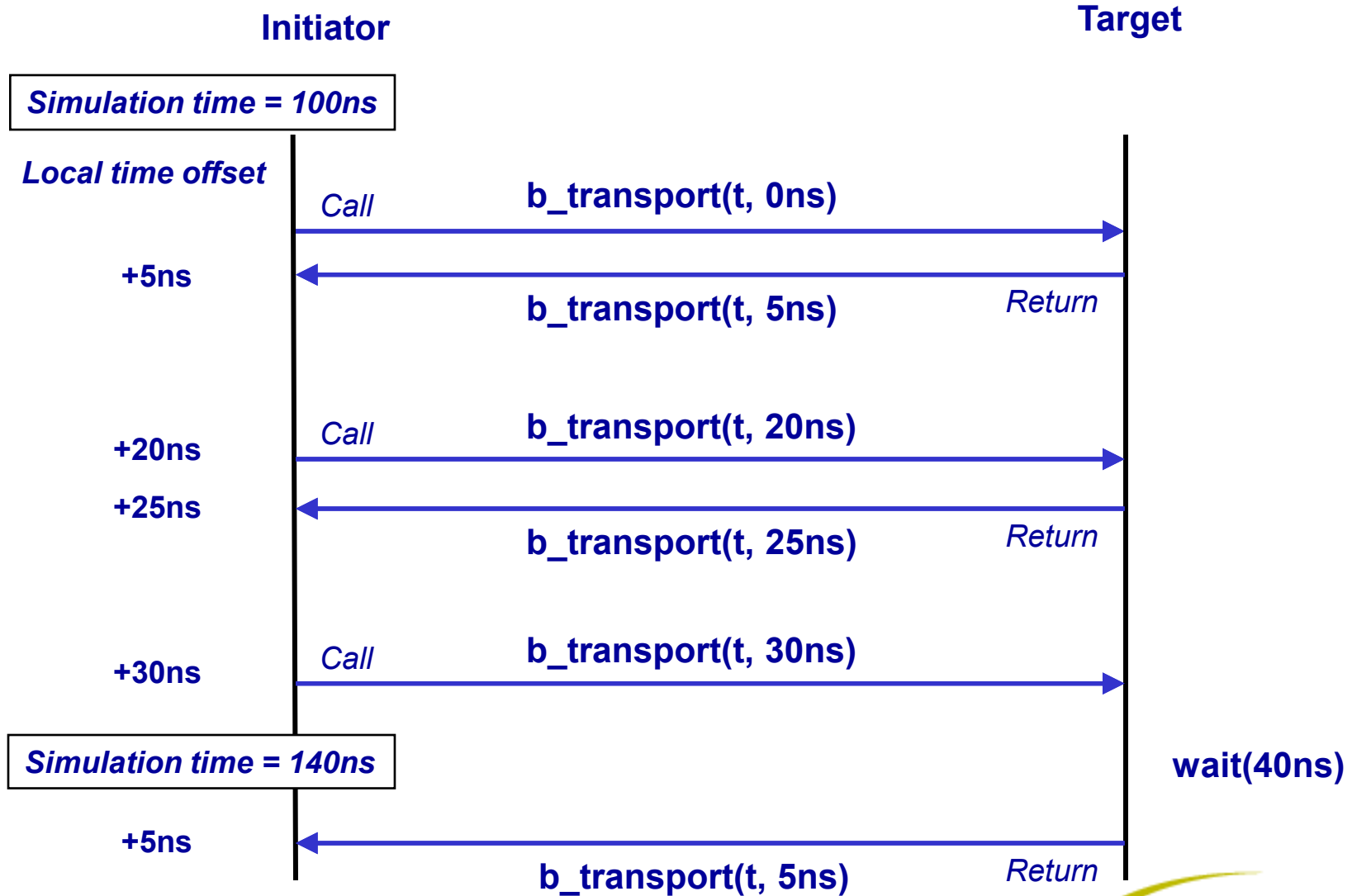
```
b_transport( transaction, delay );
```

```
// Behave as if transaction received at sc_time_stamp() + delay
```

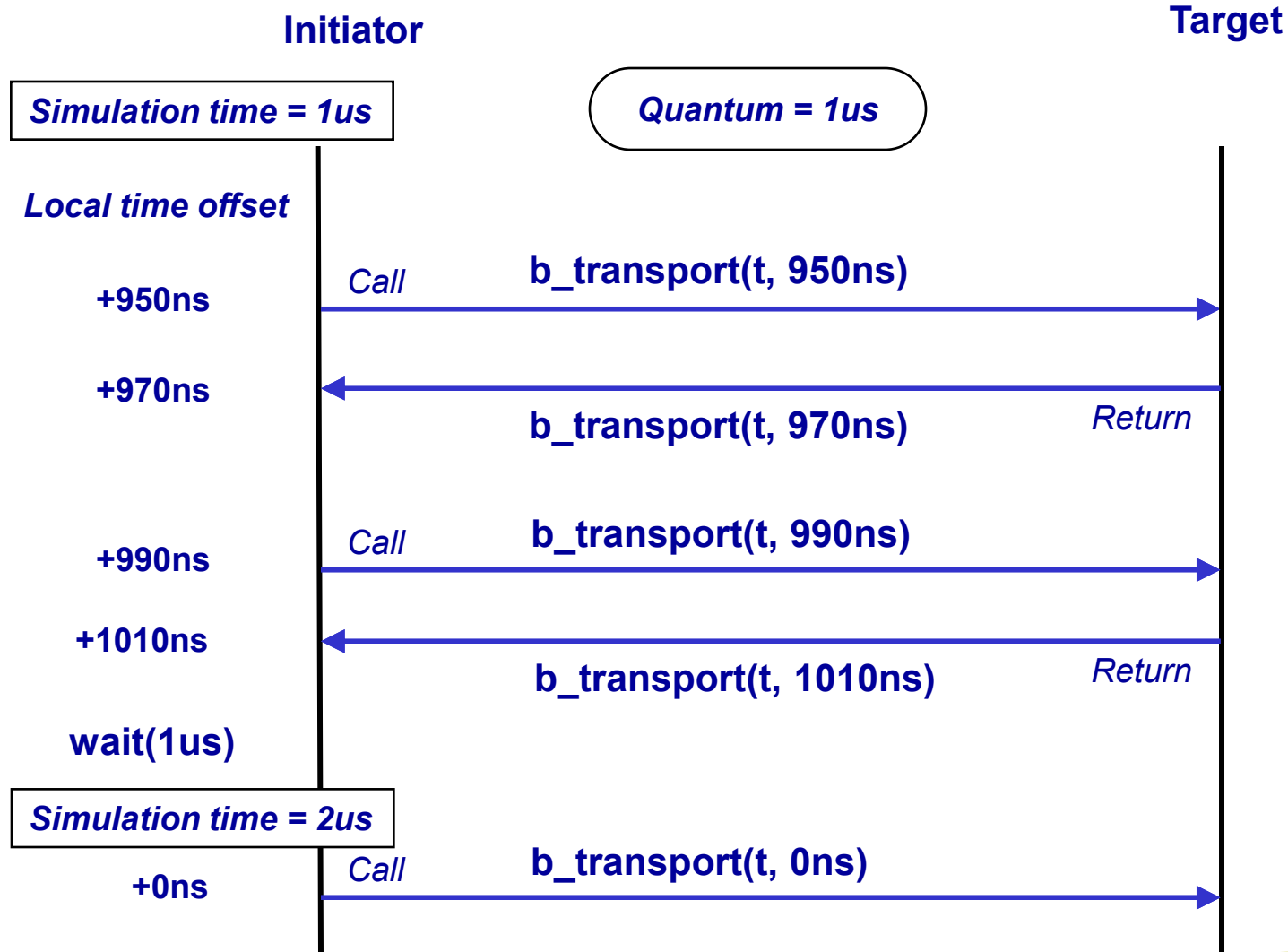
- Recipient may
  - Execute transactions immediately, out-of-order – Loosely-timed
  - Schedule transactions to execution at proper time – Approx-timed
  - Pass on the transaction with timing annotation



# Temporal Decoupling

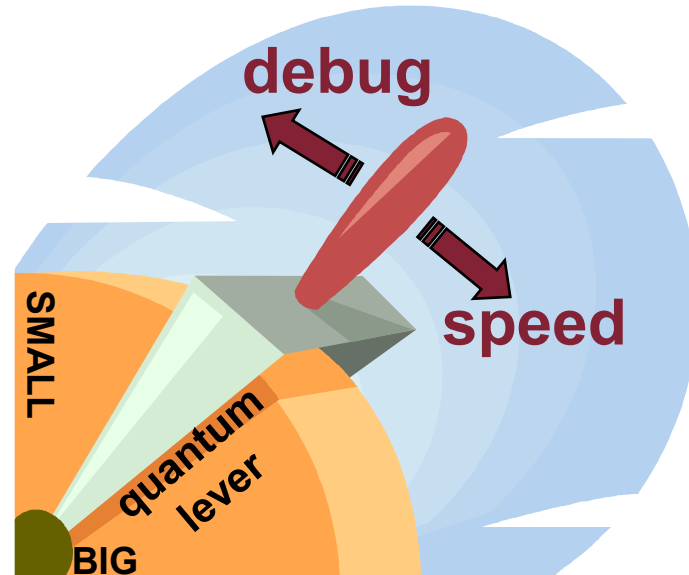


# The Time Quantum



# The Quantum Keeper (tlm\_quantumkeeper)

- Quantum is user-configurable



- Processes can check local time against quantum

# Quantum Keeper Example

```
struct Initiator: sc_module
{
    tlm_utils::simple_initiator_socket<Initiator> init_socket;
    tlm_utils::tlm_quantumkeeper m_qk;

    SC_CTOR(Initiator) : init_socket("init_socket") {
        ...
        m_qk.set_global_quantum( sc_time(1, SC_US) );
        m_qk.reset();
    }
    void thread() { ...
        for (int i = 0; i < RUN_LENGTH; i += 4) {
            ...
            delay = m_qk.get_local_time() ;
            init_socket->b_transport( trans, delay );
            m_qk.set( delay );
            m_qk.inc( sc_time(100, SC_NS) );
            if ( m_qk.need_sync() )
                m_qk.sync();
        }
    }
};
```

*The quantum keeper*

*Replace the global quantum  
Recalculate the local quantum*

*Time consumed by transport  
Further time consumed by initiator  
Check local time against quantum  
and sync if necessary*





# Non-blocking Transport

```
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(          TRANS& trans,  
                                           PHASE& phase,  
                                           sc_core::sc_time& t ) = 0;  
};
```

*Trans, phase and time arguments set by caller and modified by callee*

# tlm\_sync\_enum

## ■ TLM\_ACCEPTED

- Transaction, phase and timing arguments unmodified (ignored) on return
- Target may respond later (depending on protocol)

## ■ TLM\_UPDATED

- Transaction, phase and timing arguments updated (used) on return
- Target has advanced the protocol state machine to the next state

## ■ TLM\_COMPLETED

- Transaction, phase and timing arguments updated (used) on return
- Target has advanced the protocol state machine straight to the final phase



# Notation for Message Sequence Charts

***Simulation time = 5us***

***Local time***

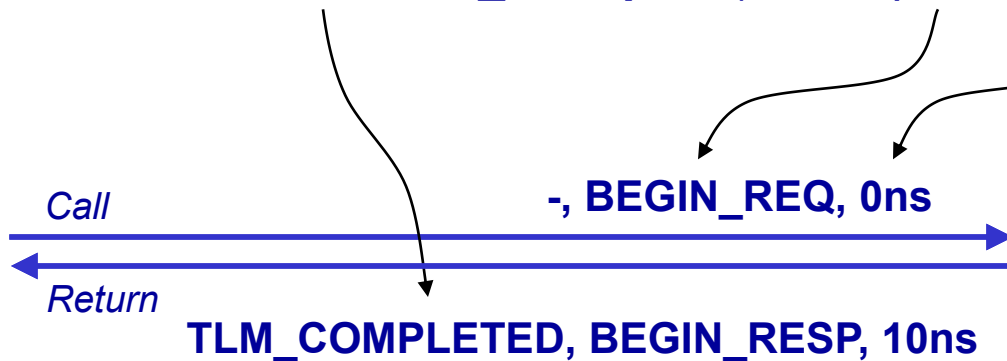
***+10ns***

***+20ns***

**= sc\_time\_stamp()**

**For temporal decoupling, local time is added to simulation time (explained on slides)**

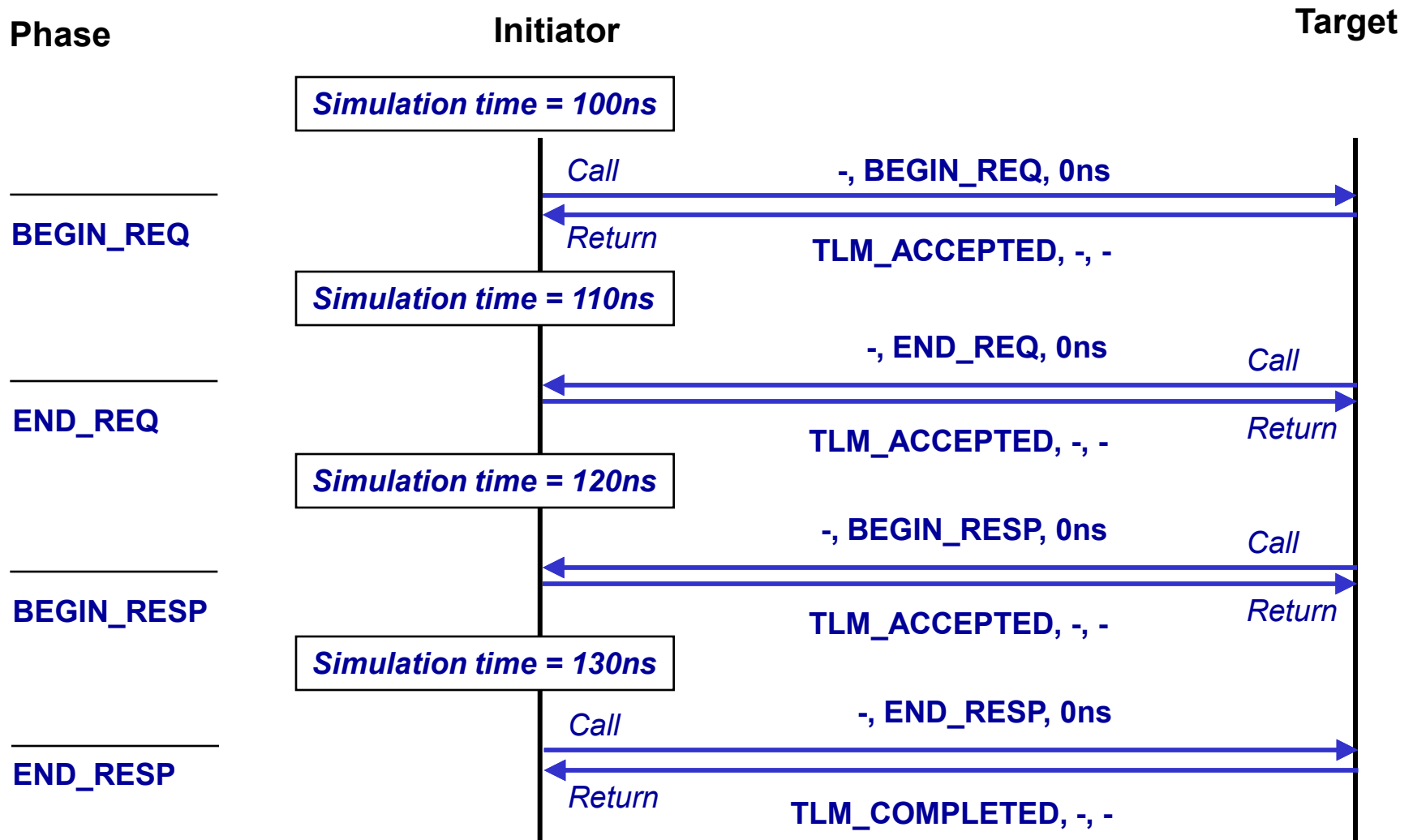
**status = nb\_transport ( trans, phase, delay ) ;**



**Arguments passed to function**

**Values returned from function**

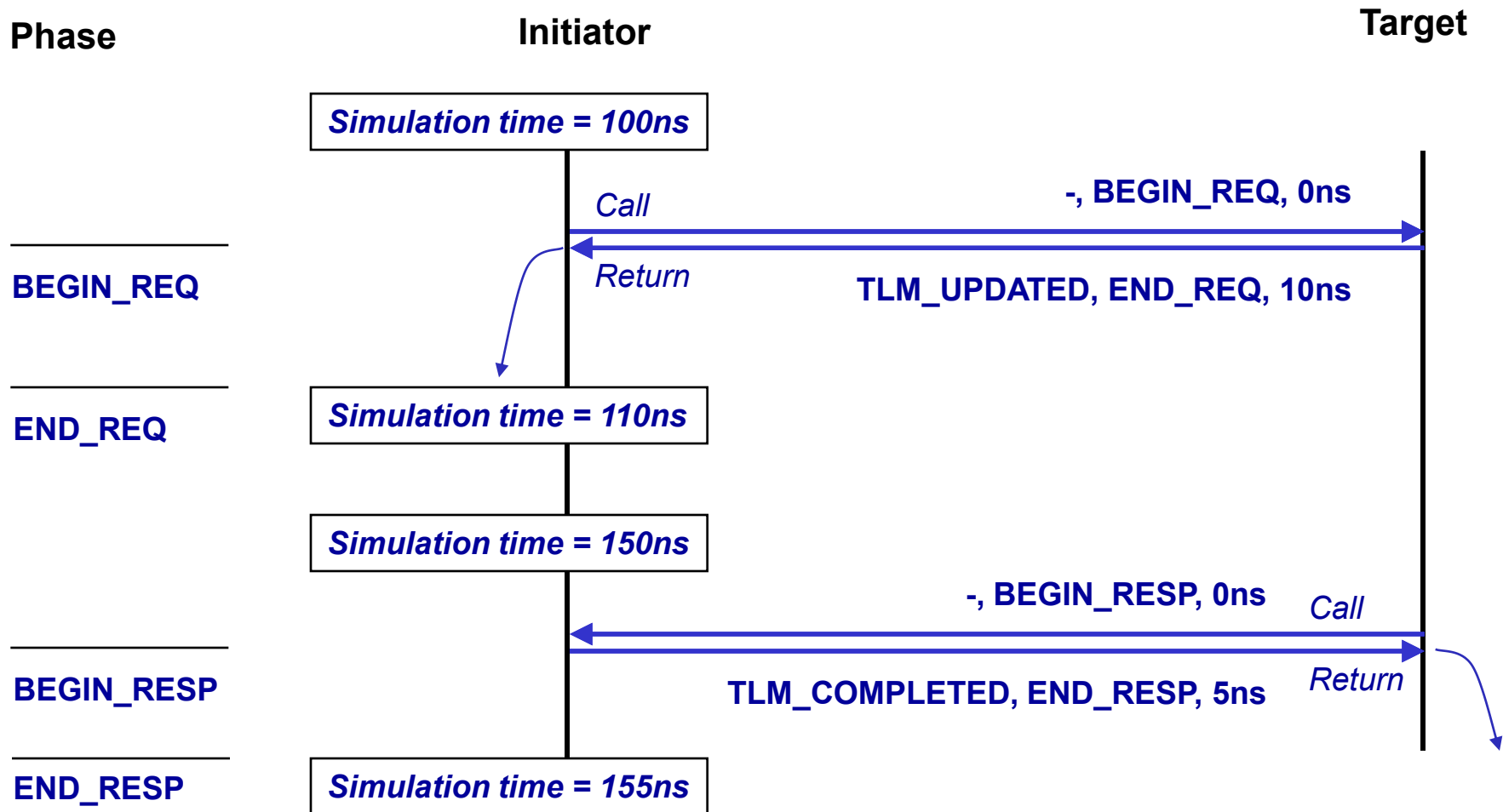
# Using the Backward Path



*Transaction accepted now, caller asked to wait*



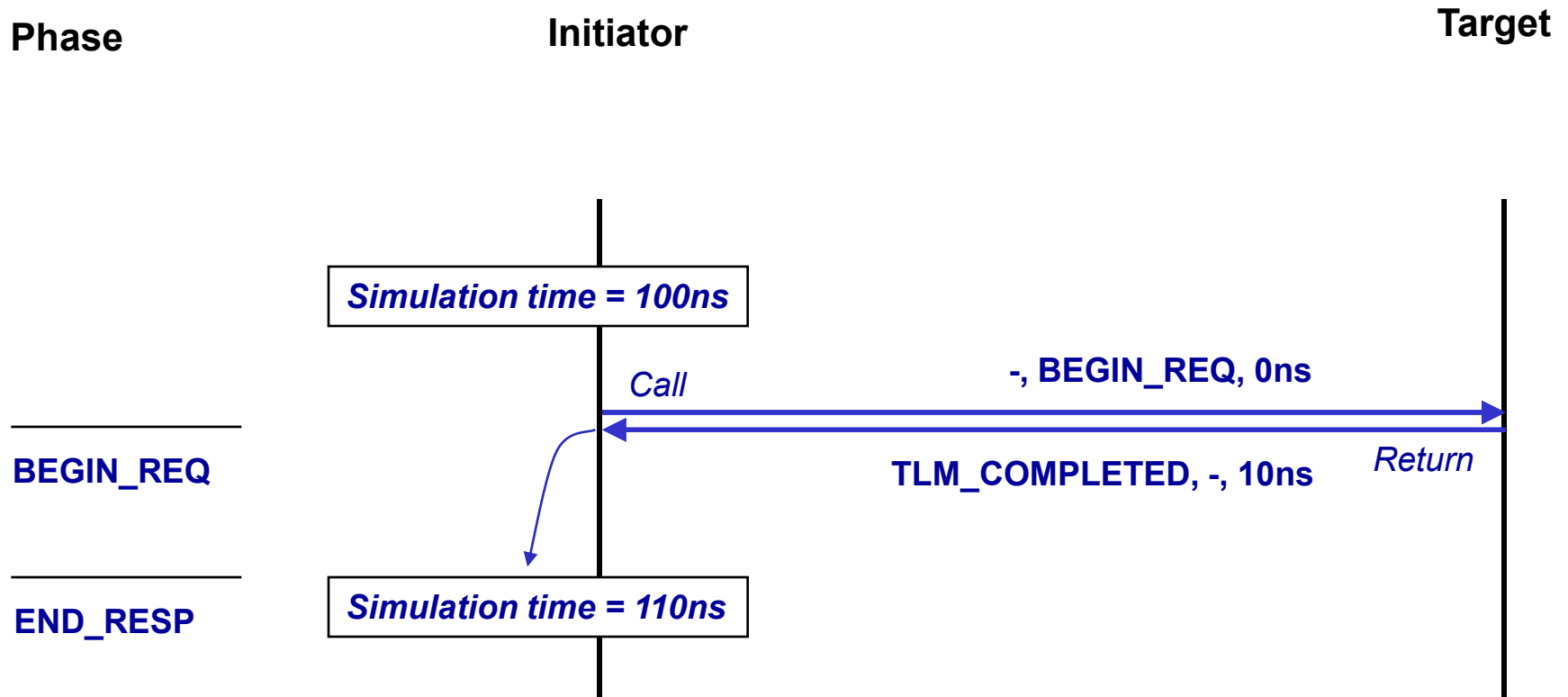
# Using the Return Path



*Callee annotates delay to next transition, caller waits*

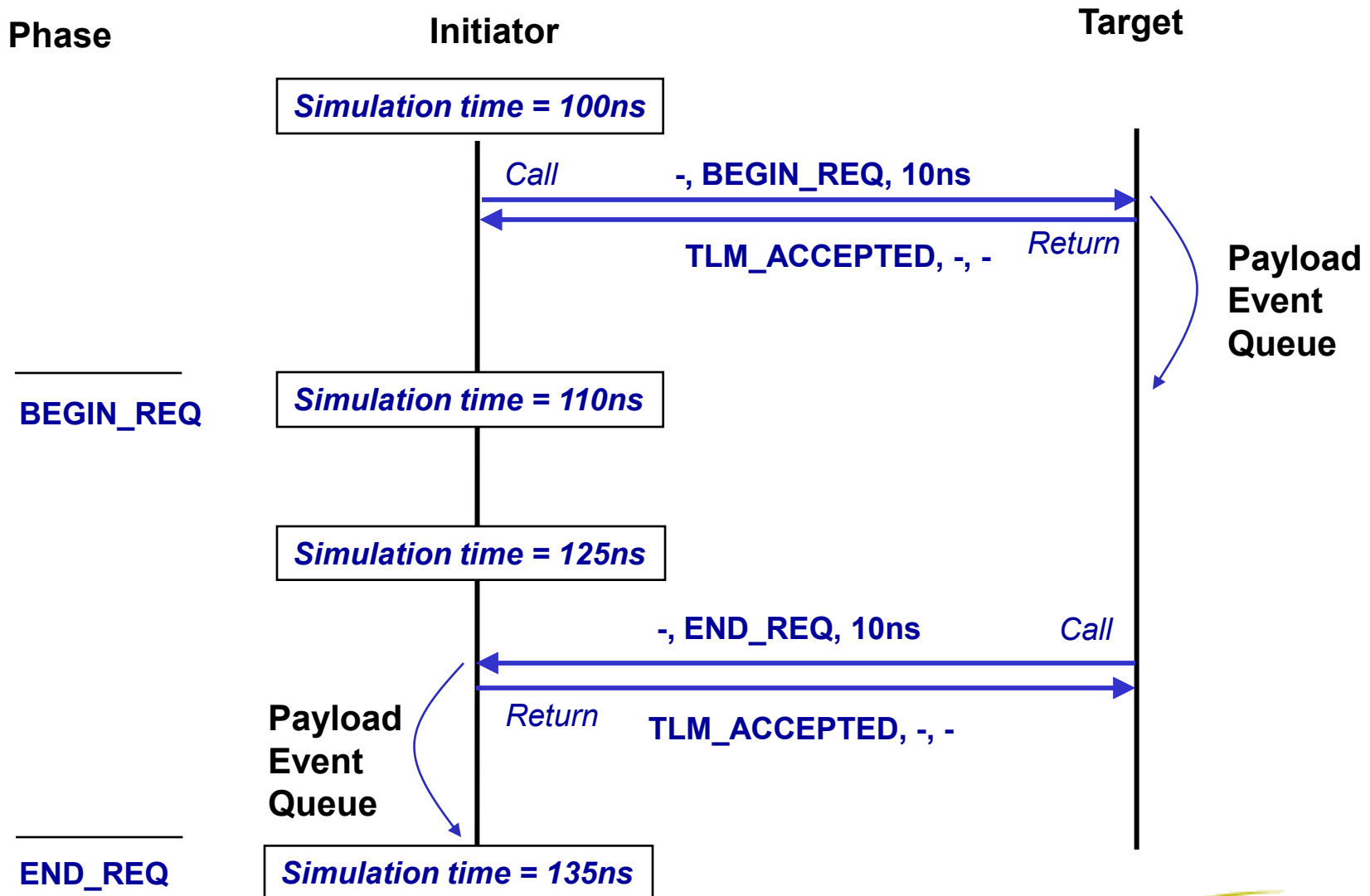


# Early Completion



*Callee annotates delay to next transition, caller waits*

# Timing Annotation



# Payload Event Queue

```
template <class PAYLOAD>
class peq_with_get : public sc_core::sc_object
{
public:
    peq_with_get( const char* name );

    void notify( PAYLOAD& trans, sc_core::sc_time& t );
    void notify( PAYLOAD& trans );

    transaction_type* get_next_transaction();
    sc_core::sc_event& get_event();
}
```

```
while (true) {
    wait( m_peq.get_event() );
    while ( (trans = m_peq.get_next_transaction()) != 0) {
        ...
    }
}
```





# DMI AND DEBUG INTERFACES

- ☐ Direct Memory Interface
- ☐ Debug Transport Interface

# DMI and Debug Transport

## ■ Direct Memory Interface

- Gives an initiator a direct pointer to memory in a target, e.g. an ISS
- By-passes the sockets and transport calls
- Read or write access by default
- Extensions may permit other kinds of access, e.g. security mode
- Target responsible for invalidating pointer

## ■ Debug Transport Interface

- Gives an initiator debug access to memory in a target
- Delay-free
- Side-effect-free

## ■ May share transactions with transport interface

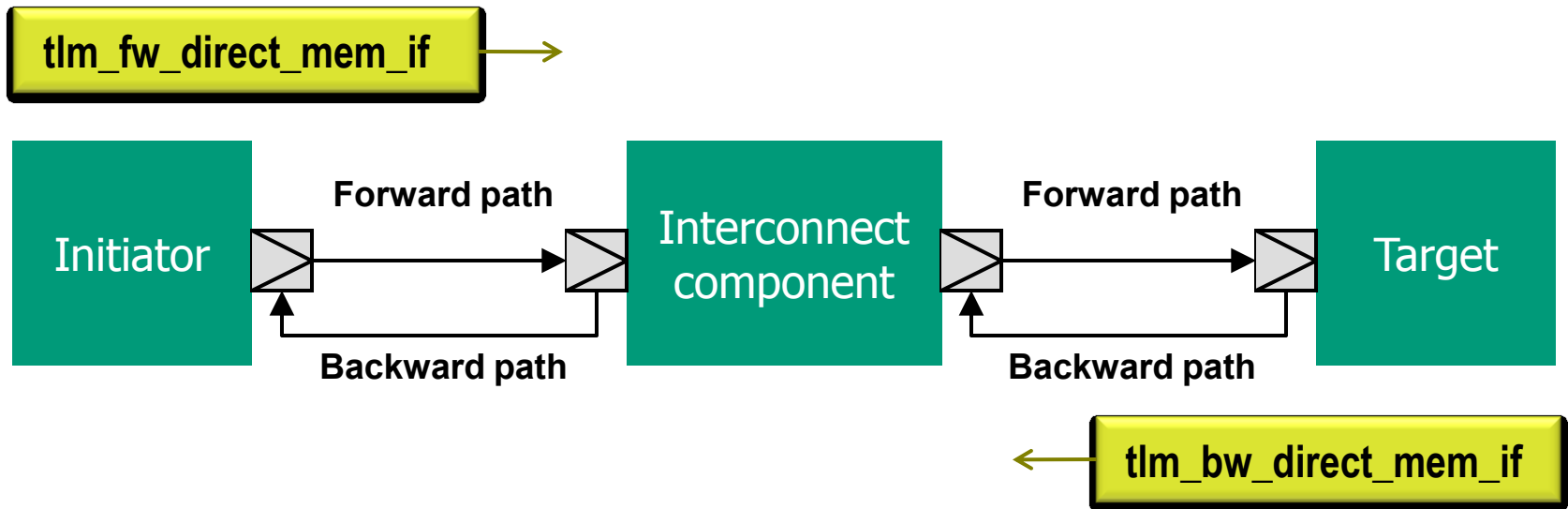


# Direct Memory Interface

*Access requested*      *Access granted*

↓                      ↓

```
status = get_direct_mem_ptr( transaction, dmi_data );
```



```
invalidate_direct_mem_ptr( start_range, end_range );
```

*Transport, DMI and debug may all use the generic payload*

*Interconnect may modify address and invalidated range*

# DMI Transaction and DMI Data

## DMI Transaction

Requests read or write access  
For a given address  
Permits extensions

## class tlm\_dmi

```
unsigned char*   dmi_ptr  
uint64           dmi_start_address  
uint64           dmi_end_address  
dmi_type_e       dmi_type;  
sc_time          read_latency  
sc_time          write_latency
```

*Direct memory pointer*



*Region granted for given access type*

*Read, write or read/write*



*Latencies to be observed by initiator*

# DMI Rules 1

- Initiator requests DMI from target at a given address
- DMI granted for a particular access type and a particular region
  - Target can only grant a single contiguous memory region containing given address
  - Target may grant an expanded memory region
  - Target may promote READ or WRITE request to READ\_WRITE
- Initiator may assume DMI pointer is valid until invalidated by target
- Initiator may keep a table of DMI pointers

# DMI Rules 2

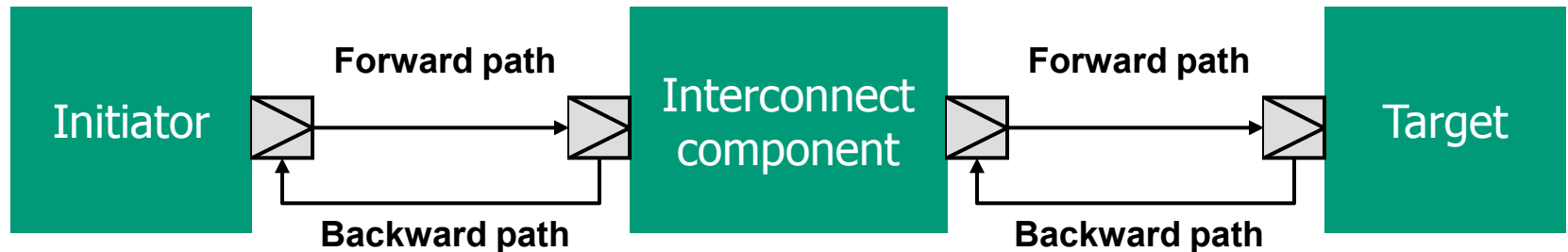
- DMI request and invalidation use same routing as regular transactions
- The invalidated address range may get expanded by the interconnect
- Target may grant DMI to multiple initiators (given multiple requests)
  - and a single invalidate may knock out multiple pointers in multiple initiators
- Use the Generic Payload DMI hint (described later)
- Only makes sense with loosely-timed models

# Debug Transport Interface

```
num_bytes = transport_dbg( transaction );
```

```
tlm_transport_dbg_if
```

Command  
Address  
Data pointer  
Data length  
Extensions



*Uses forward path only*

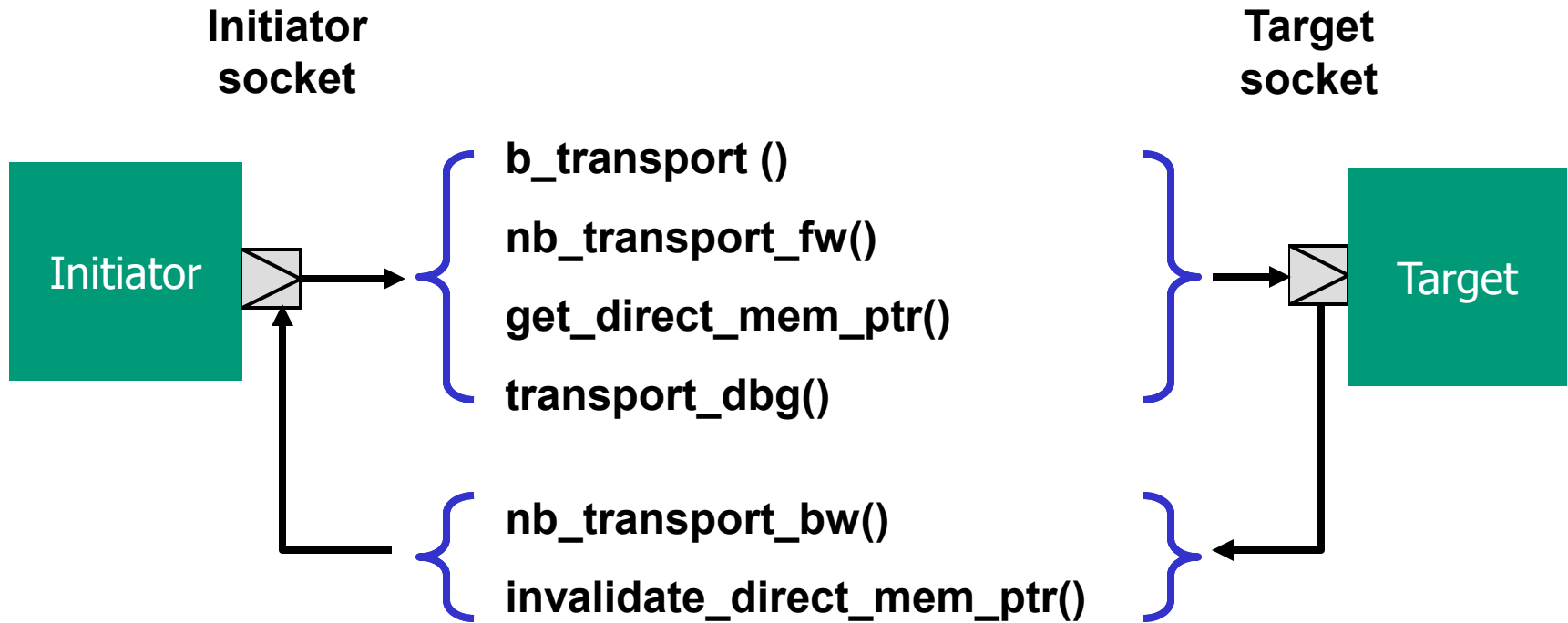
*Interconnect may modify address, target reads or writes data*

# SOCKETS

- ☐ Initiator and target sockets
- ☐ Simple sockets
- ☐ Tagged sockets
- ☐ Multi-port sockets



# Initiator and Target Sockets



*Sockets provide fw and bw paths and group interfaces*

# Benefit of Sockets

- Group the transport, DMI and debug transport interfaces
  - Bind forward and backward paths with a single call
  - Strong connection checking
  - Have a bus width parameter
- 
- Using core interfaces without sockets is not recommended

# Sockets and Transaction Types

- All interfaces templated on transaction type
- Use the generic payload and base protocol for interoperability
  - Use with transport, DMI and debug transport
  - Supports extensions
  - Even supports extended commands and phases
  - Ignorable extensions allow interoperability
  - Mechanism to disallow socket binding for non-ignorable extensions
  - Described later



# Standard Socket Classes

```
template < unsigned int  BUSWIDTH = 32,  
          typename TYPES = tlm_base_protocol_types,  
          int N          = 1,  
          sc_core::sc_port_policy POL = sc_core::SC_ONE_OR_MORE_BOUND >
```

```
class tlm_initiator_socket
```

```
...
```

```
class tlm_target_socket
```

- Part of the interoperability layer
- Initiator socket must be bound to an object that implements entire backward interface
- Target socket must be bound to an object that implements entire forward interface
- Can mix blocking and non-blocking calls – target must support both together
- Allow hierarchical binding



# Socket Binding Example 1

```
struct Initiator: sc_module, tlm::tlm_bw_transport_if<>
{
    tlm::tlm_initiator_socket<> init_socket;

    SC_CTOR(Initiator) : init_socket("init_socket") {
        SC_THREAD(thread);
        init_socket.bind( *this );
    }

    void thread() { ...
        init_socket->b_transport( trans, delay );
        init_socket->nb_transport_fw( trans, phase, delay );
        init_socket->get_direct_mem_ptr( trans, dmi_data );
        init_socket->transport_dbg( trans );
    }

    virtual tlm::tlm_sync_enum nb_transport_bw( ... ) { ... }
    virtual void invalidate_direct_mem_ptr( ... ) { ... }
};
```

*Combined interface required by socket*

*Protocol type defaults to base protocol*

*Initiator socket bound to initiator itself*

*Calls on forward path*

*Methods for backward path*

# Socket Binding Example 2

```
struct Target: sc_module, tlm::tlm_fw_transport_if<>
{
    tlm::tlm_target_socket<> targ_socket;

    SC_CTOR(Target) : targ_socket("targ_socket") {
        targ_socket.bind( *this );
    }
    virtual void b_transport( ... ) { ... }
    virtual tlm::tlm_sync_enum nb_transport_fw( ... ) { ... }
    virtual bool get_direct_mem_ptr( ... ) { ... }
    virtual unsigned int transport_dbg( ... ) { ... }
};
```

*Combined interface required by socket*

*Protocol type default to base protocol*

*Target socket bound to target itself*

*Methods for forward path*

```
SC_MODULE(Top) {
    Initiator *init;
    Target *targ;
    SC_CTOR(Top) {
        init = new Initiator("init");
        targ = new Target("targ");
        init->init_socket.bind( targ->targ_socket );
    }
};
```

*Bind initiator socket to target socket*



# Convenience Sockets

- The “simple” sockets
  - `simple_initiator_socket` and `simple_target_socket`
    - In namespace `tlm_utils`
    - Derived from `tlm_initiator_socket` and `tlm_target_socket`
  - “simple” because they are simple to use
    - Do not bind sockets to objects (implementations)
    - Instead, register methods with each socket
    - Do not allow hierarchical binding
  - Not obliged to register both `b_transport` and `nb_transport`
    - Automatic conversion (assumes base protocol)
      - Variant with no conversion – `passthrough_target_socket`



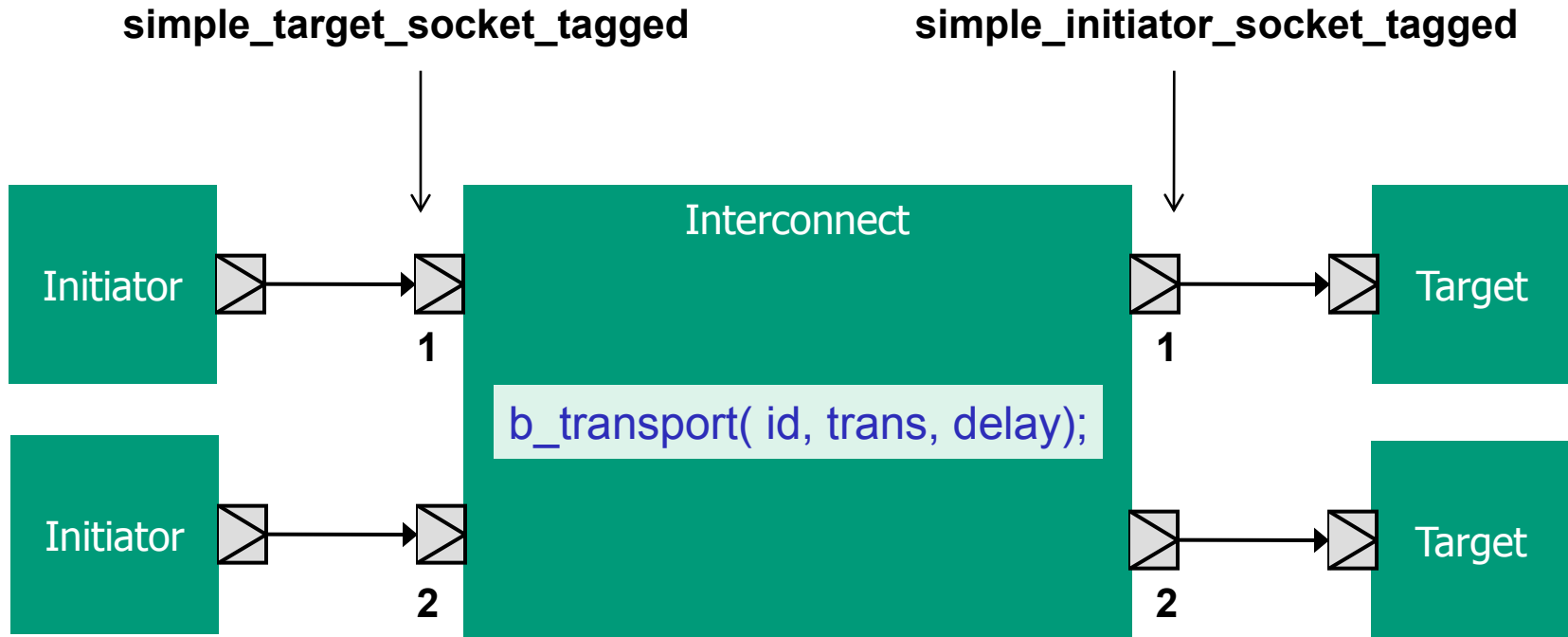
# Simple Socket Example

```
struct Interconnect : sc_module
{
    tlm_utils::simple_target_socket<Interconnect>  targ_socket;
    tlm_utils::simple_initiator_socket<Interconnect> init_socket;

    SC_CTOR(Interconnect) : targ_socket("targ_socket"), init_socket("init_socket")
    {
        targ_socket.register_nb_transport_fw(    this, &Interconnect::nb_transport_fw);
        targ_socket.register_b_transport(        this, &Interconnect::b_transport);
        targ_socket.register_get_direct_mem_ptr(this, &Interconnect::get_direct_mem_ptr);
        targ_socket.register_transport_dbg(      this, &Interconnect::transport_dbg);
        init_socket.register_nb_transport_bw(    this, &Interconnect::nb_transport_bw);
        init_socket.register_invalidate_direct_mem_ptr(
                                                    this, &Interconnect::invalidate_direct_mem_ptr);
    }
    virtual void b_transport( ... );
    virtual tlm::tlm_sync_enum nb_transport_fw( ... );
    virtual bool get_direct_mem_ptr( ... );
    virtual unsigned int transport_dbg( ... );
    virtual tlm::tlm_sync_enum nb_transport_bw( ... );
    virtual void invalidate_direct_mem_ptr( ... );
};
```



# Tagged Simple Sockets



*Distinguish origin of incoming transactions using socket id*

# Tagged Simple Socket Example

```
#include "tlm_utils/simple_initiator_socket.h"
#include "tlm_utils/simple_target_socket.h"

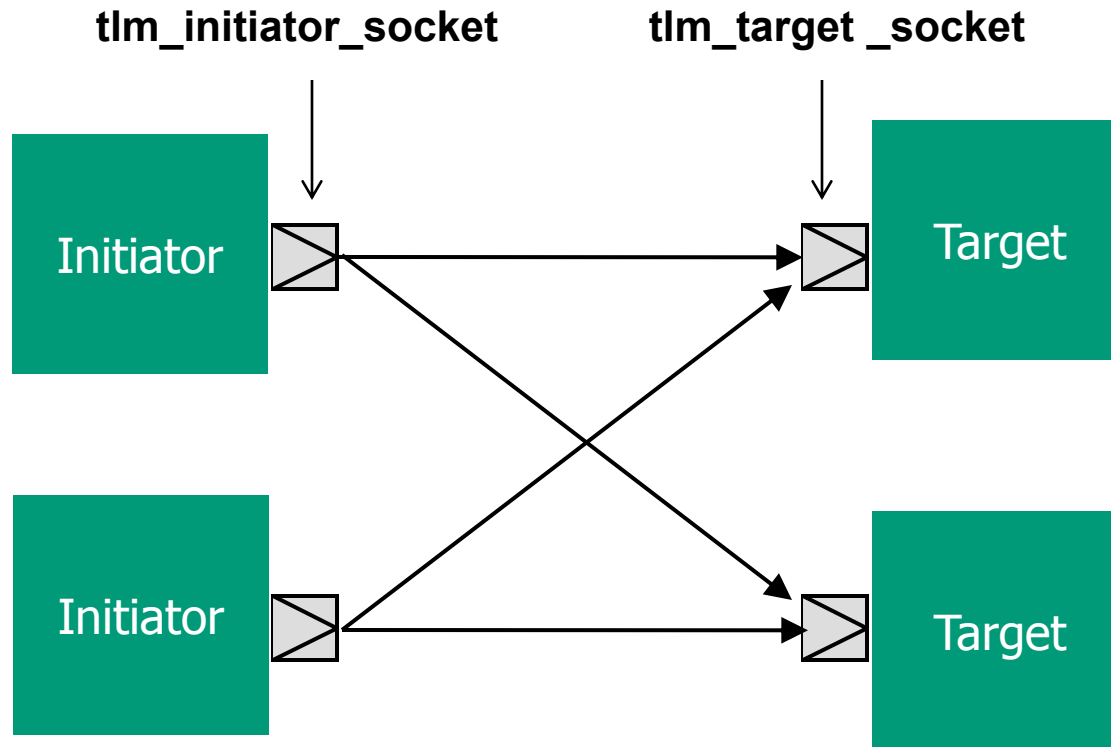
template<unsigned int N_INITIATORS, unsigned int N_TARGETS>
struct Bus: sc_module
{
    tlm_utils::simple_target_socket_tagged<Bus>*  targ_socket [N_INITIATORS];
    tlm_utils::simple_initiator_socket_tagged<Bus>*  init_socket [N_TARGETS];

    SC_CTOR(Bus) {
        for (unsigned int id = 0; i < N_INITIATORS; i++) {
            targ_socket[id] = new tlm_utils::simple_target_socket_tagged<Bus>(txt);

            targ_socket[id]->register_b_transport this, &Bus::b_transport, id );
            ...

        virtual void b_transport( int id, tlm::tlm_generic_payload& trans, sc_time& delay );
        ...
    };
};
```

# Many-to-many Binding



`init_socket[0]->b_transport(...)`  
`init_socket[1]->b_transport(...)`

`target_socket[0]->nb_transport_bw(...)`  
`target_socket[1]->nb_transport_bw(...)`

- Multi-ports – can bind many-to-many, but incoming calls are anonymous

# Multi-port Convenience Sockets

- `multi_passthrough_initiator_socket`
- `multi_passthrough_target_socket`
- Many-to-many socket bindings
- Method calls tagged with multi-port index value

# Socket Summary

class	Register callbacks?	Multi-ports?	b <-> nb conversion?	Tagged?
tlm_initiator_socket	no	yes	-	no
tlm_target_socket	no	yes	no	no
simple_initiator_socket	yes	no	-	no
simple_initiator_socket_tagged	yes	no	-	yes
simple_target_socket	yes	no	yes	no
simple_target_socket_tagged	yes	no	yes	yes
passthrough_target_socket	yes	no	no	no
passthrough_target_socket_tagged	yes	no	no	yes
multi_passthrough_initiator_socket	yes	yes	-	yes
multi_passthrough_target_socket	yes	yes	no	yes



# THE GENERIC PAYLOAD

- ☐ Attributes
- ☐ Memory management
- ☐ Response status
- ☐ Endianness
- ☐ Extensions

# The Generic Payload

- **Typical attributes of memory-mapped busses**
  - command, address, data, byte enables, single word transfers, burst transfers, streaming, response status
- **Off-the-shelf general purpose payload**
  - for abstract bus modeling
  - *ignorable* extensions allow full interoperability
- **Used to model specific bus protocols**
  - mandatory static extensions
  - compile-time type checking to avoid incompatibility
  - low implementation cost when bridging protocols

*Specific protocols can use the same generic payload machinery*



# Generic Payload Attributes

Attribute	Type	Modifiable?	
Command	tlm_command	No	
Address	uint64	Interconnect only	
Data pointer	unsigned char*	No (array – yes)	<i>Array owned by initiator</i>
Data length	unsigned int	No	
Byte enable pointer	unsigned char*	No (array – yes)	<i>Array owned by initiator</i>
Byte enable length	unsigned int	No	
Streaming width	unsigned int	No	
DMI hint	bool	Yes	<i>Try DMI !</i>
Response status	tlm_response_status	Target only	
Extensions	(tlm_extension_base*)[ ]	Yes	<i>Consider memory management</i>



# class tlm\_generic\_payload

```
class tlm_generic_payload {  
public:
```

*Not a template*

```
    // Constructors, memory management
```

```
    tlm_generic_payload () ;
```

```
    tlm_generic_payload(tlm_mm_interface& mm) ;
```

*Construct & set mm*

```
    virtual ~tlm_generic_payload ();
```

*Frees all extensions*

```
    void reset();
```

*Frees mm'd extensions*

```
    void set_mm(tlm_mm_interface* mm);
```

*mm is optional*

```
    bool has_mm();
```

```
    void acquire();
```

*Incr reference count*

```
    void release();
```

*Decr reference count, 0 => free trans*

```
    int get_ref_count();
```

```
    void deep_copy_into(tlm_generic_payload& other) const;
```

```
    ...
```

```
};
```



# Memory Management Rules

- **b\_transport** – memory managed by initiator, or reference counting (set\_mm)
- **nb\_transport** – reference counting only
  - Reference counting requires heap allocation
  - Transaction automatically freed when reference count == 0
  - free() can be overridden in memory manager for transactions
  - free() can be overridden for extensions
- When **b\_transport** calls **nb\_transport**, must add reference counting
  - Can only return when reference count == 0
- **b\_transport** can check for reference counting, or assume it could be present



# Command, Address and Data

```
enum tlm_command {  
    TLM_READ_COMMAND,  
    TLM_WRITE_COMMAND,  
    TLM_IGNORE_COMMAND  
};
```

*Copy from target to data array*  
*Copy from data array to target*  
*Neither, but may use extensions*

```
tlm_command    get_command() const ;  
void           set_command( const tlm_command command ) ;
```

```
sc_dt::uint64  get_address() const;  
void           set_address( const sc_dt::uint64 address );
```

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

*Data array owned by initiator*

```
unsigned int   get_data_length() const;  
void           set_data_length( const unsigned int length );
```

*Number of bytes in data array*



# Response Status

enum tlm_response_status	Meaning
TLM_OK_RESPONSE	Successful
TLM_INCOMPLETE_RESPONSE	Transaction not delivered to target. (Default)
TLM_ADDRESS_ERROR_RESPONSE	Unable to act on address
TLM_COMMAND_ERROR_RESPONSE	Unable to execute command
TLM_BURST_ERROR_RESPONSE	Unable to act on data length or streaming width
TLM_BYTE_ENABLE_ERROR_RESPONSE	Unable to act on byte enable
TLM_GENERIC_ERROR_RESPONSE	Any other error

# The Standard Error Response

- A target shall either
  - Execute the command and set TLM\_OK\_RESPONSE
  - Set the response status attribute to an error response
  - Call the SystemC report handler and set TLM\_OK\_RESPONSE
  
- Many corner cases
  - e.g. a target that ignores the data when executing a write – OK
  - e.g. a simulation monitor that logs out-of-range addresses – OK
  - e.g. a target that cannot support byte enables - ERROR



# Generic Payload Example 1

```
void thread_process() { // The initiator
    tlm::tlm_generic_payload trans;
    sc_time delay = SC_ZERO_TIME;
```

*Would usually pool transactions*

```
    trans.set_command( tlm::TLM_WRITE_COMMAND );
    trans.set_data_length( 4 );
    trans.set_byte_enable_ptr( 0 );
    trans.set_streaming_width( 4 );
```

```
    for ( int i = 0; i < RUN_LENGTH; i += 4 ) {
        int word = i;
        trans.set_address( i );
        trans.set_data_ptr( (unsigned char*)( &word ) );
        trans.set_response_status( tlm::TLM_INCOMPLETE_RESPONSE );
```

```
        init_socket->b_transport( trans, delay );
```

```
        if ( trans.get_response_status() <= 0 )
            SC_REPORT_ERROR("TLM2", trans.get_response_string().c_str());
```

```
        ...
```

```
    }
```



# Generic Payload Example 2

```
virtual void b_transport( // The target
    tlm::tlm_generic_payload& trans, sc_core::sc_time& t) {

    tlm::tlm_command  cmd  = trans.get_command();
    sc_dt::uint64     adr  = trans.get_address();
    unsigned char*    ptr  = trans.get_data_ptr();
    unsigned int      len  = trans.get_data_length();
    unsigned char*    byt  = trans.get_byte_enable_ptr();
    unsigned int      wid  = trans.get_streaming_width();
```

```
    if (adr+len > m_length) { Check for storage overflow
        trans.set_response_status( tlm::TLM_ADDRESS_ERROR_RESPONSE );
        return;
    }
```

```
    if (byt) { Unable to support byte enable
        trans.set_response_status( tlm::TLM_BYTE_ENABLE_ERROR_RESPONSE );
        return;
    }
```

```
    if (wid != 0 && wid < len) { Unable to support streaming
        trans.set_response_status( tlm::TLM_BURST_ERROR_RESPONSE );
        return;
    }
```



# Generic Payload Example 3

```
virtual void b_transport( // The target
    tlm::tlm_generic_payload& trans, sc_core::sc_time& t) {
    ...
    ...
    if ( cmd == tlm::TLM_WRITE_COMMAND )

        memcpy( &m_storage[adr], ptr, len );

    else if ( cmd == tlm::TLM_READ_COMMAND )

        memcpy( ptr, &m_storage[adr], len );

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

*Execute command*

*Successful completion*



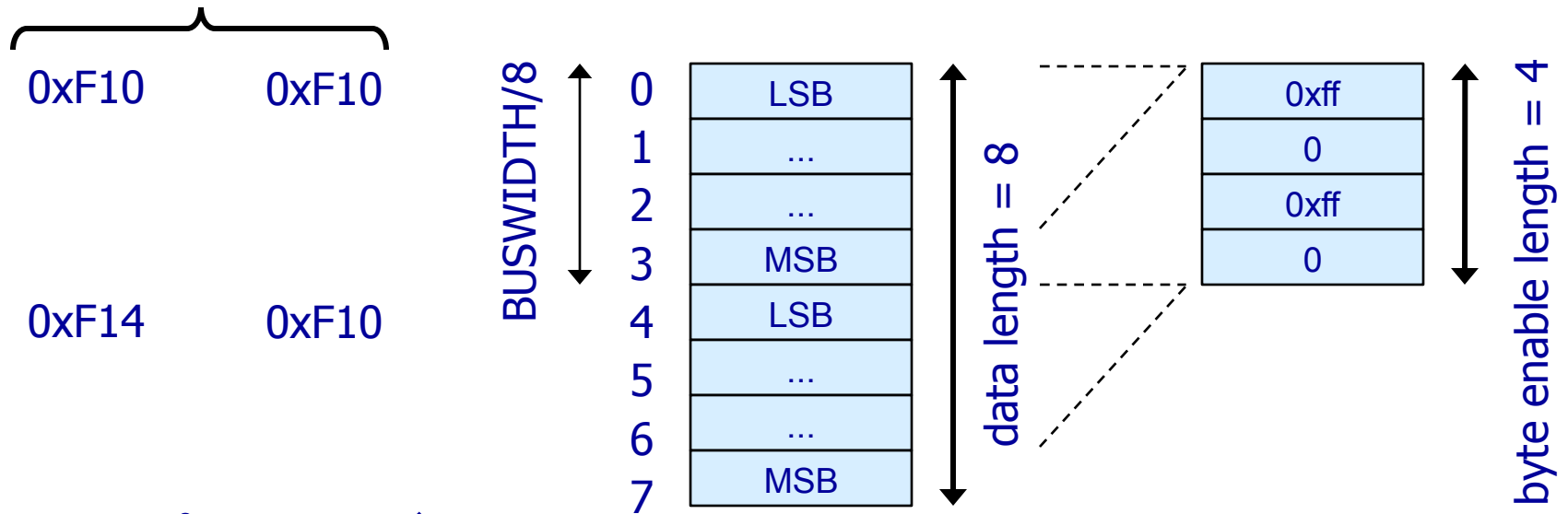
# Byte Enables and Streaming

uint64  
address

unsigned int  
index

unsigned char\*  
data

unsigned char\*  
byte\_enable



streaming width = 8

streaming width = 4

*1-enable-per-byte*

*Byte enables applied repeatedly*

*Data interpreted using BUSWIDTH*

*Streaming width > 0 => wrap address*

```
#define TLM_BYTE_DISABLED 0x0
#define TLM_BYTE_ENABLED 0xff
```



# Byte Enable Example 1

// The initiator

```
void thread_process() {  
    tlm::tlm_generic_payload trans;  
    sc_time delay;  
  
    static word_t byte_enable_mask = 0x0000ffff; Uses host-endianness MSB..LSB  
  
    trans.set_byte_enable_ptr(  
        reinterpret_cast<unsigned char*>( &byte_enable_mask ) );  
  
    trans.set_byte_enable_length( 4 );  
  
    trans.set_command( tlm::TLM_WRITE_COMMAND );  
    trans.set_data_length( 4 );  
    ...  
    for (int i = 0; i < RUN_LENGTH; i += 4) {  
        trans.set_address( i );  
        trans.set_data_ptr( (unsigned char*)&word );  
  
        init_socket->b_transport(trans, delay);  
    }  
    ...  
}
```

# Byte Enable Example 2

```
virtual void b_transport( tlm::tlm_generic_payload& trans, sc_core::sc_time& t) // The target
{
    ...
    unsigned char*      byt    = trans.get_byte_enable_ptr();
    unsigned int         bel    = trans.get_byte_enable_length();
    ...
    if (cmd == tlm::TLM_WRITE_COMMAND) {
        if (byt) {
            for (unsigned int i = 0; i < len; i++)
                if ( byt[ i % bel ] == TLM_BYTE_ENABLED)
                    m_storage[adr+i] = ptr[i];
        } else
            memcpy(&m_storage[adr], ptr, len);

    } else if (cmd == tlm::TLM_READ_COMMAND) {
        if (byt) {
            trans.set_response_status( tlm::TLM_BYTE_ENABLE_ERROR_RESPONSE );
            return tlm::TLM_COMPLETED;
        } else
            memcpy(ptr, &m_storage[adr], len);
    }
}
```

*Byte enable applied repeatedly  
byt[i] corresponds to ptr[i]*

*No byte enables*

*Target does not support read with  
byte enables*



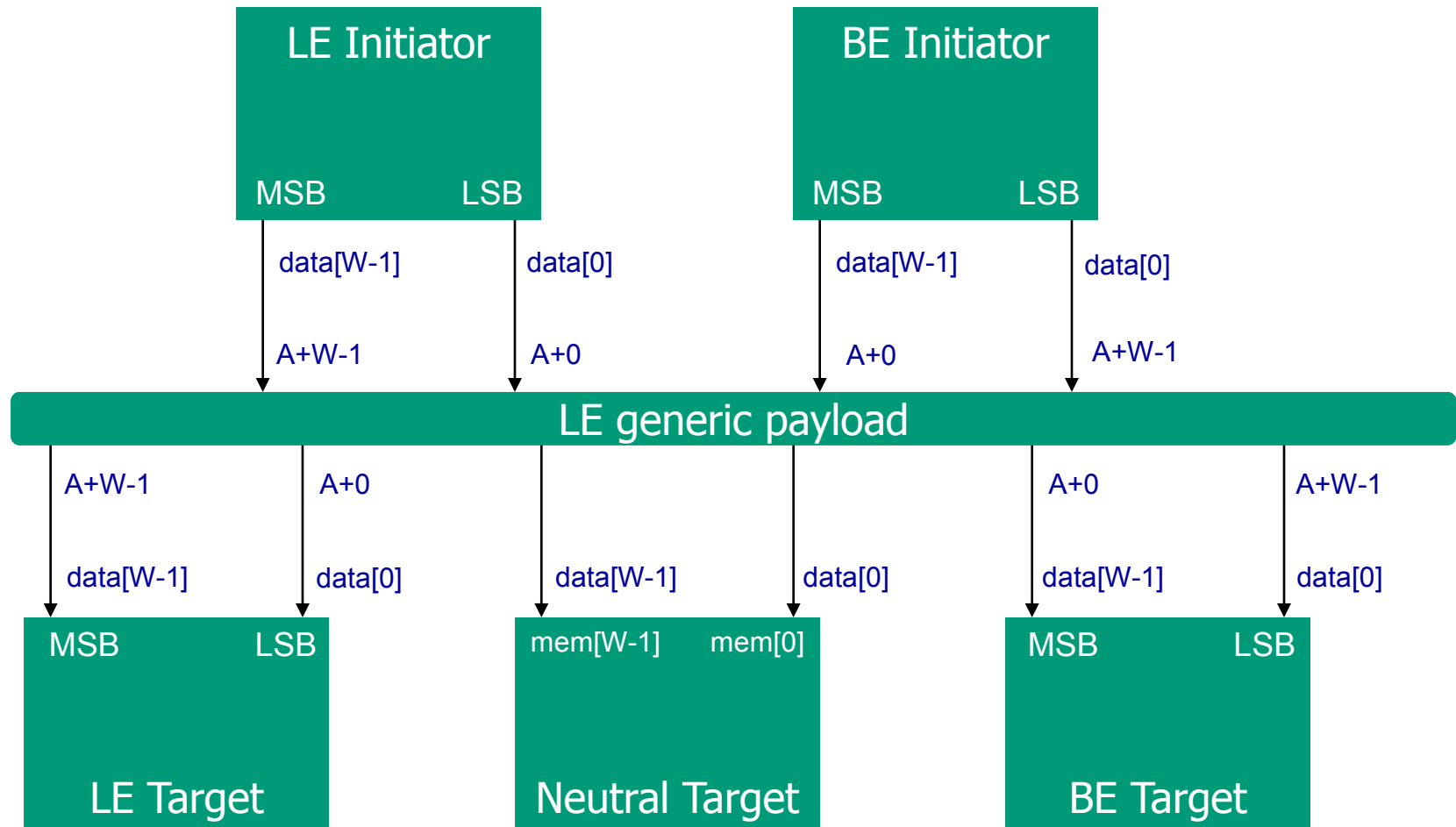
# Endianness

- Designed to maximize simulation speed
- Words in data array are host-endian
- Effective word length  $W = (\text{BUSWIDTH} + 7) / 8$
- Initiators and targets connected LSB-to-LSB, MSB-to-MSB
- Most efficient when *everything* is modeled host-endian
- Width-conversions with same endianness as host are *free*

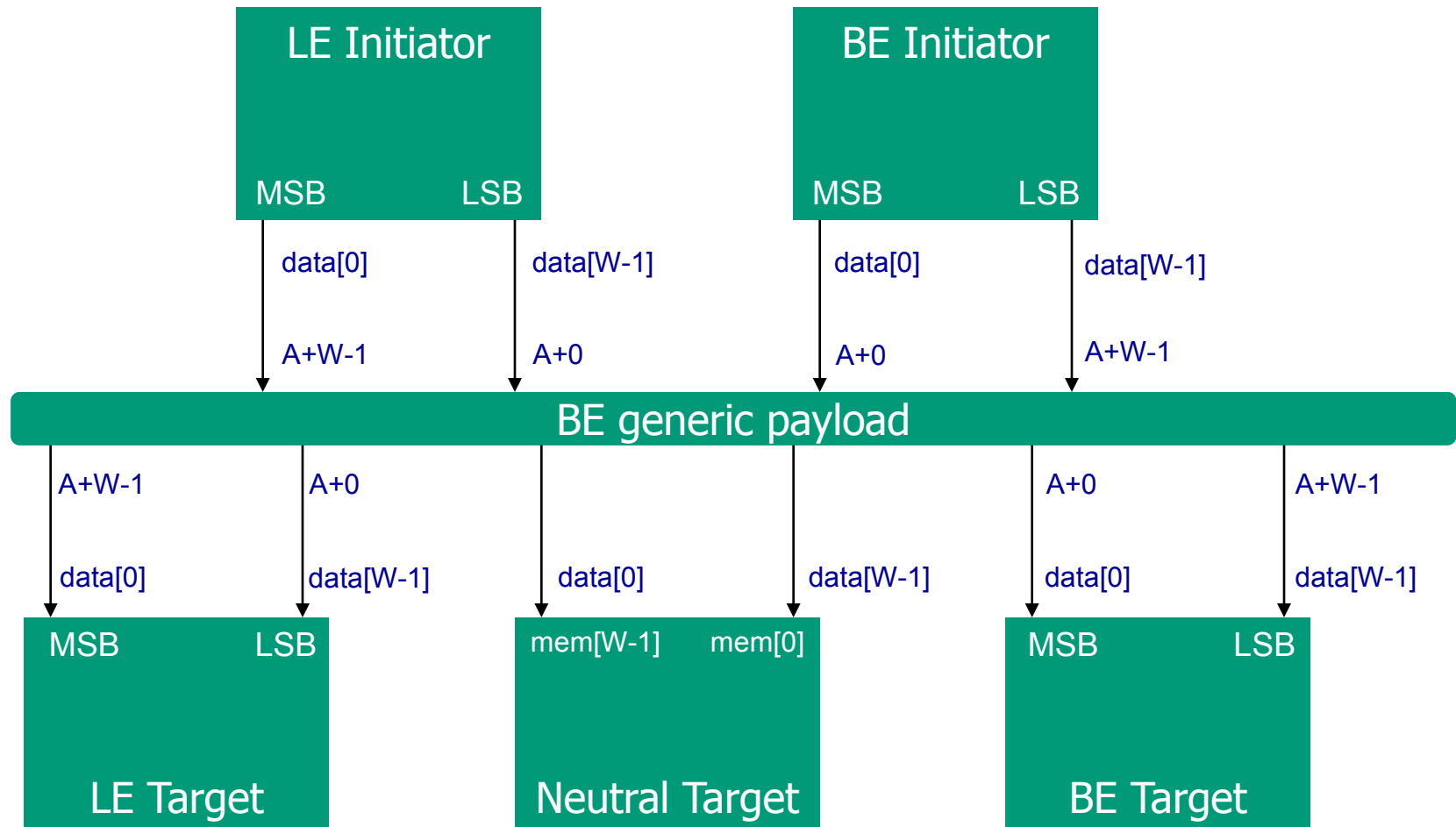
*Common transfers can use memcpy, width conversions don't modify transaction*



# Little-endian host



# Big-endian host



# Part-word Transfers

Little-endian host

$W = 4$

length = 6

address = A

data =

1	A
2	
3	
4	
5	A+4
6	

Big-endian host

$W = 4$

length = 6

address = A

data =

byte enable =

4	A	0xff
3		0xff
2		0xff
1		0xff
	A+4	0
		0
6		0xff
5		0xff

# Generic Payload Extension Methods

- Generic payload has an array-of-pointers to extensions
- One pointer per extension type
- Every transaction can potentially carry every extension type
- Flexible mechanism

template <typename T> T\* **set\_extension** ( T\* ext ); *Sticky extn*

template <typename T> T\* **set\_auto\_extension** ( T\* ext ); *Freed by ref counting*

template <typename T> T\* **get\_extension**() const;

template <typename T> void **clear\_extension** (); *Clears pointer, not extn object*

template <typename T> void **release\_extension** ();  
*mm => convert to auto  
no mm => free extn object*



# Extension Example

```
struct my_extension : tlm_extension<my_extension>
{
    my_extension() : id(0) {}
    tlm_extension_base* clone() const { ... }
    virtual void copy_from(tlm_extension_base const &ext) { ... }
    int id;
};
...
```

*User-defined extension*

*Pure virtual methods*

```
tlm_generic_payload* trans = new tlm_generic_payload( mem_mgr );
trans->acquire();
```

*Heap allocation*

*Reference counting*

```
my_extension* ext = new my_extension;
ext->id = 1;
trans.set_extension( ext );
```

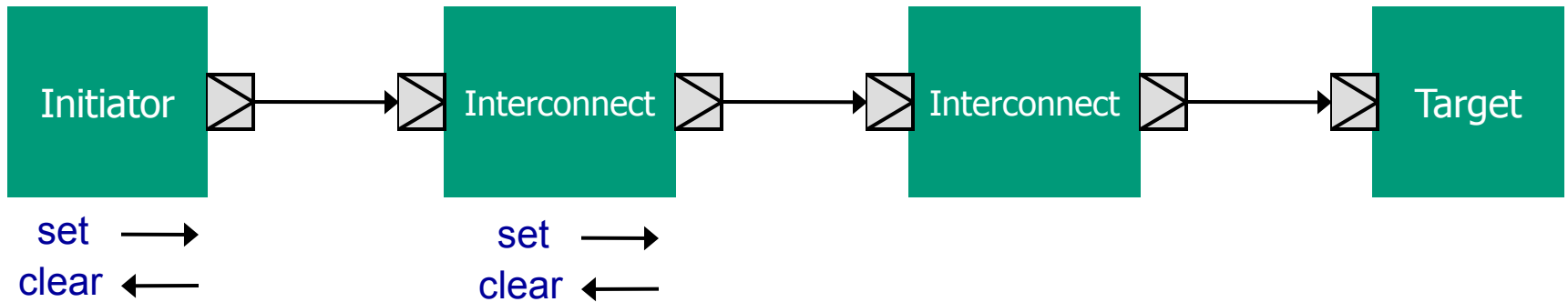
```
socket->nb_transport_fw( *trans, phase, delay );
trans.release_extension<my_extension>();
```

*Freed when ref count = 0*

```
trans->release();
```

*Trans and extn freed*

# Extension Rules



- Extensions should only be used downstream of the setter
- Whoever sets the extension should clear the extension
- If not reference counting, use `set_extension` / `clear_extension`
- If reference counting, use `set_auto_extension`
- For sticky extensions, use `set_extension`
- Within `b_transport`, either check or use `set_extension` / `release_extension`

# Instance-Specific Extensions

```
#include "tlm_utils/instance_specific_extensions.h"
```

```
struct my_extn: tlm_utils::instance_specific_extension<my_extn> {  
    int num;  
};
```

```
class Interconnect : sc_module {  
    tlm_utils::instance_specific_extension_accessor accessor;  
  
    virtual tlm::tlm_sync_enum nb_transport_fw( ... )  
    {  
        my_extn* extn;  
        accessor(trans).get_extension(extn);  
        if (extn) {  
            cout << extn->num << endl;  
            accessor(trans).clear_extension(extn);  
        } else {  
            extn = new my_extn;  
            extn->num = count++;  
            accessor(trans).set_extension(extn);  
            ....  
        }  
    }  
};
```

*Gives unique extensions  
per module instance*

# THE BASE PROTOCOL

- ☐ tlm\_phase
- ☐ Base protocol rules
- ☐ Base protocol phases
- ☐ Defining new protocol types

# Base Protocol - Coding Styles

- **Loosely-timed is typically**
  - Blocking transport interface, forward and return path
  - 2 timing points
  - Temporal decoupling and the quantum keeper
  - Direct memory interface
- **Approximately-timed is typically**
  - Non-blocking transport interface, forward and backward paths
  - 4 phases
  - Payload event queues
- **Loosely-timed and approximately-timed are only coding styles**
- **The base protocol defines rules for phases and call order**



# Base Protocol and tlm\_phase

- The base protocol = tlm\_generic\_payload + tlm\_phase
- tlm\_phase has 4 phases, but can be extended to add new phases

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

```
class tlm_phase {  
public:  
    tlm_phase();  
    tlm_phase( unsigned int id );  
    tlm_phase( const tlm_phase_enum& standard );  
    tlm_phase& operator= ( const tlm_phase_enum& standard );  
    operator unsigned int() const;  
};
```

```
#define DECLARE_EXTENDED_PHASE(name_arg) \  
class tlm_phase_##name_arg : public tlm::tlm_phase { \  
...  
}
```



# Base Protocol Rules 1

- **Base protocol phases**
  - BEGIN\_REQ → END\_REQ → BEGIN\_RESP → END\_RESP
  - Must occur in non-decreasing simulation time order
  - Only permitted one outstanding request or response per socket
  - Phase must change with each call (other than ignorable phases)
  - May complete early
- **Generic payload memory management rules**
- **Extensions must be ignorable**
- **Target is obliged to handle mixed b\_transport / nb\_transport**
- **Write response must come from target**



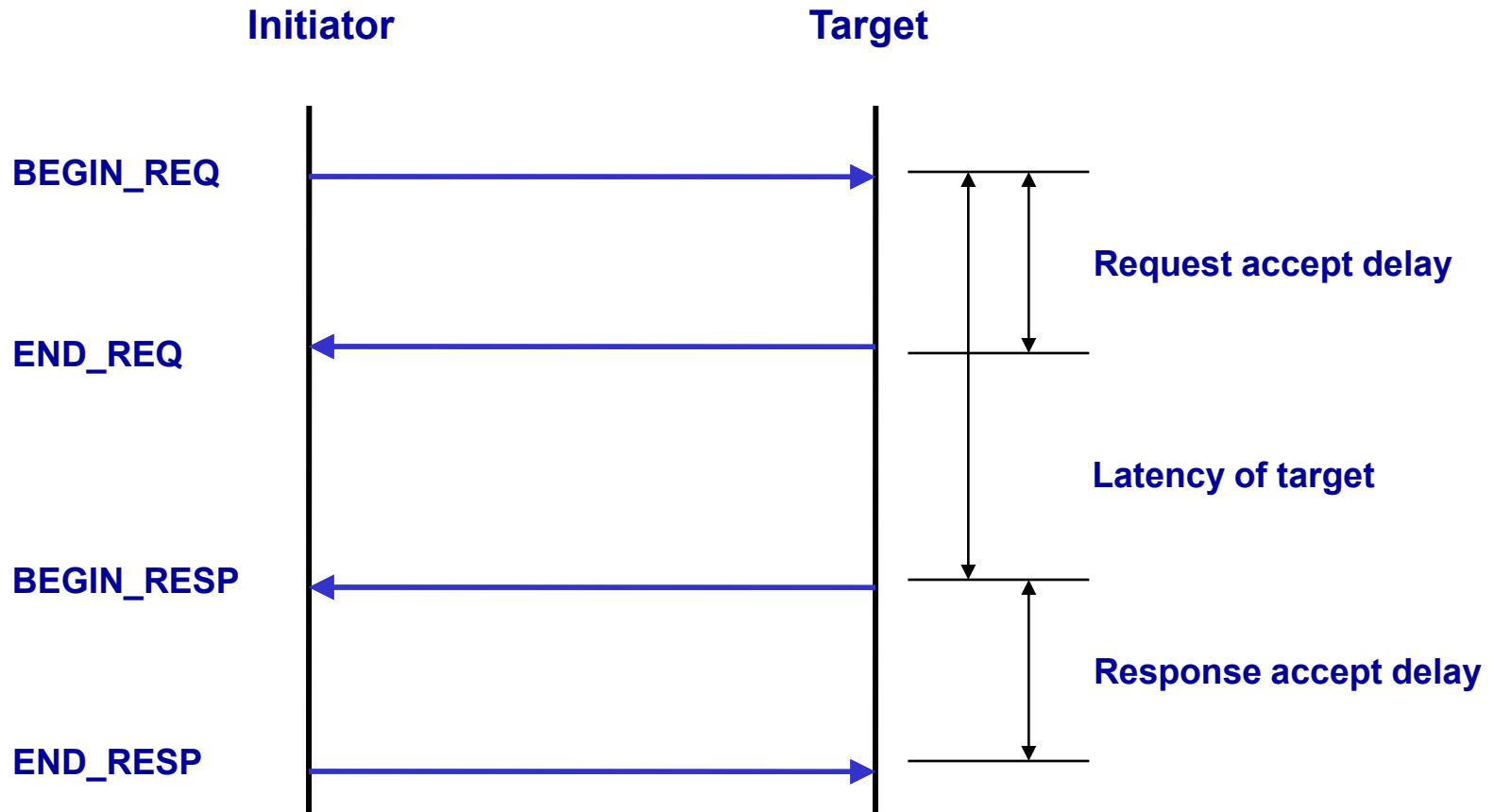
# Base Protocol Rules 2

- Timing annotation on successive calls to nb\_transport
  - for a given transaction, must be non-decreasing
  - for different transactions, mutual order is unconstrained
- Timing annotation on successive calls to b\_transport
  - order is unconstrained (loosely-timed)
- b\_transport does not interact with phases
- b\_transport is re-entrant
- For a given transaction, b\_transport / nb\_transport must not overlap





# Approximately-timed Timing Parameters



*BEGIN\_REQ must wait for previous END\_REQ, BEGIN\_RESP for END\_RESP*



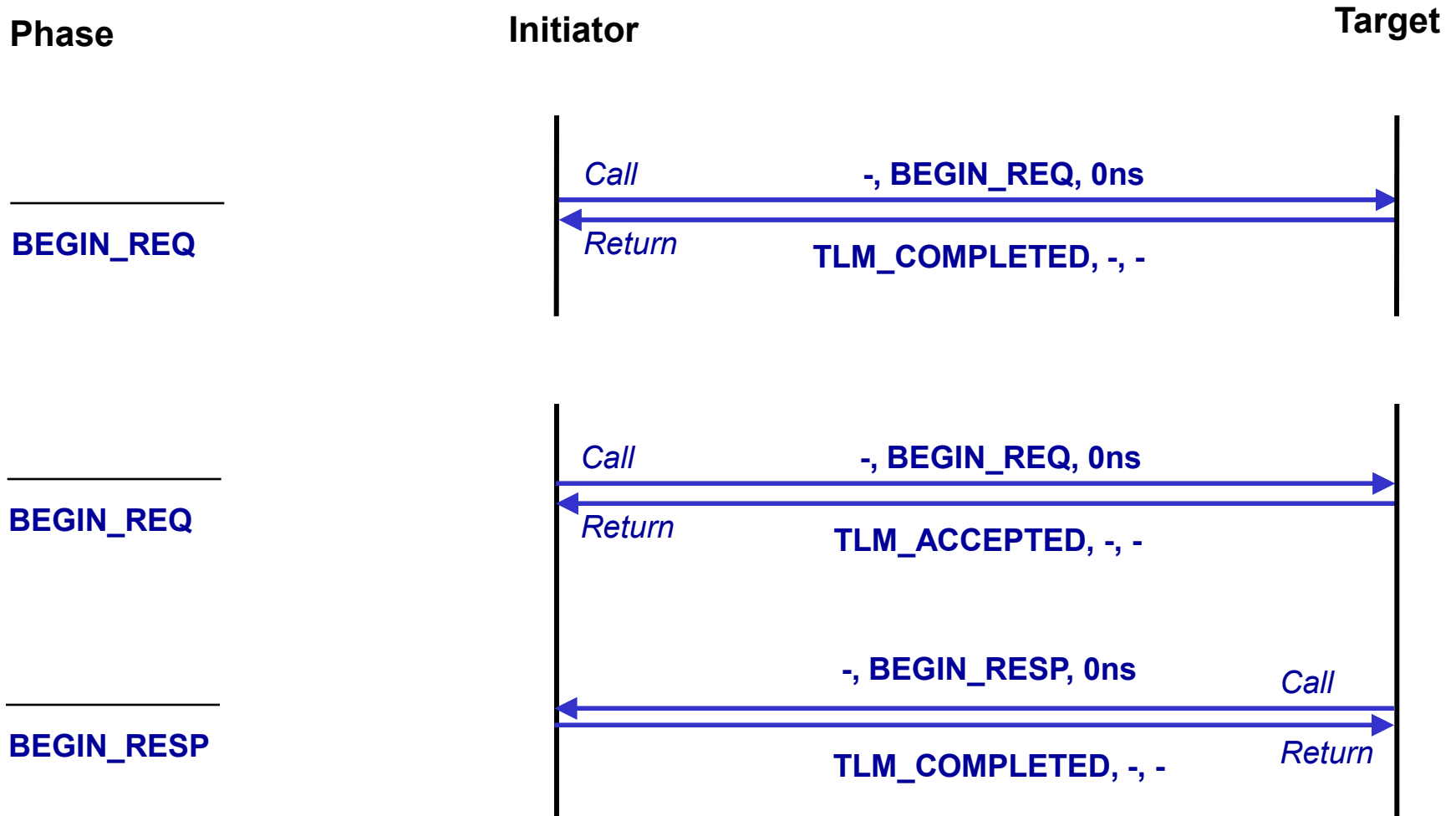
# Pre-emption and Early Completion

- Permitted phase transition sequences
  - **BEGIN\_REQ**
  - **BEGIN\_REQ** ( $\rightarrow$  END\_REQ)  $\rightarrow$  **BEGIN\_RESP**
  - **BEGIN\_REQ**  $\rightarrow$  **END\_REQ**  $\rightarrow$  **BEGIN\_RESP**
  - **BEGIN\_REQ** ( $\rightarrow$  END\_REQ)  $\rightarrow$  **BEGIN\_RESP**  $\rightarrow$  **END\_RESP**
  - **BEGIN\_REQ**  $\rightarrow$  **END\_REQ**  $\rightarrow$  **BEGIN\_RESP**  $\rightarrow$  **END\_RESP**
- **Initiator** sends **BEGIN\_REQ** and **END\_RESP**
- **Target** sends **END\_REQ** and **BEGIN\_RESP**

*Transaction completes early if nb\_transport returns TLM\_COMPLETED*



# Examples of Early Completion



# Transaction Types

- Only three recommended alternatives

- Use the base protocol directly (with ignorable extensions)

*Excellent interoperability*

- Define a new protocol type class with a typedef for `tlm_generic_payload`

*Do whatever you like with extensions*

- Define a new transaction type unrelated to the generic payload

*Sacrifice interoperability; you are on your own*

# Protocol Types Class

```
struct tlm_base_protocol_types
{
    typedef tlm_generic_payload                tlm_payload_type;
    typedef tlm_phase                          tlm_phase_type;
};

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

template <typename TYPES = tlm_base_protocol_types>
class tlm_bw_transport_if
...

```

# Defining a New Protocol Types Class

```
tlm_initiator_socket<> socket1;
```

*1. Use tlm\_base\_protocol\_types*

```
struct my_protocol_types
{
    typedef tlm_generic_payload    tlm_payload_type;
    typedef tlm_phase              tlm_phase_type;
};
```

*2. Use new protocol based on generic payload*

```
tlm_initiator_socket< 32, my_protocol_types > socket2;
```

```
struct custom_protocol_types
{
    typedef my_payload             tlm_payload_type;
    typedef my_phase              tlm_phase_type;
};
```

*3. Use new protocol unrelated to generic payload*

```
tlm_initiator_socket< 32, custom_protocol_types > socket3;
```



# Extended Protocol Example 1

// User-defined extension class

```
struct Incr_cmd_extension: tlm::tlm_extension<Incr_cmd_extension>
{
    virtual tlm_extension_base* clone() const {
        Incr_cmd_extension* t = new Incr_cmd_extension;
        t->incr_cmd = this->incr_cmd;
        return t;
    }
    virtual void copy_from( tlm_extension_base const & from ) {
        incr_cmd = static_cast<Incr_cmd_extension const &>(from).incr_cmd;
    }
    Incr_cmd_extension() : incr_cmd(false) {}
    bool incr_cmd;
};
```

struct **incr\_payload\_types**

```
{
    typedef tlm::tlm_generic_payload tlm_payload_type;
    typedef tlm::tlm_phase           tlm_phase_type;
};
```

*User-defined protocol types class using the generic payload*



# Extended Protocol Example 2

```
struct Initiator: sc_module
{
    tlm_utils::simple_initiator_socket< Initiator, 32, incr_payload_types > init_socket;
    ...
    void thread_process()
    {
        tlm::tlm_generic_payload trans;
        ...
        Incr_cmd_extension* incr_cmd_extension = new Incr_cmd_extension;
        trans.set_extension( incr_cmd_extension );
        ...
        trans.set_command( tlm::TLM_WRITE_COMMAND );
        init_socket->b_transport( trans, delay );
        ...
        trans.set_command( tlm::TLM_IGNORE_COMMAND );
        incr_cmd_extension->incr_cmd = true;
        init_socket->b_transport( trans, delay );
    }
}
```



# Extended Protocol Example 3

// The target

```
Im_utils::simple_target_socket< Memory, 32, incr_payload_types > targ_socket;
```

```
virtual void b_transport( tlm::tlm_generic_payload& trans, sc_core::sc_time& t )  
{
```

```
    tlm::tlm_command cmd = trans.get_command();
```

```
    ...
```

```
    Incr_cmd_extension* incr_cmd_extension;
```

```
    trans.get_extension( incr_cmd_extension );
```

```
    if ( incr_cmd_extension->incr_cmd ) {
```

*Assume the extension exists*

```
        if ( cmd != tlm::TLM_IGNORE_COMMAND ) {
```

```
            trans.set_response_status( tlm::TLM_GENERIC_ERROR_RESPONSE );
```

```
            return;
```

*Detect clash with read or write*

```
        }
```

```
        ++ m_storage[adr];
```

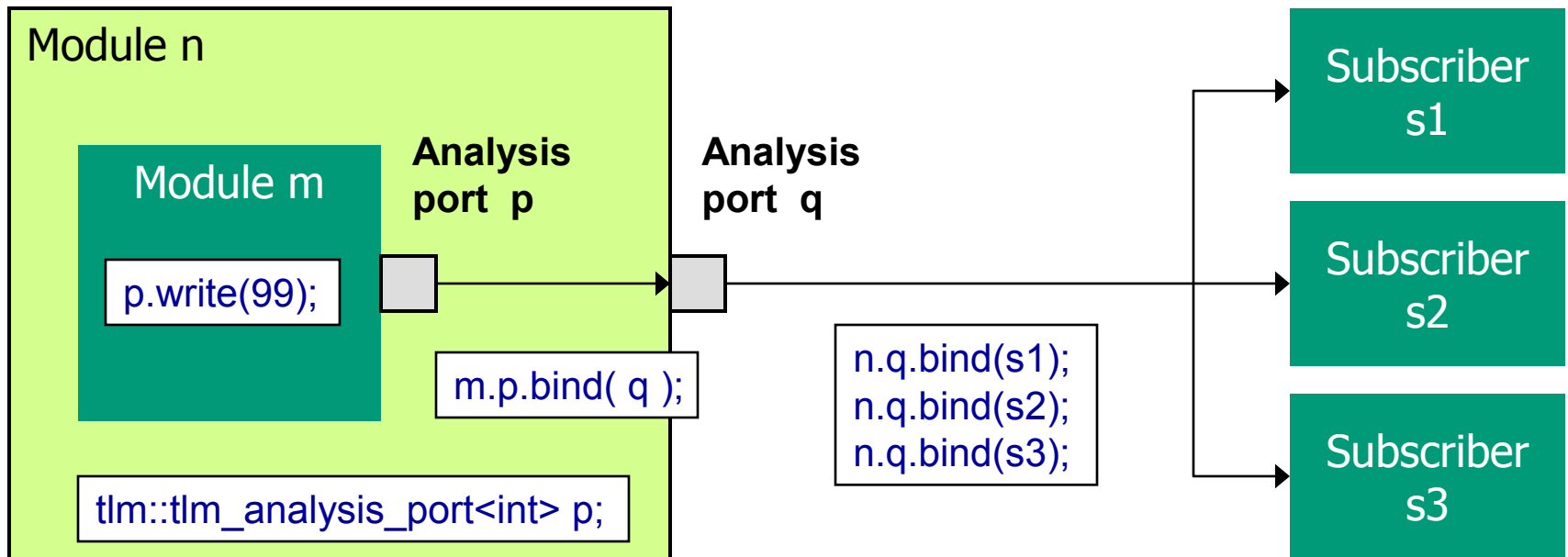
```
    }
```

```
    ...
```

# ANALYSIS PORTS

- Analysis Interface and Ports

# Analysis Ports



```
struct Subscriber: sc_object, tlm::tlm_analysis_if<int>
{
    Subscriber(char* n) : sc_object(n) {}
    virtual void write(const int& t) { ... }
};
```

*Analysis port may be bound to 0, 1 or more subscribers*



# Analysis Interface

```
template <typename T>
class tlm_write_if : public virtual sc_core::sc_interface {
public:
    virtual void write(const T& t) = 0;
};
```

*"Non-negotiated"*

```
template < typename T >
class tlm_analysis_if : public virtual tlm_write_if<T> {};
```

```
class tlm_analysis_port : public sc_core::sc_object , public virtual tlm_analysis_if< T > {
public:
    void bind( tlm_analysis_if<T> &_if );
    void operator() ( tlm_analysis_if<T> &_if );
    bool unbind( tlm_analysis_if<T> &_if );

    void write( const T &t ) {
        for( i = m_interfaces.begin(); i != m_interfaces.end();    i++ ) {
            (*i)->write( t );
        }
    }
};
```

*write() sends transaction to every subscriber*



# Analysis Port Example

```
struct Subscriber : sc_object, tlm::tlm_analysis_if<Trans> {  
    Subscriber ( const char* n ) : sc_object(n) {}  
    virtual void write( const Trans& t ) {  
        cout << "Hello, got " << t.i << "\n";  
    }  
};
```

```
SC_MODULE (M) {  
    tlm::tlm_analysis_port<Trans> ap;  
  
    SC_CTOR (M) : ap("ap") {  
        SC_THREAD (T);  
    }  
    void T () {  
        Trans t = { 999 };  
        ap.write( t );  
    }  
};
```

```
SC_MODULE (Top) {  
    M* m;  
    Subscriber* subscriber1;  
    Subscriber* subscriber2;  
    SC_CTOR(Top) {  
        m = new M("m");  
        subscriber1 = new Subscriber("subscriber1");  
        subscriber2 = new Subscriber("subscriber2");  
        m->ap.bind( *subscriber1 );  
        m->ap.bind( *subscriber2 );  
    }  
};
```

*Subscriber implements analysis interface, analysis port bound to subscriber*



# Summary: Key Features of TLM-2

- Transport interfaces with timing annotation and phases
- DMI and debug interfaces
- Loosely-timed coding style and temporal decoupling for simulation speed
- Approximately-timed coding style for timing accuracy
- Sockets for convenience and strong connection checking
- Generic payload for memory-mapped bus modeling
- Base protocol for interoperability between TL- models
- Extensions for flexibility of modeling





**For further information visit**  
**[www.systemc.org](http://www.systemc.org)**