

---

**GreenSocs „GreenBus“**

**Contribution Review Report**

---

**Contribution by** : **ST Microelectronics**  
**Reviewed by** : **Wolfgang Klingauf and Robert Günzel**  
**Date** : **2005-09-02**

<b>OVERVIEW</b>	<b>3</b>
<b>REVIEW PROTOCOL</b>	<b>4</b>
<b>Developer’s View</b>	<b>4</b>
Model of communication	4
“Location” of bus behaviour	4
Interface characteristic	4
Abstraction layer characteristics	5
Modularity and hierarchy characteristics	5
Simulation performance characteristics	5
Detailed mplementation characteristics	6
Extendability and adaptability characteristics	6
Source code documentation characteristics	7
<b>User’s View</b>	<b>7</b>
Skill requirements	7
Usability characteristics	7
Controllability characteristics	7
Extendability characteristics	8
Adaptability/flexibility characteristics	8
Error handling characteristics	8
Trace / debug support characteristics	8
API documentation	9
Overall usability	9
<b>PROS AND CONS</b>	<b>10</b>
<b>Pros</b>	<b>10</b>
<b>Cons</b>	<b>10</b>
<b>EXPERIMENTS</b>	<b>11</b>
<b>Wrapper Code</b>	<b>11</b>
Header	11
Body	13
<b>SHIP Memory Slave</b>	<b>15</b>

## Overview

The contribution from ST Microelectronics is a high level, OSCI compliant TLM framework for SystemC. It provides a set of initiator and target ports, an application-specific protocol interface (`tac_if`), a `slave_base` class and two router implementations. The framework allows for fast and systematic development of user modules that exchange data via `tac_if` interface method calls at the “programmers view” level of TLM abstraction.

To implement a `tac` master module, it has to be equipped with a initiator port from the framework. To make a user module a `tac` slave, it has to be derived from the `tac_slave_base` class and thus must implement the `tac` interface. `Tac` masters and a slaves can either be connected point-to-point by binding the master's initiator-port to a slave's target port or, for connecting multiple masters and slaves, they all must be bound to one of the provided router implementations.

Transactions take place by instructing a master module to call one of the `tac` interface methods on its initiator port. Depended on the called method, the initiator port creates a request object and transfers it to the target port by using the OSCI TLM transport interface. The `slave_base` part of the target “decodes” the request object and subsequently calls its appropriate `tac_if` method. Hence, from the users’ point of view, `tac` interface method calls on an initiator port are transparently mapped to the target module’s `tac` method implementations, thus providing a high-level view of transactions and hiding the underlying transport subsystem.

Since the TLM framework from ST Microelectronics is located in the PV and PVT levels of TLM abstraction, transaction time estimation is not supported in a cycle-accurate manner. Rather, the router implementations simulate transfer delays for each transaction “as a whole”, thus not taking lower-level bus arbitration schemes such as overlapped and pipelined transacations into account. However, as one would expect at the PVT abstraction level, by this means a fairly high simulation performance is achieved.

With the ST Microelectronics contribution, a very clear and well-coded example implementation of the OSCI TLM Working Group TLM proposal is being provided. The implementation shows how user and transport protocol can be clearly decoupled from each other using a layered approach. Moreover, the two provided router implementations outline the concept of implementing the bus functional model separate from the channel, and the high performance of `sc_port-to-sc_export` binding is illustrated.

## Review Protocol

The review protocol splits up into two parts, one analysing the features that are of importance to an engineer who wants to extend or modify the framework (*Developer's View*) and the other one analysing the qualities that matter to users of the framework (*User's View*). The review protocol is divided into several subsections of which each one deals with a specific characteristic.

### ***Developer's View***

#### **Model of communication**

The framework uses port to export bindings. Masters connect either directly to a slave (point to point connection) or they connect to a router module which itself connects to the slaves (m:n connections).

#### **“Location” of bus behaviour**

The bus behaviour is implemented in a router module. The provided routers simply forwards the incoming requests to the addressed slave and simulate transfer delays using `pv_wait()` statements.

#### **Interface characteristic**

*Available interfaces and their purposes (bus access, config...)*

The master modules use the convenience API to initiate transfers. The master's initiator port uses the `tlm_transport_if` to transfer the request to the target (router or slave). The router uses the `tlm_transport_if` to forward the requests to the slave. The slave implements the convenience API to perform the requested transfer.

There are no configuration interfaces in the system.

*Interface types (virtual functions, sc\_interface, standard base such as OSCI-TLM)*

The convenience API consists of simple method calls (it is not an `sc_interface`). A call to such a method invokes the `transport` method of the `tlm_transport_if`.

The `tlm_transport_if` is a standard OSCI-TLM interface

*Overall usability of the framework's interfaces (self-explaining function names, complexity)*

The convenience API's method names are self-explaining (`read` and `write`) but the “\_block” suffix should not be accidentally interpreted as a reference to a “blocking” method call. It indicates, that a “block of data” is to be transferred (i.e. burst).

Some of the API methods parameters are self-explaining (`data`, `address`, `error_reason`, `port_id`), others require deeper knowledge of the protocol (`byte_enable`, `mode`, `block_byte_enable`, `block_byte_enable_period`).

The usability of the `tlm_transfer_if` depends on the developer's knowledge of the OSCI TLM standard.

*Abstraction layers representation in the interfaces (function overloading, seperate functions, interface hierarchy/seperate interfaces)?*

PV and PVT both use the same API. There are no separate functions nor is there an abstraction layer related interface hierarchy.

## Abstraction layer characteristics

*Available layers (untimed (PV), timed (PVT), BA (CCATB) and/or CC)*

In the framework there are the PV and the PVT abstraction layer available.

*Layers well defined and separated*

Both PV and PVT are well defined. The PV and PVT models are separated by the fact that in PV mode `pv_waits` are randomized and in PVT mode `pv_waits` are always `SC_ZERO_TIME` waits.

*Available abstraction layer wrappers*

PV and PVT use the same API, hence wrappers are not necessary.

## Modularity and hierarchy characteristics

*Obvious basic characteristics (monolithic, manageable, too ramified)*

The framework is OSCI-TLM compliant and therefore manageable if the developer is familiar with OSCI-TLM, otherwise the hierarchy might look quite complex at a first glance.

*Means of decoupling functionality and communication*

Functionality and communication are decoupled just as proposed by OSCI TLM (PV model). This means that the `initiator_port` "wraps" the API onto a `tlm_transport_if` IMC (namely `transport(request)`). The router implements this `transport(request)` method, performs the routing by calling the `transport(request)` method of the addressed slave and simulates timing by calling `pv_wait()`. The slaves are derived from a `slave_base` class that also implements the `tlm_transport_if` (namely `transport(request)`) and this `transport(request)` method "de-wraps" the transported request into an API call. Thus transactions perambulate the initiator port, the `slave_base` and the router.

## Simulation performance characteristics

*Abstraction layer dependency*

Both existing layers perform at the same speed, as they only differ in the parameters of the used `pv_wait` statements.

*Emphasis placed on performance or model flexibility/beauty*

The performance is very high, as no events and hence no internal waits for events are used. Thus the SystemC simulation scheduler has nearly nothing to schedule. The master's API calls directly call the `tlm_transport_if` IMCs, which directly invoke the slave's API methods without scheduling anything. However, possible wait statements for transfer delay emulation in a bus functional module (e.g. the router) are ignored in this argumentation.

## Detailed implementation characteristics

### *Data flow description (end-to-end communication, fragmentation/defragmentation, number of "hops")*

As described above, a transaction is initiated by calling an API method (which is NOT a SystemC IMC as it is not part of an `sc_interface`). This API method calls a `tlm_transport_if` IMC, which is implemented in the transaction's target (with the help of an `sc_export`). The target is either the transaction's final destination (in case of a direct master-to-slave connection) or a router which forwards the request to the transaction's final destination (or another router) by calling an API method of its own initiator port which invokes the target's `transport` method. Hence a single master-to-slave data transfer consists of 1+n transactions where n is the number of routers to be passed (viz. 1+n hops are needed)

### *Control flow description (mutual exclusion, one/multiple threads, event-driven, callback functions)*

As the provided TLM model is only PV and PVT, no mutual exclusion is performed. Since the router is only used to forward the requests to the desired targets in ZERO time and afterwards put the master to sleep for a random time (in PV) or no time (in PVT), every master thread performs transaction independently.

### *Utilized SystemC constructs (ports, sc\_interface, sc\_event, sc\_thread/sc\_method)*

Initiator ports are `sc_ports`. They must be bound to a target that implements or points to the `tlm_transport_if` (channel or `sc_export`).

The `tlm_transport_if` is an `sc_interface` that binds to initiator ports.

No explicit `sc_events` are used.

The communicational part does not contain `sc_threads` or `sc_methods`.

All target ports are `sc_exports` as they have to be bound to their possessing modules.

## Extendability and adaptability characteristics

### *Necessary effort to develop a new bus or extend the existing bus's functionality*

Assuming that the concept (API to `tlm_transport_if` and back) remains unchanged, there are two possibilities:

1. In case the API shall remain unchanged, to implement a new bus behaviour, only a new router module has to be developed.
2. In case the new bus model uses another API, a new initiator port class has to be developed that wraps from the new API to the `tlm_transport_if`. A new `slave_base` class must be developed as well to wrap from the `tlm_transport_if` to the bus API. Finally, the new bus model itself has to be implemented.

### *Adding abstraction layers*

To add the BA abstraction layer, the router would have to be refined towards mutual exclusion and accurate timings.

To add the CC abstraction layer, either `pin2API` and `API2pin` wrappers had to be developed that connect to a BA router, or a router with `pin` accurate interface and cycle accurate behaviour could be implemented (boasting extremely poor performance).

### *Support of multiple abstraction layers at once*

The provided TLM model can run either in PV or PVT by toggling a static member of the `tlm_pvt_base` class from which all modules in the system should be derived. Thus the TLM model does not enable mixed-mode simulation using multiple layers of abstraction at once. However, the abstraction layer can be switched at runtime.

### **Source code documentation characteristics**

The code documentation is extremely detailed and enjoys doxygen compliance.

### **User's View**

#### **Skill requirements**

*Different user skill level support (beginner, advanced, professional)*

The provided tac API is appropriate for beginners. There are only 6 API calls, namely `read`, `write`, `read_block`, `write_block`, `get_target_inf`, and `set_target_info`. There is only one advanced API method `do_transport(request)` which calls the `transport(request)` method of the `tlm_transport_if` and does some transaction recording. If familiar with the structure of the `tlm_request` object, one could avoid calling e.g. `read(...)` and could use instead `do_transport(request)` where `request` is configured as a read request with all the other options desired. However, one cannot achieve much more than calling the high-level methods itself, since no more information as the API calls are encoded in a request.

*API structure (simple ↔ complex, intervowen ↔ layered, high/mid/low level functions)*

The API is simple and flat and consists of seven methods (6 "real" API methods and the additional advanced one described above). The API is not layered but as it only comprises seven methods it cannot be called intervowen.

#### **Usability characteristics**

*Factory for typical tasks*

There is no factory available, but as port-to-port bindings are pretty simple this is nothing to worry about.

*Meaningful warnings and error messages*

The error reason reports and configurable output levels (NONE/VERBOS/DEBUG) are very helpful.

#### **Controllability characteristics**

*Configurable parameters*

Transaction counting can be enabled via `#define`.  
PV/PVT toggling is managed by a static member of `pvt_base_class`.  
Address maps used by the router are located in a map file.  
Transaction recording is enabled via `#define` and a static member of `trans_record_database` class.  
Transaction recording output format is specified both via `#defines` and static `trans_record_database` methods.

The ports which transactions are to be recorded can be specified in a recorder config file.

*Static (compile time) or dynamic (elaboration phase, during simulation)*

Static: transaction counting, available transaction recording output formats, address map.

Dynamic: PV/PVT toggling, transaction recording enable/disable, actual transaction record output format, transaction record ports.

## **Extendability characteristics**

*Separation of user extensions (e.g. user data renders user services) from the framework*

Address and data type are template parameters of the `tac_if` and thus the user can transfer arbitrary objects that carry all the data and additional information he wants it to.

## **Adaptability/flexibility characteristics**

*Compatible to OSCI-TLM or other standards*

The provided framework is an excellent OSCI-TLM PV example.

*Available wrappers*

None in the contribution, but we implemented a simple one that wraps the tac API into another TLM interface (SHIP, E.I.S. proprietary)

## **Error handling characteristics**

*Protocol misuses detection*

Incorrect addresses (overlapping) are reported by the routers.

Incorrect access modes are reported by `slave_base`.

*Assertions*

There are no real `assert()` calls, but as mentioned above address and access mode correctness are tested.

## **Trace / debug support characteristics**

*Standard (scv) compliancy*

The framework supports SCV transaction recording, but no introspection.

The output formats can be SCV, SST2 (Cadence) or FSDB (Nova).

*Configurable trace-levels, recorded signals*

Message tracing supports three modes NONE, VERBOSE, DEBUG. With DEBUG enabled, the trace messages provide comprehensive information.

The transactions that are to be recorded are specified in a recorder config file and dumped into a data-base file.

### *Internal functions directly accessible*

All vital functions (which means functions of interest during debug) are public and therefore directly accessible.

### *Introspection/reflection*

Not available.

### **API documentation**

Just like the whole source code documentation, the API documentation is pretty detailed and also doxygen compatible.

### **Overall usability**

#### *Necessary effort to proceed to next abstraction layer*

From PV to PVT: refine user modules to PVT and toggle the mode via the static `pvt_baseclass` member.

From PVT to BA: refine modules towards bus behaviour; refine router, eventually initiator port and slave base (maybe even API) if transactions split up into phases.

From BA to CC: refine modules towards pin accurate and cycle accurate behaviour; develop `pin2BAAPI` and `BAAPI2pin` wrapper.

#### *Communication refinement automation possible?*

Since only PV and PVT are available, no communication refinement automation is needed as the API doesn't change.

#### *Limitation of systemc command set, or -- even worse -- framework-specific commands required*

Framework specific commands are `pv_wait` and `pv_next_trigger`.

#### *Design reuse, IP integration.*

As long as designs utilize the tac protocol, reuse is obviously easy.

When the design uses a different protocol or incorporates IP based on a different protocol, wrappers are necessary. As the tac protocol is closer to low level than to high level (e.g. it makes use of BE signals) wrapping to a high level TLM might be problematic.

#### *Available analysis tools*

When transaction recording output formats are SST2 or FSDB, Cadence and Nova proprietary tools can be used

#### *Gcc and systemc-2.1 compatible*

Gcc version 3.2.3 in conjunction with systemC 2.1

## Pros and Cons

In the following, some VERY DRAFT pros and cons in respect to our current idea of GreenBus requirements are discussed.

### Pros

1. Implements OSCI-TLM PV/PVT standard proposal
2. Global switch between PV and PVT, only affects router timing
3. sc\_port-to-sc\_export binding without the need for channels
4. Wraps TLM transport function to application-specific convenience API without the need for threads
5. High simulation performance
6. Shows how to use template specialization for wrapping different types of data efficiently to a user protocol (tac\_initiator\_port)
7. Slave implementation is quite easy by inheriting from the tac\_slave\_base class
8. Slave has to implement the tac\_if
9. No processes (SC\_THREAD, SC\_METHOD) needed (high simulation performance)
10. Master implementation is easy by inheriting from tlm\_module
11. Master uses SC\_THREAD and tac\_initiator\_port for communication
12. Simple m:n router implementations (portlist / multiport) with simple transaction timing / synchronisation
13. Address decoding (initiator\_port->target\_port mapping) by address map file
14. SystemC-SCV compliant transaction recording in initiator\_port and target\_port (can be enabled/disabled)
15. Comprehensive debug / trace message framework (tlm\_message)

### Cons

1. Bus functionality implementation is very “monolithic” and on a very high level of abstraction
2. Communication latency is simulated by tac\_router, which only “sees” one transport() call per transaction → timing estimation is very inaccurate
3. Typical bus features such as arbitration schemes, overlapping transfers and pipelining cannot be taken into account on this level of abstraction
4. For more accurate bus arbitration and timing simulation, the communication wrappers (slave\_base, initiator\_port) and the complete router implementation have to be changed
5. For communication refinement of system components, the convenience API has to be changed, because:
  - If the convenience API is too high-level (only one method call for per transaction), efficient TLM-to-RTL refinement of both communication and functionality is nearly impossible
  - If the convenience API is too low-level (many method calls for one transaction), a high-level implementation of the slave’s functionality is not possible
6. The convenience API is very bus-specific (read, write, read\_block, write\_block)  
put/get or send/rcv would be more flexible for mapping to arbitrary network-on-chip
7. Slaves directly inherit from communication wrappers (slave\_base)  
Slaves cannot be connected to routers at different levels of abstraction in the top-level model (as it would be possible with port→channel→port binding)  
Slave functionality is scattered over multiple functions (read, write, ...)  
For tool-supported TLM-to-RTL refinement, a single SC\_THREAD would be preferable.
8. Inheritance tree is partially complex and thus, from the developers point of view, might be difficult to understand.
9. Handling of the sc\_port->sc\_export concept is far more complicated and less “transparent” than using explicit channels
10. Large parts of the implementation rely on proprietary base classes and interfaces
11. SystemC-SCV introspection/reflection feature and sc\_report are not used

## Experiments

In this chapter we provide the source code of a protocol wrapper we wrote to analyze the necessary effort to integrate foreign IP into a tac-based system. Since it's an upward wrap, the byte-enable information gets lost in the wrapper. Hence the sending master is not allowed to use byte-enables.

In the provided tac-demo, we replaced the MEMORY1 module with our tac2SHIP-wrapper and connected a SHIP-based memory module. To avoid the above mentioned byte-enable problem, we disabled all of the traffic\_generator1's functions, but the single transaction test.

### Wrapper Code

#### Header

```
#ifndef _TAC2SHIP_WRAPPER_H_
#define _TAC2SHIP_WRAPPER_H_

/*-----
 * Includes
 *-----*/
#include <exception>
#include <iostream>
#include <iomanip>

#include "systemc.h"

#include "tlm_host_def.h"

#include "tac_slave_base.h"
#include "tac_target_port.h"
#include "ship_master_if.h"

/*-----
 * Defines
 *-----*/

using dvk_tlm_message::tlm_message;

using prt_tlm_tac::tac_mode;
using prt_tlm_tac::NO_BE;
using prt_tlm_tac::DEFAULT;
using prt_tlm_tac::BYTE_LANE_0;
using prt_tlm_tac::BYTE_LANE_1;
using prt_tlm_tac::BYTE_LANE_2;
using prt_tlm_tac::BYTE_LANE_3;

using prt_tlm_tac::tac_status;
using prt_tlm_tac::tac_error_reason;
using prt_tlm_tac::tac_metadata;
using prt_tlm_tac::tac_slave_base;
using prt_tlm_tac::tac_target_port;

using namespace std;
using namespace ship;

class tac2ship_wrapper :
    public sc_module,
    public tac_slave_base<int,int>{

protected:

    //-----
    // Module members functions & data
    //-----

    tlm_endianness m_endian;    ///< memory endianness

public :

    //-----
```

```

// Module ports
//-----

tac_target_port<int,int> target_port; ///

```

```

        const unsigned int number,
        tac_error_reason& error_reason,
        unsigned int * block_byte_enable = NULL,
        const unsigned int block_byte_enable_period = 1,
        const tac_mode mode = DEFAULT,
        const unsigned int target_port_id = 0
    );

//-----
/// Get info about target
virtual tac_status get_target_info(const int& address,
                                   tac_metadata& metadata,
                                   tac_error_reason& error_reason,
                                   const unsigned int target_port_id = 0
                                   );

//-----
/// Set info to target
virtual tac_status set_target_info(const int& address,
                                   const tac_metadata& metadata,
                                   tac_error_reason& error_reason,
                                   const unsigned int target_port_id = 0
                                   );

/* @} */
};

#endif

```

## Body

```

//-----
* Includes
*-----*/
#include "tac2ship_wrapper.h"

//-----
* Defines
*-----*/
using std::exception;

//-----
* Global Variables
*-----*/

// sc_object kind string property
const char* const tac2ship_wrapper::kind_string = "tac2ship_wrapper";

//-----
// Constructor
//-----

tac2ship_wrapper::tac2ship_wrapper(sc_module_name module_name,
                                   tlm_endianness endian
                                   ) :
    sc_module(module_name),
    tac_slave_base<int,int>(name(),kind()),
    m_endian(endian),
    target_port("target_port") {

    // Bind tac target sc_export to the slave
    target_port(*this);
}

//-----
// Destructor
//-----

tac2ship_wrapper::~tac2ship_wrapper() {

    GENERAL_REPORT("\t%s: Destructor called\n",name());
}

```

```

}

//-----
// Abstract class tac_if methods implementation
//-----

//-----
// Memory read access

tac_status tac2ship_wrapper::read(const int& address,
                                   int& data,
                                   tac_error_reason& error_reason,
                                   const unsigned int byte_enable,
                                   const tac_mode mode,
                                   const unsigned int target_port_id
                                   ) {

    tac_status status;

    MPort_addr->send(addr);
    MPort_data->request(data);

    status.set_ok();
    return(status);
}

//-----
// Memory write access

tac_status tac2ship_wrapper::write(const int& address,
                                    const int& data,
                                    tac_error_reason& error_reason,
                                    const unsigned int byte_enable,
                                    const tac_mode mode,
                                    const unsigned int target_port_id
                                    ) {

    tac_status status;

    MPort_addr->send(addr);
    MPort_data->send(data);

    status.set_ok();
    return(status);
}

//-----
// Read access block

tac_status tac2ship_wrapper::read_block(const int& address,
                                         int * block_data,
                                         const unsigned int number,
                                         tac_error_reason& error_reason,
                                         unsigned int * block_byte_enable,
                                         const unsigned int block_byte_enable_period,
                                         const tac_mode mode,
                                         const unsigned int target_port_id
                                         ) {

    //not implemented
    status.set_ok();
    return(status);
}

//-----
// Write access block

tac_status tac2ship_wrapper::write_block(const int& address,
                                         const int * block_data,
                                         const unsigned int number,
                                         tac_error_reason& error_reason,
                                         unsigned int * block_byte_enable,
                                         const unsigned int block_byte_enable_period,
                                         const tac_mode mode,
                                         const unsigned int target_port_id
                                         ) {

```

```

    ) {
        //not implemented
        status.set_ok();
        return(status);
    }

//-----
// Get info about target

tac_status
tac2ship_wrapper::get_target_info(const int& address,
                                   tac_metadata& metadata,
                                   tac_error_reason& error_reason,
                                   const unsigned int target_port_id
                                   )
{
    //not implemented
    status.set_ok();
    return(status);
}

//-----
// Set info about target

tac_status
tac2ship_wrapper::set_target_info(const int& address,
                                   const tac_metadata& metadata,
                                   tac_error_reason& error_reason,
                                   const unsigned int target_port_id
                                   ) {
    tac_status status;
    error_reason.set_reason("set info using metadata not supported");
    ERROR_REPORT(2, "\t%s: %s T:%9.9f\n", name(), error_reason.get_reason().c_str(),
                (float)(sc_time_stamp().to_seconds()));
    return(status);
}

```

## SHIP Memory Slave

```

#include "ship_slave_if.h"
#include "systemc.h"
#include "ship_structs.h"

using namespace ship;

class ship_mem_slave : public sc_module
{
public:
    sc_port<ship_slave_if<int> > SPort_addr;
    sc_port<ship_slave_if<int> > SPort_data;

    SC_HAS_PROCESS(ship_mem_slave);

    ship_mem_slave(sc_module_name name): sc_module(name){
        m_mem=new int[16384];
        SC_THREAD(main);
    }

    void main(){
        int addr;
        int data;
        ship_command comm;
        while(true){
            comm=SPort_addr->waitEvent();
            if (comm.cmd==SHIP_SEND){
                SPort_addr->recv(addr);
                //cout<<"Got addr "<<addr<<endl<<flush;
                comm=SPort_data->waitEvent();
                if(comm.cmd==SHIP_SEND){
                    SPort_data->recv(data);
                    m_mem[addr]=data;
                    //cout<<"Got data "<<data<<endl<<flush;
                }
                else if(comm.cmd==SHIP_REQUEST)
                    SPort_data->reply(m_mem[addr]);
            }
        }
    }
}

```

```
        //cout<<"Reply data "<<m_mem[addr]<<endl<<flush;
    else
        //error
    } else{
        //error
    }
}
}
protected:
    int* m_mem;
};
```