# SignalSockets

Developed by : Ashwani Singh
CircuitSutra Technologies Pvt Ltd

Document History

| Date | Author | Comment | Version |
|---|---|---|---|
| 1st July, 2009 | Puneet |  | 0.1 |

## Table of Contents

## 1. Purpose

The purpose of this document is to explain the significance and the usage of the signal-sockets.

## 2. Scope:

This document only discusses the usage of the signal-sockets. It will not go into deeper details of the implementation.

## 3. Introduction

In the existing TLM2 infrastructure, the modeling engineer has the advantage of using sockets and transport calls for memory-mapped bus communication. The predefined **tlm_generic_payload** and **tlm_phases** are given, which can also be extended to extend the base protocol.

Many IPs have various signals as input and output which are not the direct part of the memory mapped communication but they somehow enable or disable the communication or some other smaller protocols, for example, interrupt line of Interrupt Controller is not a part of memory/register read/write functionality but it forms a handshaking protocol. Similarly gate input of Timer enables/disables the counting logic of the timer. Currently systemc ports are being used to model such signals. The most commonly used ones are sc_in, sc_out etc.

The communication through SystemC ports uses event based and is bound to effect the simulation. The more the number of ports, the more it will affect the performance as for each port, we will have an update cycle.

So here we introduce the **signalsockets,** which will replace all the ports corresponding to different signals in an IP. All the signals will be sent to the IP through extensions using the signalsockets. The signalsockets will be configured to support those extensions(signals) which are part of the parent/owner IP.

**Basic Idea of the SignalSocket:**

To communicate the signal values through these sockets, a new payload is defined. The user has to add extensions to this payload corresponding to the signals that are to be communicated. The existing *tlm_generic_payload* also supports extension mechanism but the *tlm_generic_payload* contains certain fields which are never required in signal communication, like streaming width etc. Hence it is not directly used.

So to communicate value of a signal using a signal socket, the initiator has to set the appropriate extension in the payload and make a transport call on its *initiator_signal_socket*. This call will be received by appropriate *target_signal_socket*, where the target can read the value of extension. This serves the purpose of communicating the value when initiator wants to assert some value on the port.

In the connectivity using SystemC ports, the target is free to read the value of signal at a port at any time during the simulation. To support this feature, the *target_signal_socket* keeps a record of the value of the signal and provides an API to access the same. This is explained in the later section on usage.

Apart from that we have kept the idea of 'SourceID'. Each payload which is being sent across the bus, will carry a unique ID of the IP from where this payload has been originated. We required it for few specific scenarios. For example we have signals **"foo",** which originates from both **Initiators A and B.** But the **target C**, supports the **"foo"** from **Initiator A.** And **target D** supports the **"foo"** from **Initiator B.** So we will inform the **target C** to support payloads from **A.** And **D** to support payloads from **B**. So every **"foo"** signal will carry this information with it whether it is from **Initiator A** or from **Initiator B.** This way, **target C** can ignore the signal **"foo"** if it has come from **Initiator B.**

## 4. gs_generic_signal_payload

The *tlm_generic_payload* is designed to support the data communication on bus and contains generic fields like byte_enable, address, streaming width etc. Signal communication is simpler than that and does not require such attributes. The *gs_generic_signal_payload* does not need to keep these attributes instead it uses the same extension mechanism as provided in the *tlm_generic_payload*.

The extension mechanism lets the user add extensions corresponding to the signals that are there in the model. For example, if the model contains a boolean signal named IR (for which the target SystemC module would have used **sc_in<bool> IR**), then we use a **guard extension** to communicate through signal_sockets.

The gs_generic_signal_payload has following attributes

**1)** ***unsigned char* m_data_ptr :*** this is the pointer to the data which keeps value of the signal.

**2)** ***unsigned int m_src_id :*** this is the unique src id of the initiator_signal_socket from which this payload is originating.

**3)** ***bool m_ack_required :*** this flag indicates if an acknowledgement is required for the signal, useful for signals like Interrupt which have an associated Acknowledge line (INT & INTA in InterruptController and DREQ & DACK in DMA).

### 4.1 APIs

Following API's are provided to access the above fields of the paylaod:

*void set_data_ptr(unsigned char* data) :* to set the data pointer of the payload. This is a pointer to value of the signal. This is called at the initiator end.

*unsigned char* get_data_ptr() :* to get the value of the data pointer of the payload. The value of the signal is read from this location. This is called at the target end.

*void set_src_id(unsigned int &id) :* to set the unique identification of the source i.e. origin of this payload. This is called from within the initiator socket and need not be explicitly called by any module.

*unsigned int get_src_id () :* to get the id of the origin of this payload. This is called in the target socket to identify whether the payload is coming from a supported source or not.

*void set_ack_requirement(bool & flag) :* to set the ack requirement attribute of the payload. The initiator may set this if an acknowledgement is required with the transmission of this signal.

*bool is_ack_required() :* to get the ack requirement attribute at the target end. If this is true then this is a handshaking protocol and the target updates the phase accordingly.

## 5. signal_phase

Phases are provided in the TLM2 base protocol to represent various timing points in the transactions between IPs. Using phases the model can be made to behave at different abstraction levels.

When realizing the signals in some hand-shaking protocols, one can use the concept of non-blocking transfer(for capturing the timing points) by using different phases to represent the signal and its acknowldgement. For this purpose, the *signal_phase* class is provided which is similar to tlm_phase. This has the advantage of using single socket instead of two different ports, for the handshaking protocol but with two different phases.

The current enums that are suported are REQ and ACK . These can be used when using a non-blocking transport call to communicate the signal and can be extended using the same mechanism as for tlm_phase.

The gs_generic_signal_payload and signal_phase are used in the definition of structure gs_generic_signal_protocol_types, which serves as the TRAITS class in the definition of sockets.

```
struct gs_generic_signal_protocol_types {
  typedef gs_generic_signal_payload tlm_payload_type;
  typedef signal_phase tlm_phase_type;
};
```

## 6. Using initiator_signal_socket

The sockets are derived from GreenSockets and take *gs_generic_signal_protocol_types* as their protocol type. The ***initiator_signal_socket*** and ***initiator_signal_multi_socket*** are the versions available.

### 6.1 constructing the socket

Following constructor is avilable for initiator_signal_socket,

*initiator_signal_socket (const char* name, gs::socket::allocation_scheme alloc_scheme=gs::socket::GS_TXN_ONLY) ;*

The first argument is name of the socket and latter is a configuration option for socket's pool. In brief, the default value of the second argument tells the socket that we are going to get the transcation object from the socket itself and manage the memory for its data pointer at our end. For detailed description of the latter argument please refer to the GreenSocket user guide.

The constructor for the multi socket also has same signature,

*initiator_signal_multi_socket (const char* name, gs::socket::allocation_scheme alloc_scheme=gs::socket::GS_TXN_ONLY) ;*

## 6.2 Determining the number of bindings

Following API is provided to get the number of bindings,

*unsigned int size();*

For a multi socket this returns the number of bindings and for a signle socket it returns one after the socket has been bound and zero otherwise.

## 6.2 Getting name of the socket

*const char* get_name();*

will return the hierarchical name of the socket.

## 6.3 configuring the socket

Since signal sockets are derived from GreenSockets, they need to have a configuration before end_of_elaboration callback of SystemC. A single socket can only have a single configuration, while a multi socket has a configuration per bound target socket.

Following API's are provided to access the config of the signal sockets

1. *void set_config(const gs::socket::config<gs_generic_signal_protocol_types>& cfg);*

When called during construction time this functions assigns the initial configuration to both single and multi sockets. When called after construction time (i.e. the number of bindings of a multi sockets is known at this time) it overrides the configuration of a single socket, and it overrides the configurations of all bindings of a multi socket.

2. *void set_config(const gs::socket::config<gs_generic_signal_protocol_types>& cfg, unsigned int index);*

This function overrides the configuration of binding index of a multi socket. Consequently it may only be used after the number of bindings is known. Calling it on a single socket, with an index out of the bounds of a multi socket or it during construction time is considered an error.

3. *gs::socket::config<gs_generic_signal_protocol_types>& get_recent_config();*

This function returns the configuration of a single socket. For multi sockets it returns the initial configuration during construction time, and the configuration of binding zero after construction time.

4. *gs::socket::config<gs_generic_signal_protocol_types>& get_recent_config(unsigned int index);*

This function returns the configuration of binding index of a multi socket. Consequently it may only be called, after construction time, when the number of bindings has been settled.

## 6.4 Using the Transaction Memory Management

The initiator sockets provide memory management facilities to the user. However, the user is not forced to use them and may use its own pools and memory managers to allocate transactions.

The functions offered by the socket are:

1. *gs_generic_signal_payload* get_transaction();*

This function will get a transaction from the socket's pool, with it's reference count being at one. That means, initiators do not have to acquire a transaction, as the socket did it for them already.

2. void release_transaction(gs_generic_signal_payload* txn);

This function will reduce the reference count of the given payload object by one. Effectively the same can be achieved by calling txn->release(), but only one or the other should be done.

## 6.5 Registering callbacks

The user may register following TLM2.0 backward interface method call:

*template<typename MODULE> void register_nb_transport_bw(MODULE* mod, nb_cb cb)*

This function allows to register a callback for nb_transport_bw. The type of nb_cb differs between single and multi sockets. For single sockets it is

*tlm::tlm_sync_enum (MODULE::*)(transaction_type&, phase_type&, sc_core::sc_time&)*

and for multi sockets it is

*tlm::tlm_sync_enum (MODULE::*)(unsigned int, transaction_type&, phase_type&, sc_core::sc_time&)*

## 6.6 Accessing Extensions

The user may use following APIs on initiator socket to set the GUARD extensions at its end

*invalidate_extension<EXT_NAME>(*p);*

The above call will invalidate the given extension in the payload.

*validate_extension<EXT_NAME>(*p);*

The above call will validate the given extension in the payload.

By the use of these two APIs the initiator can make sure that the target can uniquely identify the signal for which value is being given.

# 7. Using target_signal_socket

The target sockets are also derived from GreenSockets and take *gs_generic_signal_protocol_types* as their protocol type. The ***target_signal_socket*** and ***target_signal_multi_socket*** are the versions available.

## 7.1 constructing the socket

The target socket takes a *MODULE* template argument which is the class that contains this socket i.e. owner of the socket.

*template <typename MODULE> class target_signal_socket;*

Following constructor is avilable for target_signal_socket,

*target_signal_socket(const char* name);*

As for initiator sockets, the declaration and construction of multi-socket version are exactly same.

## 7.2 Determining the number of bindings

Same as that for initiator sockets.

## 7.3 Getting name of the socket

Same as that for initiator sockets.

## 7.4 configuring the socket

Same as that for initiator sockets.

## 7.5 Registering callbacks

The user may register following TLM2.0 forward interface method call:

1. *template<typename MODULE> void register_nb_transport_fw(MODULE* mod, nb_cb cb)*

This function allows to register a callback for nb_transport_fw. The type of nb cb differs between single and multi sockets. For single sockets it is

tlm::tlm_sync_enum (MODULE::*)(transaction_type&, phase_type&, sc_core::sc_time&)

and for multi sockets it is

tlm::tlm_sync_enum (MODULE::*)(unsigned int, transaction_type&, phase_type&, sc_core::sc_time&)

2. *template<typename MODULE> void register_b_transport(MODULE* mod, b_cb cb)*

This function allows to register a callback for b_transport. The type of b_cb differs between single and multi sockets.

For single sockets it is

void (MODULE::*)(transaction_type&, sc_core::sc_time&)

and for multi sockets it is

void (MODULE::*)(unsigned int, transaction_type&, sc_core::sc_time&)

## 7.6 Setting the source id

Following API can be used for registering the supported initiators with the target socket:

*template<typename EXT_NAME>void set_source(std::string name)*

The template argument is the *extension* that this socket is going to accept, and the *name* argument is the name of the initiator socket from which this target socket will accept this extension(signal).

## 7.7 Accessing Extensions

The target_signal_socket receives the *gs_generic_signal_payload* which has appropriate extensions validated in it. It has the responsibility of correctly identifying the source of the transaction and forwarding the transaction to the transport call of its owner module if it is coming from a valid(pre-registered) source. For this purpose it keeps a list of supported_source_ids which is a unique identification of the initiator_signal_socket to which this target is bound.

Following API's are provided at the target socket to access the extensions:

*bool get_extension<EXT_NAME>(gs_generic_signal_payload)*

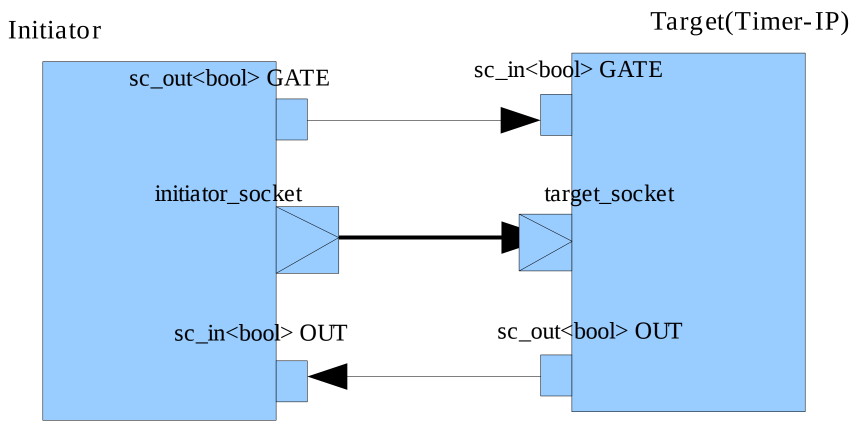
The above API will return true only if this extension was validated at initiator.

*unsigned char get_last_value<EXT_NAME>();*

The above API returns the value of the extension.

## 8. Example

In this example we describe various steps to be done for replacing the use of SystemC ports with signal_Sockets in the IPs. Let us consider a simple timer IP which takes GATE as input and generates interrupt OUT according to the values programmed in its registers by the user.



As shown in the above diagram, the integrated model contains SystemC ports to communicate the values of GATE and OUT signals and a pair tlm sockets to communicate the register read/write functions.

We intend to replace the sc_ports with green_signal_socket. The rule for this is, for all signals incoming into a module, introduce a *target_signal_socket<OWNER>* and for all the signals going out from the module introduce a *initiator_signal_socket* in the module

To instantiate these sockets use following namespace and include:

*#include "green_signal.h"*

*using namespace gs_generic_signal;*

For the example above, a pair of sockets is introduced in each of the module, i.e. The initiator gets

```
  gs_generic_signal::initiator_signal_socket out_socket;
  gs_generic_signal::target_signal_socket<sc_initiator> in_socket;
```

and similarly the Timer will get a pair of signal sockets.

The sockets of the two modules are bound to each other in the testbench(where instances of the two modules are created):

```
InitInst.out_socket(TimerInst.in_socket);
TimerInst.out_scoket(InitInst.in_socket);
```

After instantiating the sockets, they are configured in the constructor of the owner module. For both type of sockets the configuration is provided specifying the extensions that are expected and for the target_signal_socket one needs to additionally bind the appropriate functions for transport calls.

```
class sc_initiator : public sc_module {
  gs_generic_signal::initiator_signal_socket out_socket;
  gs_generic_signal::target_signal_socket<sc_initiator> in_socket;
  sc_initiator (sc_module_name nm) : sc_module(nm), in_socket("target_sock"),
      out_socket("init_sock")  {
   in_socket.register_b_transport(this, &sc_initiator::b_trans);
    gs::socket::config<gs_generic_signal_protocol_types> inCnf;
    inCnf.use_mandatory_extension<OUT>();
    in_socket.set_config(inCnf);
    std::string srcName("Timer.out_sock");
    in_socket.set_source<OUT>(srcName);

    gs::socket::config<gs_generic_signal_protocol_types> outCnf;
    outCnf.use_mandatory_extension<GATE>();
    out_socket.set_config(outCnf);
 }
};
```

So for the *in_socket*, the config mandates an <OUT> extension. This corresponds to the OUT input from the Timer IP. So whenever Timer wishes to write on its OUT port, now it will generate a transaction and send it on its *out_socket*. This transaction must have the <OUT> extension validated otherwise the receiver will not be able to correctly identify for which signal the value is sent.

And for the out_socket, the config mandates a <GATE> extension which is the signal that this initiator drives for the Timer.

Similar configuration is also done at the Timer end.

Please note that we have used *set_source(string)* on the target_signal_socket in the constructor of the module itself. This is not necessary or possible always. Instead in the testbench where the modules are instantiated, one can do:

*TimerInst.in_socket.set_source<GATE>(InitInst.out_socket.name());*

The *set_src_id(unsigned int)* API for gs_generic_signal_payload, mentioned above, is called for the payload within the initiator_signal_socket transport calls. This helps the target identify the source of payload and decide whether or not to forward the transaction to its owner module.

So when initiator wants to write a value on its GATE port it will do the following:

```
{
  gs_generic_signal_payload* p = out_socket.get_transaction();
  unsigned char data = 0x1; // to raise the signal
  p->set_data_ptr(&data);
  //out_socket.invalidate_extension<other_exts>(*p);
  out_signal_port.validate_extension<GATE>(*p);
  sc_core::sc_time t = sc_core::SC_ZERO_TIME;
  out_signal_port->b_transport(*p, t);
  out_signal_port.release_transaction(p);
}
```

Please note that if there are multiple signals going from initiator to the Timer then there shall be multiple extensions that the target_socket of Timer is configured to accept. So the initiator should only validate the particular extension that it wishes to send.

At the Timer end the target_signal_socket receives the above b_transport call, and can use following API's to access the value

```
{
  if (in_socket.get_extension<GATE>(trans)) { //true only if this extension was validated at initiator
```

```
    bool vaue = in_socket.get_last_value<GATE>(); //get value set for extension
  }
}
```

Note that the *get_last_value<EXT>()* does not require the *trans* argument, so at the target end, one can read the value of a signal at any time during simulation and is not restricted to do so only when the initiator makes a transport call.

Finally, before using any of the extensions in the configuration of sockets or transmission or reception of payload containing extensions, the extensions need to be declared at a common place visible to both the initiator and target. Following declarations are required for the above example.

```
GS_GUARD_ONLY_EXTENSION(OUT)
GS_GUARD_ONLY_EXTENSION(GATE)
```

## 9. References:

1. OSCI TLM2 User Manual (version JA22) [http://www.systemc.org/home ]
2. GreenSockets User Guide
3. SignalSockets examples