GreenSocket - Overview

Robert Günzel (GreenSocs)

Abstract

GreenSocket is intended to be the base for more advanced and customized sockets. It offers memory management facilities for transactions and extensions, based on the configuration of the socket, as well as a callback registration for the interface method calls of TLM2.0. During runtime two connected GreenSockets exchange configuration information to determine the largest possible intersection of used extensions and phases. Afterwards the socket calls virtual functions on "'itself"' to indicate a successful binding. Sockets derived from GreenSocket can override those functions to e.g. inform the owning modules about the configuration intersection or to perform additional bindability checks.

Hence, a plain (un-derived) GreenSocket will not provide this information to the owning modules. However, modules can actively get the resolved configuration at start_of_simulation or later.

Together with GreenSocket comes a special TLM-2.0 compliant, non-restricting extension type philosophy (See chapter 2 of the GreenSocket Overview for details) as well as a TLM-2.0 compliant, non-restricting extension access philosophy (namely GVI*, see chapter 3.4 of the GreenSocket Overview for details).*

Contents

1 Preface

This document gives a detailed overview over the API of GreenSocket. It can either be read from top to bottom, or can be read in a use case based way. That means, you should find your use case in section *Inclusions and namespaces*, read about it there and follow the links from there to more detailed information about your use case. Whenever you find things missing in the user guide, please contact the author.

GreenSocket comes in three flavors: once as an initiator socket and once as a target socket and once as a bidirectional socket, which is a unification of the former two sockets. The initiator socket allows for using the TLM2.0 forward interface and for registering callbacks for the TLM2.0 backwards interface. The target socket does the same but the other way around. Finally the bidirectional socket allows for both.

In the Pipe: Upcoming versions of this document will contain

- Information on how to monitor GreenSocket connections.
- Information on how to customize GreenSocket monitoring.
- A sophisticated example.

2 Inclusions and namespaces

Greensocket is distributed as a header only release, so after getting it from GreenSocs you only have to include the correct files to start using it. There are a number of files within the package, but only few are meant to be used directly. This section lists the most important ones and are sorted by use cases. If you find your use case missing in this list, please contact the author of this document.

Note that inclusions with larger major enumeration numbers include all the files with lower major enumeration numbers.

1. Defining own extensions:

   When defining your extensions you will need to include

   `#include "greensocket/generic/gs_extension.h"`

   Afterwards you will have access to all members of namespace `gs::ext` and the extensions definition macros, as mentioned in section *Defining your Extensions* below.

2. Working with GreenSocket configurations:

   When working with configurations for GreenSocket you will have to include

   `#include "greensocket/generic/green_socket_config.h"`

   Afterwards you will have access to the class `config` in namespace `gs::socket`, as described in section *Working with Configurations* below.

3. Using or extending a basic GreenSocket:

   You have the choice of including one, two or all three flavors of GreenSocket.

   (a) Initiator socket that can be bound to exactly one target:

   `#include "greensocket/initiator/single_socket.h"`

   Afterwards you will have access to the class `initiator_socket` in namespace `gs::socket`, as described in section *Working with Initiator Sockets* below.

   Initiator socket that can be bound to many targets:

   `#include "greensocket/initiator/multi_socket.h"`

   Afterwards you will have access to the class `initiator_multi_socket` in namespace `gs::socket`, as described in section *Working with Initiator Sockets* below.

   (b) Target socket that can be bound to exactly one initiator:

   `#include "greensocket/target/single_socket.h"`

   Afterwards you will have access to the class `target_socket` in namespace `gs::socket`, as described in section *Working with Target Sockets* below.

   Target socket that can be bound to many initiators:

   `#include "greensocket/target/multi_socket.h"`

   Afterwards you will have access to the class `target_multi_socket` in namespace `gs::socket`, as described in section *Working with Target Sockets* below.

(c) Bidirectional socket that can be bound to exactly one bidirectional socket:

`#include "greensocket/bidirectional/single_socket.h"`

Afterwards you will have access to the class `bidirectional_socket` in namespace `gs::socket`, as described in section *Working with Bidirectional Sockets* below.

Bidirectional socket that can be bound to many bidirectional sockets:

`#include "greensocket/bidirectional/multi_socket.h"`

Afterwards you will have access to the class `bidirectional_multi_socket` in namespace `gs::socket`, as described in section *Working with Bidirectional Sockets* below.

3 Defining your Extensions

First of all it is important to mention that the GreenSockets can only exploit their runtime bindability check capabilities if all extensions come from a single shared extension repository. When defining your GreenSocket configuration you should first visit such a repository and look if there already is an extension that matches your needs. Only if you do not find what you need, you should define a new one, and submit it together with a description back into the repository. Otherwise we might end up having a `foo_priority` and a `bar_priority` extension which are functionally equivalent but will fail to bind because GreenSocket bind checks are type based, and the types differ.

3.1 Guard Only Extensions

To define a guard only extension use the macro `GS_GUARD_ONLY_EXTENSION(name)`.

For example `GS_GUARD_ONLY_EXTENSION(cacheable);` will define a guard extension type named cacheable.

3.2 Data Only Extensions

To define a data only extension use the macro `GS_DATA_ONLY_EXTENSION(name)`. As you can see there is no terminating `;` because this time you will have to provide the body for the extension.

The things you have to provide are:

1. `void copy_from(tlm::tlm_extension_base const & ext)`

   Within this function you will have to copy the data members of function argument `ext` into the local members of this extension.

2. `tlm::tlm_extension_base* clone() const`

   Within this function you will have to create a new instance of this extension, copy the local members of this extension into the new instance and return the pinter to this instance.

3. `gs_extension_base* construct() const`

   Within this function you shall create an fresh and new instance of this extension and return the pointer to it.

4. `std::string dump() const`

   Within this function you shall dump the current state of the extension. This has to be done in a special format, so that it matches the complete transaction format when printed e.g. into a debug file.

   The format is

   ```
   </extension name="typename" type="data"
   member1="value_of_member1_as_string"
   member2="value_of_member2_as_string"
   member3="value_of_member3_as_string"
   ...
   >
   ```

   Note that the line feeds are only there to make the format fit into this document. The actual returned string must not contain newlines, and that the type is fixed. For example if your extension was called `baz` and it has two members `int j;` and `char c;` the returned string should be

   `</extension name="baz" type="data" j="2" c="a">`

   Given that j is 2 and c is 'a' at the time `dump` is called.

Example: Defining a data only extension for priority:

```
GS_DATA_ONLY_EXTENSION(priority){
   void copy_from(tlm::tlm_extension_base const & ext){
     const priority* tmp=static_cast<const priority*>(&ext);
     this->prio=tmp->prio;
   }
   tlm::tlm_extension_base* clone() const {
     priority* tmp=new priority();
     tmp->prio=this->prio;
     return tmp;
   }
   gs_extension_base* construct() const{
     return new priority();
   }
   std::string dump() const{
     std::stringstream s;
     s<<"</extension name=\"priority\" type=\"data\" prio=\""<<prio<<"\">";
     return s.str();
   }
   unsigned int prio;
 };
```

Since in the majority of cases extensions carry just a single member, we also provide the macro `SINGLE_MEMBER_DATA(name, type);`, which will set up a data only extension with name `name` and a single member of type `type`. The member's name will be `value`. Note that only types are supported for which `std::ostream& operator<<(std::ostream &, const type &)` is available.

For example `SINGLE_MEMBER_DATA(priority, unsigned int);` would define a priority extension just like in the example above, but as mentioned before the actual member of the extension would be called `value`.

3.3 Guarded Data Extensions

Guarded data extensions are defined exactly like data only extensions, however the macro to use is `GS_GUARDED_DATA_EXTENSION(name)`, and the type that is part of the dump format is `guarded_data`.

Example: Defining a data only extension for burst sequence:

```
GS_GUARDED_DATA_EXTENSION(burstsequence){
   void copy_from(tlm::tlm_extension_base const & ext){
     const burstsequence* tmp=static_cast<const burstsequence*>(&ext);
     this->seq=tmp->seq;
   }
   tlm::tlm_extension_base* clone() const {
     burstsequence* tmp=new burstsequence();
     tmp->seq=this->seq;
     return tmp;
   }
   gs_extension_base* construct() const{
     return new burstsequence();
   }
   std::string dump() const{
     std::stringstream s;
     s<<"</extension name=\"burstsequence\" type=\"guarded_data\" seq=\""<<seq<<"\">";
     return s.str();
   }
   unsigned int seq;
 };
```

Since in the majority of cases extensions carry just a single member, we also provide the macro `SINGLE_MEMBER_GUARDED_DATA(name, type);`, which will set up a guardeddata extension with name `name` and a single member of type `type`. The member's name will be `value`. Note that only types are supported for which `std::ostream& operator<<(std::ostream &, const type &)` is available.

For example `SINGLE_MEMBER_GUARDED_DATA(burstsequence, unsigned int);` would define a burst sequence extension just like in the example above, but as mentioned before the actual member of the extension would be called `value`.

3.4 Bindability enum

Together with the macros a enumerated type is defined that defines the possible bindability states of extensions (see the GreenSocket Overview for details): `gs::ext::gs_extension_bindability_enum;`.

The enumerated values are:
`gs_optional=0, gs_mandatory=1, gs_reject=2`

4 Working with Configurations

All GreenSockets share the same configuration facility. It allows for configuring the socket, with respect to the used extensions and phases. The configuration of a GreenSocket is an object that shall be instantiated by the user, be set up according to the users needs, and then shall be assigned to the socket. Whenever two GreenSockets are bound they compare their configurations, try to find the largest possible working intersection of their configurations, and if they succeed, they will change their configurations to represent the intersection.

The user can then read the configuration back from the socket, to investigate the resulting intersection.

The class name is `template<typename TRAITS> gs::socket::config`. The template argument shall be the traits class (e.g. `tlm::base_protocol_types`) used by the socket to which the configuration shall be applied.

4.1 Construction

The provided constructor does not take any arguments `config()`. For example creating a GreenSocket configuration named `cfg` on the stack would be done by

```
gs::socket::config<tlm::base_protocol_types> cfg;
```

while the construction on the heap would be done by

```
gs::socket::config<tlm::base_protocol_types>* cfg=
                new gs::socket::config<tlm::base_protocol_types>();
```

Also a copy constructor is available: `config(const config&)`.

4.2 Configuring Use of Extensions

Note that GreenSocket configurations only support extensions defined by the macros mentioned in the chapter above. The functions provided to configure the use of extensions are

1. `template <typename T> void use_mandatory_extension();`

   This function will add extension type T as mandatory to the configuration. Example: Add extension type `cacheable` as mandatory to the configuration.

   ```
   gs::socket::config<tlm::base_protocol_types> conf;
   conf.use_mandatory_extension<cacheable>();
   ```

2. `template <typename T> void use_optional_extension();`

   This function will add extension type T as optional to the configuration. Example: Add extension type `cacheable` as optional to the configuration.

   ```
   gs::socket::config<tlm::base_protocol_types> conf;
   conf.use_optional_extension<cacheable>();
   ```

3. `template <typename T> void reject_extension();`

   This function will add extension type T as rejected to the configuration. Example: Add extension type `cacheable` as rejected to the configuration.

   ```
   gs::socket::config<tlm::base_protocol_types> conf;
   conf.reject_extension<cacheable>();
   ```

4. `template <typename T> void remove_extension();`

   This function will remove extension type T from the configuration. Example: Remove extension type `cacheable` from the configuration.

   ```
   gs::socket::config<tlm::base_protocol_types> conf;
   conf.remove_extension<cacheable>();
   ```

5. `template <typename T> gs::ext::gs_extension_bindability_enum has_extension() const;`

   This function will return how the configuration treats the extension type T. That might be helpful to find out if a certain extension can be used or not after binding has completed. Example: Find out how the configuration deals with extension type `cacheable`

```
switch(conf.has_extension<cacheable>()){
  case gs::ext::gs_optional: std::cout<<"It's up to me to"; break;
  case gs::ext::gs_mandatory: std::cout<<"I'll have to"; break;
  case gs::ext::gs_reject: std::cout<<"I must not"; break;
}
std::cout<<" use the cacheable extension."<<std::endl;
```

   Note: If you use this function with a TLM-2 ignorable extension (i.e. it is not part of the bindability checks and configs) you will get back the treatment value for unknown extension (see section *Configuring How to Handle Unknown Phases or Extension*).

4.3 Configuring Use of Phases

The functions provided to configure the use of phases (both TLM base protocol phases or additional phases that were created using the TLM macro `DECLARE_EXTENDED_PHASE`) are

1. `void use_mandatory_phase(unsigned int ph1)`

   This function will add phase `ph1` as mandatory to the configuration. Example: Add phase `BEGIN_REQ` as mandatory to the configuration.

```
gs::socket::config<tlm::base_protocol_types> conf;
conf.use_mandatory_phase(tlm::BEGIN_REQ);
```

2. `void use_optional_phase(unsigned int ph1)`

   This function will add phase `ph1` as optional to the configuration. Example: Add phase `BEGIN_REQ` as optional to the configuration.

```
gs::socket::config<tlm::base_protocol_types> conf;
conf.use_otpional_phase(tlm::BEGIN_REQ);
```

3. `void reject_phase(unsigned int ph1)`

   This function will add phase `ph1` as rejected to the configuration. Example: Add phase `BEGIN_REQ` as rejected to the configuration.

```
gs::socket::config<tlm::base_protocol_types> conf;
conf.reject_phase(tlm::BEGIN_REQ);
```

4. `void remove_phase(unsigned int ph1)`

   This function will remove phase `ph1` from the configuration. Example: Remove phase `BEGIN_REQ` from the configuration.

```
gs::socket::config<tlm::base_protocol_types> conf;
conf.remove_phase(tlm::BEGIN_REQ);
```

5. `gs::ext::gs_extension_bindability_enum has_phase(unsigned int ph) const;`

   This function will return how the configuration treats phase `ph`. That might be helpful to find out if a certain phase can be used or not after binding has completed. Example: Find out how the configuration deals with phase `BEGIN_REQ`

```
switch(conf.has_phase<tlm::BEGIN_REQ>()){
  case gs::ext::gs_optional: std::cout<<"It's up to me to"; break;
  case gs::ext::gs_mandatory: std::cout<<"I'll have to"; break;
  case gs::ext::gs_reject: std::cout<<"I must not"; break;
}
std::cout<<" use phase BEGIN_REQ."<<std::endl;
```

   Note: If you use this function with a TLM-2 ignorable phase (i.e. it is not part of the bindability checks and configs) you will get back the treatment value for unknown phases. (see section *Configuring How to Handle Unknown Phases or Extension*).

4.4 Configuring How to Handle Unknown Phases or Extension

When GreenSockets are bound to each other, it may happen that one socket knows about extensions/phases the other socket doesn't know about. You can tell the configuration what to do in this case.

1. `void treat_unknown_as_optional();`

   This function will tell the configuration to treat unknown extensions/phases as optional, when being compared during the bindability check. Example:

   ```
   gs::socket::config<tlm::base_protocol_types> conf;
   conf.treat_unknown_as_optional();
   ```

2. `void treat_unknown_as_rejected();`

   This function will tell the configuration to treat unknown extensions/phases as rejected, when being compared during the bindability check. Example:

   ```
   gs::socket::config<tlm::base_protocol_types> conf;
   conf.treat_unknown_as_rejected();
   ```

3. `bool unknowns_are_optional();`

   This function will return true if unknowns are treated as optional, and false if they are treated as rejected.

4.5 Auxiliary functions

Some functions are in place to help working with configurations.

1. `config& operator=(const config&);`

   Copies of configurations are allowed.

2. `std::string to_string() const;`

   This function will return a string representation of the configuration. It might be helpful for debugging and analysis.

   Example:

   ```
   #include "greensocket/generic/green_socket_config.h"

   GS_GUARD_ONLY_EXTENSION(foo);
   SINGLE_MEMBER_GUARDED_DATA(bar, int);
   SINGLE_MEMBER_DATA(baz, int);

   DECLARE_EXTENDED_PHASE(ZOO);
   DECLARE_EXTENDED_PHASE(GOO);
   DECLARE_EXTENDED_PHASE(BOO);

   int main(int, char**){
     gs::socket::config<tlm::tlm_base_protocol_types> cfg;
     cfg.use_mandatory_extension<foo>();
     cfg.use_optional_extension<bar>();
     cfg.reject_extension<baz>();
     cfg.use_mandatory_phase(ZOO);
     cfg.use_optional_phase(GOO);
     cfg.reject_phase(BOO);
     cfg.treat_unknown_as_optional();

     std::cout<<cfg.to_string()<<std::endl;
   }
   ```

   The output will be

   ```
   Note: Unassigned configuration.
     Treats unknown as optional.
     Used extensions:
       foo used as mandatory.
       bar used as optional.
   ```

```
    baz used as rejected.
    bar_guard used as optional.
  Used phases:
    BOO used as rejected.
    GOO used as optional.
    ZOO used as mandatory.
```

Note that the configuration has detected that it has not been assigned to a socket yet, and that there is an extension `bar` and an extension `bar_guard`. The former is the data part of the guarded data extension `bar`, while the latter is its guard. Don't be scared! That is the only time you will explicitly notice the split personality of a guarded data extension.

4.6 Advanced functions

Some functions are provided for advanced users, especially for those who want to develop sockets on top of GreenSocket.

1. `void set_force_reeval(bool val);`

   When a configuration is assigned to socket after it was previously configured, the socket will reevaluate the binding only if the new configuration differs from one that is already in the socket. However, customized sockets may use the callbacks that are generated due to a bindability check to trigger certain activities in the other socket. In this case you can use this function to enforce GreenSocket to reevaluate the binding and to trigger all callbacks again, even if the GreenSocket configuration did not change.

2. `bool merge_with(const char*, const char*, config&, bool abort_at_mismatch=true);`

   This function 'merges' a configuration with another. The function is used by GreenSocket to do the bindability check, but it might be useful for users that want to determine the largest possible intersection of two configurations. The first const char* shall identify the owner of the configuration on which the function is called, the second const char* shall identify the owner of the configuration that is passed as the third argument (these const chars* will be used in the error messages in case the two configurations are not bindable). The final boolean value determines whether to abort the simulation in case of a mismatch or not. The return value will return true if there was a mismatch and false if the merge succeeded.

   The merge `A.merge_with("A","B", B)` will only be successful if

   - all mandatory extension and phases of A are mandatory or optional in B
   - no rejected extension or phase of A is mandatory in B
   - A treats unknown extensions/phases as rejected but there are no unknown extension/phases in B

   After the merge configuration A changes:

   - mandatory extensions stay mandatory (no change)
   - rejected extensions stay rejected (no change)
   - optional extensions that were rejected in B become rejected in A
   - optional extension that were mandatory in B are now mandatory in A
   - optional extension that are optional in B stay optional
   - if B treats unknowns as rejected, optionals unknown to B become rejected in A
   - if B treats unknowns as optional, optionals unknown to B stay optional in A
   - if A treats unknowns as optional, unknowns in B are added like the are in B to A
   - all the above points for phases
   - if B treats unknowns as rejected A will do so now

5 Working with Initiator Sockets

The two flavors of initiator sockets (single or multi socket) will be described in this section. The single socket class is defined as

```
template <unsigned int BUSWIDTH=32,typename TRAITS=tlm::tlm_base_protocol_types>
  class initiator_socket;
```

and the multi socket is defined as

```
template <unsigned int BUSWIDTH=32,typename TRAITS=tlm::tlm_base_protocol_types>
  class initiator_multi_socket;
```

both reside in namespace `gs::socket`. The former must be bound to exactly one target socket, while the latter may be bound to an unlimited number of target sockets, but at least one.

For those who have had a look into the code: There are more template arguments than the ones listed here, but those should not be touched. Never.

Throughout this chapter, differences between the two flavors will be mentioned explicitly, otherwise the single socket will be used in examples.

5.1 Construction

The provided constructors for the sockets do not differ in their signatures:

```
initiator_socket(const char* name,
    gs::socket::allocation_scheme alloc_scheme=gs::socket::GS_TXN_ONLY);
initiator_mutli_socket(const char* name,
    gs::socket::allocation_scheme alloc_scheme=gs::socket::GS_TXN_ONLY);
```

The first argument shall be the name of the socket, just as known from naming `sc_ports`, while the latter will define what the socket's pool will be doing. The options are:

1. `gs::socket::GS_TXN_ONLY`

   The socket will just pool allocate the transactions of the type specified in the traits class (see the TLM-2.0 reference manual to understand what the traits class is).

2. `gs::socket::GS_TXN_WITH_DATA`

   The socket will pool allocate the transactions of the type specified in the traits class, and in addition will prepare to pool allocate data arrays if needed.

3. `gs::socket::GS_TXN_WITH_BE`

   The socket will pool allocate the transactions of the type specified in the traits class, and in addition will prepare to pool allocate byte enable arrays if needed.

4. `gs::socket::GS_TXN_WITH_BE_AND_DATA`

   The socket will pool allocate the transactions of the type specified in the traits class, and in addition will prepare to pool allocate byte enable arrays and data arrays if needed.

See this chapter's subsection *Using the Transaction Memory Management* for more details on how to deal with the transaction pool.

Example: A module that instantiates an initiator socket with its pool being able to pool transactions, data arrays and byte enable arrays.

```
#include "greensocket/initiator/single_socket.h"

SC_MODULE(foo){
  gs::socket::initiator_socket i_socket;

  SC_CTOR(foo) : i_socket("i_socket", gs::socket::GS_TXN_WITH_BE_AND_DATA)
  {
  ...
  }
  ...
};
```

5.2 Public Type Definitions

The sockets make some type definitions publicly available. They are:

1. `typedef TRAITS traits_type;`

   This type definition matches what was passed as the template argument `TRAITS` to the socket.

   Example: A module that gets a complete socket as a template argument and instantiates a potentially different socket with the same traits class.

```
template <typename FIRSTSOCKET>
class foo : public sc_core::sc_module
{
public:
  FIRSTSOCKET sock1;
  gs::socket::initiator_socket<32, typename FIRSTSOCKET::traits_type> sock2;

  foo(sc_core::sc_module_name name)
    : sc_core::sc_module(name)
    , sock1("sock1", gs::socket::GS_TXN_ONLY)
    , sock2("sock2", gs::socket::GS_TXN_ONLY)
  {
  }
};

int main(int, char**){
  //instantiate a foo module with a multi socket and a single socket
  foo<gs::socket::initiator_multi_socket<32, tlm::tlm_base_protocol_types> > my_foo1("my_foo1");

  //instantiate a foo module with two single sockets
  foo<gs::socket::initiator_socket<32, tlm::tlm_base_protocol_types> > my_foo2("my_foo2");

}
```

2. `typedef typename TRAITS::tlm_payload_type payload_type;`

   This type definition allows to get the type of payload that is going over a socket.

   Example: A module templated on a traits class that it will pass on to its socket and that instantiates a transaction of the correct type for its socket.

```
template <typename TRAITS>
class foo : public sc_core::sc_module
{
public:
  gs::socket::initiator_socket<32, TRAITS> sock;

  foo(sc_core::sc_module_name name)
    : sc_core::sc_module(name)
    , sock("sock", gs::socket::GS_TXN_ONLY)
  {
  }

  //instantiate a transaction that fits socket 'sock'
  typename gs::socket::initiator_socket<32, TRAITS>::payload_type m_txn;
};
```

3. `typedef typename TRAITS::tlm_phase_type phase_type;`

   This type definition allows to get the type of phase that is going over a socket.

   Example: A module templated on a traits class that it will pass on to its socket and that instantiates a phase of the correct type for its socket.

```
template <typename TRAITS>
class foo : public sc_core::sc_module
{
public:
  gs::socket::initiator_socket<32, TRAITS> sock;
```

```
  foo(sc_core::sc_module_name name)
    : sc_core::sc_module(name)
    , sock("sock", gs::socket::GS_TXN_ONLY)
  {
  }

  //instantiate a phase that fits socket 'sock'
  typename gs::socket::initiator_socket<32, TRAITS>::phase_type m_ph;
};
```

5.3 Determining the Number of Binding

To determine the number of bindings of a multi socket `unsigned int size();` will return the number of bindings. Note that this will only return a reliable value after construction time (just as for `sc_ports`). Calling it on a single socket will always return one after the socket has been bound and zero otherwise.

5.4 Getting the Name of a Socket

`const char* get_name();` will return the hierarchical name of the socket.

5.5 Getting the Bus Width of a Socket

`unsigned int get_bus_width();` will return the bus width that was passed as a template argument to the socket.

5.6 Getting the Connected Socket

After `end_of_elaboration` has been called on a socket, the function
`gs::socket::bindability_base<TRAITS>* get_other_side(int a, int& b);`
will return the pointer to the socket that is bound at index `a`. The integer that has to be provided by reference as argument `b` will be set to the index at which the socket is bound on the other side.

The `bindability_base` offers some functions that may be of help and shall be listed here:

- `const char* get_name();`

  This function returns the hierarchical name of the connected socket.

- `sc_core::sc_object* get_parent();`

  This function returns the `sc_object` that is the owner of the GreenSocket.

5.7 Configuring the Socket

A GreenSocket has to have a configuration before the `end_of_elaboration` callback of SystemC. However, you may still modify the configuration of a socket from within the `end_of_elaboration` callback or in response to the completion of another binding (which is also at `end_of_elaboration`). You may not change the configuration later on.

A single socket can only have a single configuration, while a multi socket has a configuration per bound target socket.

The functions provided for configuration are:

1. `void set_config(const gs::socket::config<TRAITS>& 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<TRAITS>& 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<TRAITS>& 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<TRAITS>& 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.

Example: Assigning an initial configuration to a multi socket and a single socket, and investigating the bind results for the multi socket at `start_of_simulation`.

```
GS_GUARD_ONLY_EXTENSION(compressed);

class foo : public sc_core::sc_module
{
public:
  typedef gs::socket::initiator_socket<> single_socket_type;
  typedef gs::socket::initiator_multi_socket<> mutli_socket_type;
  single_socket_type sock;
  mutli_socket_type mul_sock;

  SC_HAS_PROCESS(foo);

  foo(sc_core::sc_module_name name)
    : sc_core::sc_module(name)
    , sock("sock", gs::socket::GS_TXN_ONLY)
    , mul_sock("mul_sock", gs::socket::GS_TXN_ONLY)
  {
    gs::socket::config<single_socket_type::traits_type> cfg;

    //a tlm base protocol only configuration
    cfg.use_mandatory_phase(tlm::BEGIN_REQ);
    cfg.use_mandatory_phase(tlm::END_REQ);
    cfg.use_mandatory_phase(tlm::BEGIN_RESP);
    cfg.use_mandatory_phase(tlm::END_RESP);
    cfg.treat_unknown_as_rejected();

    //the single socket can only talk OSCI BP
    sock.set_config(cfg);

    //OSCI BP + one extension
    cfg.use_optional_extension<compressed>();

    //the multi socket could issue compressed transfers
    mul_sock.set_config(cfg);

    SC_THREAD(run);
  }

  void start_of_simulation(){
    //if we reach this point without aborting the
    // simulation, all targets had fitting configurations
    // now let's see which one of them supports compression
    m_can_do_compression.resize(mul_sock.size());
    for (unsigned int i=0; i<mul_sock.size(); i++)
      m_can_do_compression[i]=mul_sock.get_recent_config(i).has_extension<compressed>()!=gs::ext::gs_reject;
    //now during simulation we have very quick access to
    // the info at which link we can send compressed data
    // and on which we have to send uncompressed data
  }

  void run(){...}

  std::vector<bool> m_can_do_compression;
};
```

5.8 Using the Transaction Memory Management

The initiator sockets provide memory management facilities to the user. However, the user is not forced to use them. He may also choose to use its own pools and memory managers, to stack or to heap allocate transactions. The functions offered by the socket are:

1. `payload_type* 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.

   Note that no reset is performed on the payload object, so you will have to set each member of it by hand. The only known fact is that all no guard or guarded data extensions set.

   An example can be found when looking a the `release_transaction` function.

2. `void release_transaction(payload_type* 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. The main reason for the existence of the function is symmetry: If I need to do a call on the socket to get a transaction, there should be a call to 'return' the transaction to the socket.

   Example: Getting a transaction from the socket, working with it and releasing it.

```
class foo : public sc_core::sc_module
{
public:
  typedef gs::socket::initiator_socket<> socket_type;
  socket_type sock;

  SC_HAS_PROCESS(foo);

  foo(sc_core::sc_module_name name)
    : sc_core::sc_module(name)
    , sock("sock", gs::socket::GS_TXN_ONLY)
  {
    SC_THREAD(run);
  }

  void run(){
    socket_type::payload_type* txn=sock.get_transaction();
    do_fancy_communication(txn);
    sock.release_transaction(txn);
  }

  void do_fancy_communication(socket_type::payload_type* txn){
    ...
  }
};
```

3. `void reserve_data_size(payload_type& txn, unsigned int bytes);`

   This function reserves at least `bytes` bytes for the data array of `txn`. This includes setting the data pointer and data length inside the payload object.

   The reservation is persistent, i.e. when the payload object comes out of the pool again, it will still have the same data size reservation. Subsequent calls to this function will always set up the data length and data pointer, but memory allocation will only be necessary when `bytes` is larger than what was reserved previously.

   Note that this function will only operate correctly when the socket was constructed with `gs::socket::GS_TXN_WITH_DATA` or `gs::socket::GS_TXN_WITH_BE_AND_DATA`. This is guarded by assertions, so will be easily detectable in debug compilations.

   Note that you can still set your own data pointer and data length if desired. You are never forced to use the memory reservation mechanism.

   You will find an example for this function in the description of the `reserve_be_size` function.

4. `void reserve_be_size(payload_type&, unsigned int);`

   This function reserves at least `bytes` bytes for the byte enable array of `txn`. This includes setting the byte enable pointer and byte enable length inside the payload object.

   The reservation is persistent, i.e. when the payload object comes out of the pool again, it will still have the same byte enable size reservation. Subsequent calls to this function will always set up the byte enable length and byte enable pointer, but memory allocation will only be necessary when `bytes` is larger than what was reserved previously.

   Note that this function will only operate correctly when the socket was constructed with `gs::socket::GS_TXN_WITH_BE` or `gs::socket::GS_TXN_WITH_BE_AND_DATA`. This is guarded by assertions, so will be easily detectable in debug compilations.

   Note that you can still set your own byte enable pointer and byte enable length if desired. You are never forced to use the memory reservation mechanism.

   Example: Reserving data and byte enable sizes.

```
class foo : public sc_core::sc_module
{
public:
  typedef gs::socket::initiator_socket<> socket_type;
  socket_type sock;

  SC_HAS_PROCESS(foo);

  foo(sc_core::sc_module_name name)
    : sc_core::sc_module(name)
    , sock("sock", gs::socket::GS_TXN_WIDTH_BE_AND_DATA)
  {
    SC_THREAD(run);
  }

  void run(){
    socket_type::payload_type* txn=sock.get_transaction();
    do_fancy_communication(txn);
    sock.release_transaction(txn);
  }

  void do_fancy_communication(socket_type::payload_type* txn){
    //make space for 10 bytes of data
    sock.reserve_data_size(*txn, 10);
    fill_data(txn, 10);
    //we wanna disable every second byte
    sock.reserve_be_size(*txn, 2);
    txn->get_byte_enable_ptr()[0]=TLM_BYTE_ENABLED;
    txn->get_byte_enable_ptr()[1]=TLM_BYTE_DISABLED;
  }

  void fill_data(socket_type::payload_type*, unsigned int){
  }
};
```

5. `unsigned int get_reserved_data_size(payload_type&);`

   This function returns the currently reserved data size. It may be used to find out if a call to `reserve_data_size` is required, but then you will have to set the data length and data pointer manually, when you did not call `reserve_data_size`.

6. `unsigned int get_reserved_be_size(payload_type&);`

   This function returns the currently reserved byte enable size. It may be used to find out if a call to `reserve_be_size` is required, but then you will have to set the byte enable length and byte enable pointer manually, when you did not call `reserve_be_size`.

5.9 Registering Callbacks

You may register callbacks for all interface method calls of the TLM-2.0 backwards interface:

1. `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&)
   ```

   There is no need to explain the three parameter and the return value of the first signature, as this is done by the TLM-2.0 reference manual. The `unsigned int` in the multi socket signature will carry the index of the binding over which the call came in, so that the callers can be distinguished.

2. `template<typename MODULE> void register_invalidate_direct_mem_ptr(MODULE* mod,`*inval_dmi_cb* `cb)`

   This function allows to register a callback for `invalidate_direct_mem_ptr`. The type of *inval_dmi_cb* differs between single and multi sockets.

   For single sockets it is

   ```
   void (MODULE::*)(sc_dt::uint64, sc_dt::uint64)
   ```

   and for multi sockets it is

   ```
   void (MODULE::*)(unsigned int, sc_dt::uint64, sc_dt::uint64)
   ```

   There is no need to explain the two parameter of the first signature, as this is done by the TLM-2.0 reference manual. The `unsigned int` in the multi socket signature will carry the index of the binding over which the call came in, so that the callers can be distinguished.

Example: Registering callback for `nb_transport_bw` for both a multi and a single socket

```
class foo : public sc_core::sc_module
{
public:
  typedef gs::socket::initiator_socket<> single_socket_type;
  typedef gs::socket::initiator_multi_socket<> mutli_socket_type;
  single_socket_type sock;
  mutli_socket_type mul_sock;

  SC_HAS_PROCESS(foo);

  foo(sc_core::sc_module_name name)
    : sc_core::sc_module(name)
    , sock("sock", gs::socket::GS_TXN_ONLY)
    , mul_sock("mul_sock", gs::socket::GS_TXN_ONLY)
  {
    sock.register_nb_transport_bw(this, &foo::my_single_socket_nb_transport_bw);
    mul_sock.register_nb_transport_bw(this, &foo::my_multi_socket_nb_transport_bw);
  }

  tlm::tlm_sync_enum my_single_socket_nb_transport_bw(
    single_socket_type::payload_type& txn,
    single_socket_type::phase_type&   ph,
    sc_core::sc_time&                 t)
  {
    return tlm::TLM_ACCEPTED;
  }

  tlm::tlm_sync_enum my_multi_socket_nb_transport_bw(
    unsigned int                      index,
    single_socket_type::payload_type& txn,
    single_socket_type::phase_type&   ph,
    sc_core::sc_time&                 t)
  {
    return tlm::TLM_ACCEPTED;
  }
};
```

5.10 Calling Transport Functions

The initiator sockets allow for calling the functions of the TLM-2.0 forward interface, just like 'normal' TLM-2.0 sockets.

1. `tlm::tlm_sync_enum nb_transport_fw(payload_type&,phase_type&,sc_core::sc_time&)`
2. `void b_transport(payload_type&,sc_core::sc_time&)`
3. `bool get_direct_mem_ptr(payload_type&,tlm::tlm_dmi&)`
4. `unsigned int transport_dbg(payload_type&)`

Single socket shall use the `->` operator to call the functions, while multi sockets shall use the `[]` operator in conjunction with the `->` operator. When the `->` is used without the `[]` operator by a multi socket, the call will be performed on binding zero.

Example: Calling `b_transport` on a single and a multi socket.

```
class foo : public sc_core::sc_module
{
public:
  typedef gs::socket::initiator_socket<> single_socket_type;
  typedef gs::socket::initiator_multi_socket<> mutli_socket_type;
  single_socket_type sock;
  mutli_socket_type mul_sock;

  SC_HAS_PROCESS(foo);

  foo(sc_core::sc_module_name name)
    : sc_core::sc_module(name)
    , sock("sock", gs::socket::GS_TXN_ONLY)
    , mul_sock("mul_sock", gs::socket::GS_TXN_ONLY)
  {
    SC_THREAD(run);
    configure_sockets();
  }

  void run(){
    //we stack allocate a non memory managed txn
    //for b_transport
    //we will use it both for the single and the
    //multi socket, since both payload types
    //are the same
    single_socket_type::payload_type txn;
    sc_core::sc_time t=sc_core::SC_ZERO_TIME;

    //first the single socket
    setup_txn(txn);
    sock->b_transport(txn,t);
    check_results(txn);

    //now the multi socket (one transfer per link)
    for (unsigned int i=0; i<mul_sock.size(); i++){
      t=sc_core::SC_ZERO_TIME;
      setup_txn(txn);
      sock[i]->b_transport(txn,t);
      check_results(txn);
    }
  }

  void configure_sockets(){...}
  void setup_txn(single_socket_type::payload_type& txn){...}
  void check_results(single_socket_type::payload_type& txn){...}
};
```

5.11 Accessing Extension

As mentioned in the GreenSocket Overview the GreenSockets use s smaller and simpler set of functions to access extensions. This is only possible when operating with guard, data or guarded data extensions only. When using other extensions than GreenSocket extension (which is of course possible) you are responsible for their memory management, and to set and release/clear them properly, by using the member functions of the payload object directly.

The functions that operator with GreenSocket extensions are:

1. `template <typename T> bool get_extension(T*& ptr, payload_type& txn);`

   Use this function to get an extension from a transaction and it can be used for all types of extensions (guard, data, guarded data). The return value shows if the extension was present or not.

   For guards the `ptr` argument is just used to determine the type but it should not be used after the call.

   For data extensions the return value will always be true, because their validity state cannot be determined. `ptr` will always point to a valid extension, and you can directly access it after the call of `get_extension`.

   For guarded data extensions, the return value shows if the extension is valid or not, but in either case `ptr` will point to a ready-to-use extension after the call. If the return value was true, the content of `ptr` can be considered valid, otherwise the content will have no meaning.

   Example: Code snippets per type.

   ```
   //Example guard extension
   my_guard_extension* tmp;  //just used to determine the type
   if (socket.get_extension(tmp, txn) guard_was_set_action();
   else guard_was_not_set_action();

   //Example data extension:
   my_data_extension* tmp;
   socket.get_extension(tmp, txn);  //return value ignored
   tmp->value=33; //set the data of the extension

   //Example guarded data extension:
   my_guarded_data_extension* tmp;
   if (socket.get_extension(tmp, txn)){
     //extension was already set
     work_with_provided_data_action(tmp->value);
   }
   else {
     //extension was not set, so we set it now
     socket.validate_extension<my_guarded_data_extension>(txn);
     tmp->value=33; //set the data of the extension
   }
   ```

2. `template <typename T> T* get_extension(payload_type& txn);`

   This function can only be used for data only and guard extensions. It provides simplified access, where the former call would introduce avoidable overhead.

   ```
   //Example guard extension:
   if (socket.get_extension<my_guard_extension>(txn)) guard_was_set_action();
   else guard_was_not_set_action();

   //Example data extension:
   socket.get_extension<my_data_extension>(txn)->value=42; //just set the data
   ```

3. `template <typename T> bool validate_extension(payload_type& txn, unsigned int index=0);`

   This function immediately validates a guard or a guard of a guarded data extension. It returns true if there was memory management and false if there wasn't The return value is only of value within `b_transport` (because within `nb_transport` there is always memory management). If there was memory management, the memory manager will clean off the extension when the reference count hits zero, if not, it has to be removed after a `b_transport` call returned.

   Example: Code snippets per type, and on how to use the return value

```
//Example guard extension nb_transport:
socket.validate_extension<my_guard_extension>(txn); //set the guard

//Example guarded data extension nb_transport:
socket.validate_extension<my_guarded_data_extension>(txn); //set the guard of the guarded data

//Example b_transport:
void b_transport(txn, ...){
  bool needs_invalidate=!socket.validate<my_guard_extension>(txn);
  out_socket->b_transport(txn, ...);
  if (needs_invalidate) socket.invalidate<my_guard_extension>(txn);
}
```

4. `template <typename T> bool invalidate_extension(payload_type& txn, unsigned int index=0);`

   Example: Code snippets per type, and on how to use the return value

```
//Example guard extension:
socket.invalidate_extension<my_guard_extension>(txn); //set the guard

//Example guarded data extension:
socket.invalidate_extension<my_guarded_data_extension>(txn); //set the guard of the guarded data
```

5.12 Deriving from the Socket

GreenSocket is intended to form the base of protocol specific sockets. Such derived sockets should provide the configuration to the underlying GreenSocket, ensure that access rights to transaction members are respected, and they may support the user with time management (e.g. protocol specific PEQs). Design guide lines on how to build a protocol socket on top of GreenSocket can be found in the GreenSocket methodology documentation (still to come...).

This section will only describe what has to be done technically to derive from GreenSocket.

1. Derive publicly.

   The GreenSocket has to be a public base of the socket to allow the compiler to do implicit type conversions into GreenSocket. If that is not possible binding the derived socket to another GreenSocket will be impossible. If it is absolutely necessary to make the GreenSocket a protected or private base, you will have to provide a type conversion operator into GreenSocket (which would actually render the protected/private derivation useless).

2. Override mandatory virtual function `const std::string& get_type_string();`

   There is one virtual function that you have to override to make your derived socket work properly. This function returns the type string of your socket. That string shall uniquely identify the type of your socket (see function `bound_to` for the meaning of that string). E.g. if your socket's class definition is

```
//a derived socket for OSCI BP traits class only
template< unsigned int BUSWIDTH>
class my_protocol_initiator_socket
: public gs::socket::initiator_socket<BUSWIDTH>
{
...
};
```

   then `get_type_string` should return `"my_protocol_initiator_socket"`.

3. Choose which optional virtual functions to override.

   There are a number of virtual functions you may choose to override, or that you have to override in certain cases. Check the following list to find out which ones you want to/have to override.

   (a) `void bound_to(const std::string&, gs::socket::bindability_base<TRAITS>*, unsigned int);`

   This function is called by the socket as soon as it has successfully passed the bindability check with its connected socket. The string argument is the type string of the other side, the second argument is a pointer to the socket that has just been bound and the third argument is the index of the binding that has just passed the bindability check.

The string can be used to guide the cast on the second argument. E.g. if the type string is a know type, the second argument can be statically casted into the appropriate type.

The pointer to the other side is of type `gs::socket::bindabiliy_base<TRAITS>`, where `TRAITS` matches the traits class of the connection. It will be `NULL` if the socket was bound to a non-GreenSocket TLM-socket. If it was not `NULL` it may be casted into a derived socket type, either trying dynamic casts or using the string argument of the call to perform static casts. Finally the third argument is always zero for single sockets, and it carries the index of the binding for multi sockets.

The `bound_to` callback could be used to e.g. inform the connected module about the resolved configuration via a callback, to perform some special data exchange with sockets whose types are known and could be casted into some derived sockets, and so on.

Call the base implementation of this call before doing anything else in your own implementation.

4. `void before_end_of_elaboration();`

   This is the standard SystemC callback. If you choose to use it in your derived socket you have to call the base implementation to make your socket work properly. Ideally you call the base implementation before doing anything else in the call.

5. `void end_of_elaboration();`

   This is the standard SystemC callback. If you choose to use it in your derived socket you have to call the base implementation to make your socket work properly. Ideally you call the base implementation before doing anything else in the call.

6. `void start_of_simulation();`

   This is the standard SystemC callback. If you choose to use it in your derived socket you have to call the base implementation to make your socket work properly. Ideally you call the base implementation before doing anything else in the call.

7. `void free(payload_type* txn);`

   This function gets called when the reference count of a transaction hits zero. You have to call the base implementation of the function, ideally at the end of the function because it returns the transaction back to the pool.

   You might want to override this function to do some pool profiling in your derived socket, or to trigger an event/callback towards the owner of the socket.

8. `sc_core::sc_object*  get_parent();`

   The default implementation of `get_parent` casts the socket into an `sc_object` and returns the parent of this object. If this will fail for the object hierarchy you built for your derived socket, you will have to override this function and return the pointer to the `sc_object` that really is the owner of your socket.

6 Working with Target Sockets

The two flavors of target sockets (single or multi socket) will be described in this section. The single socket class is defined as

```
template <unsigned int BUSWIDTH=32,typename TRAITS=tlm::tlm_base_protocol_types>
  class target_socket;
```

and the multi socket is defined as

```
template <unsigned int BUSWIDTH=32,typename TRAITS=tlm::tlm_base_protocol_types>
  class target_multi_socket;
```

both reside in namespace `gs::socket`. The former must be bound to exactly one initiator socket, while the latter may be bound to an unlimited number of initiator sockets, but at least one.

Throughout this chapter, differences between the two flavors will be mentioned explicitly, otherwise the single socket will be used in examples.

6.1 Construction

The provided constructors for the sockets do not differ in their signatures:

```
target_socket(const char* name);
target_mutli_socket(const char* name);
```

The only argument shall be the name of the socket, just as known from naming `sc_ports`.
Example: A module that instantiates a target socket.

```
#include "greensocket/target/single_socket.h"

SC_MODULE(foo){
  gs::socket::target_socket t_socket;

  SC_CTOR(foo) : t_socket("t_socket")
  {
  ...
  }
  ...
};
```

6.2 Public Type Definitions

The public type definitions match the ones from the initiator socket.

6.3 Determining the Number of Binding

This is the same like for initiator socket.

6.4 Getting the Name of a Socket

This is the same like for initiator socket.

6.5 Getting the Bus Width of a Socket

This is the same like for initiator socket.

6.6 Getting the Connected Socket

This is the same like for initiator socket.

6.7 Configuring the Socket

This is the same like for initiator socket.

6.8 Registering Callbacks

You may register callbacks for all interface method calls of the TLM-2.0 forward interface:

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&)
   ```

   There is no need to explain the three parameters and the return value of the first signature, as this is done by the TLM-2.0 reference manual. The `unsigned int` in the multi socket signature will carry the index of the binding over which the call came in, so that the callers can be distinguished.

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&)
```

There is no need to explain the two parameters of the first signature, as this is done by the TLM-2.0 reference manual. The `unsigned int` in the multi socket signature will carry the index of the binding over which the call came in, so that the callers can be distinguished.

3. `template<typename MODULE> void register_transport_dbg(MODULE* mod,`*dbg_cb* `cb)`

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

   For single sockets it is

   ```
   unsigned int (MODULE::*)(transaction_type&)
   ```

   and for multi sockets it is

   ```
   unsigned int (MODULE::*)(unsigned int, transaction_type&)
   ```

   There is no need to explain the parameter and the return value of the first signature, as this is done by the TLM-2.0 reference manual. The `unsigned int` in the multi socket signature will carry the index of the binding over which the call came in, so that the callers can be distinguished.

4. `template<typename MODULE> void register_get_direct_mem_ptr(MODULE* mod,`*get_dmi_cb* `cb)`

   This function allows to register a callback for `invalidate_direct_mem_ptr`. The type of *get_dmi_cb* differs between single and multi sockets.

   For single sockets it is

   ```
   bool (MODULE::*)(transaction_type&, tlm::tlm_dmi&)
   ```

   and for multi sockets it is

   ```
   bool (MODULE::*)(unsigned int, transaction_type&, tlm::tlm_dmi&)
   ```

   There is no need to explain the two parameter and the return value of the first signature, as this is done by the TLM-2.0 reference manual. The `unsigned int` in the multi socket signature will carry the index of the binding over which the call came in, so that the callers can be distinguished.

An example on how to register callbacks can be found in the initiator socket's appropriate section.

6.9 Calling Transport Functions

The target sockets allow for calling the functions of the TLM-2.0 forward interface, just like 'normal' TLM-2.0 sockets.

1. `tlm::tlm_sync_enum nb_transport_bw(payload_type&,phase_type&,sc_core::sc_time&)`
2. `void invalidate_direct_mem_ptr(sc_dt::uint64, sc_dt::uint64)`

Single socket shall use the `->` operator to call the functions, while multi sockets shall use the `[]` operator in conjunction with the `->` operator. When the `->` is used without the `[]` operator by a multi socket, the call will be performed on binding zero.

An example for calling transport functions can be found in the appropriate initiator socket section,

6.10 Accessing Extension

That is the same like for initiator sockets.

6.11 Deriving from the Socket

This is the same like for initiator sockets, but you do not have the option to override `free()`, as the target sockets do not provide memory management of transactions.

7 Working with Bidirectional Sockets

The two flavors of bidirectional sockets (single or multi socket) will be described in this section. The single socket class is defined as

```
template <unsigned int BUSWIDTH=32,typename TRAITS=tlm::tlm_base_protocol_types>
  class bidirectional_socket;
```

and the multi socket is defined as

```
template <unsigned int BUSWIDTH=32,typename TRAITS=tlm::tlm_base_protocol_types>
  class bidirectional_multi_socket;
```

both reside in namespace `gs::socket`. The former must be bound to exactly one bidirectional socket, while the latter may be bound to an unlimited number of bidirectional sockets, but at least one.

For those who have had a look into the code: There are more template arguments than the ones listed here, but those should not be touched. Never.

Throughout this chapter, differences between the two flavors will be mentioned explicitly, otherwise the single socket will be used in examples.

Bidirectional sockets are basically a unification of the initiator and the target socket. They are intended to be used for packet based communication, where each module is both target and initiator.

7.1 Construction

The provided constructors for the sockets do not differ in their signatures:

```
bidirectional_socket(const char* name,
    gs::socket::allocation_scheme alloc_scheme=gs::socket::GS_TXN_ONLY);
bidirectional_mutli_socket(const char* name,
    gs::socket::allocation_scheme alloc_scheme=gs::socket::GS_TXN_ONLY);
```

The first argument shall be the name of the socket, just as known from naming `sc_ports`, while the latter will define what the socket's pool will be doing. This is just the same as for initiator sockets.

7.2 Public Type Definitions

This is just the same like for initiator sockets.

7.3 Determining the Number of Binding

This is just the same like for initiator sockets.

7.4 Getting the Name of a Socket

This is just the same like for initiator sockets.

7.5 Getting the Bus Width of a Socket

This is just the same like for initiator sockets.

7.6 Getting the Connected Socket

This is just the same like for initiator sockets.

7.7 Configuring the Socket

This is just the same like for initiator sockets.

7.8 Using the Transaction Memory Management

This is just the same like for initiator sockets.

7.9 Registering Callbacks

You may register callbacks for all interface method calls of the TLM-2.0 backwards and forward interface.

See in the appropriate initiator/target sections for how to register callbacks for the interface method calls.

7.10 Calling Transport Functions

The initiator sockets allow for calling the functions of the TLM-2.0 forward and backwards interface. To distinguish between the forward and backward call, prefix the `->` or `[]` operators with either `.bw` or `.fw`.

Single socket shall use the `->` operator to call the functions, while multi sockets shall use the `[]` operator in conjunction with the `->` operator. When the `->` is used without the `[]` operator by a multi socket, the call will be performed on binding zero.

Example: Calling `b_transport` on a single and a multi socket.

```
class foo : public sc_core::sc_module
{
public:
  typedef gs::socket::bidirectional_socket<> single_socket_type;
  typedef gs::socket::bidirectional_multi_socket<> mutli_socket_type;
  single_socket_type sock;
  mutli_socket_type mul_sock;

  SC_HAS_PROCESS(foo);

  foo(sc_core::sc_module_name name)
    : sc_core::sc_module(name)
    , sock("sock", gs::socket::GS_TXN_ONLY)
    , mul_sock("mul_sock", gs::socket::GS_TXN_ONLY)
  {
    SC_THREAD(run);
    configure_sockets();
  }

  void run(){
    //we stack allocate a non memory managed txn
    //for b_transport
    //we will use it both for the single and the
    //multi socket, since both payload types
    //are the same
    single_socket_type::payload_type txn;
    sc_core::sc_time t=sc_core::SC_ZERO_TIME;

    //first the single socket
    setup_txn(txn);
    sock.fw->b_transport(txn,t);
    check_results(txn);

    //now the multi socket (one transfer per link)
    for (unsigned int i=0; i<mul_sock.size(); i++){
      t=sc_core::SC_ZERO_TIME;
      setup_txn(txn);
      sock.fw[i]->b_transport(txn,t);
      check_results(txn);
    }
  }

  void configure_sockets(){...}
  void setup_txn(single_socket_type::payload_type& txn){...}
  void check_results(single_socket_type::payload_type& txn){...}
};
```

7.11 Accessing Extension

This is the same like for initiator sockets.

7.12 Deriving from the Socket

This is the same like for initiator sockets.