

PCIeSocket PCI Express User's Guide (PCIeSocket v.2.0.1)

Copyright GreenSocs Ltd 2007-2009

Developed by
Christian Schröder and Wolfgang Klingauf, Robert Günzel
Technical University of Braunschweig, Dept. E.I.S.

12th March 2009

Contents

| | | |
|----------|---|----------|
| 1 | Introduction | 2 |
| 1.1 | Terms, Acronyms, Documentation and Implementation Conventions | 2 |
| 1.2 | Namespace | 2 |
| 1.3 | PCI Express | 3 |
| 2 | General Project Handling | 4 |
| 2.1 | Global Project Settings | 4 |
| 2.2 | Include PCIe.h | 5 |
| 2.3 | Generic Devices in PCIe | 5 |
| 3 | PCIe specific Classes | 6 |
| 3.1 | Transactions and Accesses | 6 |
| 3.2 | Using the PCIe API | 8 |
| 3.2.1 | Processing incoming TLPs | 8 |
| 3.2.2 | Creating and sending TLPs | 8 |
| 3.3 | Configuration Space | 9 |
| 3.4 | Addressing | 9 |
| 3.5 | Switch | 12 |
| 3.6 | Root Complex | 12 |
| 3.7 | Top-Level testbench | 12 |
| 3.8 | Mix Generic and PCIe Devices | 13 |
| 3.8.1 | How to connect Generic Devices to PCIe Switches | 15 |
| 3.8.1.1 | Introduction | 15 |
| 3.8.1.2 | Connect a generic bidirectional device | 16 |
| 3.8.1.3 | Connect a generic slave device | 16 |
| 3.8.1.4 | Connect a generic master device | 17 |

| | | |
|----------|--|-----------|
| 3.8.1.5 | Connect a generic master/slave device | 17 |
| 3.8.1.6 | Manual Address Range | 18 |
| 3.8.2 | How to connect PCIe Devices to Generic Routers | 18 |
| 3.8.3 | Rules | 19 |
| 3.9 | Specials | 20 |
| 3.9.1 | Interrupts | 20 |
| 3.9.2 | Power Management Turn Off Message and Ack | 20 |
| 4 | How-to | 21 |
| 4.1 | Multiple independent PCIe topologies in one simulation | 21 |

Chapter 1

Introduction

This is the User's Guide of the PCI Express (PCIe) implementation for TLM2.0 named *PCIESocket*. The implementation uses GSGPSockets which are TLM2.0 sockets (GreenSockets) with a mostly GreenBus-compatible API. This documentation expects the reader to be familiar with SystemC and - for deeper understanding - GSGPSocket / formerly GreenBus. GreenSocket is the universal TLM2.0 bus implementation by GreenSocs with GSGPSocket adding the GreenBus compatible API with three-phase-protocol.

The PCIESocket PCIe implementation allows the user to model a PCIe topology at the programmer's view (PV) abstraction level.

The documentation is related to the GSGPSocket 1.x.x version.

1.1 Terms, Acronyms, Documentation and Implementation Conventions

| | |
|------------------|---|
| PCIe | PCI Express |
| TLP Transaction | Layer Packet which is modeled with a PCIe Transaction |
| PCIe Transaction | One PCIe TLP or several TLPs (correlated Request, Completions) |
| Generic device | A generic device is a device (or SystemC module) using the standard GSGP-Socket API without any further API specialization (like PCIe). |
| Root Complex | A Root Complex is a device with an internal PCIe Switch sitting on top of the PCIe hierarchy. |

1.2 Namespace

The implementation of all not-user classes according PCIe are located in the namespace `gs::PCIe`.

Legacy Note

Namespace

If you need to use the GreenBus PCIe namespace `tlm::PCIe` include the header file

 `pcie_namespace_compatibility.h`

1.3 PCI Express

This PCIe implementation is based on the *PCI Express Base Specification Revision 2.0* and related documents.

Chapter 2

General Project Handling

This chapter is a short introduction how to prepare your project for PCIe.

2.1 Global Project Settings

Here we act on the assumption that you use a global project settings file (e.g. `globals.h`) which is included by each header file before any other includes (except maybe `systemc`).

Additional PCIe Checks

The PCIe implementation is prepared to check additional rules apart from the obligatory ones defined in the PCIe specification. They help to find bad devices or faulty usage. To enable these additional checks, use

```
1 #define CHECK_RULES_ENABLE
```

Note that these checks need some more run time. Disable these checks for large simulations where you already know that no failures occur.

Debug Output

In the global setting file you can switch on the PCIe debug terminal outputs by defining

```
1 #define PCIeDEBUG_ON
```

To enable further GSGPSSocket outputs use (see GSGPSSocket manual)

```
1 #define GS_VERBOSE
```

2.2 Include PCIe.h

In *each header* file of the PCIe project include  PCIe.h . The examples all first include a user specific  globals.h .

```
#include "globals.h"
#include "pciesocket/transport/PCIe.h"
```

The PCIe include defines the PCIe Transaction and access classes.

Legacy Note

make EXTENSION=PCIe

There is no longer the need to compile with different transaction types – like it was needed with the GreenBus PCIe project.

2.3 Generic Devices in PCIe

Combining PCIe peripherals with generic ones is easily possible. Any further device using another protocol and API which is compatible with the GSGPSSocket should be compatible with PCIe as well.

Anyway only the memory transactions (and not e.g. ID based ones) are compatible to generic devices in both directions.

Refer the GSGPSSocket Documentation¹ how to create generic devices.

See section 3.8 for a guide how to connect PCIe and generic devices to each other.

Legacy Note

Static/dynamic casts

There are no longer different version handling transactions of different types differently.

¹GSGPSSocket User's Guide:

<http://www.greensocs.com/projects/GreenSocket/GSGPSSocket/docs/GSGPSSocketUser039sGuide>

Chapter 3

PCIe specific Classes

This chapter leads through the process of creating a project with PCIe Devices, Switches, Root Complex etc. and giving background information.

In the end there is the section [3.8](#) about how to connect generic devices to PCIe and vice versa.

3.1 Transactions and Accesses

Figure [3.1](#) shows the derivation tree for PCIe Transactions and Accesses.

The quark access methods in the PCIe Transaction class are protected to avoid the direct usage of the Transaction class. The user has to use the Access class which matches the actual usage type: When creating a transaction use the `PCIEMasterAccessHandle`, when processing a received transaction, use the `PCIESlaveAccessHandle`. Accesses are temporary objects that should not be stored to keep access to a transaction!



Legacy Note

Deprecated

`getMasterAccess()` and `getSlaveAccess()` are deprecated!

How to use the access methods also see section [3.2](#).

In a PCIe peripheral module the user instantiates the PCIe API (see section [3.2](#)).

When creating a PCIe TLP the PCIe Master Access class provides convenience methods for all existing TLP types, e.g.:

```
1 init_Memory_Read(/* address, data, data size, requester func. no. */);  
  init_Memory_Write(/* address, data, data size, requester func. no. */);  
  init_IO_Read(/* address, data, data size, requester function no. */);  
  init_IO_Write(/* address, data, data size, requester function no. */);  
5 init_Configuration_Read(/* bus no., dev. no., func. no.,..., data,...*/);  
  init_Configuration_Write(/* bus no., dev. no., func. no.,..., data,...*/);  
  init_Message(/* message code, requester func. no. */);
```


PCIe Transaction Derivation Tree

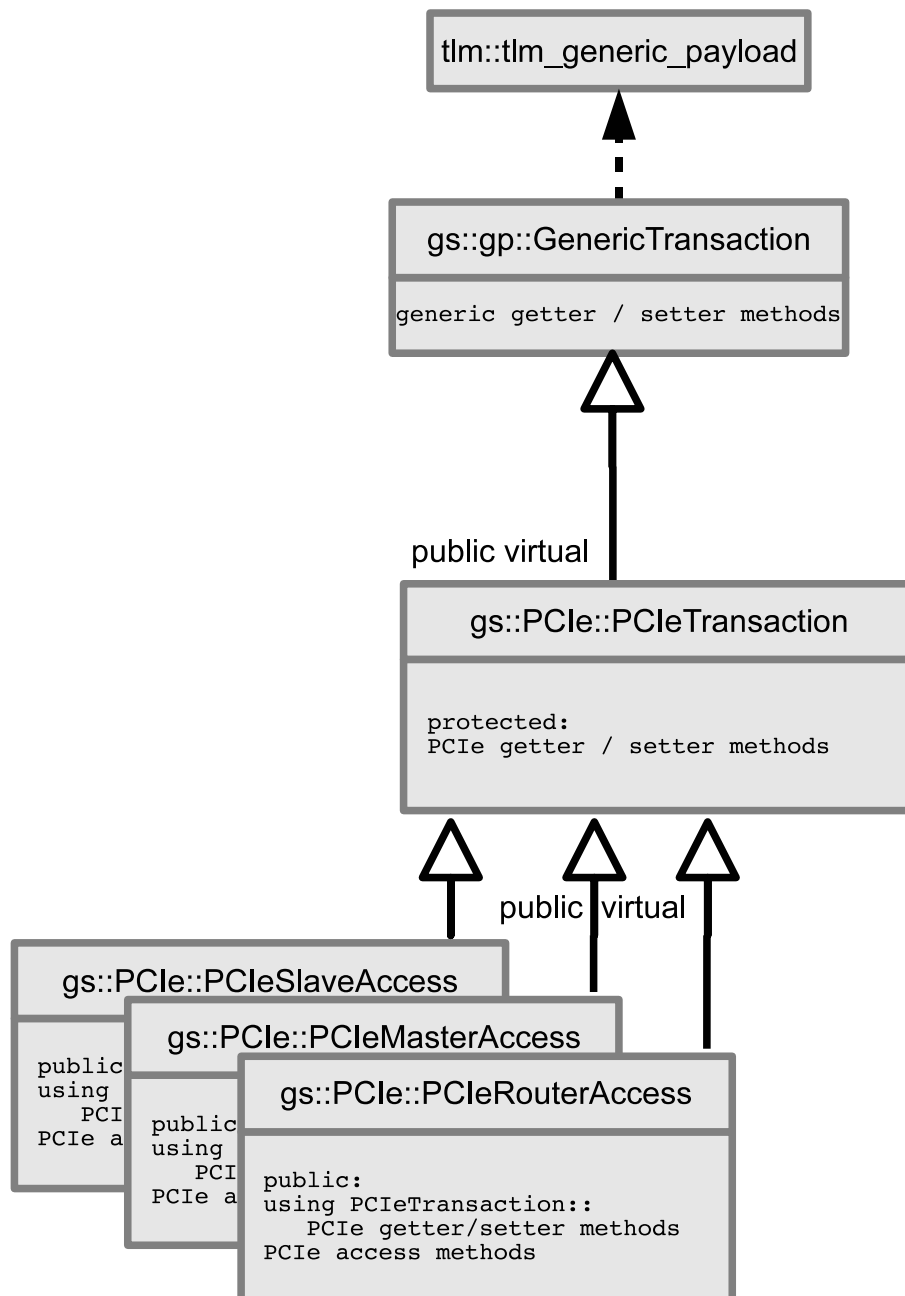



Figure 3.1: Derivation tree Transactions, Accesses.

When receiving and processing a PCIe TLP the PCIe Slave Access class provides convenience methods, e.g.:

```
1 get_TLP_type();  
  get_addr();  
  init_Memory_Read_Completion(/* completion status, completer func. no. */);  
  // ...  
5 set_Completion_Status(/* status */) ;  
  init_Unsupported_Request();
```

For the detailed and complete list of access methods look at the doxygen API reference.

3.2 Using the PCIe API

The PCIe framework provides the PCIeAPI for PCIe peripherals independent from their main behavior as a master or slave since each PCIe device is a master *and* a slave in GSGP Socket terms. Include  pciesocket/api/PCIeAPI.h before using the API.


Implementation Note


Bidirectional PCIe Ports

Because each PCIe device is a bus master and a slave the PCIe API has one bidirectional port which acts as a master *and* a slave. The bidirectional port (PCIeBidirectionalPort) is a `gs::gp::GSGPBidirectionalSocket`

3.2.1 Processing incoming TLPs

The PCIe API creates a PCIe Configuration Space Header of type 0 (for PCIe device Functions) and handles the incoming Configuration Request TLPs automatically. The user module needs not to care about them. Details about Configuration Space see section 3.3.

The API does some additional checks for incoming transactions. Some of the rule checks can be switched off by removing the define `PRECHECK_ENABLE` in  pciesocket/api/PCIeAPI.h . After these first checks the transaction is forwarded to the `b_transact` method which has to be implemented by the user module. The user device has to implement the `PCIe_rcv_if` interface and give the this pointer to the API during its construction.

After having received a transaction, use the convenience methods of the Slave Access class to process the TLP in the switch statement. An example implementation of a sending and receiving PCIe device is presented in the example listing in figure 3.2. See an example for a receiving device e.g. in  pciesocket/examples/platform/PCIeRecvinfoDevice.h .

3.2.2 Creating and sending TLPs

The same PCIe API can be used for creating and sending TLPs.

Call the `create_transaction()`-method on the API to get a Master Access Handle to the TLP which should be sent. Afterward call the convenience method(s) according to the TLP type you want to send. Call `myAPI.send_transaction(myMasterAccessHandle)` on the API to send the TLP. In a final step process the completion of the TLP if there is one again using the convenience methods of the Access class.

See figure 3.2 for an example listing. Find an example for a sending device e.g. in `pciesocket/examples/platform/PCleSendDevice.h`.

If a device should only send but not process received transactions the constructor of the API can be called with `NULL` instead of the `this` pointer. Then the API will internally process the transactions (e.g. handle Configuration Requests and manipulate the Configuration Space) but will not try to forward them to the user device implementation. If a transaction is received which needed user processing, an `sc_warning` will be reported.

3.3 Configuration Space

TODO: The Configuration Space is not yet implemented. Needs to be done with GreenReg.

3.4 Addressing

The PCIe framework supports the three address (routing) types of PCIe: address based, ID based and implicit routing.

The PCIe Transaction has two boolean variables which store the kind of used address type due to fast access (figure 3.3):

| | Address based Routing | ID based routing | Implicit routing |
|---------------------------------|-----------------------|------------------|------------------|
| <code>mlsIDbasedRouting</code> | false | true | false |
| <code>mlsImplicitRouting</code> | false | false | true |

Figure 3.3: Routing types.

Address based routing

The address based routing is divided into two independent address spaces: *Memory address space* and *I/O address space*. The routing information is transported in the quark `mAddr` which is the default quark for addresses in GreenBus. GreenBus routing with `MAddr` matches address based routing. Other

```
1 #include "my_globals.h"
#include "pciesocket/transport/PCIE.h"
#include "pciesocket/api/PCIEAPI.h"
using namespace gs::PCIE;

5
class MyPCIEDevice
: public sc_module,
  public PCIE_recv_if
public:
10   PCIEAPI myAPI;

   SC_HAS_PROCESS(MyPCIEDevice);
   MyPCIEDevice(sc_module_name name)
   : sc_module(name),
15     myAPI("PCIEAPI", this) // for sending and receiving
   {
       SC_THREAD(send_action); // for sending
   }

20   /// Method for receiving ability
   virtual void b_transact(PCIEAPI::accessHandle ah) {
       PCIEDEBUG("b_transact: received transaction");

       switch((unsigned int)ah->get_TLP_type()) {
25         case MemWrite:
             // ...
       } }

   /// Thread for sending ability
30   void PCIESendDevice::main_action() {
       gs::MAddr addr = 0x01; unsigned int data_size = 100;
       // create handle for transaction and empty data
       PCIEAPI::accessHandle ah;
       gs::GSDataType::dtype *dat;
35       dat = new gs::GSDataType::dtype(data_size);
       ah = mAPI.create_transaction();
       // prepare transaction
       ah->init_Memory_Read( addr, *dat, data_size ); // read data_size bytes
           from address targetAddr to vector dat
       if (lock) ah->lock_Memory_Read();
40       mAPI.send_transaction(ah);
       // process completion
       if (ah->get_Completion_Status() == SuccessfulCompletion) {
           cout << "read data:" << endl;
           for (unsigned int i = 0; i < data_size; i++) {
45               cout << (unsigned int)(dat->at(i)) << " ";
           } else
           cout << " Completion status: not successfull"<<endl;
           delete dat; dat = NULL;

50   } }
```

Figure 3.2: Example listing PCIe device.


routing mechanisms cannot be mapped to MAddr field routing because all 64 bit are needed by address based routing.

ID based routing

The PCIe ID based routing is mapped to the quarks mBusNo, mDevNo, mFuncNo and mRegNo. ID based routing is needed for Configuration Requests. The Requester ID is the ID of a TLP's sender. The user needs not to handle these addresses. Only the Root Complex implementation gets in contact with the IDs. Requester IDs are set automatically by the API.

Implementation Note

Bus ID assignment

By default the bus numbers are automatically assigned during construction with a global static counter (m_max_bus_no) in the address map ( PCIeAddressMap.h).

If you need to influence the assignment (e.g. you have two independent topologies in one simulation and each one needs to have its own bus numbering), you may give a reference to an own `int` counter in the router's and root complex' constructors. You must initialize the counters to -1! Ensure you give the correct counters to all devices (routers as well as root complexes). See section 4.1 for an example.

Implicit routing

Implicit routing is done with the quark mMessageType. The user needs not to handle implicit routing. The message creating convenience methods set the correct values.

| | | |
|--------------------------------|-----|---|
| RoutedToRootComplex | 000 | Route to Root Complex |
| RoutedByAddress | 001 | Never used (mIsIDbasedRouting=mIsImplicitRouting=false) |
| RoutedByID | 010 | Route normally by ID (set mIsIDbasedRouting=true instead of mIsImplicitRouting!) |
| BroadcastFromRootComplex | 011 | Broadcast from Root Complex to all devices |
| LocalTerminateAtReceiver | 100 | Local - Not forwarded at the next switch |
| GatheredAndRoutedToRootComplex | 101 | see PME_TO_Ack Message Code: Collect Acks from Downstream Ports and send one to Upstream Port |

Figure 3.4: Implicit routing types.

3.5 Switch

The PCIe Switch class is named `PCIERouter` following the GSGPSocket terms. A Switch has an Upstream Port (`upstream_port`) which has to be connected to another bidirectional PCIe port: a Downstream Port of another Switch or a Root Complex. The Downstream Port of a Switch is a multi port. The user can connect one or more bidirectional PCIe ports to this port: mainly that will be PCIe device's or other Switch's Upstream Ports. (The Downstream port is not needed to be connected to any port. It may left unbound.)

Generic devices can be connected to the Downstream Port according the rules in section 3.8.

3.6 Root Complex

To help the user with implementing the Root Complex there is a class `PCIERootComplex` (see `pciesocket/api/PCIERootComplex.h`). This derives from the `PCIERouter` class, accordingly the user is able to connect devices and switches to its Downstream Port. Do not connect the Upstream Port: it is bound internally to the `down_to_router_port` which is another PCIeAPI port. That port is the connection from the software/CPU to PCIe and should be used to implement the Root Complex functionality.

At minimum the user has to implement the method `PCIERootComplex::down_to_router_port_b_transact(PCIETransactionHandle th)`. To be able to extend the Root Complex, the user should derive from this class, e.g. `MyPCIERootComplex : public PCIERootComplex`. See `pciesocket/examples/platform` for an example.

- The `PCIERootComplex` is a (derives from) `PCIERouter`.
- Use Downstream Port to connect the PCIe tree topology.
- Do not use the Upstream Port which is connected internally.
- Use the port `down_to_router_port` to access the PCIe topology.
- Implement `down_to_router_port_b_transact` to handle incoming TLPs from the PCIe topology.
- Derive from `PCIERootComplex` to extend functionality, e.g. to add an `SC_METHOD`.

3.7 Top-Level testbench

In the examples the PCIe topology is built in the top-level testbench.

Rules:

- Create exactly one Root Complex instance.
- Create Switches if needed.
- Connect each Switch's Upstream Port to another Switch (Downstream Port) or to the Root Complex (Downstream Port).

- Create PCIe peripherals (devices) and connect their Upstream Ports either to a Switch's or a Root Complex' Downstream Port.

Figure 3.6 shows an PCIe example testbench with connected Root Complex, Switches and devices. The example can be found in the platform example `pciesocket/examples/platform` and is illustrated in figure 3.5.

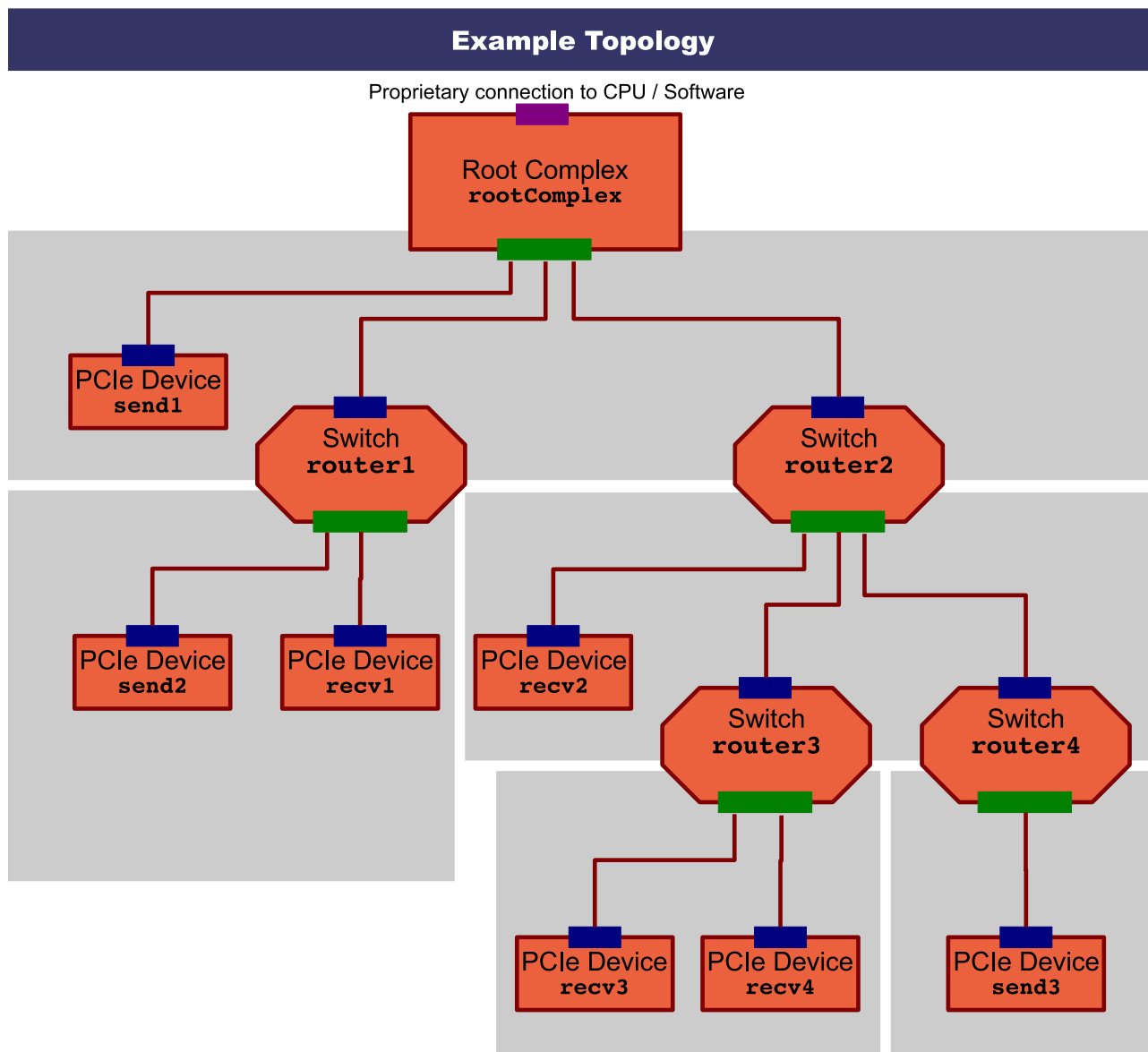


Figure 3.5: Example topology.

3.8 Mix Generic and PCIe Devices

This section describes how a generic device can be connected to a PCIe topology.

```

1 #include "my_globals.h"
#include "greencontrol/all.h"
#include "pciesocket/transport/PCIE.h"
#include "pciesocket/transport/PCIERouter.h"
5 #include "MyPCIERootComplex.h"
#include "PCIESendDevice.h"
#include "PCIRecvDevice.h"

int sc_main(int, char**)
10 {
    // instantiate partizipants
    MyPCIERootComplex rootComplex("rootComplex");
    PCIRecvInfoDevice recv1("recv1");
    PCIRecvInfoDevice recv2("recv2");
    15 PCIRecvInfoDevice recv3("recv3");
    PCIRecvInfoDevice recv4("recv4");
    PCIESendDevice send1("send1");
    PCIESendDevice send2("send2");
    PCIESendDevice send3("send3");
    20 PCIERouter router1("router1");
    PCIERouter router2("router2");
    PCIERouter router3("router3");
    PCIERouter router4("router4");

    25 // connect Devices
    rootComplex.downstream_port(send1.mAPI);

    router1.upstream_port(rootComplex.downstream_port);
    router1.downstream_port(send2.mAPI);
    30 router1.downstream_port(recv1.mAPI);

    router2.upstream_port(rootComplex.downstream_port);
    router2.downstream_port(recv2.mAPI);

    35 router3.upstream_port(router2.downstream_port);
    router3.downstream_port(recv3.mAPI);
    router3.downstream_port(recv4.mAPI);

    router4.upstream_port(router2.downstream_port);
    40 router4.downstream_port(send3.mAPI);

    // insert manual address settings here if needed

    // start simulation
    45 sc_start();
    return 0;
}

```

Figure 3.6: Example top-level testbench.

Easily connect directly:

- TLM 2.0 BaseProtocol GreenSockets (with bidirectional ports) to PCIe ports,
- Bidirectional GSGPSocket to PCIe ports

3.8.1 How to connect Generic Devices to PCIe Switches

The main precondition for getting generic devices compatible to be connected to a PCIe topology is that the generic device is only allowed to use extension that are compatible to the Generic Protocol. See the GreenSocket documentation for details on (in)compatible extensions.

When using raw GSGPSockets without further extensions they are connectable to PCIe ports easily.

3.8.1.1 Introduction

Generic transactions are address based, so only address based PCIe transactions are compatible with generic devices and generic device's transactions are received as address based TLPs by PCIe.

Implementation Note


Generic quarks in PCIe transactions

To make PCIe transaction compatible to generic ones the quark access functions (like `setMPCIECmd(...)`) are setting the generic quarks as well. E.g. `setMPCIECmd(...)` sets the generic quark `MCmd` additional to the PCIe quark `MPCIECmd`. PCIe uses an own Command enumeration because the generic one is not extendable. The two analogical transactions have the same number.

Generic Read and Write Transactions have an equivalent in PCIe: Memory Read and Write Transactions. Due to this analogy it is possible to map generic `Generic_MCMD_RD` and `Generic_MCMD_WR` transactions to PCIe `MemRead` and `MemWrite` transactions.

The supported features are: Read, Write Transactions, Error handling, Completion.

Generic devices with bidirectional sockets can be connected directly to the PCIe port of the router. If the generic device has other ports like only one master port, one slave port or master and slave port (e.g. `GenericMasterPort` and `GenericSlavePort`) it has to be connected by using the `GreeSocket` `bidir_unidir_wrapper`. This is no issue for configuration because generic devices don't understand any ID routed TLPs anyway.

To connect modules which use generic ports (`GenericMasterPort`, `GenericSlavePort`, ...) to PCIe modules or routers, the user has to follow the steps in the next subsections (see example  `pciesocket/examples/mixed/`). Figure 3.7 illustrates the different connections.

Connecting Generic Devices to a PCIe Router

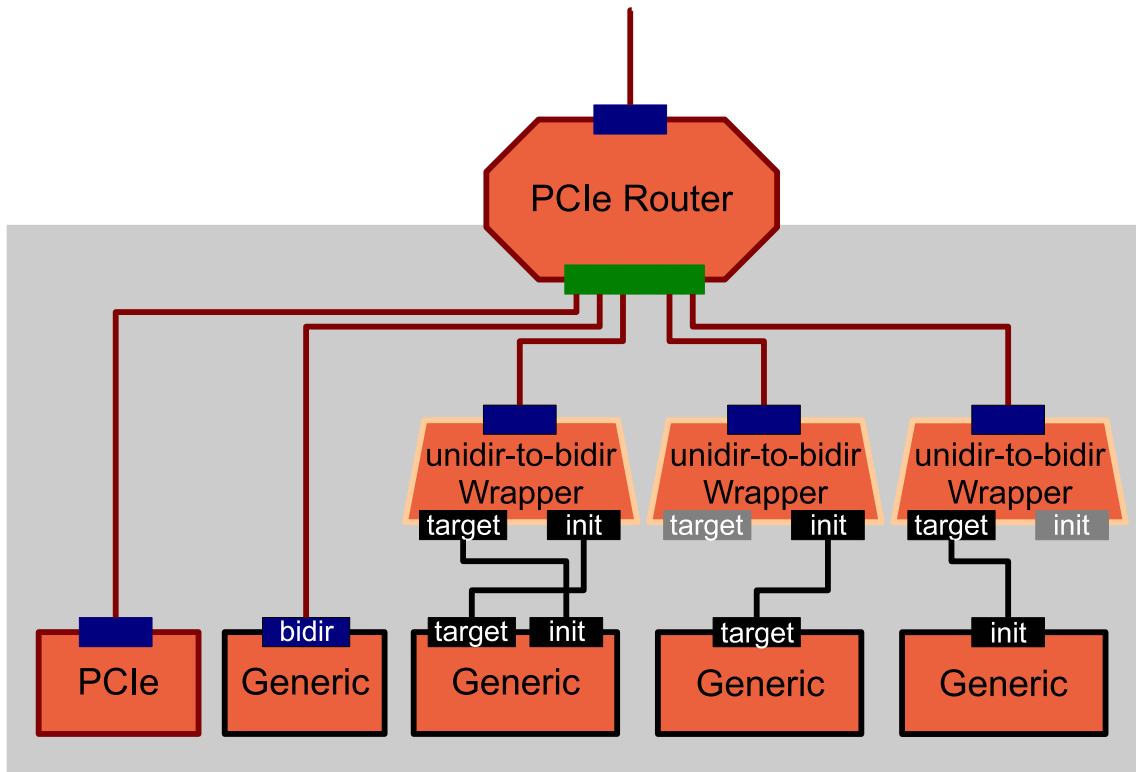


Figure 3.7: Alternative connections Generic Device to PCIe Switch

3.8.1.2 Connect a generic bidirectional device

Here a generic bidirectional master and slave device (`generic_dev`) shall be connected to a PCIe router (`router`):

- Instantiate the (PCIe and generic) modules and routers as usual in the testbench.
- Manually set address range(s) (see subsection 3.8.1.6).
- Connect the generic device directly to the PCIe Router Downstream Port, e.g.

```
router.downstream_port(generic_dev.bidir_socket);
```

3.8.1.3 Connect a generic slave device

Here a generic slave device (`generic_slave`) shall be connected to a PCIe router (`router`) using a wrapper (`adapt1`):

- Instantiate an unidir-to-bidir-wrapper (`undir_bidir_wrapper`). The first bool parameter defines if to use the init port (true when connecting a target/slave), the second bool parameter defines if to use the target port (false when connecting only a target/slave, will be dummy connected internally).

```
1 gs::socket::bidir_unidir_wrapper <> adapt1("Adapter1", true, false);
```

- (Optionally) manually set address range(s)i in the device's socket. This has no importance for the PCIe router, just maybe important for the user code in the device.
- Manually set same address range(s)i in the wrapper bidir socket (e.g. by copying the value from the device's socket) and the router (use subsection [3.8.1.6](#)).
- Connect the device to the wrapper, e.g.

```
1 adapt1.init_socket(generic_slave.slave_port);
```

- Connect the wrapper to the PCIe Router Downstream Port, e.g.

```
1 router.downstream_port(adapt1.bidir_socket);
```

3.8.1.4 Connect a generic master device

Here a generic master device (generic_master) shall be connected to a PCIe router (router) using a wrapper (adapt2):

- Instantiate an unidir-to-bidir-wrapper (undir_bidir_wrapper). The first bool parameter defines if to use the init port (false when connecting only an initiator/master, will be dummy connected internally), the second bool parameter defines if to use the init port (true when connecting a initiator/master).

```
1 gs::socket::bidir_unidir_wrapper <> adapt2("Adapter2", false, true);
```

- Connect the device to the wrapper, e.g.

```
1 generic_master.init_port(adapt2.target_socket);
```

- Connect the wrapper to the PCIe Router Downstream Port, e.g.

```
1 router.downstream_port(adapt2.bidir_socket);
```

3.8.1.5 Connect a generic master/slave device

Here a generic device with dedicated master and slave socket (generic_ma_sl) shall be connected to a PCIe router (router) using a wrapper (adapt3):

- Instantiate an unidir-to-bidir-wrapper (undir_bidir_wrapper). The first bool parameter defines if to use the init port (true when connecting a target/slave), the second bool parameter defines if to use the target port (true when connecting a initiator/master).

```
1 gs::socket::bidir_unidir_wrapper <> adapt1("Adapter3", true, true);
```

- (Optionally) manually set address range(s) in the device's target socket. This has no importance for the PCIe router, just maybe important for the user code in the device.
- Manually set same address range(s) in the wrapper bidir socket (e.g. by copying the value from the device's target socket) and the router (use subsection 3.8.1.6).
- Connect the device to the wrapper, e.g.

```
1 adapt3.init_socket(generic_ma_sl.slave_port);
   generic_ma_sl.init_socket(adapt3.slave_port);
```

- Connect the wrapper to the PCIe Router Downstream Port, e.g.

```
1 router.downstream_port(adapt3.bidir_socket);
```

3.8.1.6 Manual Address Range

- Manually set the address range of the generic *slave/bidirectional* socket and make it available to the router it is connected to:

- Simple (single) address range:

```
1 generic_dev.bidir_socket.base_addr = 0x0000000000000000LL;
   generic_dev.bidir_socket.high_addr = 0x000000000000FFFFLL;
   router.add_Memory_address_range(generic_dev.bidir_socket);
```

- Multiple address ranges (two ranges in this example):

```
1 // multiple address ranges, address 1
   router.add_Memory_address_range(generic_dev.bidir_socket ,
                                   0x000000000000B100LL , 0x000000000000B1FFLL);
   // multiple address ranges, address 2
5 router.add_Memory_address_range(generic_dev.bidir_socket ,
                                   0x000000000000D100LL , 0x000000000000D1FFLL);
```

Note

Manual address ranges for PCIe devices

The same way generic devices get their manual address range, PCIe devices' address ranges can be set. Simply replace `bidir_socket` with the PCIe API, e.g. `mApi`.

3.8.2 How to connect PCIe Devices to Generic Routers

A PCIe Device can be connected to a generic topology (e.g. router). The PCIe Device only may send Memory Read and Memory Write TLPs. Reads and writes being sent by generic devices to a PCIe device are mapped to PCIe Memory Reads and Memory Writes.

When connecting to a router owning initiator and target sockets (usual way), the unidir-to-bidir-wrapper which was introduced in the previous sections (3.8.1.2 to 3.8.1.5) needs to be used. Connecting to any generic bidirectional socket can be done directly.

Anyway the user should set the `base_addr` and `high_addr` parameters of the PCIe API socket (and the wrapper socket) manually (see section 3.8.1.6).

Figure 3.8 shows the available alternatives connecting a PCIe device to a generic router.

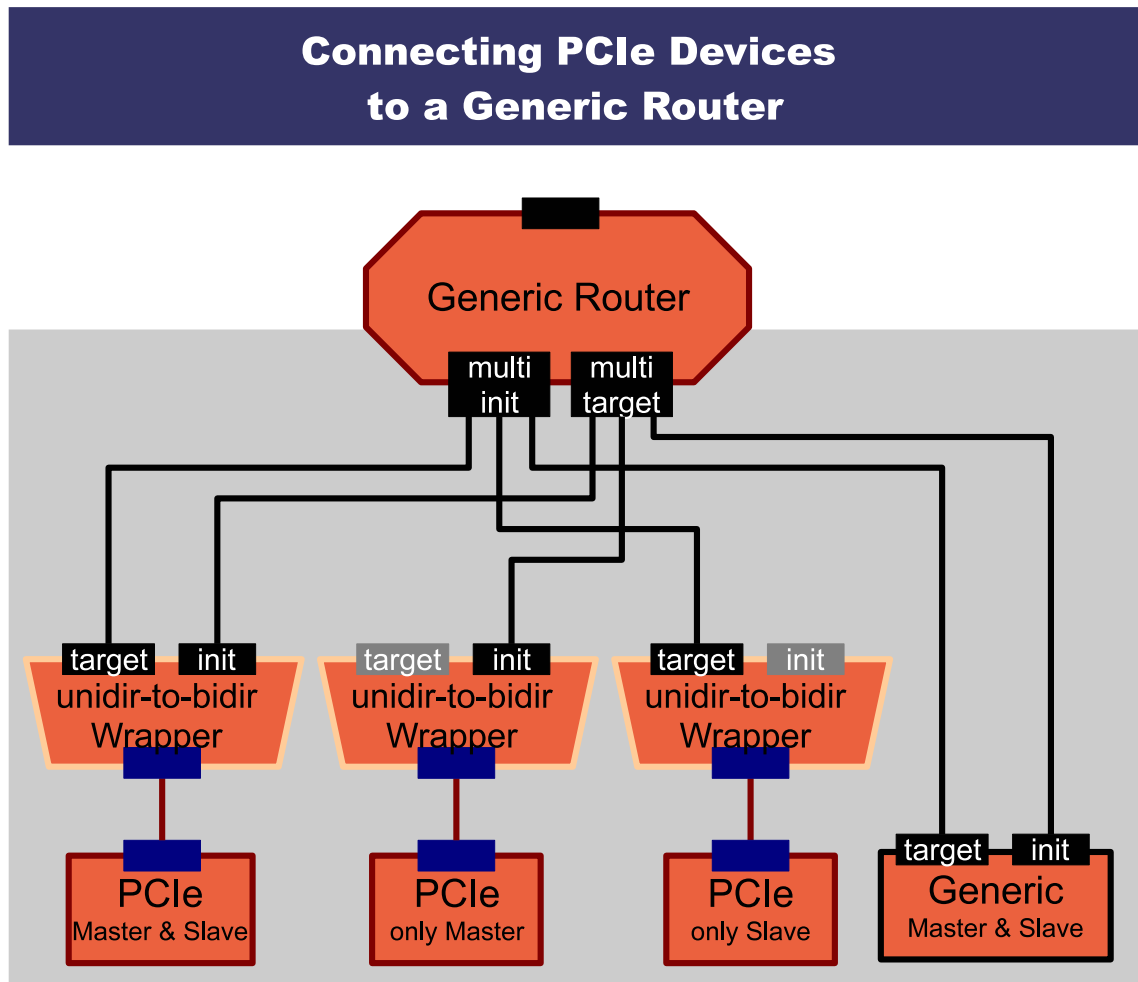


Figure 3.8: Alternative connections PCIe Device to Generic Router

3.8.3 Rules

- When a PCIe device sets a completion status error, the GSGP quarks `mError` and `sError` are set to `GenericError::Generic_Error_AccessDenied`
- When a PCIe sending device gets the completion status:
 - an `UnsupportedRequest` is returned if the GSGP quarks `mError` or `sError` contain a failure,
 - if no completion status has been set by the receiver the default return value is `SuccessfulCompletion` except for a GSGP command idle (which means the used command was not supported by generic devices).

3.9 Specials

This sections is a collection of some special TLPs etc.

3.9.1 Interrupts

Legacy support interrupts are provided by with INTx Messages (Assert/Deassert) which can be generated with the transaction access methods. The PCIe Router has to handle these Interrupt messages in a special way.

The PCIe Router supports a basic mechanism handling INTx Assert and Deassert Messages: An (Downstream) incoming Interrupt Message is terminated at the receiver (Switch) (see MessageType). The Interrupt is forwarded after calculating the mapping. This approach does not match all requirements (see Specification pages 72f.) but will care for forwarding the correct Message. (Not supported e.g.: Interrupt Disable bit of the Command register, Deassert Message in the case of DL_Down, Root Complex support.)

3.9.2 Power Management Turn Off Message and Ack

The acknowledgment to the Power Management message PME_Turn_Off is a special routing case: The Turn_Off message is broadcasted to all devices and each device has to react with a PME_TO_Ack message. A Switch has to forward the Ack only after having received all Acks from its downstream devices. A Switch counts the incoming PME_TO_Ack messages and sends the PME_TO_Ack message to its upstream port after having received Acks from all Downstream Ports.

Chapter 4

How-to

4.1 Multiple idenpendent PCIe topologies in one simulation

By default it is not possible to have multiple topologies with independent bus numbers in one simulation because there is a global static member function in the address map class which counts (increments) the assigned bus numbers (also see [3.4](#)).

To overcome this issue you can give integer references to the constructors of the routers (and root complex) to have independent counters for each topology in your simulation.

Use it like this:

- For two topologies, in your testbench create two int:

```
1      int bus_counter_topologyA = -1;
      int bus_counter_topologyB = -1;
```

- Give the counters to the root complexes (if you are using the GreenSocs ones):

```
1      PCIeRootComplex rootA("RootComplexA", bus_counter_topologyA);
      PCIeRootComplex rootB("RootComplexB", bus_counter_topologyB);
```

- Give the counters to the routers:

```
1      PCIeRouter router1A("router1A", bus_counter_topologyA);
      PCIeRouter router2A("router2A", bus_counter_topologyA);
      PCIeRouter router3A("router3A", bus_counter_topologyA);
      PCIeRouter router1B("router1B", bus_counter_topologyB);
5      PCIeRouter router2B("router2B", bus_counter_topologyB);
```