

GreenConfig Tutorial (GreenConfig v.4.1.0)

Copyright GreenSocs Ltd 2007-2009

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

23rd February 2010

Contents

Chapter 1

GreenConfig Tutorial

1.1 Introduction

This document introduces SystemC developers to GreenConfig: A means of controlling parameters and configurable elements of a SystemC model within the GreenControl framework.


In this tutorial an example system will be created which uses the GreenBus framework for communication between the modules. The example is used throughout to explain the important points of how to use the configuration framework.

For completeness, the tutorial will describe the connection between the modules of the system (using GreenBus) and their functionality. In more detail we will describe how the parameters of the GreenConfig framework are used in the demonstration platform.


The platform is composed of some user modules connected with a bus: A traffic generator (TG1), a cache (Cache1), and two devices which the traffic generator accesses, a memory (Mem) and a device called PCIeDevice2. The traffic generator and the cache are connected directly over SimpleBus ports and the cache and the other modules communicate through a SimpleBus (which is provided with GreenBus).

The usage of GreenConfig APIs for parameter access through the command line, a config file or an internal command line will be explained.

The intended audience for this tutorial have to be familiar with C++ and SystemC.

 **Note:** Beware the example's bus might be outdated and already been changed or updated in the current example. This makes no difference for the configuration part this tutorial should show.

1.2 Tutorial Full Example

Figure ?? shows an overview of the demonstration platform we aspire to build in this tutorial. A sample implementation is in the directory  `greencontrol/examples/demonstration_platform` . The modules with the main configurable values are:

- *Random traffic generator (TG)*: Acts as bus master and creates test traffic to the memory and the PCIe device. Its bound should be set with configurable parameters (and can be changed even during runtime).
- *Memory (Mem)*: This memory module is a simple double data rate random access memory (DDR RAM). It is a bus slave. Its size and base address should be configured.
- *Cache (Cache1)*: This cache is a write-back LRU cache whose size and line size is configurable during runtime. It also can be disabled during runtime.
- *PCIe device (PCIeDevice2)*: This bus slave's functionality is just a stub with registers which can be written and read. The base address should be configured.
- *Switch*: The switch is modeled by the SimpleBus router and need not be developed as user module.

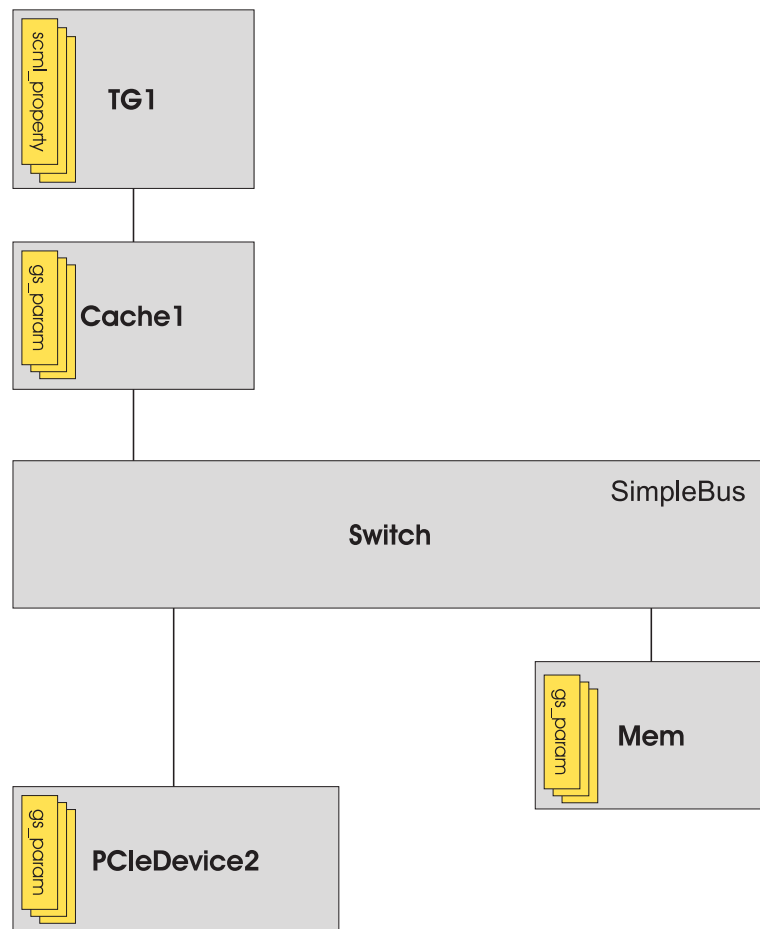


Figure 1.1: Demonstration Platform built in this tutorial.

Main steps for creating our configurable demonstration platform:

1. Create user modules with desired functionality:

In our case for slaves

- implement the SimpleBus interface

```
class ddr : public sc_module, public virtual simplebus_if,
```
- instantiate a slave port as member variable

```
simplebusSlavePort slave_port;
```
- in the constructor connect `this` to the port

```
slave_port.slave_port(*this);
```
- also in the constructor set the address range of this slave (we use `gs_params` for this)

```
slave_port.base_addr = base_addr;  
slave_port.high_addr = high_addr;
```

In our case for masters

- instantiate a master port as member variable

```
simplebusMasterPort master_port;
```

2. Concurrently include the GreenConfig parameters and make use of them:

- Include the framework:

```
#include "greencontrol/config.h".
```
- Optionally make "using" statements for configuration (and SimpleBus tlm):

```
using gs::gs_param;  
using gs::gs_param_base; //e.g. needed for callbacks  
using namespace tlm; //for SimpleBus tlm
```
- Usage of parameters:
 - declare them as member variables and initialize them in the constructor:

```
1 class MyClass
  : public sc_module
  {
  public: (or private:)
5     gs::gs_param<unsigned int> my_param;
  public:
    MyClass(sc_module_name name)
      : sc_module(name),
        port("PortName"),
10        my_param("my_param", 100) // gs_param with initial value
    { [...] }
  };
```

- declare and initialize them as local variables:

```
gs::gs_param<int> my_local_param("my_local_param", 50);
```

3. Connect the user modules in the testbench with the SimpleBus.

Demonstration Platform

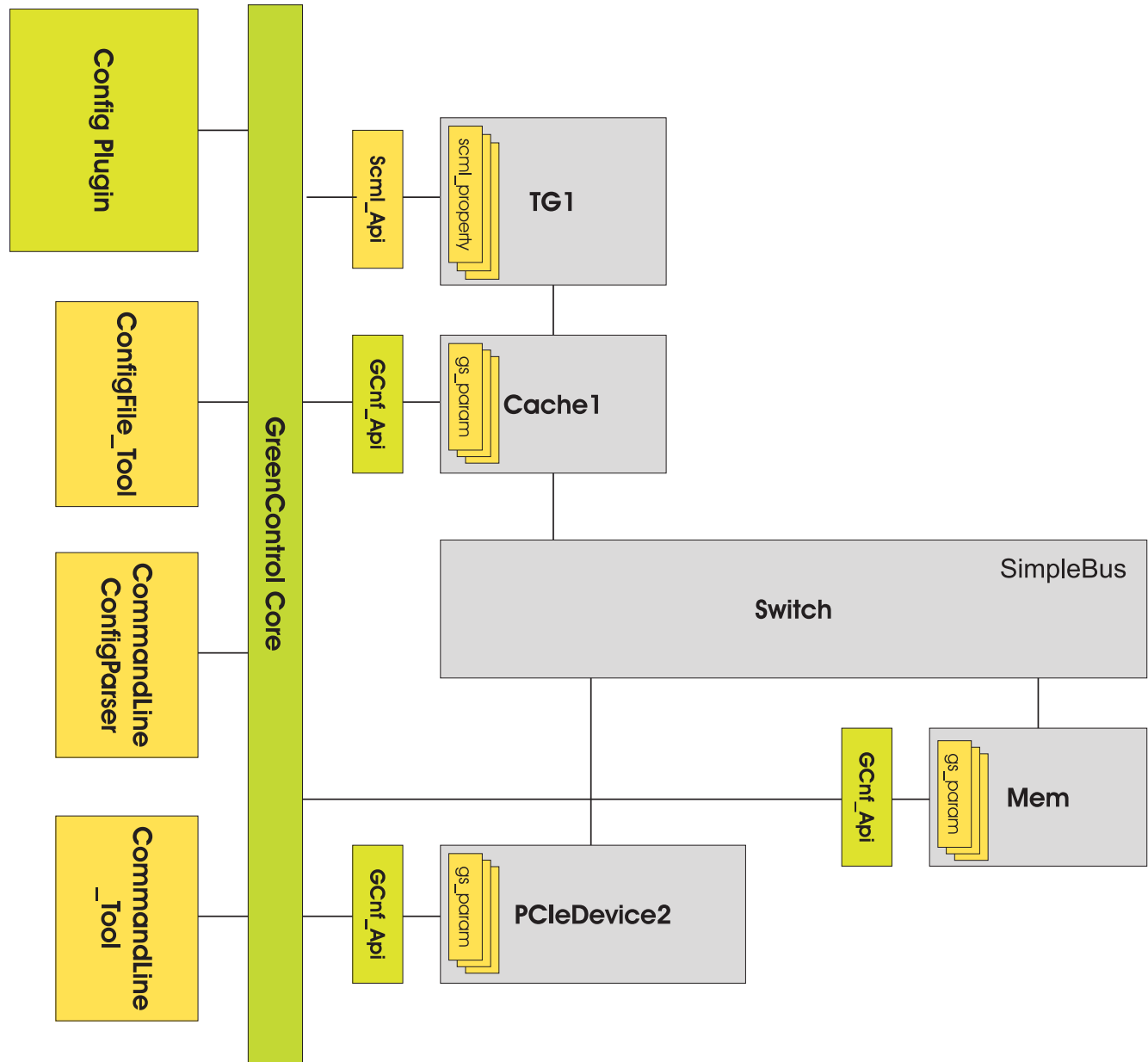


Figure 1.2: Demonstration Platform with configuration framework.

1.2.1 Create bus master module Traffic Generator (TG)

The traffic generator module (instance name TG1) is developed with the steps above and results are in the files `TG.h` and `TG.cpp`.

The following listing is a shortened extract of the header file. For the sake of clarity, most of the parts which are needed for the functionality of the traffic generator are left out here.

In this module different parameters are used: Member variable parameters (e.g. `init_size`) of type `scml_property` are declared private and get their default values inside the constructor. A local parameter variable (`tmp_param`) of GreenConfig parameter type `gs_param` is declared inside the constructor.

All member parameters have to be set at construction time to avoid undefined behavior. The full hierarchical name of the parameters is `TG1.param_name`.

Apart from the normal usage of `scml` parameters (`init_size` etc.) there is one interesting point: The reading of external parameters from the module whose instance name is `PCIeDevice2`. This is done in order to get the address and size of this device. The GCnf API is used to get the values of the `gs_param` parameters `PCIeDevice2.base_addr` and `PCIeDevice2.mem_size`. A local pointer to a GreenConfig parameter is used to parse the parameter base pointer to the correct data type.

To get access to these foreign parameters we need a GCnf API instance which provides the method `getPar(string param_name)` which returns a parameter base pointer. The default way getting a reference to the GCnf API is:

■ Get the API instance with the static call `gs::cnf::GCnf_Api::getApiInstance(this)`.

To get the values (and not the string representations) the temporary parameter `tmp_param` is used and the parameter base pointer is casted to the correct parameter pointer type. The string value of any parameter in the system can be accessed through the GCnf API with the call `getPar("module.param_name").getString()`.

TODO: new options getting values/params

```
1  [...]
   #include "demo_globals.h"
   #include "greencontrol/core/helpfunctions.h"
5  // SimpleBus API
   #include "userAPI/simplebusAPI.h"
   // GreenControl and GreenConfig
   #include "greencontrol/config.h"
10  [...]
   class TG
   : public sc_module
   {
15  public:
       tlm::simplebusMasterPort port;
   private:
```

```

20  scml_property<gs_uint32> init_size;    // data vector size (1MB)
    /*[some more gs_param parameters]*/

    gs::gs_uint64 write_addr[2];        // addresses were to write to
    gs::gs_uint64 read_addr[2];        // addresses were to read from
25  gs::gs_uint64 write_addr_size[2];    // size of the address space 2
        write_addr[i]
    gs::gs_uint64 read_addr_size[2];    // size of the address space read_addr 2
        [i]

    unsigned int target;                // target which is addressed

30 public:
    SC_HAS_PROCESS(TG);

    /// Constructor
    TG(sc_module_name name)
35     : sc_module(name),
        port("iport"),
        init_size("init_size", 1048576),
        [some more scml_property parameters]
    {
40     /*[...]*/

        m_GCnf_Api = gs::cnf::GCnf_Api::getApiInstance(this);

        // Pointer to a gs_uint64 parmeter
45     gs::gs_param<gs_uint64> *tmp_param;



        // Addresses for PCIeDevice2
        tmp_param = m_GCnf_Api->getPar("PCIeDevice2.base_addr")->get_gs_param<2
            gs_uint64>();
        write_addr[0] = read_addr[0] = *tmp_param;

50     /*[Addresses and sizes for devices]*/
    }

private:
    /*[some methods]*/
55 private:
    gs::cnf::cnf_api *m_GCnf_Api;
    /*[some methods and member variables]*/
};

```

1.2.2 Create bus slave module (Mem)

The memory module (instance name Mem) is located in the files  Mem.h and  Mem.cpp . It uses an instance of a DDR RAM class ddr to store words.

The memory has configurable addresses `base_addr` and `high_addr` of type `gs_param` which have to be set during construction time when they are used to set the addresses in the slave port. Changes during runtime are not handled by this model.

The parameters can be set with their full hierarchical names `Mem.param_name`.

The DDR RAM module has GreenConfig parameters, too.

The following listing shows a shortened part of the header file `Mem.h`:

```

1 class Mem
  : public sc_module
  {
  public:
5   /// SimpleBus slave port
    tlm::simplebusSlavePort slave_port;

    gs::gs_param<gs_uint64> base_addr;
    gs::gs_param<gs_uint64> high_addr;

10   SC_HAS_PROCESS(Mem);

    /// Constructor
    Mem(sc_module_name name)
      : sc_module(name),
        slave_port("simplebus_slaveport"),
        base_addr("base_addr", 0x200),
        high_addr("high_addr", 0x400),
        ddr_mem("memory")
20   {
        ddr_mem.size = high_addr - base_addr;
        slave_port.slave_port(ddr_mem);
        slave_port.base_addr = tlm::MAddr(base_addr);
        slave_port.high_addr = tlm::MAddr(high_addr);
25   }

  private:
    /// Memory
    ddr ddr_mem;

30  };

```

1.2.3 Create bus slave module PCIeDevice

The module `PCIeDevice` is a little memory and can be found in the files `PCIeDevice.h` and `PCIeDevice.cpp`. The instance is named `PCIeDevice2` accordingly its parameters can be accessed with `PCIeDevice2.param_name`. This module also uses the DDR RAM which was introduced in section ?? and is similar to the memory module.

The provided parameters are `base_addr` and `mem_size`. The memory size is used to calculate the high address for the slave port to make it different from the memory. These parameters have to be set before

construction of this module. How the parameters may be used, and how they allow changes during runtime, is shown in the next section ??.

1.2.4 Create cache module (Cache)

The cache module consists of two nested classes. The module which is instantiated in the testbench is the `Cache` class in the files `Cache.h` and `Cache.cpp`. This module owns a `SimpleBus` slave and a master port to receive and send reads and writes.

The address range of the slave port is fixed to the whole address range (0x00000000 to 0xFFFFFFFF).

Submodule `LRU_cache`

The `Cache` module owns an instance of the cache module `LRU_cache`. This module is a word addressable LRU (Least Recently Used) cache with configurable parameters for its size and its behavior (write-back or write-through). The `GreenConfig` parameters for sizes are not used directly in the cache to allow runtime changes. If the cache size parameter `cache_size` changes during runtime, all lines are written back and the cache is reset and re-sized.

The cache supports write-back and write-through. The parameter `param_write_back` is used directly. So a change to this parameter takes effect directly on the behavior (write-back or write-through). If the behavior is changed from write back to write through the cache will reset itself and write back dirty lines. In the write-through-mode data is written to the memory word-wise but read line-wise.

The following parameters have affect on the size of the cache:

- `cache_line_size` is the size of one cache line (block)
- `cache_lines` is the number of cache lines in the cache
- `cache_size` is the entire size of the cache.

When using this cache module in this platform be aware of some important design rules:

- The size parameters must match each other: The size of all cache lines together has to be equal or greater than the cache size and the cache lines must not be able to store more than $cache_size + line_size - 1$ words.
- The address ranges of two target modules must not be inside a cache line. The cache line size has to be chosen carefully to guarantee that. The cache reads and writes entire cache lines from and to the modules. So either the module has to be able to deal with out-of-bound-queries or the cache lines have to match the address borders.
- When changing the size parameters during runtime, first change `cache_line_size` and `cache_lines`, *afterwards* change the parameter `cache_size`.

Cache module

The Cache module demonstrates how the runtime configuration can be used by the developer in a module.

The following listing shows an extract of the Cache code illustrating the usage of parameters changing at runtime and registering callbacks. Callback registering modules must call the macro `GC_HAS_CALLBACKS()` in its body and must call `GC_UNREGISTER_CALLBACKS()` during destruction.

```

1 class Cache
  : public sc_module, /*[...]*/
  {
    /*[...]*/
5 private:
    gs::gs_param<bool> enable_cache;

    public:
    GC_HAS_CALLBACKS();
10    /*[...]*/
    Cache(sc_module_name name)
      : sc_module(name), /*[...]*/
        enable_cache("enable_cache", true), /*[...]*/
    {
15        /*[...]*/
        // register callback
        GC_REGISTER_PARAM_CALLBACK(&enable_cache, Cache, 2
            parameter_changed_callback);
    }
    ~Cache() {
20        GC_UNREGISTER_CALLBACKS();
    }
    /*[...]*/
    /// Callback functions for changed enable_cache-parameter
    /** Write back all lines and disable or enable cache. */
25    void parameter_changed_callback(gs::gs_param_base &par);

    private:
    LRU_cache cache;
    };
30

void Cache::parameter_changed_callback(gs::gs_param_base& par) {
    if (!par.is_destructing()) {
        // notify cache to make write backs and reset.
        cache.create_cache_event.notify();
35    }
}

```

Inside the callback function an event in the LRU cache is notified which initiates a write back and resets the cache. The reset method may not be invoked directly because *no waits are allowed in the callback function*.

1.2.5 Testbench and module connection

This important section is about how to include the framework and needed plugins and APIs in a testbench and how to instantiate them.

The code snippets in this section are ordered as they appear in the testbench file of the demonstration platform, testbench_demonstration_platform.cpp.

The define at the beginning of the testbench is needed for the command line tool, see section ??.

```
1 #define TEST_TOOL_POLL_TIME sc_time(1, SC_MS)
```

Includes

To be able to use the SimpleBus for the connection of the modules we have to include a router, a scheduler and the SimpleBus protocol:

```
1 #include "greenbus/transport/genericRouter.h"
#include "greenbus/protocol/SimpleBus/simpleBusProtocol.h"
#include "greenbus/scheduler/fixedPriorityScheduler.h"
```

GreenConfig including the GreenControl base is included by:

```
1 #include "greencontrol/config.h"
```

Afterwards some APIs which are used in the testbench are included. We use the config file parser tool (see section ??) which allows reading configuration files and the command line config parser (see section ??) which allows setting of parameters on the command line. We also use the command line tool (see section ??) which allows the user to access parameters during runtime by an internal command line at the simulation terminal. Other APIs may be included in the modules which make use of them.

```
1 #include "greencontrol/config_api_config_file_parser.h"
#include "greencontrol/config_api_command_line_parser.h"
#include "greencontrol/config_api_command_line_tool.h"
```

The include order of plugins and APIs is not fixed.

In the end we include our user modules we developed for the demonstration platform above:

```
1 #include "Mem.h"
#include "TG.h"
#include "Cache.h"
#include "PCIeDevice.h"
```

Instantiations

On the first usage the Core and the config Plugin are instantiated automatically. There are ways to manually instantiate them, see the User's Guide for more information.

Instantiate the two APIs we want to use to set initial parameter values:

```
1 // Configuration with GreenConfig config files
gs::cnf::ConfigFile_Tool configTool("ConfigFileTool");
configTool.parseCommandLine(argc, argv); // parses the command line for -->
    configfile
configTool.config("demoplatform.cfg"); // reads in a config file
5
// Configuration with GreenConfig command line options
gs::cnf::CommandLineConfigParser configParser("ConfigParser");
configParser.parse(argc, argv); // parses the command line
10
// Command line tool which provides a command line to access parameters
gs::cnf::CommandLine_Tool commandLineTool("CommandLineTool");
```

Now the initial values of the modules are set by the APIs inside the configuration plugin as *implicit parameters*. That means that they already exist with a value and even can be accessed by any other API, but they have not yet been added by the owner module itself.

Now we can instantiate the owner modules (the demonstration platform modules). They add their parameters using the `gs_param` API and hence their state is changes from *implicit* to *explicit parameters*.

We instantiate the user modules Mem, Cache, PCIeDevice and TG.

In the end we connect our modules with the SimpleBus. The TG and the cache are connected directly over their SimpleBus ports without a SimpleBus instance between them. Between the cache and the devices a SimpleBus instance takes care of communication. With the usage of SimpleBus ports between the TG and the cache slave we have the opportunity to leave out the cache by simply connecting the TG to the bus.

The instantiation and connection of the SimpleBus is done as presented in section ??.

1.2.6 Usage of the config file parser

The config file parser API (class `ConfigFile_Tool`) reads in a configuration file and sets the parameters in the `ConfigPlugin`.

After instantiating an instance of the API (e.g.) in the testbench there are two different ways of reading one or more configuration files:

- The name of the configuration file which should be parsed can be hard coded in the testbench by using the method call `my_configFileTool_instance.config("config_file.cfg");`. This method may be invoked multiple times in order to parse more than one configuration file.
- A configuration file may be specified at command line with the option `--configfile <config_file_name>`.
The parsing of the command line is initiated in the testbench with the call `my_configFileTool_instance.parseCommandLine(argc, argv)`.
You are allowed to mix these two techniques. The last called has priority.

The configuration file is a simple text file with one line per parameter:

```
1 ## comment line
hierarchical.par_name value # comment ignored
myModule.my_SubModule.parameter 18349
string_parameter "String value"
```

For details especially concerning string values see the User's Guide.

1.2.7 Usage of the command line config parser

First, the parsers must be instantiated and initiated as explained in section ???. Afterwards parameters may be set in the command line at the start of the simulation.

Setting parameters from the command line can be done with the command line option `--param parname=value`. String values with spaces may be set in quotes. Quotes in string values have to be written as `'\"'`. For more details see the User's Guide.

1.2.8 Usage of the command line tool

The command line tool provides a command line interface during simulation in the simulation terminal. After having instantiated the class `CommandLine_Tool` some simple commands may be used: *get*, *set*, *list*, *h* (help), *q* (quit):

```
1  h   : this help
   q   : quit
   set <param_name> <value> : Set value <value> of parameter <param_name>
   get <param_name>         : Get value of parameter <param_name>
5  list                               : List all parameters
   list <modname>             : List all parameters of module <modname>
   list <modname>.*           : List all parameters of module <modname> and
                               its children
```

Dont worry if the command line (`toolThread (h=help)>`) is not shown (it may be hidden in some output that the simulation produces), just type the command and press enter.

The only additional special use case is the `#define TEST_TOOL_POLL_TIME`. This sets the interval the command line's `SC_THREAD` polls for new data. For untimed models use `SC_ZERO_TIME`, for timed models use a reasonable time, e.g. `sc_time(1, SC_MS)`. In this demonstration platform we use `#define 2 TEST_TOOL_POLL_TIME sc_time(1, SC_MS)` which is set in the testbench.

1.3 Parameter Callbacks and Notification

Prepare modules to use callbacks

How to prepare a module to be able to register callbacks on parameters.

- Include the GreenConfig framework:
`#include "greencontrol/config.h"`
- Use macro `GC_HAS_CALLBACKS()` in the module body.
- Use macro `GC_UNREGISTER_CALLBACKS()` in the module destructor.
- Define a callback function with the signature
`void func_name(gs::gs_param_base&)`
- Check each call of the callback function if the called parameter is just being destroyed. In that case make sure not to use any parameter reference/pointer to that parameter any longer!
- If desired use `par.getType()` or `getTypeString()` to cast to the correct parameter type.

Register callbacks

- Register a callback for the parameter `my_param` using the macro
`GC_REGISTER_PARAM_CALLBACK(&my_param, MyModule, parameter_callback)`

Unregister callbacks

You *may* unregister callbacks but you need not! If you are not interested any longer in getting the callbacks for a specific parameter you may unregister the callback:

- Store the callback adapter pointer when registering the callback:
`boost::shared_ptr<gs::cnf::ParamCallbAdapt_b> callbadapt
= GC_REGISTER_PARAM_CALLBACK(&my_param, MyModule, parameter_callback)`
- Unregister the callback using the macro
`GC_UNREGISTER_CALLBACK (callbadapt)`

There are alternative ways unregistering parameters, please see the *GreenConfig User's Guide*.

Code example

```
1 // GreenControl and GreenConfig
#include "greencontrol/config.h"

class MyModule
5 : public sc_module
{
public:
    GC_HAS_CALLBACKS();
    SC_HAS_PROCESS(MyModule);
10 MyModule(sc_module_name name)
    : sc_module(name),
      my_param("my_param")
    {
        // register callback
15 GC_REGISTER_PARAM_CALLBACK(&my_param, MyModule, parameter_callback);
    }
    ~MyModule() {
        GC_UNREGISTER_CALLBACKS();
    }
20 /// Callback function
    void parameter_callback(gs::gs_param_base &par);
private:
    gs::gs_param<int> my_param;
};
25
void MyModule::parameter_callback(gs::gs_param_base& par) {
    if (!par.is_destructing()) {
        /* do something but do NO time! */
        switch(par.getType()) {
30         case gs::cnf::PARTYPE_INT: {
            gs::gs_param<int>* tmp_param = par->get_gs_param<int>();
            /* do something */
            break; }
        /* other cases */
35     }
    }
}
```

⚠ Deprecated:

You may get an event which is notified when a parameter changes using `par.getUpdateEvent()`.

1.4 Parameter Arrays

GreenConfig provides two different types of arrays:

- Parameter arrays with *unnamed* members each of the *same data type*: *Simple Parameter Array* (see section ??).
- Parameter arrays with *named* members of *different data types*: *Extended Parameter Array* (see section ??).

1.4.1 Simple Parameter Arrays

Instances:

```
1      class IP1 {  
2          [...]  
3          gs::gs_param<int*>    my_int_array;  
4          gs::gs_param<float*>  my_float_array;  
5          gs::gs_param<string*> my_string_array;  
6          gs::gs_param<unsigned int*> my_uint_array;  
7      }
```

Constructor:

```
1      IP1()  
2      : my_int_array("my_int_array", 3), // with size=3  
3        my_float_array("my_float_array"), // with default size  
4        my_string_array("my_string_array"),  
5        my_uint_array("my_uint_array", 3) // with size=3  
6      { ... }  
7  
8      void any_method() {  
9          std::vector<int> intvec;  
10         for( int i = 0; i < 20; i++ ) intvec.push_back( 110 + i );  
11         // constructor with default values in a vector  
12         gs::gs_param<int*> my_new_arr("my_new_arr", intvec);  
13     }  
14 }
```

Access:

```
1  any_function() {  
    // set members individually  
    my_int_array[0] = 100;  
    my_int_array[1] = 101;  
5   my_int_array[2] = 102;  
    // resize and set the new member  
    my_int_array.resize(4);  
    my_int_array[3] = 103;  
    // set all members concurrently and resize automatically  
10  my_int_array.setString("{10 12 13 14 15}");  
}
```

Iterate: Example for iterating through a Simple Parameter Array. Simple Parameter Arrays need no iterators because they can be accessed with the position.

```
1  for (int i = 0; i < arr.size(); i++) {  
    cout << "member #" <<i<< " = " << arr[i].getString() <<endl;  
}
```

Config file Usage:

```
1  IP1.my_uint_array.init_size 3  ## optional. Otherwise the config 2  
    file calculates the size  
    IP1.my_uint_array.0 1000  
    IP1.my_uint_array.1 2200  
    IP1.my_uint_array.2 3330
```

Sets the array size either to `init_size` (if given) or automatically to the highest member (here `size=3`). This syntax has priority to the `{...}` notation.

1.4.2 Extended Parameter Arrays

Example of usage: Alternatively the array member pointers could be class members:

```
1 class IP1 {
  gs::gs_param_array myTopArr;

  IP1()
5  : myTopArr("myTopArr")
  {
    gs::gs_param_array *subArr0
      = new gs::gs_param_array("my2ndArray0", &myTopArr);
    gs::gs_param_array *subArr1
10     = new gs::gs_param_array("my2ndArray1", &myTopArr);

    gs::gs_param<int> *par0
      = new gs::gs_param<int>("myIntPar0", 10, &subArr0);
    gs::gs_param<int> *par1
15     = new gs::gs_param<int>("myIntPar1", 12, &subArr0);
    gs::gs_param<string> *par2
      = new gs::gs_param<string>("myStringPar", "Def.", &subArr0);
      );

    gs::gs_param<int> *otherIntPar
20     = new gs::gs_param<int>("myOtherIntPar", 500, &subArr1);
    gs::gs_param<int> *otherIntPar1
      = new gs::gs_param<int>("myIntPar1", 600, &subArr1);
  }
}
```

Resulting structure:

```
1 myTopArr {
  'my2ndArray0',
  'my2ndArray1'
5 }
my2ndArray0 {
  myIntPar0 = 10,
  myIntPar1 = 12,
  myStringPar = "Def."
10 }
my2ndArray1 {
  myOtherIntPar = 500,
  myIntPar1 = 600,
}
```

Access: Access an Extended Parameter Array in any module in the simulation (not the owner module which can access directly).

```
1 any_function() {  
    gs::gs_param_base *tmp = m_Api.getPar( "Owner.myTopArr" );  
    gs::gs_param_array &topArr  
        = *(dynamic_cast<gs::gs_param_array*>(tmp));  
5    gs::gs_param_array::iterator it;  
    it = topArr.getMemberArray( "my2ndArray0" )->find( "myIntPar1" );  
}
```

Config file Usage:

```
1 IP1.myTopArr.my2ndArray0.myIntPar0 100  
IP1.myTopArr.my2ndArray0.myIntPar1 111  
IP1.myTopArr.my2ndArray0.myStringPar "Hello world!"  
IP1.myTopArr.my2ndArray1.myIntPar1 50000
```

Or use the lua config file parser ( luafile_tool.h).

Iterators

The `gs_param_array` class has an iterator class `gs_param_array::iterator`.

The iterators support the operators `=`, `==`, `!=`, `++` (prefix), `-` (prefix), `*`

The iterator class has the functions

- `begin()` returns an iterator which points to the first member.
- `end()` returns an iterator which points behind the last member.
- `find(string local_name)` returns an iterator which points to the member with the local name.

Example usage of the array iterator:

```
1 gs::gs_param_array::iterator it;  
for (it = myArr.begin(); it != myArr.end(); ++it) {  
    cout << (*it).getName() << " = " << (*it).getString() << endl;  
}
```

1.5 Frequently Asked Questions

Simulation crash during destruction

Question: My simulation crashes during destruction within the macro `GC_UNREGISTER_CALLBACKS()`.

Answer: You missed to use the `GC_HAS_CALLBACKS()` macro in the body of your module.

Simulation crash inside parameter callback function

Question: My simulation crashes inside a parameter callback function.

Answer: Maybe you missed to check the `is_destructing()` flag of the changed parameter. If a parameter does a callback during destruction you must not access its value any more (even not in this callback).

Simulation crash inside parameter callback function

Question: Compiler error within macro `GC_REGISTER_PARAM_CALLBACK`:

error: 'gc_add_ParamCallbAdapt' was not declared in this scope

Answer: You missed to use the `GC_HAS_CALLBACKS()` macro in the body of your module.

Simulation crash when sending transaction

Question: My simulation crashes (segmentation fault) within an API in a line which calls `m_gc_port 2 .init_port.out->notify(transaction, PEQ_IMMEDIATE);`.

Answer: Make sure that the Core and the Plugins are still existing. They need to be created before any other GreenControl elements and have to be deleted as last. E.g. try to use new instead of variables in the main function.

To be continued...

Appendix A

Appendix

A.1 Introduction to SimpleBus

In this tutorial we developed a platform with devices which communicate using transaction over a bus. This bus is modeled by the SimpleBus which is a protocol in the GreenBus framework developed by GreenSocs.

Although the GreenControl framework uses the GreenBus aswell, the SimpleBus communication is not used and not part of GreenControl. The SimpleBus is used to achieve a realistic demonstration platform. This section is a brief overview of how to use its API.

The SimpleBus has a bus oriented API. A master can access an addressed slave by reading and writing data of a chosen length.

A slave module has to implement the interface `simplebus_if`. The method `bool read(std::vector<gs_uint8> &data, const gs_uint32 addr, const gs_uint32 length)` allows a master to read data from the SimpleBus slave. The method `bool write(const std::vector<gs_uint8> &data, const gs_uint32 addr, const gs_uint32 length)` allows a master to write data to the SimpleBus slave. The transactions may be of arbitrary length. The implementation of these methods has to be blocking and must not return until the data has been completely received by the slave. The methods return if a slave is present at the target address. The address in these calls is the bus address of the transaction less the base address of the slave. The data types are GreenSocs typedefs, see `gs_datatypes.h`.

A master is able to call similar read and write methods in its `simplebusMasterPort`. The address is different from the address the slave gets: Here the bus address of the slave the transaction should be delivered to has to be set.

A master module simply has to have a `simplebusMasterPort` as a public member variable which can be connected in the testbench to the router.

Slave modules have to implement the interface `simplebus_if`. Hence, the `simplebusSlavePort` in `myslaveport` has to be called: `myslaveport.slave_port((simplebus_if)ddr_mem)`.

- Either the module owning the bus port may implement the interface itself and call `slave_port(*this)`
- or another module has to have the bus port

- or the testbench instantiates the slave port and calls it with the interface implementing object of the module.

In the testbench the connection is done as follows:

```
1  // SimpleBus
SimpleBusProtocol p("Protocol", sc_time(10, SC_NS));
fixedPriorityScheduler s("Scheduler");
GenericRouter r("Router");
5  r.protocol_port(p);
p.router_port(r);
p.scheduler_port(s);

10 // Connect Masters
tg_master->mySimplebusMasterPort(r.target_port);

// Connect Slaves
r.init_port(testbenchSimplebusSlavePort);
r.init_port(mem.mySimplebusSlavePort);
```