Getting Started with the SPEC

A tutorial for the Simple PCI Express Carrier project newcomers
Version 1.0 (March, 2014)

Javier D. Garcia-Lasheras for CERN (BE-CO-HT)
Table of Contents

1 Introduction ................................................................. 1
   1.1 Prerequisites ............................................................... 1
   1.2 License and Copyright .................................................... 2

2 The SPEC hardware architecture ......................................... 3
   2.1 Main features ............................................................... 4
   2.2 PCB Design files ............................................................ 5
   2.3 FMC interface ............................................................... 6
   2.4 Operation Modes ........................................................... 7
       2.4.1 The SPEC as a stand-alone board ....................................... 7
       2.4.2 The SPEC as a PCI Express Slave ....................................... 8

3 Designing your own SPEC Gateware ...................................... 11
   3.1 The Wishbone Bus and the OHR Core libraries ......................... 11
       3.1.1 The Gennum GN4124 Core ............................................. 11
       3.1.2 The General Cores library ........................................... 12
   3.2 Building a simple Wishbone Slave ....................................... 13
       3.2.1 Introducing the Wishbone Slave Generator project ..................... 13
       3.2.2 Installing the Wishbone Slave Generator wbgen2 tool ................. 14
   3.3 Code overview of the SPEC Demo HDL examples .......................... 14
       3.3.1 The SPEC Demo User gateware ..................................... 15
       3.3.2 The SPEC Demo User + DIO gateware .................................. 17
   3.4 Compiling the HDL demos ................................................ 20
       3.4.1 Installing the Xilinx ISE Design Software ............................. 20
       3.4.2 Automated synthesis with Hdlmake .................................. 20
       3.4.3 FPGA bitstream generation .......................................... 21

4 SPEC control from the Host user-space ................................ 23
   4.1 Software support for the SPEC board ................................... 23
       4.1.1 Accessing the SPEC from sysfs ...................................... 23
       4.1.2 Installing the SPEC-SW package ...................................... 24
       4.1.3 The SPEC-SW user-space tools ...................................... 25
   4.2 Interactive control of the SPEC board from Python ..................... 26
       4.2.1 The Spec() Python class ............................................. 27
       4.2.2 The SpecDemoUser() Python class .................................... 28
       4.2.3 The SpecDemoUserDIO() Python class ................................ 30
Chapter 1: Introduction

1 Introduction

This *Getting started with the SPEC* project provides a detailed tutorial on how to get ready to work with the *Simple PCI Express Carrier* [http://www.ohwr.org/projects/spec/wiki](http://www.ohwr.org/projects/spec/wiki).

The *Getting started with the SPEC* project is hosted at [http://www.ohwr.org/projects/spec-getting-started](http://www.ohwr.org/projects/spec-getting-started) and is composed of two main parts:

- Document covering the process of building a full SPEC design, from designing the gateware to handling the board from a user-space application running on a Linux Host PC.
- Companion step-by-step demos, demonstrating how to use the different development tools that are required for bringing up a new project in the SPEC board from the ground up.

The document and demos assume that the user has no previous or very little experience inside the *Open Hardware Repository* initiative. For this reason, the tutorial introduces different resources from *OHR* while covering the following design issues:

- An in depth explanation of the SPEC hardware board, exposing its main features, introducing the PCB design and the FMC standard and available modules and explaining the different operation modes.
- How to design an FPGA gateware for the SPEC board, from introducing the Wishbone bus and its associated OHR tools to building a bitstream from the included HDL demos.
- How to handle the SPEC board from a Linux Host, including the use of the official user-space C software support, direct access to the PCI bus and the interactive control of the board from a Python shell.

1.1 Prerequisites

In order to perform the *Getting Started with the SPEC* tutorial, the following prerequisites need to be fulfilled (note that some application specific tools will be installed later):

**Commercial hardware**

- Simple PCI Express Carrier (mandatory)
- fmc-dio-5chtlla (optional)

**32/64-bits Linux Host PC with a free PCI Express slot**

Designed to be Kernel independent for a maximum compatibility, the tutorial has been successfully performed in every Linux distribution in which has been tested and there are at this moment no incompatible distributions known.

**Clone the Getting Started with the SPEC git repository**

```bash
user$ git clone git://ohwr.org/fmc-projects/spec/spec-getting-started.git
```

**Python environment**

Minimum Python 2.x and optional Python 3.x (tested on Python 2.6.6, 2.7.5, 2.7.6 and Python 3.3.2, 3.3.3).

**Packages dependencies for building the code:**

Debian derivatives:
user$ sudo apt-get install git gcc make

RHEL derivatives:
root# yum install git gcc make

Package dependencies for building the documentation:
Debian derivatives:
user$ sudo apt-get install emacs texinfo texlive texi2html
user$ sudo apt-get install libpng-dev libjpeg-dev libgif-dev

RHEL derivatives:
root# yum install emacs texinfo texlive texinfo-tex texi2html
root# yum install libpng-devel libjpeg-devel giflib-devel

1.2 License and Copyright
This document is licensed under the Creative Commons Attribution-ShareAlike 4.0 International License. To view a copy of this license, visit http://creativecommons.org/licenses/by-sa/4.0/deed.en_US.

The licenses for the associated source code and demos can be found as a companion of the code itself and are specified in the delivered works.

Copyright Notice: CERN, the European Organization for Nuclear Research, is the first and sole owner of all copyright of both this document and the associated source code deliverables.
2 The SPEC hardware architecture

The Simple PCI Express Carrier is an FMC carrier that can hold one FMC card and an SFP connector. On the PCIe side it has a 4-lane interface, while the FMC mezzanine slot uses a low-pin count connector. This board is optimised for cost and is usable with most of the FMC cards designed within the OHR project (e.g. ADC cards, Fine Delay).
Chapter 2: The SPEC hardware architecture

2.1 Main features

**PCIe Support:** The SPEC includes a 4-lane PCIe interface based in the Gennum GN4124. This chip is designed to work as a companion for FPGA devices to provide a complete bridging solution for general applications.

**Programmable Logic:** The SPEC is powered by a Xilinx Spartan6 FPGA (XC6SLX45T-3FGG484C). As a member of the Spartan-6 LXT family this device is intended for High-speed serial connectivity and the main resources that can be used by the user in order to build custom gateware designs are:

- 43,661 equivalent Logic Cells.
- 401 Kb of maximum distributed RAM.
- 58 DSP48A1 slices (each slice contains an 18x18 multiplier, an adder and an accumulator).
- 4x Clock Management Tiles (each CMT contains two DCM and one PLL).

**FMC Slot:** The SPEC mounts a low pin count (LPC) connector which main features are:

- Vadj fixed to 2.5V.
- FMC connectivity: all 34 differential pairs connected, 1 GTP transceiver with clock, 2 clock pairs, JTAG.
- No dedicated clock signals from Carrier to FMC (only available on HPC pins).

**Clocking Resources:** The SPEC design includes a plethora of clocking sources that can be used to develop highly precise applications:

- 1x 10-280 MHz I2C Programmable XO Oscillator, starts up at 100 MHz (Silicon Labs Si570).
- 1x 25 MHz TCXO controlled by a DAC with SPI interface (AD5662).
- 1x 20 MHz VCXO controlled by a DAC with SPI interface (AD5662).
- 1x low-jitter frequency synthesizer (TI CDCM61004, fixed configuration, Fout=125 MHz).

**On-board Memory:** The SPEC mounts both volatile and non volatile memory ICs:

- 1x 2 Gbit (256 MByte) DDR3 (MT41J128M16HA-15E).
- 1x SPI 32 Mbit flash PROM (M25P32-VMF6P) for multiboot FPGA powerup configuration, storage of the FPGA firmware or of critical data.

**Miscellaneous:** The SPEC includes the following extra functionalities:

- on-board thermometer IC (DS18B20U*).
- unique 64-bit identifier (DS18B20U*).

**Front Panel:** The mechanical design features a front panel that fits in standard PCI slots and contains the following items:

- 1x Small Formfactor Pluggable (SFP) cage for fibre-optic transceiver.
- Programmable Red and Green LEDs.
- FMC front panel.

**Internal Connectors:** Several interfaces are available in the SPEC’s PCB out of the front panel:

- 1x JTAG header for Xilinx programming during debugging.
- 2x SATA connector.

**FPGA configuration:** The Spartan-6 device can optionally be programmed in different ways that can be selected by using the GN4124 configurable GPIO:

- GN4124 SPRIO interface (bitstream loaded by software).
- JTAG header (bitstream loaded using specific programming hardware).
• SPI 32Mbit flash PROM.

Stand-alone Features: The SPEC board includes different features intended for a better stand-alone operation:

• External 12V power supply connector.
• mini USB connector (USB-UART bridge).
• 4x auxiliary user LEDs.
• 2x auxiliary user buttons.

Power Consumption: The SPEC board power consumption ranges from 5 to 12 Watt depending on both the application and the operation mode (stand-alone vs PCIe slave).

Optimised for cost: The SPEC’s PCB uses a 6-layer stack-up for a lower production cost.

2.2 PCB Design files

The SPEC design is licensed under the terms of CERN Open Hardware License (CERN-OHL). In this way, when a SPEC unit is purchased, the user is granted with full access to all the design files, including:

• Schematics.
• PCB layout.
• Assembly.
• Manufacturing.
• Mechanical parts.

The specific version of the CERN-OHL covering the design must be indicated by the licensor/manufacturer. In order to check the different rights and responsibilities that CERN-OHL implies for both the licensor and the licensee, more information can be found in http://www.ohwr.org/projects/cernohl/wiki.

The SPEC design files are placed online in the file tab of the official project website (http://www.ohwr.org/projects/spec/files). The current SPEC version is 4.0 and the associated design files can be downloaded by using any of the following links:

• http://www.ohwr.org/attachments/download/918/SPEC_V4.tar.bz2.

In order to open and modify the SPEC design project, Altium Designer schematic capture and layout editor tool packages are required. In any case, a complete documentation set for the design is released in pdf format too.
Chapter 2: The SPEC hardware architecture

3D view of the SPEC design in Altium Designer

Note: in the SPEC project site hosted on OHR you may find a SVN repository in which some outdated SPEC design files are hosted. If you have any doubts about the specific design version and associated documentation that matches your SPEC unit, contact your SPEC board provider in order to get more information as granted by CERN-OHL.

2.3 FMC interface

The SPEC board is designed to act as a carrier for a low pin count (LPC) FPGA Mezzanine Card (VITA 57). In order to get more information about the VITA 57.1 FPGA Mezzanine Card (FMC) standard, visit FMC Projects in the Open Hardware Repository (http://www.ohwr.org/projects/fmc-projects).

In this getting started guide for the SPEC board, two different FMC use cases will be considered in the demo examples:

- No FMC module is plugged in.
- fmc-dio-5chttla is plugged in (check http://www.ohwr.org/projects/fmc-dio-5chttla).

The fmc-dio-5chttla 5-channel digital I/O module has been selected as a demonstrative platform and because it is commercially available from different hardware providers. In addition, this FMC module is licensed under CERN-OHL too.
2.4 Operation Modes

The SPEC board is designed to allow the operation in both stand-alone and PCIe slave modes. In this getting started guide, we will consider only the case in which the SPEC board is attached to a PC through the PCI Express port. This option has been chosen because it does not require additional third-party hardware tools to handle the SPEC and because it makes the things easier for a newcomer, as the control is on the Host PC side and the gateware designs can be simplified.

2.4.1 The SPEC as a stand-alone board

The SPEC board can be deployed as an independent stand-alone unit. In order to enable this operation mode, the Simple PCI Express Carrier includes the following functionalities:

- External 12V power supply connector.
- mini USB connector (USB-UART bridge).
- 4x user LEDs.
- 2x user buttons.

By embedding a soft processor in the Spartan-6 FPGA, the SPEC unit can be used as an FMC enabled Single Board Computer (SBC). This topic is not covered in this guide, but deploying a Wishbone bus enabled LM32 processor may be an appropriated choice when working in the OHR environment. You can find more info about this IP-core and its companion Wishbone peripherals in the Platform Independent Core Collection project on the Open Hardware Repository [http://www.ohwr.org/projects/general-cores/wiki](http://www.ohwr.org/projects/general-cores/wiki).

In order to program and debug the SPEC board when operating in stand-alone mode, the recommended third-party tool is Xilinx’s Platform Cable USB [http://www.xilinx.com/products/boards-and-kits/HW-USB-II-G.htm](http://www.xilinx.com/products/boards-and-kits/HW-USB-II-G.htm).
2.4.2 The SPEC as a PCI Express Slave

The Simple PCI Express Carrier can act as a PCI Express slave. In order to do that, the SPEC includes a 4-lane PCIe interface based in the Gennum GN4124. In this way, you can plug your SPEC board in any PCIe compliant standard port with x4 lane width or greater (x4, x8, x16 or x32). If the slot lane width is greater than x4, the Gennum GN4124 and the PC host will automatically negotiate the highest mutually supported lane count, i.e. x4.

It’s important to highlight in this point that the SPEC board does not fit in a PCIe x1 slot. Because this kind of ports are widely used in x86 embedded mother boards, you should be cautious when selecting a new one that claims to be PCIe enabled if you are planning to use it with the SPEC.

Note that multiple SPEC boards can be attached to a single Host if multiple x4 compatible PCIe slots are available. In this getting started guide, only one free PCIe port will be required as only one SPEC board is used in the demonstration.

In the following pictures, a SPEC board mounting an fmc-dio-5chttla module is shown while being attached to a standard desktop PC PCIe slot. This is the full hardware setup required to go through all the getting started guide.

*Mother Board supporting x16 PCIe and x1 PCIe slots*
Chapter 2: The SPEC hardware architecture

SPEC board being attached to the compatible x16 PCIe slot

SPEC board already installed in the x16 PCIe slot
Chapter 2: The SPEC hardware architecture

Rear view of the PC showing the SPEC + FMC DIO

Closer view of the SPEC + FMC DIO front panel
3 Designing your own SPEC Gateware

In this chapter we will cover the topic of building some simple gateware examples to be loaded on the SPEC’s Spartan-6 FPGA. In order to do that we will learn how to design and synthesize a simple Wishbone based embedded system by using standard and custom HDL cores and specific tools hosted in the Open Hardware Repository.

3.1 The Wishbone Bus and the OHR Core libraries

The Wishbone Bus is an open source hardware computer bus intended to let the parts of an integrated circuit communicate with each other. The aim is to allow the connection of different cores to each other inside of a chip/FPGA.

Initially designed for being used as the standard bus in the OpenCores project, the Wishbone Bus is the mainstream bus used in most of the Open Hardware Repository projects too. In this way, Wishbone provides a standard way for designers to combine already available generic hardware logic designs (called "cores") with custom ones designed for application specific purposes.

While Wishbone is defined to have 8, 16, 32, and 64-bit buses, most of the cores used in the Open Hardware Repository projects use a 32-bit bus. This is because the reference processor used in OHR flagship projects is the LM32 (LatticeMico32), an Open Source 32-bit RISC processor that has been designed for efficient FPGA implementation.

For an in-depth description of the Wishbone bus internals, you can get the full Wishbone specification by following the following link:


In the demos covered by this tutorial, different cores from the Open Hardware Repository are used for building the HDL embedded system examples. In order to fetch the code of these required core libraries, the git submodule mechanism is used.

In order to do that, once the spec-getting-started repository has been cloned, we must jump to the main folder inside a shell and execute the following git command sequence:

```
user$ cd spec-getting-started
user$ git submodule init
user$ git submodule update
```

Once the update command has finished, two new folders are created in hdl/ip_cores directory. These are general-cores and gn4124-core, and their contents are described in the coming sections.

3.1.1 The Gennum GN4124 Core

When designing a gateware intended to run in the SPEC board while working as a PCIe slave, deploying an interface between the Gennum GN4124 device and the FPGA embedded system is a mandatory task.

For this purpose, the gn4124-core project is included in the Open Hardware repository. This project aims to provide a Wishbone master generic interface for FMC projects controlled by a PCI express access.
Chapter 3: Designing your own SPEC Gateware

Overview of the GN4124 Core architecture

The gn4124-core is used in the Getting Started with the SPEC demos in order to communicate from the Host PC to the SPEC board. This is the case too for all the projects included in the OHR that make use of the Gennum GN4124 device for building the hardware bridge.

A more detailed description of the gn4124-core project can be found in its official OHR project site:


The git repository in which the stable code is hosted, can be accessed by using the following Git Read-Only URL:

- [git://ohwr.org/hdl-core-lib/gn4124-core.git](git://ohwr.org/hdl-core-lib/gn4124-core.git)

### 3.1.2 The General Cores library

The general-cores project, A.K.A. Platform-Independent Core Library, Generic Cores Library or GenCores provides a number of common VHDL components used in various projects hosted in the OHR.

The library comprises 3 different packages:

- **gencores.pkg**: simple cores (synchronizer chain, delay generator, pulse extender, PI controller, CRC generator, etc.)
- **genrams.pkg**: collection of platform-independent wrappers for RAMs and FIFOs provided by the FPGA vendors (currently supported: Altera Cyclone3, Arria 2 GX and Xilinx Spartan6/Virtex6)
- **wishbone.pkg**: set of commonly used Wishbone modules (UART, SPI, I2C, OneWire, GPIO, Timer, Interrupt controller, LM32 CPU, Pipelined WB Interconnect)

In the Getting Started with the SPEC project demos, both gencores.pkg and wishbone.pkg libraries are needed, the first for getting some simple standard HDL building blocks and the latter for cleaner Wishbone bus signals handling.
Chapter 3: Designing your own SPEC Gateware

The official project site for *general-cores* can be found at the OHR link:

The **git** repository in which the stable code is hosted, can be accessed by using the following Git Read-Only URL:
- [git://ohwr.org/hdl-core-lib/general-cores.git](http://git://ohwr.org/hdl-core-lib/general-cores.git)

### 3.2 Building a simple Wishbone Slave

After reviewing the different HDL core libraries from OHR that are used in the *Getting Started with the SPEC* gateware demos, we will be building a new Wishbone compatible custom core from the ground up. While designing an application specific HDL core is a task that implies an inevitable design effort (e.g. when a new FMC module is going to be handled), linking this core by hand to the Wishbone bus supposes an engineering cost that should be avoided.

#### 3.2.1 Introducing the Wishbone Slave Generator project

In order to solve this issue, the *Wishbone Slave Generator* project has been developed inside the *Open Hardware Repository* initiative. Current version of this tool, A.K.A. *wbgen2*, allows for generating VHDL/Verilog cores which implement Wishbone bus slaves with certain registers, memory blocks, FIFOs and interrupts. The input is a C-like syntax file with an abstract description of what do we want to have in the slave.

As a result, we get:
- Customisable register types, with multiple access options and multiple clocking schemes
- Configurable memory blocks
- Peripheral-level interrupts via Embedded Interrupt Controller
- Generation of VHDL/Verilog synthesizable code
- Automatic address space layout generation
- Generation of C header files containing memory map consistent with the HDL core
- Support for popular synthesizable data types

In this way, when a slave core is generated we get an HDL black-box in which the primitives are accessible from outside as VHDL/Verilog signals, both the Wishbone bus interface and the user defined ones.

![General structure for a slave core generated by wbgen2](image-url)
More information about the Wishbone Slave Generator can be found in the official OHR project site:

- http://www.ohwr.org/projects/wishbone-gen/wiki

wbgen2 is actively used across the OHR repository, and this is the case for the custom demo cores in the Getting Started with the SPEC examples. In the coming sections, the Wishbone generator descriptions for the different demo slave cores will be introduced, but we must learn how to install this tool as a previous step.

### 3.2.2 Installing the Wishbone Slave Generator wbgen2 tool

The stable code for the wbgen2 can be found in the git repository tab in the Wishbone Slave Generator project. In this section, we will download, build, and deploy the tool.

First of all, we need to install the wbgen2 dependencies. This tool is written in lua language, so we need to install the appropriated software packages for our Linux distribution.

In Ubuntu Linux and other Debian derivatives, this can be made by issuing the following apt command in a shell:

```
user$ sudo apt-get install lua5.1
```

In Red Hat Enterprise Linux derivatives, such as Scientific Linux, this can be made by issuing the following yum command in a shell:

```
root$ yum install lua
```

Once we have this package installed, we can clone and build the wbgen2. For this purpose, we place ourselves in the workspace folder in which we want to deploy the tool and execute:

```
user$ git clone git://ohwr.org/hdl-core-lib/wishbone-gen.git
user$ cd wishbone-gen
user$ make
```

After building the tool, the wbgen2 binary is available inside the wishbone-gen folder. The next step is including it inside the wishbone-gen directory. For this purpose, we will need to export the wishbone-gen directory with the following command:

```
user$ export PATH=$PATH:<wishbone-gen dir>
```

This must be exported every time we are going to use the wbgen2 tool or, alternatively, can be appended at the end of the .bashrc file for every user in the system that is going to run the tool.

### 3.3 Code overview of the SPEC Demo HDL examples

In this section, we will review the code structure for the Getting Started with the SPEC HDL examples. First of all, we must note that the HDL code includes two different examples:

- **Demo User**: this demo makes basic SPEC features accessible throughout the Wishbone to GN4124 bridge.
- **Demo User + DIO**: this demo expands the Demo User functionality by providing a very simple interface to the FMC DIO 5Ch TTL module.

In this way, the Getting Started with the SPEC allows both the use of a PCIe attached SPEC board with or without an available Digital I/O FMC module.

The directory layout for the provided HDL code is shown in the following scheme:

- spec-getting-started/
  - hdl/
The purpose for each of these folders is:

**ip_cores/**
The directory hosts the cores that are fetched from already available OHR libraries. In both of the demos, the gn4124-core and some elements from general-cores are required, as previously explained when the Wishbone architecture has been presented.

**modules/**
The directory hosts the custom cores that has been specifically designed for being used in the Getting Started with the SPEC demos. Both reset_generator and spec_user_interface are used in the two different demos, but fmc_dio_ch5_ttl is only required by the demo supporting the Digital I/O FMC module.

**top/**
The directory hosts the top level design files for both of the demo designs, that are placed in independent folders. In each of the demo specific folders, we will find a VHDL spec_top.vhd file that specifies the top level entity architecture, and a Xilinx user constraints spec_top.ucf file that properly assigns the top port signals to the physical pins in the Spartan-6 FPGA.

**syn/**
This directory is intended to host the full bitstream synthesis process, including an independent folder for each of the demos. In the demo specific folders we can find the information that is required for running an automated synthesis, as will be explained later when introducing the gateware building process.

### 3.3.1 The SPEC Demo User gateware

The Demo User gateware exposes basic SPEC board functions throughout the Gemnum GN4124 bridge by mapping some custom registers in the Wishbone bus. By using this HDL design, we can control the Green/Red LEDs in the SPEC front panel, read the SPEC PCB version or access to the internal pushbuttons and LEDs.
In order to do this, the Demo User includes the `spec_user_interface` custom core, that has been implemented by using the *Wishbone Slave Generator* tool. The most important files that can be found inside this module directory are:

- **`build_wb.sh`**
  A simple shell script including the `wbgen2` commands for generating both the VHDL Wishbone slave core and its documentation in texinfo and html format from the *Wishbone Slave Generator* description file.

- **`spec_user_interface.wb`**
  The *Wishbone Slave Generator* description file in which the different register primitives for handling the basic SPEC board functions are indicated.

- **`spec_user_interface.vhd`**
  The actual Wishbone slave VHDL code that is automatically generated by `wbgen2` from the *Wishbone Slave Generator* description file.

In the rest of this subsection, the memory map summary and the role of each of the Wishbone registers that `spec_user_interface` exposes are detailed.

**Memory map summary**

<table>
<thead>
<tr>
<th>Address</th>
<th>Type</th>
<th>Prefix</th>
<th>Name</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x0</td>
<td>REG</td>
<td>ctrl</td>
<td>Control register</td>
</tr>
<tr>
<td>0x4</td>
<td>REG</td>
<td>aux</td>
<td>Auxiliary interface</td>
</tr>
</tbody>
</table>

**ctrl - Control register**

A register containing basic SPEC controls

<table>
<thead>
<tr>
<th>Bits</th>
<th>Access</th>
<th>Prefix</th>
<th>Default</th>
<th>Name</th>
</tr>
</thead>
<tbody>
<tr>
<td>3...0</td>
<td>R/O</td>
<td>VER</td>
<td>X</td>
<td>PCB Version</td>
</tr>
<tr>
<td>4</td>
<td>R/W</td>
<td>LED_GREEN</td>
<td>0</td>
<td>Front panel green LED</td>
</tr>
<tr>
<td>5</td>
<td>R/W</td>
<td>LED_RED</td>
<td>0</td>
<td>Front panel red LED</td>
</tr>
</tbody>
</table>
### Field Description

- **ver**: This field accesses to the hard-wired 4 bits that indicate the PCB version
- **led_green**: Control bit for the green LED placed on the SPEC front panel
- **led_red**: Control bit for the red LED placed on the SPEC front panel

#### aux - Auxiliary interface

A register mapping the SPEC internal auxiliary interface

<table>
<thead>
<tr>
<th>Bits</th>
<th>Access</th>
<th>Prefix</th>
<th>Default</th>
<th>Name</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>R/O</td>
<td>BUTTON1</td>
<td>X</td>
<td>User Push-button 1</td>
</tr>
<tr>
<td>1</td>
<td>R/O</td>
<td>BUTTON2</td>
<td>X</td>
<td>User Push-button 2</td>
</tr>
<tr>
<td>5...2</td>
<td>R/W</td>
<td>LEADS</td>
<td>0</td>
<td>User Leds</td>
</tr>
</tbody>
</table>

### 3.3.2 The SPEC Demo User + DIO gateware

The SPEC Demo User + DIO demo expands the Demo User functionality by providing a very simple interface to the FMC DIO 5Ch TTL module. In this way, in addition to the basic SPEC controls provided by Demo User, a set of registers handling the Digital I/O module are exposed in the Wishbone bus. By accessing to these registers, we can configure the DIO pin direction, read/write each pin value or configure the termination resistor. In addition, the Top/Bottom LEDs in the FMC DIO front panel can be controlled too.

![Block diagram for the SPEC Demo User + DIO gateware](image-url)
In order to do this, the Demo User + DIO adds to the previous demo Wishbone bus a new `fmc_dio_ch5_ttl` custom core that has been implemented by using the *Wishbone Slave Generator* tool. The most important files that can be found inside this module directory are:

- **build_wb.sh**
  A simple shell script including the `wbgen2` commands for generating both the VHDL Wishbone slave core and its documentation in texinfo and html format from the *Wishbone Slave Generator* description file.

- **wb_slave_fmc_dio_ch5_ttl.wb**
  The *Wishbone Slave Generator* description file in which the different register primitives for handling the basic DIO FMC functions are indicated.

- **wb_slave_fmc_dio_ch5_ttl.vhd**
  The actual Wishbone slave VHDL code that is automatically generated by `wbgen2` from the *Wishbone Slave Generator* description file.

- **fmc_dio_ch5_ttl.vhd**
  A VHDL code that connects the Wishbone slave registers to the FPGA. This code includes the Wishbone slave as a component and is the file that is actually instantiated from the top level design VHDL description.

In the rest of this subsection, the memory map summary and the role of each of the Wishbone registers that `fmc_dio_ch5_ttl` exposes are detailed. Note that in the Demo User + DIO, a basic Wishbone bus multiplexor is implemented in order to allow the access to both the `fmc_dio_ch5_ttl` and the `spec_user_interface` cores from the GN4124 bridge. In this way, the DIO FMC registers are placed at the 0x100 base address, while the SPEC user interface remains at 0x000 base address.

**Memory map summary**

**Base Address: 0x100**

<table>
<thead>
<tr>
<th>Address</th>
<th>Type</th>
<th>Prefix</th>
<th>Name</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x0</td>
<td>REG</td>
<td>ddr</td>
<td>Pin direction register</td>
</tr>
<tr>
<td>0x4</td>
<td>REG</td>
<td>psr</td>
<td>Pin state register</td>
</tr>
<tr>
<td>0x8</td>
<td>REG</td>
<td>pdr</td>
<td>Pin output register</td>
</tr>
<tr>
<td>0xc</td>
<td>REG</td>
<td>term</td>
<td>Pin termination register</td>
</tr>
<tr>
<td>0x10</td>
<td>REG</td>
<td>sopr</td>
<td>Set output pin register</td>
</tr>
<tr>
<td>0x14</td>
<td>REG</td>
<td>copr</td>
<td>Clear output pin register</td>
</tr>
<tr>
<td>0x20</td>
<td>REG</td>
<td>leds</td>
<td>LED signaling interface</td>
</tr>
</tbody>
</table>

**ddr - Pin direction register**

A register defining the direction of the DIO pins.

<table>
<thead>
<tr>
<th>Bits</th>
<th>Access</th>
<th>Prefix</th>
<th>Default</th>
<th>Name</th>
</tr>
</thead>
<tbody>
<tr>
<td>4...0</td>
<td>R/W</td>
<td>DDR</td>
<td>0</td>
<td>Pin directions</td>
</tr>
</tbody>
</table>

**Field**

**ddr**

Each bit in this register defines the direction of the corresponding pin in the DIO. 1 means the pin is an OUTPUT, 0 means the pin is an INPUT.

**psr - Pin state register**
A register containing the current state of the DIO pins.

<table>
<thead>
<tr>
<th>Bits</th>
<th>Access</th>
<th>Prefix</th>
<th>Default</th>
<th>Name</th>
</tr>
</thead>
<tbody>
<tr>
<td>4...0</td>
<td>R/O</td>
<td>PSR</td>
<td>X</td>
<td>Pin input state</td>
</tr>
</tbody>
</table>

**Field** | **Description**
---|---
psr     | Each bit in this register reflects the state of the corresponding pin in the DIO. 1 means the pin is HIGH, 0 means the pin is LOW

**pdr - Pin output register**
A register that allows changing the value of the DIO pins by means of a direct write access

<table>
<thead>
<tr>
<th>Bits</th>
<th>Access</th>
<th>Prefix</th>
<th>Default</th>
<th>Name</th>
</tr>
</thead>
<tbody>
<tr>
<td>4...0</td>
<td>W/O</td>
<td>PDR</td>
<td>0</td>
<td>Pin output value</td>
</tr>
</tbody>
</table>

**term - Pin termination register**
A register defining the use of the 50 Ohm termination

<table>
<thead>
<tr>
<th>Bits</th>
<th>Access</th>
<th>Prefix</th>
<th>Default</th>
<th>Name</th>
</tr>
</thead>
<tbody>
<tr>
<td>4...0</td>
<td>R/W</td>
<td>TERM</td>
<td>0</td>
<td>Pin terminations</td>
</tr>
</tbody>
</table>

**Field** | **Description**
---|---
term | Writing '1' activates the termination resistor

**sopr - Set output pin register**
Writing '1' sets the corresponding DIO pin to '1'

<table>
<thead>
<tr>
<th>Bits</th>
<th>Access</th>
<th>Prefix</th>
<th>Default</th>
<th>Name</th>
</tr>
</thead>
<tbody>
<tr>
<td>4...0</td>
<td>W/O</td>
<td>SOPR</td>
<td>0</td>
<td>Set output pin register</td>
</tr>
</tbody>
</table>

**copr - Clear output pin register**
Writing '1' clears the corresponding DIO pin

<table>
<thead>
<tr>
<th>Bits</th>
<th>Access</th>
<th>Prefix</th>
<th>Default</th>
<th>Name</th>
</tr>
</thead>
<tbody>
<tr>
<td>4...0</td>
<td>W/O</td>
<td>COPR</td>
<td>0</td>
<td>Clear output pin register</td>
</tr>
</tbody>
</table>

**leds - LED signaling interface**
Writing '1' activates the corresponding LED

<table>
<thead>
<tr>
<th>Bits</th>
<th>Access</th>
<th>Prefix</th>
<th>Default</th>
<th>Name</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>R/W</td>
<td>BOT</td>
<td>0</td>
<td>FMC DIO Bottom LED</td>
</tr>
<tr>
<td>1</td>
<td>R/W</td>
<td>TOP</td>
<td>0</td>
<td>FMC DIO Top LED</td>
</tr>
</tbody>
</table>

**Field** | **Description**
---|---
bot  | Control bit for the bottom LED placed on the FMC DIO front panel
top | Control bit for the top LED placed on the FMC DIO front panel
3.4 Compiling the HDL demos

In this section, the process of building a bitstream from the HDL code is explained. In order to do this, we are going to use two different tools, the Xilinx ISE Design Software and the OHR’s HdlMake.

3.4.1 Installing the Xilinx ISE Design Software

In order to synthesize, map, place & route and generate the bitstream for the Xilinx Spartan-6 FPGA, we need the Xilinx ISE proprietary toolchain. The good news are that the specific Spartan-6 XC6SLX45T-3FGG484C device mounted by the SPEC board is supported by the ISE WebPACK edition, a free, fully featured front-to-back FPGA design solution for Linux and Windows.

In this way, before building our gateware designs, we need to download and install the ISE WebPACK Design Software for Linux, that can be found in the following link: www.xilinx.com/products/design-tools/ise-design-suite/ise-webpack.htm.

The install binary for Linux supports both 32 and 64 bits Operating Systems, and is compatible with mainstream Linux distributions right out-of-the-box. This include both Debian derivatives such as Ubuntu and RHEL derivatives such as Scientific Linux.

Note that, once the ISE software has been installed, pointing to a valid WebPACK license file is mandatory before running any of the Xilinx tools. You can request a WebPACK license by using a valid Xilinx registration account in the following link: http://www.xilinx.com/getlicense.

When working in Linux environments, we need to update the shell configuration with the appropriate Xilinx toolchain information before using any of the tools included in the Xilinx ISE Design Software. In order to do this, we need to source the ISE settings file that matches with our Operating System shell and processor architecture.

As an example, when the ISE Design Software had been installed in the default path and the ISE version was 14.7, the command required to load the settings in a 32 bits Operating System would be:

```
user$ source /opt/Xilinx/14.7/ISE_DS/settings.sh
```

while in a 64 bits Operating System this would be:

```
user$ source /opt/Xilinx/14.7/ISE_DS/settings64.sh
```

If we want to avoid launching this command everytime we start a shell terminal, we can append it to the end of the .bashrc file placed in our user home directory.

3.4.2 Automated synthesis with Hdlmake

Despite the fact that Xilinx ISE includes very powerful graphic user interface tools, sometimes we feel that working with HDL code just in the way we do with software code will be very handy. For this purpose, the HdlMake project has been created inside the Open Hardware Repository initiative.

Hdlmake generates multi-purpose makefiles for FPGA projects. It supports local and remote synthesis, simulation, fetching module dependencies from repositories, creating Quartus/ISE project files. All of this can be done with a makefile command or with Hdlmake directly. It supports modularity, scalability, use of revision control systems and code reuse.

In order to get more information about Hdlmake, you can visit the official project site hosted on the Open Hardware Repository.

Chapter 3: Designing your own SPEC Gateware

**Note:** Current Hdlmake version is under heavy development. For this reason, we will use an older but stable Hdlmake release for synthesizing our demos.

As Hdlmake is written as a Python application, before installing and running Hdlmake, a valid Python distribution must be available on the Operating System. Python is included as an off-the-shelf basic package in almost all of the mainstream Linux distribution, so this shouldn’t be a problem.

Once this requisite is met, we can clone and build Hdlmake. For this purpose, we place ourselves in the workspace folder in which we want to deploy the tool and execute:

```
user$ git clone -b isyp git://ohwr.org/misc/hdl-make.git
user$ cd hdl-make
user$ make
```

After building, the hdlmake Python app is available inside the hdl-make folder. The next step is including it inside the user shell path. For this purpose, we will need to export the hdl-make directory with the following command:

```
user$ export PATH=$PATH:<hdl-make dir>
```

This must be exported everytime we are going to use the wbgen2 tool or, alternatively, can be appended at the end of the .bashrc file for every user in the system that is going to run the tool.

### 3.4.3 FPGA bitstream generation

Once both the Xilinx ISE and Hdlmake tools are available from a user shell, we are ready to run the automated synthesis process. For this purpose, we should go to the specific syn/ folder associated to the demo that is going to be synthesized.

In order to exemplify this, we are going to detail the steps for building the Demo User gateware. For building the Demo User + DIO, the process is exactly the same.

```
user$ cd spec-getting-started/hdl/syn/spec/demo_user
```

Inside this folder, we can see that a Manifest.py file is placed. Similar files can be found across the whole hdl/ directory, as these are used by Hdlmake in order to get information about the synthesis process, from pointing to the HDL packages/files that must be included to specifying the parameters of the target hardware.

The Manifest.py placed in the syn folder contains all the required information for generating a new spec_top.xise Xilinx ISE project and a companion Makefile from the ground up. In order to do this, we must execute:

```
user$ hdlmake --make-ise --ise-proj
```

If we open the spec_top.xise with a text editor, we will check that this contains only very basic information (Note that we could open this project with the Xilinx ISE graphic interface, but this is not necessary when working with Hdlmake).

Once we have a valid Makefile and spec_top.xise ISE project, we are ready to generate the bitstream by just launching a make command:

```
user$ make
```

This process can take some minutes depending on the Host CPU speed and the complexity of the HDL design synthesized. Once the synthesis, map, P&R... processes have finished, we have a lot of new files in our folder, including a spec_top.bit bitstream that can be loaded into the SPEC board by using Xilinx programmer.
In order to generate an additional `spec_top.bin` bitstream file that can be loaded to the FPGA throughout the PCI Express bridge, we need to open the `spec_top.xise` project file in a text editor. By doing this, we can check that a lot of new configuration parameters have been added to the project file in the first synthesis run.

Once the `spec_top.xise` is opened, we must search the Create Binary Configuration File property and change its value from `false` to `true`. Then, by just executing `make` again, the `spec_top.bin` will be generated too.

Now that we have a valid `spec_top.bin` file, we can jump to the software side in order to program and control the SPEC board from the Host user space.
Chapter 4: SPEC control from the Host user-space

4 SPEC control from the Host user-space

In this chapter we will learn how to control a SPEC board attached to the PCIe slot of the a Linux Host PC. For this purpose, we will introduce the official OHR software support for the SPEC project and the user-space tools it provides, will learn how to get direct access to the SPEC address space by using the Sysfs PCI bus interface and will interactively handle the SPEC board from a Python environment.

4.1 Software support for the SPEC board

The software companion of the Simple PCI Express Carrier in the OHR is the Software support for the SPEC board project, A.K.A. spec-sw. This software package is a complete solution including kernel and user-space Linux code, and is actively used by the most important SPEC based projects hosted in the Open Hardware Repository.

In the Getting Started with the SPEC project demos, we will use the user-space Linux code side of the spec-sw package, as the kernel support is not mandatory for basic SPEC handling and is intended for most advanced designs. In addition, by not using the kernel code, we can deploy and run the SPEC demos in a wider spectrum of Linux distributions, as building the kernel modules in the spec-sw package requires kernel versions greater than 2.6.36 (e.g. current version of Red Hat Enterprise Linux and its derivatives include kernel 2.6.32 by default).

In order to get more information about the Software support for the SPEC board project, you can visit its official site in the OHR:


4.1.1 Accessing the SPEC from sysfs

Before going deeper into how we can handle the SPEC board from a Linux Host, we must clarify the mechanism in which the SPEC is identified by the PCI system. In a Linux Operating System, this task is performed by the Sysfs virtual filesystem, which exports information about devices and drivers from the kernel device model to user space.

In order to access a SPEC board attached to a PCI Express slot of a Host, we must check the devices that are present in the PCI bus. This is an example of what we can expect to find inside the Sysfs path in which the PCI bus devices are exported:

```
user$ ls /sys/bus/pci/devices/
0000:00:00.0 0000:00:1c.0 0000:00:1d.0 0000:01:00.0 0000:06:00.0
0000:00:01.0 0000:00:1c.4 0000:00:1f.0 0000:01:00.1
0000:00:16.0 0000:00:1c.5 0000:00:1f.2 0000:02:00.0
0000:00:1a.0 0000:00:1c.6 0000:00:1f.3 0000:04:00.0
0000:00:1b.0 0000:00:1c.7 0000:00:1f.5 0000:05:00.0
```

Each one of the PCI devices listed, can be pointed in the PCI system by the bus and devfn numbers. e.g. if we knew that our SPEC board is in the 0000:02:00.0 folder, the associated values would be bus=0x02 and devfn=0x00.

A simple way of checking if a PCI device at the Sysfs PCI system is a SPEC board, is checking the content of its associated device and vendor files. If these values match with the known identifiers for the SPEC board, then we know that the PCI device is actually a SPEC board. e.g.:

```
user$ cat /sys/bus/pci/devices/0000\:02\:00.0/device
0x018d
user$ cat /sys/bus/pci/devices/0000\:02\:00.0/vendor
0x10dc
```
There are two different sets of identifiers associated to the SPEC board, so we must note that we can potentially find any of them for a given SPEC.

- **Newer SPEC PCB Versions:**
  - PCI_VENDOR_ID_CERN = 0x10dc
  - PCI_DEVICE_ID_SPEC = 0x018d

- **Older SPEC PCB Versions:**
  - PCI_VENDOR_ID_GENNUM = 0x1a39
  - PCI_DEVICE_ID_GN4124 = 0x0004

So, in our example, we would have verified that the PCI device at 0000:02:00.0 folder is one of the newer SPEC boards.

Finally, once we have identified a SPEC board in the *Sysfs* PCI bus, we can explore the content of its interface:

```
user$ ls /sys/bus/pci/devices/0000\:02\:00.0
broken_parity_status enable power subsystem
class fmc-0200 remove subsystem_device
config irq rescan subsystem_vendor
consistent_dma_mask_bits local_cpulist reset uevent
d3cold_allowed local_cpulist resource vendor
device modalias resource0
dma_mask_bits msi_bus resource2
driver numa_node resource4
```

From the user-space point of view, the most important files in the exported *Sysfs* folder for the SPEC board are the `resource0` and `resource4` files.

- **resource0**
  - This is the Base Address Register 0 area (BAR0). In this resource, the FPGA Wishbone bus registers are mapped. By accessing to this area, we can control the internal FPGA embedded system from the gn4124-core Wishbone master.

- **resource4**
  - This is the Base Address Register 4 area (BAR4). In this resource, the internal Gennum GN4124 control registers are mapped. By accessing to this area, we can perform system control tasks such as choosing the boot mode or loading a new bitstream in the FPGA.

In this way, by memory mapping this files in any programming language, e.g. C, Python, we can control any SPEC board functionality by performing low level accesses to the different mapped registers. In the coming sections, we will introduce how we can use the *spec-sw* user-space code for a higher level operation.

**Note:** By default, the access to the *Sysfs* devices, including `resource0` and `resource4` files, requires super user privileges.

### 4.1.2 Installing the SPEC-SW package

In order to install the *spec-sw* user-space tools, we need to clone the full repository, but then we only need to build the user-space code placed in the `spec-sw/tools` folder.
Note: As part of the *Getting Started with the SPEC* development, the *spec-sw* code has been upgraded in order to support a fully kernel independent operation. In addition, the building process has been modified in order to generate a *libspec.so* shared object library from the standard and already available *libspec.a* static library. Both of this new features are required in the *Getting Started with the SPEC* demos, so until being merged into main *spec-sw* git repository, we will point to the development branch in which these have been introduced and tested.

```
user$ git clone -b jdgl-140206 git://ohwr.org/fmc-projects/spec/spec-sw.git
user$ cd spec-sw/tools
user$ make
```

Once the build process has finished, the user-space tools and libraries can be found inside the *spec-sw/tools* folder. Now, if we want to execute some of these applications, we must remember that *super user privileges* are required by default in order to access to the SPEC resources.

In this way, if the *Sysfs* SPEC device permissions are not changed, we can use both the root account or the sudo command for direct access from inside the *spec-sw/tools* folder. e.g.

```
user$ cd spec-sw/tools
user$ sudo ./specmem 0x04
user$ su
root# ./specmem 0x04 0xff
```

If we want to add the *spec-sw/tools* folder to the execution path, we must issue the following command, considering that `<spec-sw/tools dir>` is the absolute path to this folder.

```
user$ export PATH=$PATH:<spec-sw/tools dir>
root# export PATH=$PATH:<spec-sw/tools dir>
```

If we want this path to be included everytime a new shell instance is created, we must append the export command to the end of the .bashrc file in the user folder and, if desired, to the root one.

Note: some Linux distributions, in order to make the extended user path available when using a sudo command, need the edition of the /etc/sudoers file by using visudo for commenting out the following line if present:

```
Defaults secure_path="...
```

### 4.1.3 The SPEC-SW user-space tools

The *tools* subdirectory of the *spec-sw* includes a few host-side programs that may be useful while working with the SPEC device. They are all based on the same *speclib*, part of the same directory, so all of them accept some parameters in common, in order to identify one specific SPEC card if you have more than one:

- **-b <bus>**
  
  This option specifies the bus number

- **-d <devfn>**
  
  This is used to specify the device and function number, but it is expected to be 0 on most if not all the computers. You won’t generally need to specify the *devfn* value.

If no arguments are supplied, the tools act on the *first* device if more than one is plugged. The meaning of *first* is actually undefined and system-dependent.
From the set of userspace tools, these are the most interesting ones when working in simple gateware designs such as the included in the *Getting Started with the SPEC* project:

**specmem**

The program acts like *devmem* or *devmem2* but in a simplified way. It receives one or two command line arguments: one for reading and two for writing. Both arguments are used as hex numbers, whether or not the leading 0x is specified. The program makes a single 32-bit access to BAR0 of the Gennum PCI bridge; the first argument is the address, and the second argument is the value to be written. The **VERBOSE** environment variable makes the tool slightly more verbose. If you pass `-g` you will access the Gennum registers (for example, for GPIO access).

**spec-fwloader**

This is a userspace loader for the gateware file. It simply receives a file name argument (after the optional bus number for the device).

In this way, by using these tools, we can take the previously generated *spec_top.bin* bitstream files and load it to the FPGA from the shell. Then, we can perform direct Read/Write access to the mapped Wishbone registers in order to control the internal registers of our design.

### 4.2 Interactive control of the SPEC board from Python

The *Getting Started with the SPEC* project includes a set of Python files that has been designed for demonstrating how a SPEC board can be handled in an interactive way from a Python shell. In order to do that, the code includes a collection of *Python classes* that can be used to create SPEC objects in which different high level methods has been defined.

The directory layout for the provided Python code is shown in the following scheme:

```
− spec-getting-started/.
− sw/
− python/
− spec_libc.py
− spec_demo_user.py
− spec_user_user_dio.py
```

Now, the specific role for each of these Python files will be introduced:

**spec_libc.py**

This Python package defines a general purpose Spec() Python class. The Spec() class requires the *spec-sw/tools/libspec.so* dynamic shared library and imports the basic *speclib* library methods from C to Python Language. This class provides general methods for both loading a bitstream into the FPGA and accessing the user Wishbone registers by using 32-bit and bitwise operations.

**spec_demo_user.py**

This Python package defines the SpecDemoUser() Python class, which inherits the Spec() class and includes specific methods for handling the functionalities included in the user interface defined by the *Spec Demo User gateware*.

**spec_demo_user_dio.py**

This Python package defines the SpecDemoUserDIO() Python class, which inherits the Spec() and SpecDemoUser() classes and adds specific methods for handling the Digital I/O functionalities exposed in the *Spec Demo User+DIO gateware*. 
The Python code in *Getting Started with the SPEC* project has been designed for being compatible with both Python 2.x and Python 3.x. In this way, the Python classes has been heavily tested on Python 2.7.6 and Python 3.3.3.

Note that, in order to run the demos, we must open a shell inside the spec-getting-started/sw/python folder. Once placed in this folder, we must run a Python 2.x or 3.x interactive session with super-user privileges. For doing this, we can alternatively run:

```
user$ sudo python
user$ sudo python3
```

As being a key component of the base Spec() Python class, from which the other demo classes inherit, we must provide a valid libspec.so library when creating any of the objects included in the *Getting Started with the SPEC* project. In order to do this, we can specify the path to a valid libspec.so as a parameter every time a new object is created or we can just copy the libspec.so into the /usr/lib/ folder, in which the Spec() class searchs for the library by default.

```
user$ sudo cp spec-sw/tools/libspec.so /usr/lib/
```

Note: when introducing the following demos, we will assume that a valid libspec.so library is placed into the /usr/lib/ folder.

### 4.2.1 The Spec() Python class

In this section, we are going to explain by example how the Spec() Python class must be used from the interactive Python 2/3 shell. In this way, we will create a new Spec() object, then we will load the bitstream binary for the Spec Demo User gateware, toggle the Red and Green LEDs in the SPEC front panel and finally destroy the Spec() object.

Note that we have included a `help(spec)` in our example. By doing this, we get printed in the Python shell the help for all the methods included on Spec() class (‘q’ for exit).

```
>>> from spec_libc import *
>>> spec = Spec()
a new SPEC object has been created
>>> spec.specLoadBitstream('../hd1/syn/spec/demo_user/spec_top.bin')
the bitstream has been found...
the bitstream has been successfully loaded
>>> spec.specToggleBit(0x0, 5)
>>> spec.specToggleBit(0x0, 5)
>>> spec.specToggleBit(0x0, 4)
>>> spec.specToggleBit(0x0, 4)
>>> help(spec)
>>> del(spec)
The SPEC object has been destroyed
```

**Help on Spec in module spec_libc object:**

```
class Spec(builtins.object)
    | A Class that creates SPEC objects. This includes basic controls
    | and methods in order to handle a PCIe attached SPEC board.
    | Access to a valid libspec.so is mandatory for a proper operation.
    | NOTE: user write/access permission to the SPEC dev is mandatory.
```
Chapter 4: SPEC control from the Host user-space

Attributes:
- **bus, dev**  bus and device for the requested SPEC device.
  - If one (or both) parameters are < 0,
    - takes first available card.
- **libc**  Path to a valid libspec.so shared library.
  - If empty, it points to /usr/lib/libspec.so

Methods defined here:
- **__del__(self)**
  - Destroying the SPEC card object
- **__init__(self, bus=-1, dev=-1, libc='/usr/lib/libspec.so')**
- **specClearBit(self, address, offset)**
  - Clear to 0 the bit placed at offset in the register at address
- **specLoadBitstream(self, bitstream)**
  - Load a new bitstream into the SPEC throughout GN4124 interface.
    - bitstream is the path to the .bin bitstream that must be loaded.
- **specReadL(self, address, hexformat=True)**
  - Read a 32 bit integer data from the register at address.
    - Return an hex string if hexformat=True, an integer otherwise
- **specSetBit(self, address, offset)**
  - Set to 1 the bit placed at offset in the register at address
- **specTestBit(self, address, offset)**
  - Test the bit placed at offset in the register at address.
    - returns a nonzero result, 2**offset, if the bit is 1
- **specToggleBit(self, address, offset)**
  - Toggle/invert the bit placed at offset in the register at address
- **specWriteL(self, address, data)**
  - Write a 32 bit integer data into the register at address

### 4.2.2 The SpecDemoUser() Python class

In this section, we are going to explain by example how the SpecDemoUser() Python class must be used from the interactive Python 2/3 shell. In this way, we will create a new SpecDemoUser() object, then we will load the bitstream binary for the Spec Demo User gateware, read the PCB version, toggle the Red and Green LEDs in the SPEC front panel, write/read the 4x internal LEDs and finally destroy the object.

Note that we have included a `help(spec)` in our example. By doing this, we get printed in the Python shell the help for all the methods included on the SpecDemoUser() class and those inherited from the Spec() class. In this section, we have only included the SpecDemoUser() specific help. ('q' for exit).
>>> from spec_demo_user import *
>>> spec = SpecDemoUser()
a new SPEC object has been created
>>> spec.specLoadBitstream('.../hdl/syn/spec/demo_user/spec_top.bin')
the bitstream has been found...
the bitstream has been successfully loaded
>>> spec.specGetPCBVersion()
3
>>> spec.specToggleGreenLED()
>>> spec.specToggleRedLED()
'0x0'
>>> spec.specSetAuxLEDs(0xf)
>>> spec.specGetAuxLEDs(0xf)
'0xf'
>>> help(spec)
>>> del(spec)
The SPEC object has been destroyed

Help on SpecDemoUser in module spec_demo_user object:

class SpecDemoUser(spec_libc.Spec)
 | Simple demo application demonstrating the use of Spec class.
 | This is part of the getting started with the SPEC project.
 | By creating a SpecDemoUser object, access to the basic functions
 | covered in the gateware HDL Spec Demo User is granted.
 | In this way, the SPEC FPGA must be programmed with the specific bitstream.
 | For this purpose, use the specLoadBitstream() method inherited from Spec.
 |
 | Method resolution order:
 | SpecDemoUser
 | spec_libc.Spec
 | builtins.object
 |
 | Methods defined here:
 |
 | __init__(self, bus=-1, dev=-1, libc='''/usr/lib/libspec.so')
 |
 | specClearGreenLED(self)
 | Clear to OFF the SPEC Front Panel Green LED.
 |
 | specClearRedLED(self)
 | Clear to OFF the SPEC Front Panel Red LED.
 |
 | specGetAuxLEDs(self, hexformat=True)
 | Return the hex value corresponding to the four auxiliary LEDs.
 | If hexformat=True, the returned value is a string in hex format.
 | Note that the auxiliary LEDs are active low.
 |
 | specGetButton1(self)
 | Test the state of the internal auxiliary Button1.
 | Return 0 when pressed and nonzero when released.
specGetButton2(self)
Test the state of the internal auxiliary Button2.
Return 0 when pressed and nonzero when released.

specGetGreenLED(self)
Test the status of the SPEC Front Panel Green LED.
returns a nonzero result if the LED is ON, 0 if the LED is OFF.

specGetPCBVersion(self)
Return the SPEC PCB version as an integer value.

specGetRedLED(self)
Test the status of the SPEC Front Panel Red LED.
returns a nonzero result if the LED is ON, 0 if the LED is OFF.

specSetAuxLEDs(self, value)
Set the hex value corresponding to the four auxiliary LEDs.
Note that the auxiliary LEDs are active low.

specSetGreenLED(self)
Set to ON the SPEC Front Panel Green LED.

specToggleRedLED(self)
Toggle/change the SPEC Front Panel Red LED.

4.2.3 The SpecDemoUserDIO() Python class
In this section, we are going to explain by example how the SpecDemoUserDIO() Python class must be used from the interactive Python 2/3 shell. In this way, we will create a new SpecDemoUserDIO() object, then we will load the bitstream binary for the Spec Demo User+DIO gateware, toggle the top led in the DIO module front panel, configure the pin4 direction as output and then set pin4 to '1' and finally destroy the object.

Note that we have included a help(spec) in our example. By doing this, we get printed in the Python shell the help for all the methods included on the SpecDemoUserDIO() class and those inherited from the Spec() and the SpecDemoUser() classes. In this section, we have only included the SpecDemoUserDIO() specific help. ('q' for exit).

>>> from spec_demo_user_dio import *
>>> spec = SpecDemoUserDIO()
a new SPEC object has been created
>>> spec.specLoadBitstream('..../hdl/syn/spec/demo_user+dio/spec_top.bin')
the bitstream has been found...
the bitstream has been successfully loaded
>>> spec.dioToggleTopLED()
>>> spec.dioWriteDirection('01000')
'01000'
>>> spec.dioReadDirection()
'00000'
'00000'
>>> spec.dioSetOutput('01000')
>>> spec.dioReadValue()
'01000'
>>> help(spec)
>>> del(spec)
The SPEC object has been destroyed

Help on SpecDemoUserDIO in module spec_demo_user_dio object:

class SpecDemoUserDIO(spec_demo_user.SpecDemoUser)
 | Simple demo application demonstrating the use of Spec class.
 | This is part of the getting started with the SPEC project.
 | By creating a SpecDemoUserDIO object, access to the basic functions
 | covered in the gateware HDL Spec Demo User DIO is granted.
 | In this way, the SPEC FPGA must be programmed with the specific bitstream.
 | For this purpose, use the specLoadBitstream() method inherited from Spec.
 |
 | Method resolution order:
 | SpecDemoUserDIO
 | spec_demo_user.SpecDemoUser
 | spec_libc.Spec
 | builtins.object
 |
 | Methods defined here:
 |
 | __init__(self, bus=-1, dev=-1, libc='/usr/lib/libspec.so')
 |
 | dioClearBottomLED(self)
 | Clear to OFF the DIO Front Panel Bottom LED.
 |
 | dioClearOutput(self, value)
 | Clear to 0 the selected DIO pins as a 5 chars string.
 | Write char 1 for selected and 0 for unchanged.
 | e.g. Pin 5 clear, others unchanged: value = '10000'.
 |
 | dioClearTopLED(self)
 | Clear to OFF the DIO Front Panel Top LED.
 |
 | dioGetBottomLED(self)
 | Test the status of the DIO Front Panel Bottom LED.
 | returns a nonzero result if the LED is ON, 0 if the LED is OFF.
 |
 | dioGetTopLED(self)
 | Test the status of the DIO Front Panel Top LED.
 | returns a nonzero result if the LED is ON, 0 if the LED is OFF.
 |
 | dioReadDirection(self)
 | Return the DIO pin directions as a 5 chars string.
 | Read char 1 for output and 0 for input.
 | e.g. Pin 5 output, others input: return '10000'.
 |
 | dioReadTermination(self)
Chapter 4: SPEC control from the Host user-space

Return the DIO pin termination resistor as a 5 chars string.
Read char 1 for termination ON and 0 for termination OFF.
e.g. Pin 5 termination ON, others termination OFF: return ‘10000’.

dioReadValue(self)
Return the DIO pin values as a 5 chars string.
Read char 1 for HIGH and 0 for LOW.
e.g. Pin 5 HIGH, others LOW: return ‘10000’.

dioSetBottomLED(self)
Set to ON the DIO Front Panel Bottom LED.

dioSetOutput(self, value)
Set to 1 the selected DIO pins as a 5 chars string.
Write char 1 for selected and 0 for unchanged.
e.g. Pin 5 set, others unchanged: value = ‘10000’.

dioSetTopLED(self)
Set to ON the DIO Front Panel Top LED.

dioToggleBottomLED(self)
Toggle/change the DIO Front Panel Bottom LED.

dioToggleTopLED(self)
Toggle/change the DIO Front Panel Top LED.

dioWriteDirection(self, value)
Set the DIO pin directions as a 5 chars string.
Write char 1 for output and 0 for input.
e.g. Pin 5 output, others input: value = ’10000’.

dioWriteTermination(self, value)
Set the DIO pin termination resistor as a 5 chars string.
Write char 1 for termination ON and 0 for termination OFF.
e.g. Pin 5 termination ON, others termination OFF: value = ’10000’.

dioWriteValue(self, value)
Set the DIO pin values as a 5 chars string.
Write char 1 for HIGH and 0 for LOW.
e.g. Pin 5 HIGH, others LOW: value = ’10000’.