



The Abdus Salam
International Centre
for Theoretical Physics

First Steps with FreeRTOS

Advanced Workshop on FPGA-based Systems-On-Chip for
Scientific Instrumentation and Reconfigurable Computing

Trieste, 26st November-7 December 2018

Contents

Objectives	3
Building the system hardware	3
Questions.....	5
FreeRTOS BSP generation	6
FreeRTOS Application	8
Questions.....	8
FreeRTOS Application – part 2.....	9

Objectives

After the completion of the lab, you will be able to:

- Generate and configure a FreeRTOS BSP
- Write a simple FreeRTOS application that interacts with the buttons and leds in the board.

Building the system hardware

Before proceeding to the development of the FreeRTOS application, a previous step should be the definition of the hardware architecture of the system.

To illustrate the interaction between the operating system and the reconfigurable hardware in the PL, the architecture of the system will be the following one:

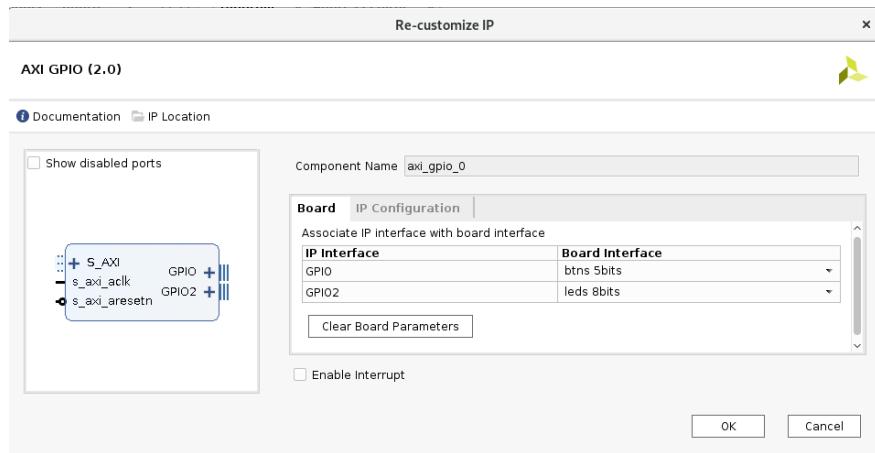
- In the PS side, the first ARM core will provide the execution environment for the operating system and the simple application to be developed in this lab.
- In the PL side an AXI_GPIO core will be used for the monitoring and control of the buttons and leds available in the Zedboard. This core will be connected to the PS side through one of the GP AXI buses.

The following steps describe the procedure for the creation of the hardware platform described in the Vivado environment.

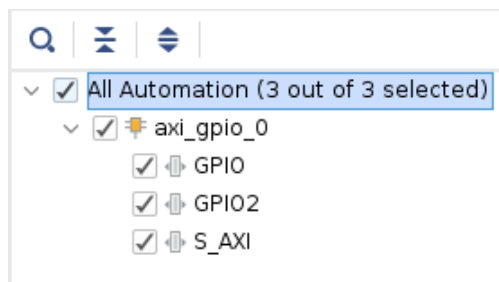
1. Run the Vivado tool and create an empty new project targeted to the Zedboard.
2. Once the project has been created, choose **Create Block Design** from the **IP INTEGRATOR** menu, in the left side of the tool.
3. Once in the *Diagram* window click on the “+” icon to *add an IP* to the empty schematic, and choose the **ZYNQ7 Processing System**, to include the block that corresponds to the PS part of the architecture¹.
4. Repeat the operation (**add IP**) but this time insert an AXI GPIO core.
5. Double-click on the AXI GPIO block to open the configuration window for the IP and instead of *custom* board interfaces for the two available GPIOs in the core:
 1. Select **btns 5bits** and **leds 8bits** for GPIO0 and GPIO2 respectively.
 2. Check in the *IP Configuration* tab that the sizes in bits of the GPIO buses have changed to 5 and 8, with is the number of buttons and

¹ Remember that the ARMs in the PS part are not reconfigurable cores, but already built-in as hard-cores. The purpose of the block included in the schematic is just to generate the appropriate configuration code for the final configuration selected of peripheral and buses. That code is automatically included in the BSP generated by the SDK.

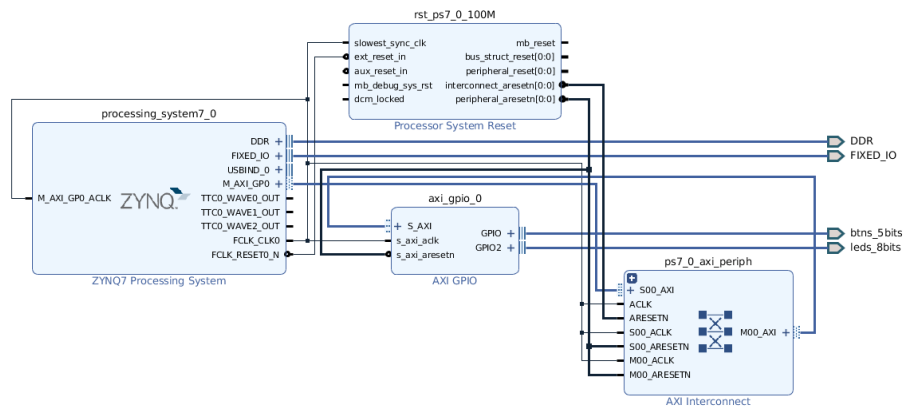
leds available in the Zedboard.



3. Finally click on **OK** to finish the configuration process.
6. After the insertion of the cores, a notification message should have appeared near the top of the *diagram* window, signaling the availability of the assistant for the automatic setup of the connections and configuration of the PS block
7. Click on the **Run Block Automation** first, and simply select OK with the defaults provided in the emerging window. This will autoconfigure the PS, and will connect the memory and fixed IO ports.
8. Now click on the **Run Connection Automation** message. And select the **All Automation** option in the left, which should automatically check the rest of the boxes in the hierarchy. Click then on **OK** which will produce the following effect:



1. Generate an AXI bus arbiter, and connect both the PS and GPIO IPs to the arbiter
2. Create two external ports for the **btns_5bits** and **led_8bits** signals, and connect them to the corresponding GPIO ports
9. The final architecture should be similar to the one depicted in the following figure:



10. Remember now to **Save** and then run **Tools**→**Validate Design** to check that everything is in place.
11. Click on the **Sources** tab of the *Block Design*, select the just captured design (generally *design_1.bd*), and **right-click** on it. Then Select **Create HDL Wrapper** to build the top level file of the project. You should see a design hierarchy such as the one following:



12. Now that the design is complete, run **PROGRAM DEBUG**→ **Generate Bitstream** to complete the back-end flow.
13. Once finished we just need to **Export**→**Export Hardware** (also selecting **include bitstream**) to generate the necessary files for the software development part of the project using the SDK

Questions

1. Which are the channels of the GPIO where both the leds and buttons have been connected?

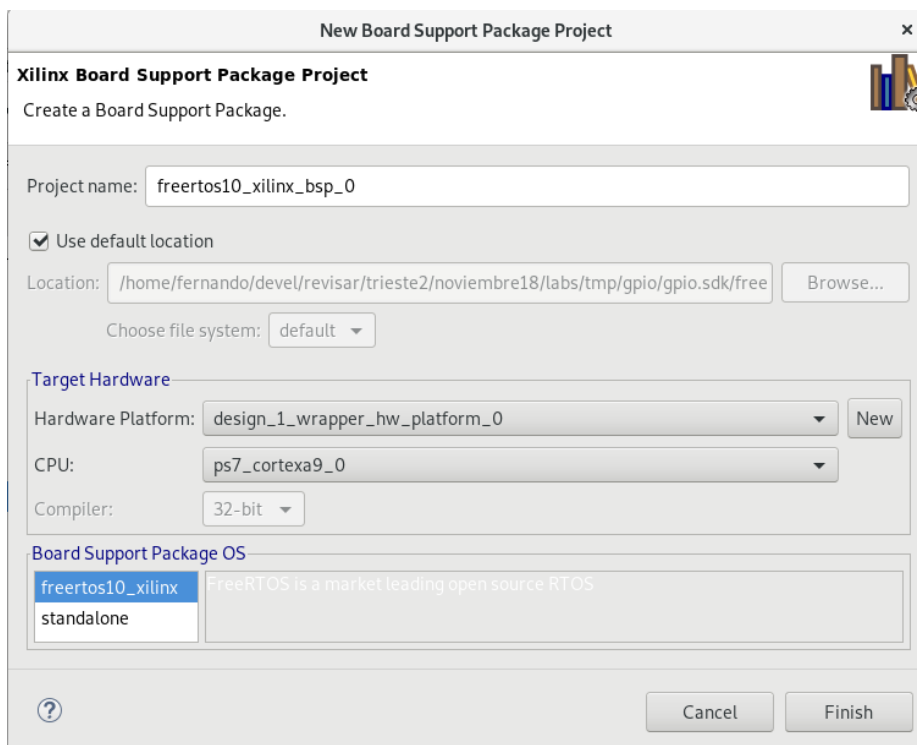
FreeRTOS BSP generation

Once the hardware platform has been generated, the rest of the development will continue in the SDK environment. You can launch SDK directly from the Vivado tool by executing File→Launch SDK. You will then be requested to select the *exported location* and *workspace*, which should be local to the project by default. Just click on OK, and the SDK will be launched.

During the first execution of the SDK, the exported hardware configuration will be detected, which will result in the generation of the platform configuration code. This code will later be regenerated if the original architecture is modified in Vivado.

Once the platform specification is available, it is the time to generate the FreeRTOS Board Support Package, upon which the FreeRTOS applications will later be developed. The BSP will provide the FreeRTOS run-time, as well as the Application Programming Interface (API) that the applications require to make of the services and resources provided by the Operating System:

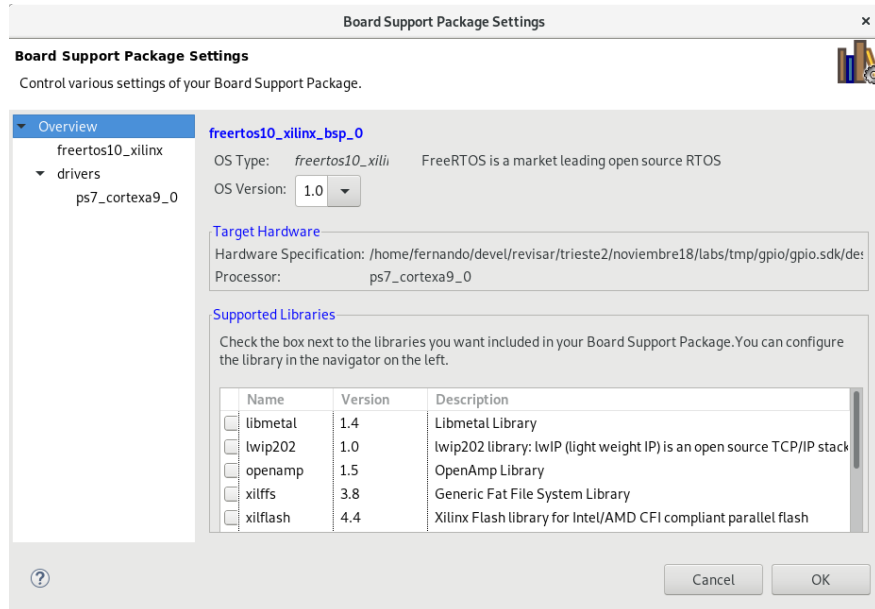
1. Click on **File→New→Board Support Package** and select **freertos10_xilinx** in the *Board Support Package OS* entry in the bottom left side of the resulting window. The 10 in the name refers to the FreeRTOS version, and may vary depending on the Vivado Tools version installed. Finally, click on Finish to generate the BSP.



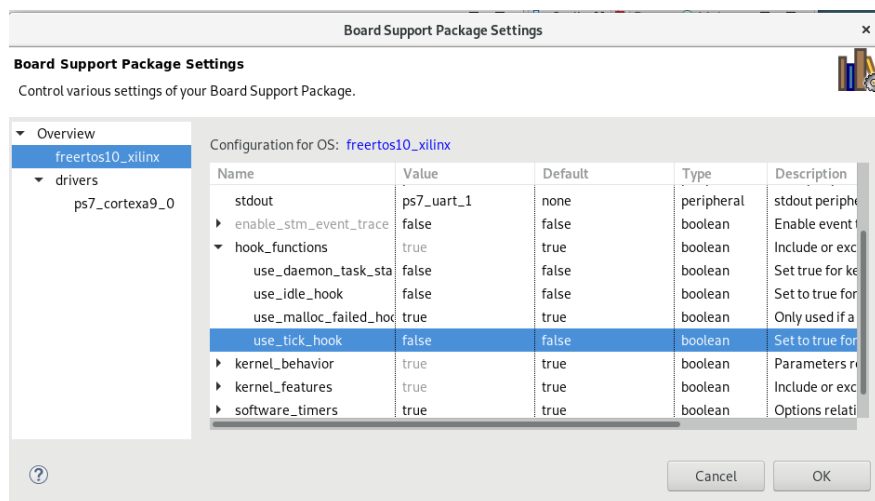
2. Once the generation has been completed, a new window will pop-up

showing the *Board Support Package Settings*. There are a number of issues that can be configured at this point:

1. In the *Overview* section you can find a number of *Supported Libraries* that can be added to the basic FreeRTOS kernel, such as a TCP/IP networking stack, a FAT filesystem, etc.



2. in the *freertos10_xilinx* section it is possible to tune the operation of the FreeRTOS itself. Most of these parameters will translate into the C code in file FreeRTOS_Config.h. Note how there is a category called *hook_functions* where it is possible to enable the activation or certain function calls every time an event takes place during the execution of the software. For example, if enabled, the *use_tick_hook* function will be called every time the OS timer generates a tick.



3. Leave the configuration by default and click on **Ok** to finish. You can later modify the settings or regenerate the code by selecting the *system.mss* file included in the BSP

FreeRTOS Application

After the generation of the BSP, we are ready to build the application program. The easiest procedure is to first generate the typical `hello_world` application and then customize it for the new purpose, which in this case it will be reading the state of the buttons and reflect their status using the leds in the board:

1. File→New→Application Project.
 1. *Project Name*: **led_buttons**
 2. *OS platform*: **freertos10_xilinx**
 3. *board support package*: **use existing freertos10_xilinx_bsp_0**. The one that we just generated in the previous step.
 4. Now click on **Next** to select the type of initial project to generate from a list of *Templates*. Here choose the **FreeRTOS Hello World** application.
 5. By clicking on **Finish** the application will be generated.
2. Run the project to check that the hello world application is working properly. This step does not require the configuration of the PL part, since it will just make use of the UART communication in the PS to show some text:
 1. Configure the SDK Terminal (find the tab in the lower part of the SDK), clicking on the “+” icon and choosing the right port and baud rate (115200 by default). This is necessary since the standard input and output of the project are mapped to the PS UART, and therefore we would need a serial terminal in the PC to have access to them.
 2. Select the application project (`led_buttons`) on the left side of the SDK (*Project Explorer*).
 3. Run→Run, and in the pop-up window select **Launch on Hardware**. After a few seconds you should see on the SDK Terminal ??

Questions

Proceed to the source code file of the application (*freertos_hello_world*), which you will find under the *led_buttons*→*src* hierarchy. Double-click on the file to open the contents. Have a look at the structure and then answer the following questions:

1. How many tasks are being created? What is the purpose of each task?
2. What is the queue used for?
3. Do both tasks have the same priority?
4. Which is the purpose of the timer? How many messages will be produced from the Tx to the Rx task during the execution of the application?
5. How does the application finish message transmission? Is it by canceling the timer?

FreeRTOS Application – part 2

Taking the generated code as the starting point, we'll proceed to the modification to reflect the following behavior:

- Every time the Tx task is enabled, it will read the status of the 5 buttons in the board. The resulting value will be send to the queue used for task communication.
- Every time the Rx task is enabled, it will pop a new value from the queue and display it though the activation of the appropriate leds. Since there are only 5 buttons available in the board, only 5 leds will be really used for the display.
- This behavior should be keep indefinitely.

These are the list of modifications required in order to reflect the above behavior:

1. Include the reference to the GPIO driver library at the top of the file:

```
#include "xgpio.h"
```

2. Locate the programmatic IDs for both the GPIO. This information is included in the *xparameters.h* file that was generated from the hardware platform configuration in Vivado, therefore, it reflects the actual structure of the hardware. To do so:
 1. navigate to the top of the file and locate the `#include "xparameter.h"` code line.
 2. Place the cursor above any part of the *xparamters* text, and hit F3. This will open the file in a different tab.
 3. Look for the `/* Definitions for driver GPIO */` section.
 4. Once there you should find a definitions similar to `XPAR_AXI_GPIO_0_DEVICE_ID`. Check it really matches that name. Otherwise take note of the real ones since it will be required in the next step.
 5. Back in the *freertos_hello_world* file, declare the variable **gpio** of type **XGpio**. This variable will held the reference to the GPIO driver. We

need the declaration to be a **global** one, so insert the following text just **before the main function**:

```
XGpio gpio;
```

6. Get into the main function and insert the following code to configure the GPIO peripheral:

```
XGpio_Initialize(&gpio, XPAR_AXI_GPIO_0_DEVICE_ID);
XGpio_SetDataDirection(&gpio, 1, 0xff);
XGpio_SetDataDirection(&gpio, 2, 0x00);
```

The *Initialize* function stores the reference to the peripheral in the *gpio* variable

The *SetDataDirection* configures the *gpio* peripheral in channel 1 (GPIO 0 in the Vivado core) as an input channel (0xFF sets a 1 for every direction bit, which implies being an input).

The *SetDataDirection* for channel 2 (GPIO2 in the Vivado core) is 0 since all leds are outputs.

7. The next step, also in the main function, will be modification of the queue to be created. In the hello world example, the queue was used to pass strings from the Tx to the Rx task. Now we only need to copy an integer. Therefore, look for the call to *xQueueCreate*, and replace it with:

```
xQueue = xQueueCreate( 1, sizeof( unsigned int ) );
```

Here the depth of the queue can be kept to 1, since each data produced is immediately consumed by the Rx task.

8. Let's switch to the *prvTxTask* function. Here, instead of a fixed string, we'll send the value obtained from reading the current status of the buttons. To do so:

1. include the following declaration before the *for* clause:

```
unsigned int value;
```

2. read the status of the buttons (inside the *for* clause):

```
value = XGpio_DiscreteRead(&gpio, 1);
```

3. and replace the *xQueueSend* call with this one:


```
xQueueSend( xQueue, &value, 0UL );
```


9. In the *prvRxTask* we should do the complimentary changes:

1. include again the declaration of *value*:

```
unsigned int value;
```

2. replace the *xQueueReceive* call with this one:

 Note that the *gpio* variable is always passed by reference (&).

 Notice *value* is again passed by reference.

```
xQueueReceive( xQueue, &value, portMAX_DELAY );
```

3. and finally write the value into the leds:

```
xQueueReceive( xQueue, &value, portMAX_DELAY );
```

10. The last modification will consist in removing the limitation of just 10 iterations of the producer/consumer tasks. Since this is performed through a callback fired by a timer created in the main function, the solution will consist in:
 1. The removal of the *xTimerCreate* function call in the *main* function, as well as the *assert* and *xTimerStart* function that immediately follow.
 2. The complete removal of the *vTimerCallBack* function at the end of the source code file.
11. One final step would be the removal of some unnecessary code that remains after the changes and removals, such as the declaration of the *HwString* or the *RxTaskCntr*. The warning icons on the source code editor will give you a clue on this.

Questions

1. You should observe that the response from the board is not immediate, but there is some delay from the button has been pressed until the value is reflected on the leds. Even the leds are not immediately turn off after releasing the button. Why do you think this is happening?
2. How can you fix such high latency between the pressing and the display?

Optional

Write a new task that should also consume the same data from the queue than the RxTask. This third task should print the value to the serial console (*xil_printf*).

Take into account that only one of the consumer tasks can really remove the value from the queue, otherwise the second consumer won't have access to it, since it would have disappeared. This can easily be done using the *xQueuePeek* function which is equivalent to the *xQueueReceive*, but just keeps a copy of the value without really removing it.

An additional issue that you should consider is that the task doing the peek should have higher priority than the one receiving, otherwise data can be consumed before than expected.

