José Antonio de la Torre las Heras
Universidad de Castilla-La Mancha

SMR3249

A PyQT GUI for a SoC design

Contents

Introduction

In this tutorial, you will learn how to communicate external devices like pmods sensors, and leds from Zedboard to a PC (frontend). In order to avoid privative software like: Matlab, Visual Basic, Labview... In this laboratory, we are going to use Python and different libraries which are completely free and open source. The advantages of using these kind of technologies are the following ones: you have control over all parts of your system and you don't need to trust in external companies and private design cycles.

In this project, you will learn how to integrate a fully functional project from a reconfigurable part to high level programming in Python. The main objective of the project is that the student, at the end, understands how to integrate all the parts to create a final product.

Objectives

- To design a fully functional GUI (Graphical User Interface)
- To create a controller of the GUI
- To connect a Zedboard to the GUI
- To control a Zedboard from the GUI
- To get data and plot from Zedboard

Procedure

This project has different parts. In the first part, the student will create a Vivado project and configure a Zedboard to use a GPIOs (General Purpose Input Outputs) and a PMOD temperature sensor. In the second part, a SDK project will be created in order to obtain temperature values. During the third, very simple, but functional, serial protocol will be added to send data to Zedboard and control it from a PC. Once it has been completed, the student will move on to a completely different set of tools to create a GUI in QT. That part only puts controls in a window, but it will be in the fifth part when the GUI will be alive. In fifth part, the main objective is to program a controller in Python language and send values to the board in order to turn on and off the leds. Finally, in the last part, the plot will be added in order to plot data that comes from the board.

Design flow

In Figure 1 it is depicted a general architecture of the project.

As it can be seen, data will be read from a PMOD sensor using SPI. In order to avoid some problems with embedded SPI controller inside the PS, we will use a SPI IP controller inside PL. As an advanced exercise the student could extend the project to use a internal SPI inside the PS.

Once in the PL, it is necessary to setup several parameters of the SPI controller like in any classic microcontroller design. Also, will be necessary to "talk" directly with the PMOD sensor in order to obtain the temperature. Finally, in the GUI located in a PC, the Python controller of the GUI will read the data from the PL and plot it in a graph. Moreover, in order to see how to communicate in both ways, the GPIOs configured in the Vivado project will be written from the PC.

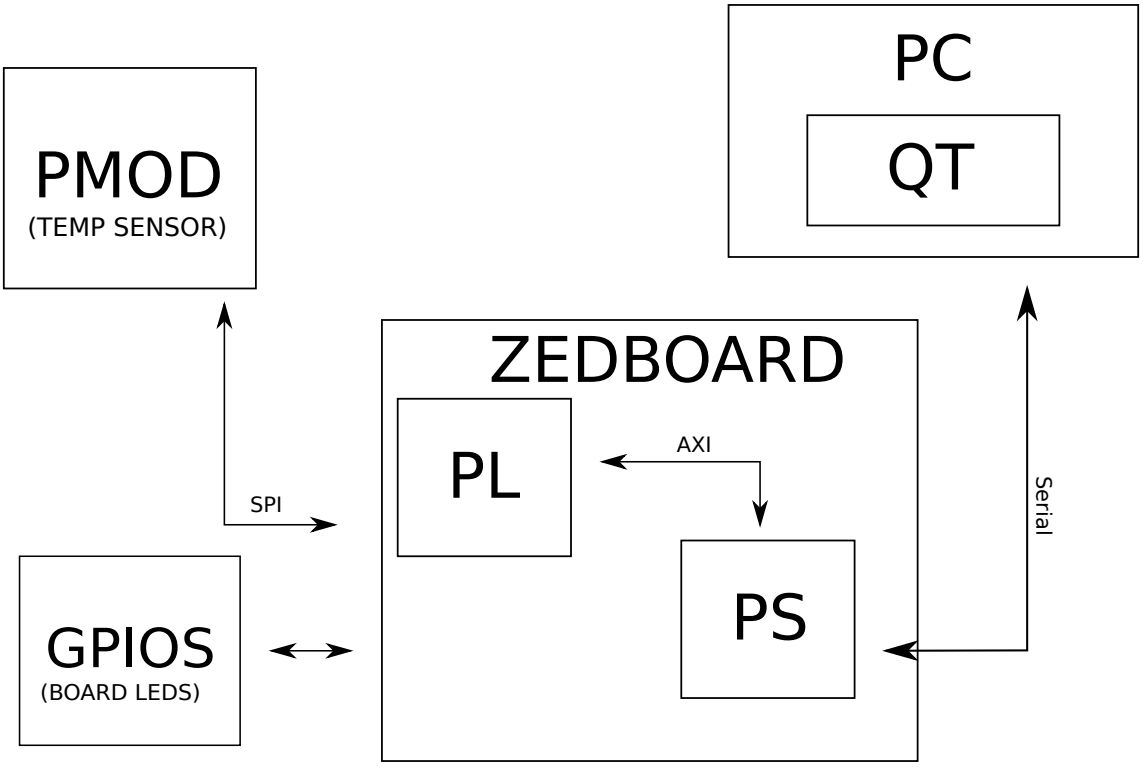

Figure 1: General architecture of lab

1 Vivado and SDK

First of all, this laboratory will start from a Vivado project already created with all the parts needed in order to communicate with the PMOD temperature sensor and with the GPIOs. It is supposed that the student already knows the basic architecture of this type of Vivado project.

The project can be found in the shared folder. It is important to move the project to a folder in which the user has permission for reading it and writing it. Normally the desktop is enough.

Once the project has been copied, open it with Vivado. In windows you should double click over the .xpr file and Vivado will appears. If for some reason, it does not work, open Vivado and click in the menu "File → Project → open" (Figure 2) and select the TutorialTempLedPyQt.xpr file.

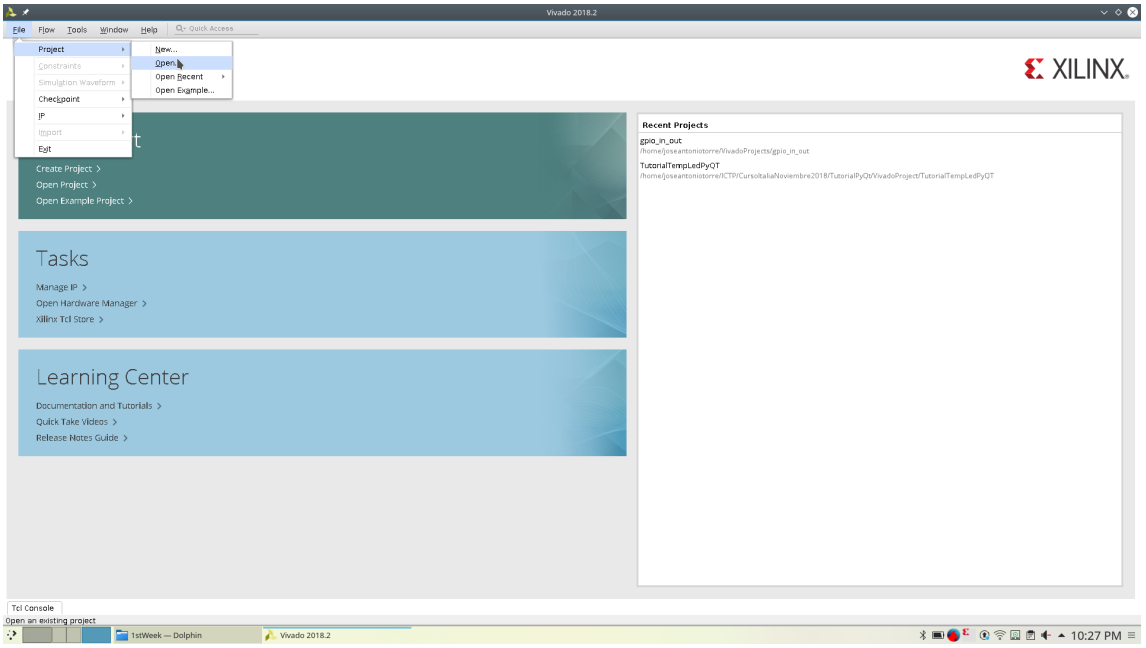

Figure 2: Opening a project in Vivado

When Vivado opens the project you should see something similar to Figure 3. Please, note that paths could change, it doesn't matter, it is only important that the content is the same. As you can see in the upper right corner, the bitsteam has been already generated for you. It is not necessary to generate it again but if you feel ready you can try!.

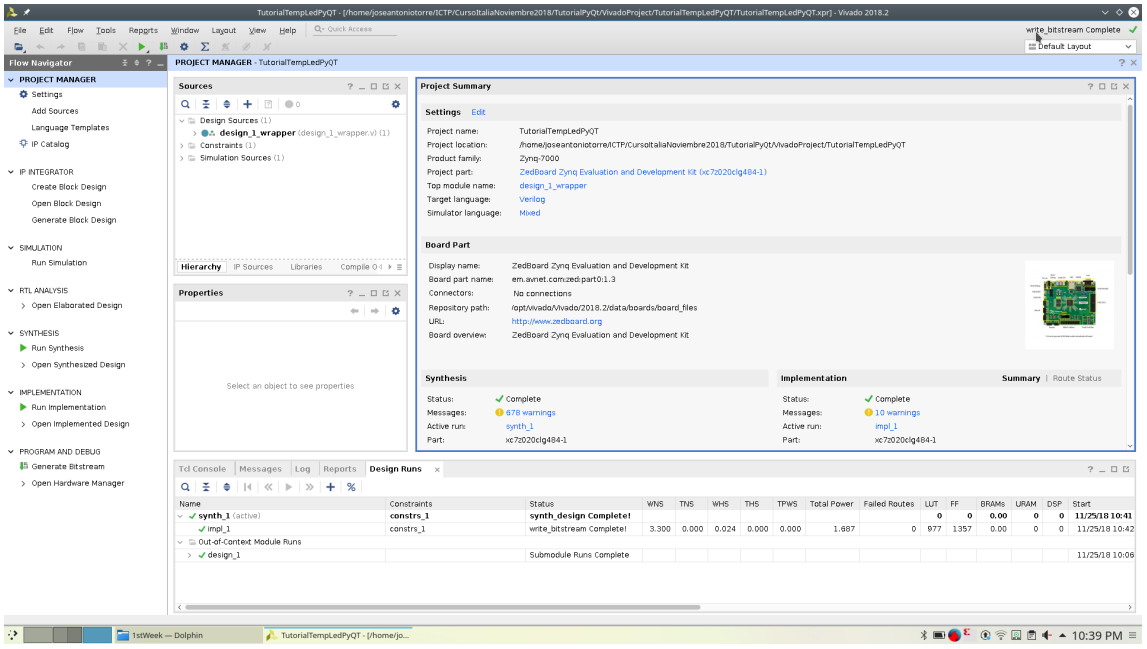

Figure 3: Vivado project

In the Figure 5 you can see how the project looks like. To open the block design you should click over “Open block design” in the “IP integrator” submenu, as it is depicted in Figure 4

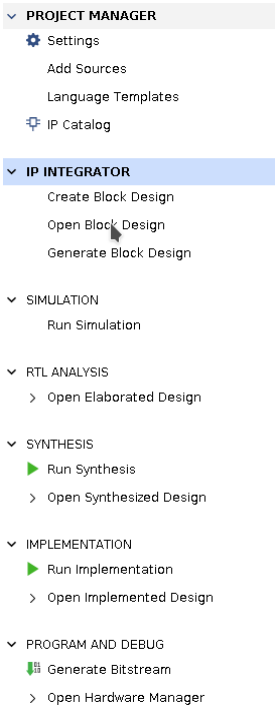

Figure 4: Opening a block design

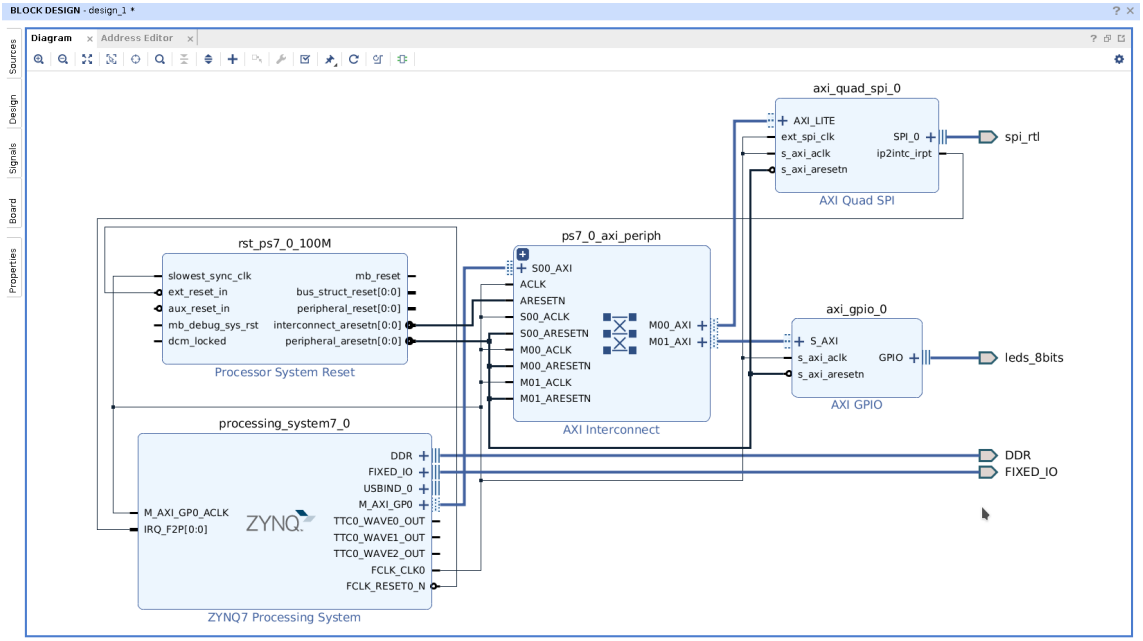

Figure 5: Block Design

The project is very simple and the block design too. It is composed of 3 main IP cores:

- axi_quad_spi: This block is the manager of the SPI protocol. PS part will use that in order to communicate with the temperature sensor.
- axi_gpio_0: The GPIO controller has the responsability to control the GPIO associated to the boards leds. In general that, GPIO controller could be mapped to any pin inside the board but in this case it is used to board leds.
- processing_system7_0: Processing system or PS is the heart of the application. It has the responsability to understand packets coming from the pc and perform control the GPIOs and PMODs to reply those packets.

Once you have understand the design, let us move to the next step that normally should be create the bitstream but, in this case, it is not necessary because it has been already created for you in order to reduce the time needed to start working in the project.

Click on menu "File → Launch SDK" (Figure 6. Once opened it should looks similar to Figure 7

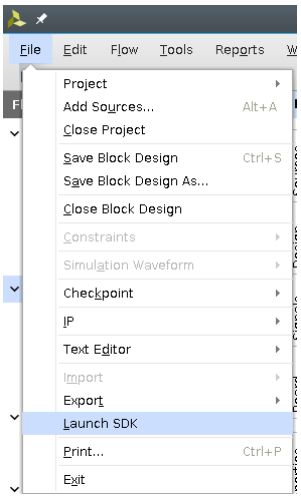

Figure 6: Open SDK

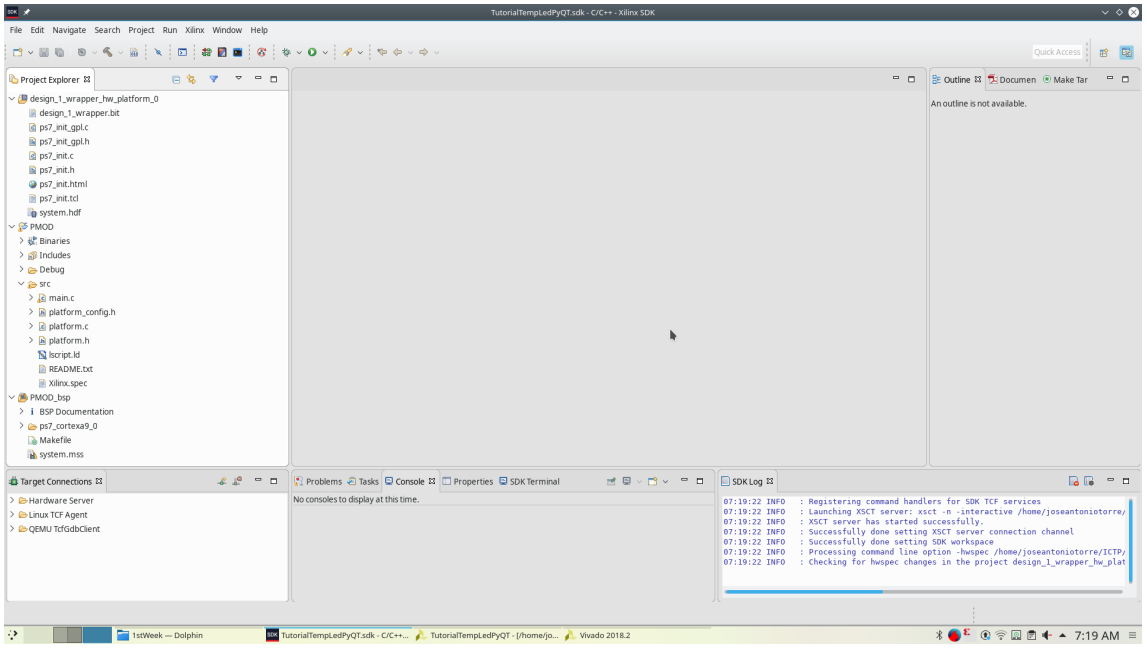

Figure 7: SDK Opened

In the left part of the SDK you can see the main projects included in the workspace. If in your SDK you don't see the full list, click over the arrow in order to extend the tree.

The main projects are:

1. design_1_wrapper_hw_platform_0: This project is generated by Vivado when hardware is exported. In this project, you have not exported because it has been done for you before. In the tree you will find:
   - design_1_wrapper.bit: Bitstream used to configure FPGA part.
   - ps7_init_gpl.c ps7_init_gpl.h ps7_init ps7_init.h: This code is responsible of starting ARM processing system before execute the main program. You should not edit these files.
   - ps7_init.html: Summary of resources loaded in PS part.
   - ps7_init.tcl: Exactly the same as ps7_init.c but in tcl scripting language.
   - system.hdf: Hardware Definition File, in this file you can see all IPs loaded in your design and the registers they have.
2. PMOD: Main project, this will be the project in which you will be working. It has the regular files in this type of projects. You will use only main.c. In that file you will find all the necessary parts to start working in the project. Take a look before going to the next step in order to understand how it is developed.
3. PMOD_BSP: This project it is generated by Vivado in order to create necessary drivers to each one of the cores loaded in the design.

2 main.c

As you can note, main.c is the principal source code of this project. You should open it and see what is inside. In summary, these are the most important functions implemented:

- check_communication: It reads two characters from serial device and returns one if the pc wants to receive data or two if pc wants to write data. The protocol designed to this project is too easy. In order to get the data, PS should read "gd", and to set data, it should read "sd".
- set_gpios: It reads 3 characters from serial device and converts them in to a int, then it writes in the GPIO controller.
- send_temperature: It sends temperature read from SPI temperature sensor in format 2.3f that means 3 decimals and 2 integers.
- main.c: It is developed to configure each core and start a while loop in which it reads serial communication and send or received the data.

Take some time to understand the code and ask all questions you have in this step.

3 Opening development environment

In order to get all the software ready to be used and also with the intention to use only free software, a Debian virtual machine has been created and placed in C:\SW2018\PyQTDebian . This machine has mainly the following software:

- Debian Stretch OS: A free operating system.
- Python 3: Last version of Python.
- PyQT4: Python binding to Qt.
- (Recommended) Visual Code: Free text editor that simplify the work of developing python code.
- PyCharm: Another text editor of python.

All those programs could be installed in any operating system like MAC or Windows but it won't be covered within this laboratory.

To open this machine you will need two programs:

1. Virtualbox: Free software to virtualize machines.
2. Virtualbox Extension Pack: Free of charge program installed in top of virtualbox that brings USB support to virtual machines.

Again, those programs are already installed in lab's computers so you don't need to worry.

Once you have located the virtualbox, you can open it by clicking over the blue icon as showed in Figure 8. When virtualbox appears, right-click over PyQtDebian and then start and normal start.

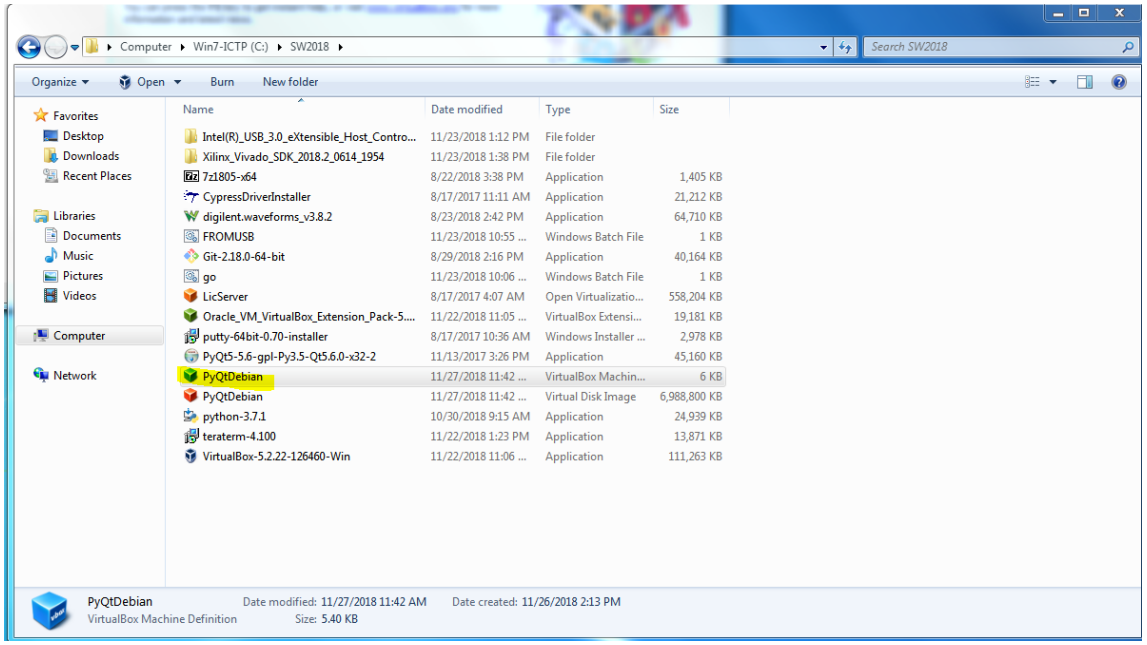

Figure 8: Virtual machine

It may take some time but at the end a login screen will appear (Figure 9). There you should enter following user and password:

- user: ictp
- password: ictp

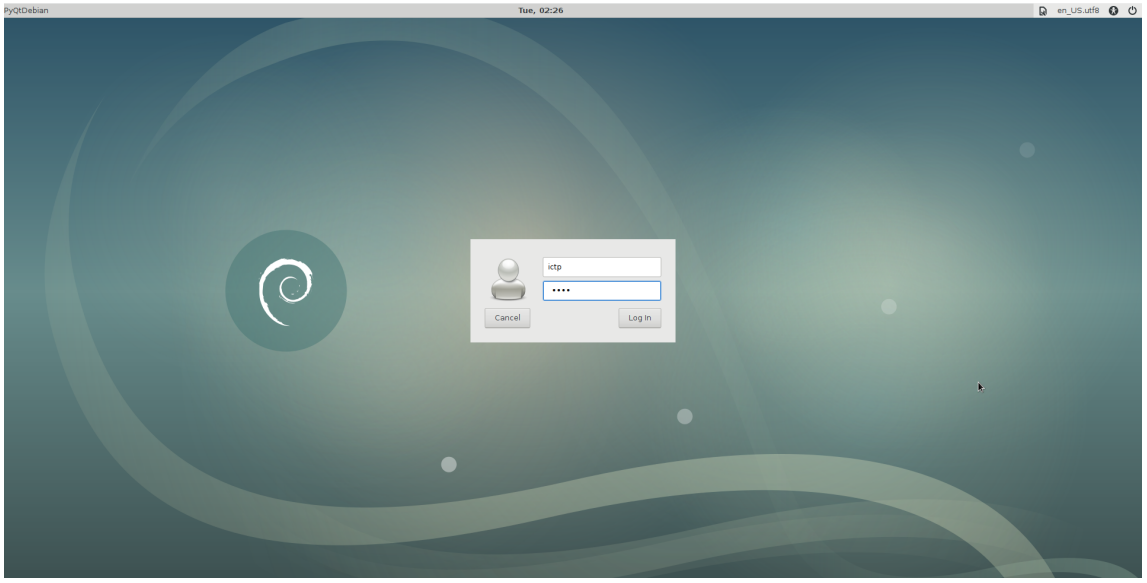

Figure 9: Login

At this point, you have completed this step. If you have time, try to explore the virtual machine to familiarize yourself with it.

4 Designing the interface

Designing a good interface is a hard process. Normally, there are a lot of professionals working in that in order to make a fully responsive interface. The objective of this laboratory is not to make the best interface, but to create a functional one with the latest technology in the area.

Qt is one of the most used technology in the world of interface development. GTK is another alternative but in the last years Qt has advanced in embedded design and companies like Tesla has adopted it.

For the design of a typical interface, some steps should be carried out. Following there is a list summarizing each one:

- Mockup: Typically, it involves draw some drafts in order to have and idea of the final window.
- Design with a WYSIWYG (What you get is what you see): The output of this project is normally a HTML or XML describing how the computer should compose the interface. In this laboratory we will use a QTDesigner that it is completely free and easy to use.
- Create a controller: WYSIWYG normally outputs a description of the window but it can not be used to make any action. In order to make it alive it is necessary to program a controller. The controller role is to connect each button, textbox and, in general, each control with a function in Python, C, C++, Java... In this laboratory we will use Python.

So, in order to follow those steps and ignoring the mockup, we will start designing the interface.

4.1 Design in QtDesigner

First of all, you need to open QtDesigner. It should be in desktop, so click over it. Once opened, you should see something similar to Figure 10. Please, select widget and default size and then click create. At the end, you should have a clean window ready to be design.

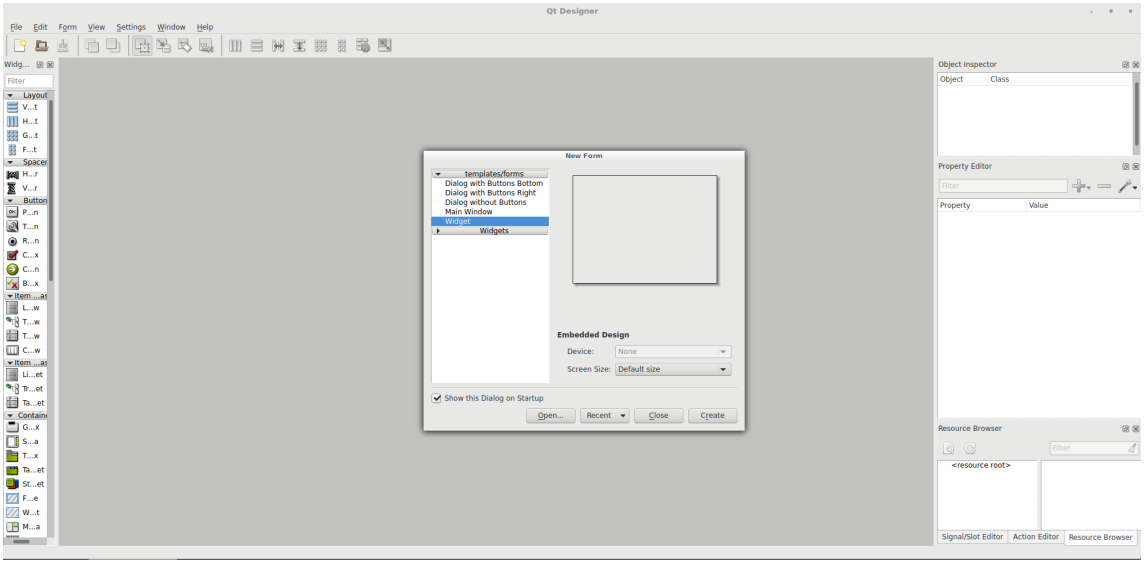

Figure 10: Qt Designer

In Figure 11 and Figure 12 you can see how the final interface should looks like.

Figure 11: First tab of final interface

Figure 12: Second tab of final interface

As you can see in Figure 11 the interface has tabs. These tabs allow us to change to different "subwindows" and separate each part. In the first tab we will control the leds and in the second one, we will plot data coming from the PMOD.

In order to create the tabs, go to left part of qtdesign and then, search for "tab" in the search field (Figure 13). Once you have found it, move it to the canvas as shown in Figure 13. As can be shown in that figure, it is possible to resize the tabs but it has the problem that it will have a fixed size and if you resize the window it will no grow. To obtain that behavior, in Qt, you must fix a layout. To do that click over a blank area and click over layouts and then layout horizontally (Figure 14).

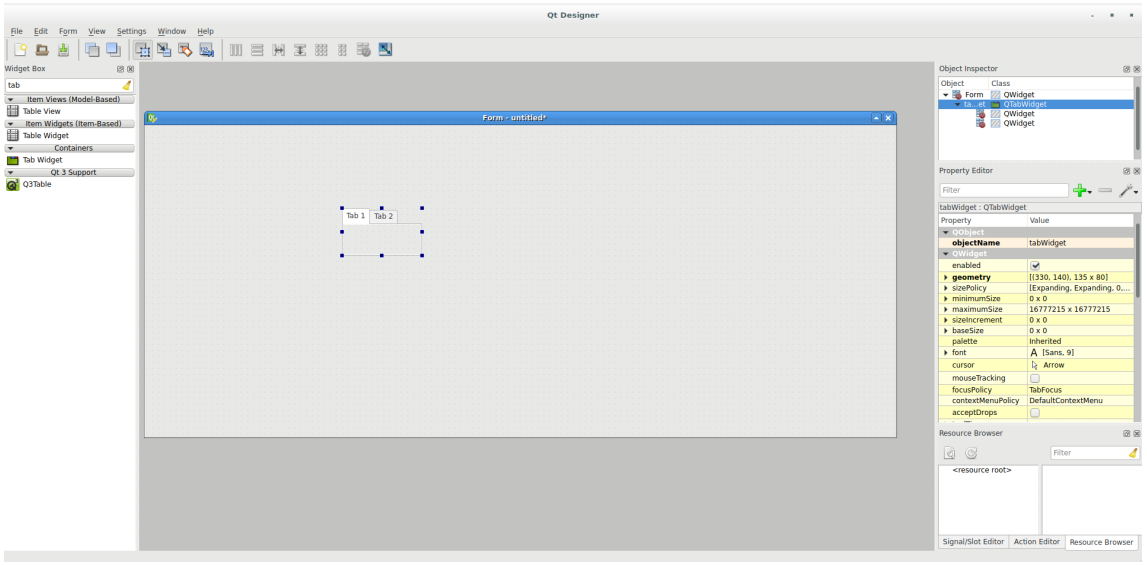

Figure 13: Searching a control in qt designer

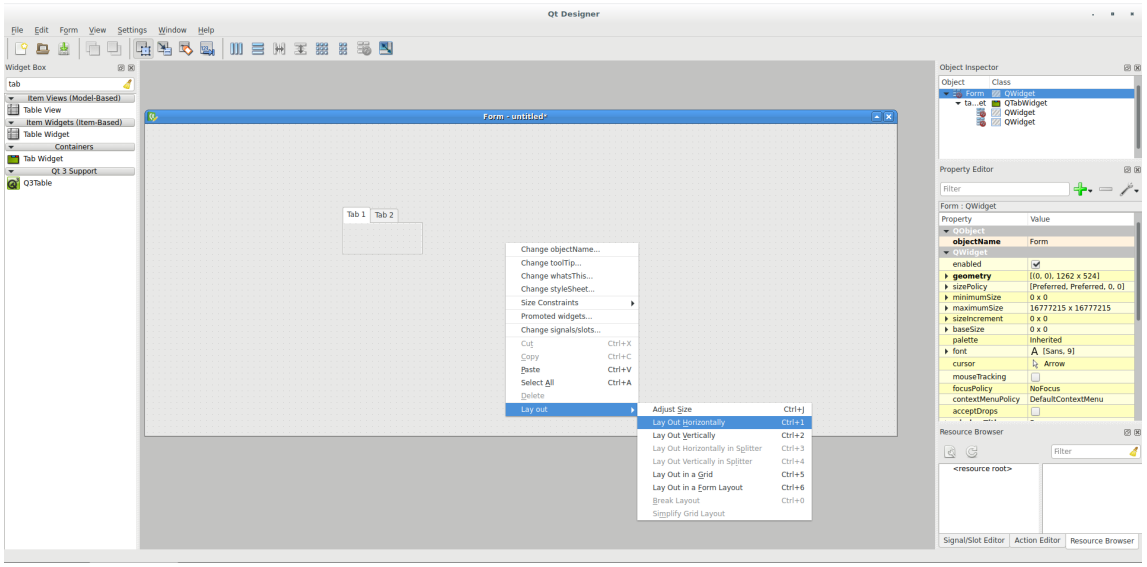

Figure 14: Making responsive

Now, inside the first tab we will put eight checkbox that will turn on/off each led in the board. Again, search in the left panel as in Figure 13 but in this case, search for a “radio” control. Then, move to the canvas and click over it. When clicked in the lower right window you will see a properties window like one in Figure 15. Search for text property and remove it.

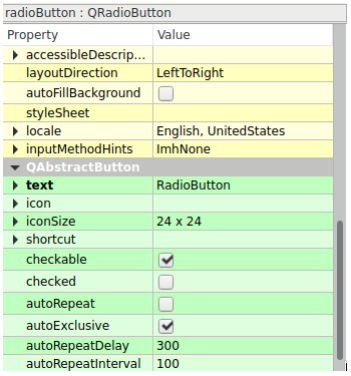

Figure 15: Changing properties

Now, in the same window, search for an "autoexclusive" property. That property makes the radio buttons exclusive. That means, that you can not select more than one at the same time. When properties have been selected click over the radio button and copy and paste until you have eight buttons. You can use Figure 11 as example of the final application but you can arrange it as you want.

Once you have the buttons prepared it is **very important** to change the object name in order to call it from the Python Application. To do that go click over each button and in properties windows search for "objectName" property. This property will be used in python code. In order to make it easier, to help you in case of problems, please put "led1_2" to the left led, "led2_2" to right one and so on.

At the end, you should have something similar to Figure 16.

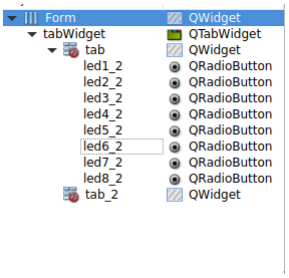

Figure 16: Object inspector

Now, with the same procedure you can create following two components:

- QPushButton: Change the text with a "SEND BUTTON" text and "objectName" to sendButton.
- QTextButton: Change the "objectName" to ledInformation.

At the end, you should have something similar to Figure 17.

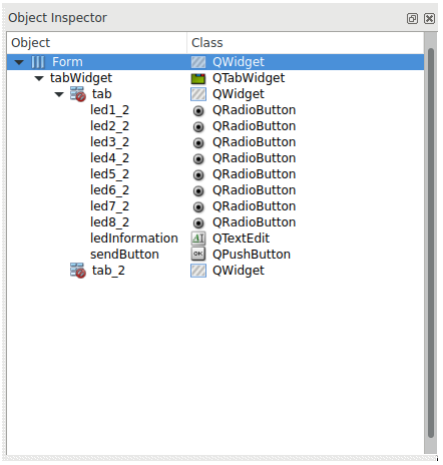

Figure 17: Object inspector of first tab

Now, when the first tab has been completed, we will move to the next one. In that tab, we will plot data coming from the zedboard.

In order to plot data in PyQt4, you should use an external library or make all by yourself. In this case, we have chosen PyQtGraph. PyQtGraph is a very powerful library that has GPU acceleration and a lot of features like FFT already ready to use.

As we have told, PyQtGraph is an external library, that means that it will not appear in QtDesigner as a control and also, we will need to make some hacks in order to integrate it.

First of all, let us move to next tab. Click over second tab and you should see another canvas (or more precisely a QWidget). In this canvas, we will design another interface similar to Figure 12.

At this point, you should know how to move controls and change their object name and text. In the Figure 18 you have a summary of the primary components you shall create. Please, try to hold same “object names”.

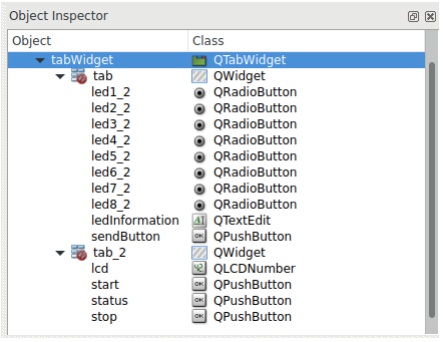

Figure 18: Object inspector of second tab

Once you have completed it, you are ready to prepare the PyQtGraph.

As it has been told before, PyQtGraph is not a standard library so we will make some hacks. First, move a “Graphics View” and change “objectName” to “graphComponent”. Then, right click over it and select “Promote to” option as show in Figure 19.

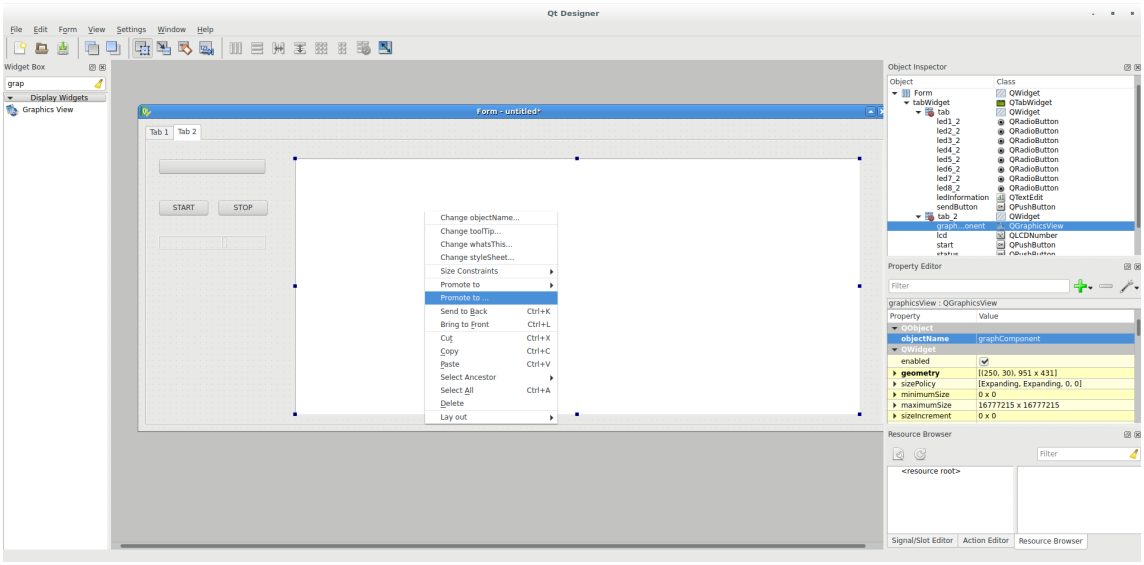

Figure 19: Promoting graphics to plot

Fill the modal dialog as shown in Figure 20. These values come from PyQtGraph documentation,so, feel free to go and search it. When filled, click over “add” and later over “promote”.

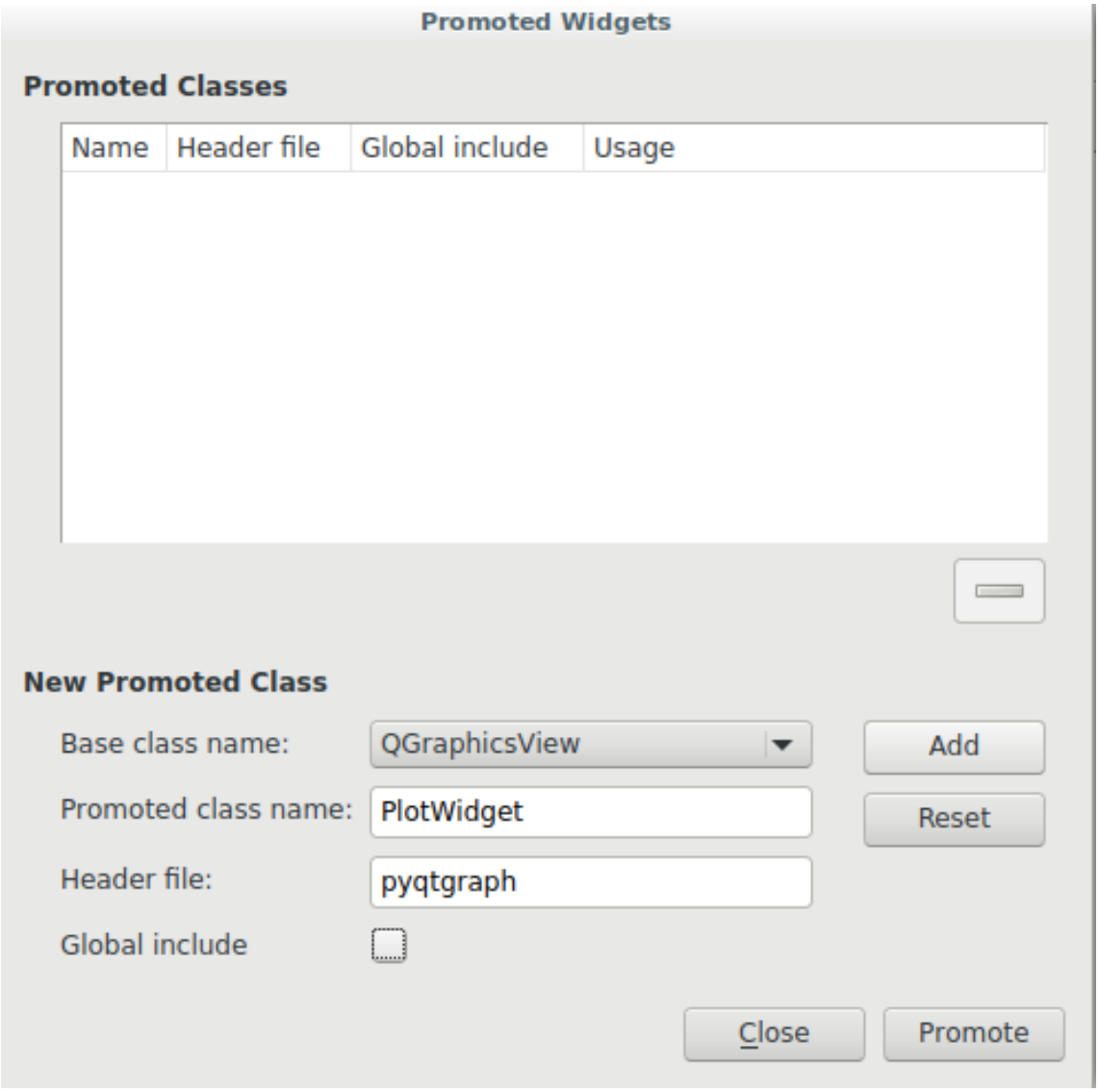

Figure 20: Values to promote

Now, you have completed the design of the interface.

4.2 Exporting the interface

Until now, we have design the interface in a WYGIWYS program but python can not understand it. In order to prepare the Interface for Python, we should convert that controls to an intermediate language. Qt has chosen XML as a definition language but other frameworks could use other languages.

To export the interface go to File\Save and type “interface.ui”. In order to simplify next steps save it in a “home” folder as shown in Figure 21

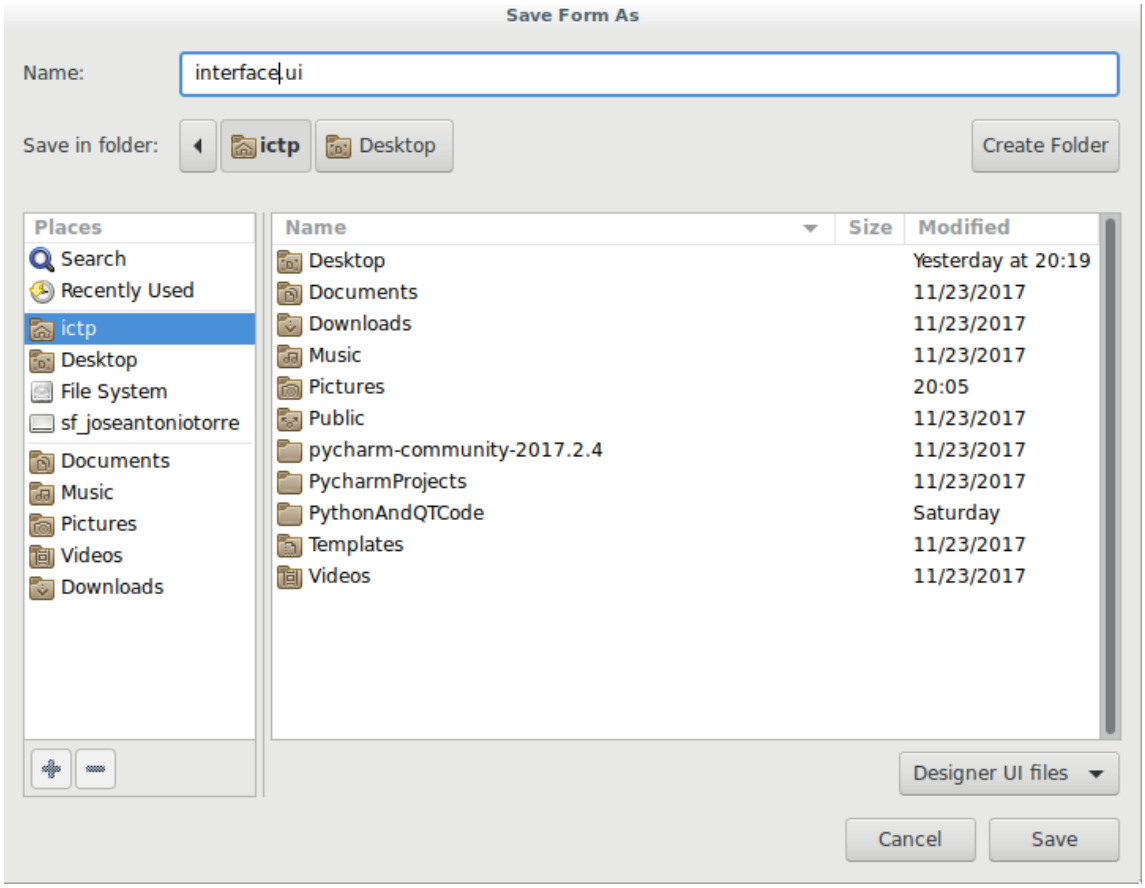

Figure 21: Exporting interface

Now, you are ready to go to python and start working in the controller.

5 Programming the controller

At this point we have an interface exported in XML format (with .ui extension) and we are ready to work with Python.

PyQt comes with a utility that converts XML code to Python code. To make this step, open a terminal and type "pyuic4 -x interfaz.ui -o interfaz.py". pyuic4 is the compiler that takes XML code and generate python equivalent code. "-x" is an option to make it executable, "interfaz.ui" has to be the same the one you saved from qtdesigner. Finally, "-o" is an option that has an argument to tell the compiler which is the name and the location of the generated code.

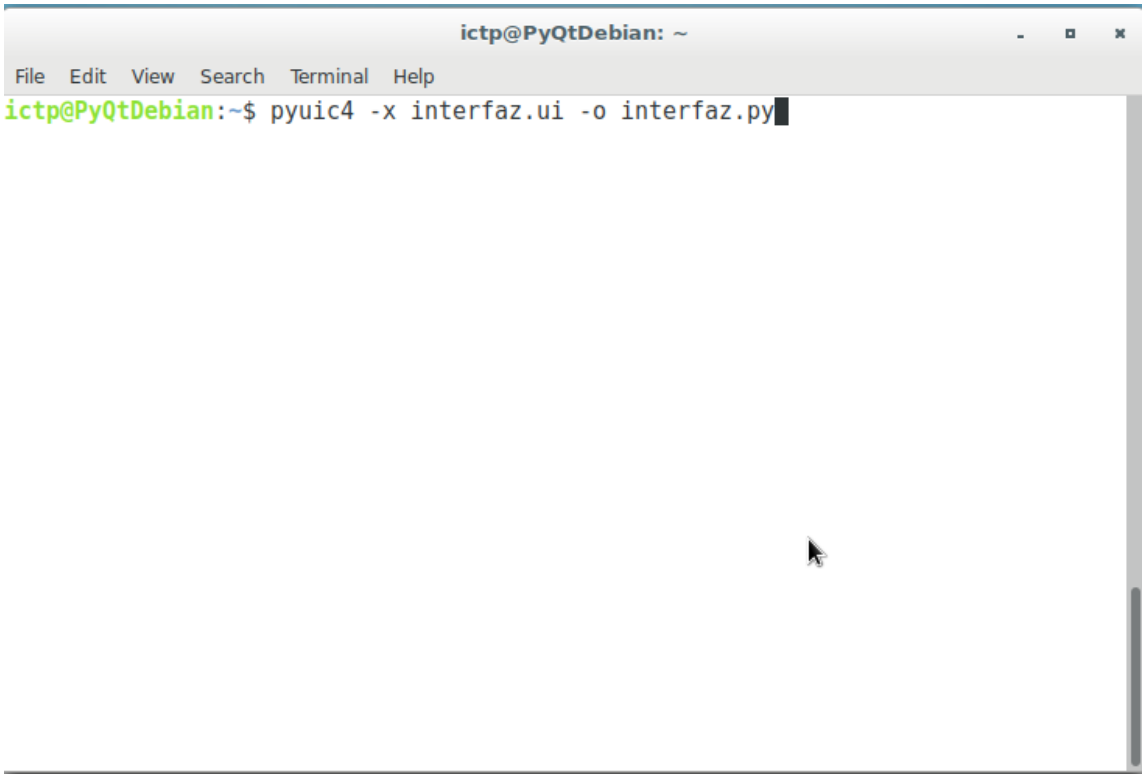

Figure 22: Compile interface

In order to test it, once you have generated python code, type "python3 interfaz.py" and you should see the interface.

Looking back, until now, we have completed following items:

- Design interface in WYGIWYG program.
- Export the interface to .xml.
- Compile the interface to python code.

If you open the python interface with any editor installed in the virtual machine you will see something similar to the Listing 1

Source Code 1: Interface code

```
1  try:
2      _fromUtf8 = QtCore.QString.fromUtf8
3  except AttributeError:
4      def _fromUtf8(s):
5          return s
6
7  try:
8      _encoding = QtGui.QApplication.UnicodeUTF8
9      def _translate(context, text, disambig):
```

```
10          return QtGui.QApplication.translate(context, text, disambig, _encoding)
11  except AttributeError:
12      def _translate(context, text, disambig):
13          return QtGui.QApplication.translate(context, text, disambig)
14
15  class Ui_Dialog(object):
16      def setupUi(self, Dialog):
17          Dialog.setObjectName(_fromUtf8("Dialog"))
18          Dialog.resize(1207, 519)
19          sizePolicy = QtGui.QSizePolicy(QtGui.QSizePolicy.Expanding, QtGui.QSizePolicy.Expanding)
20          sizePolicy.setHorizontalStretch(0)
21          sizePolicy.setVerticalStretch(0)
22          sizePolicy.setHeightForWidth(Dialog.sizePolicy().hasHeightForWidth())
23          Dialog.setSizePolicy(sizePolicy)
24          self.horizontalLayout_3 = QtGui.QHBoxLayout(Dialog)
25          self.horizontalLayout_3.setObjectName(_fromUtf8("horizontalLayout_3"))
26          self.ledTab = QtGui.QTabWidget(Dialog)
27          self.ledTab.setObjectName(_fromUtf8("ledTab"))
28          self.tab = QtGui.QWidget()
29          self.tab.setObjectName(_fromUtf8("tab"))
30          self.horizontalLayout_4 = QtGui.QHBoxLayout(self.tab)
```

All of that code has been already generated by the pyqt compiler. In other scenarios or frameworks where there is not any compiler or designer tool you must do it by yourself.

If you see the Code 1 you could recognize, in line 15 the definition of a class. The name of the class is very important because we will use it in the next steps. Please, if you have other name, note it and in following steps use that.

Now that you have verified that the interface looks as expected it, it is time to make it alive with some logic. Before doing that it is important to note how each piece of code fits together to understand what we will do.

In the Figure 23 you can see how the application architecture looks like.

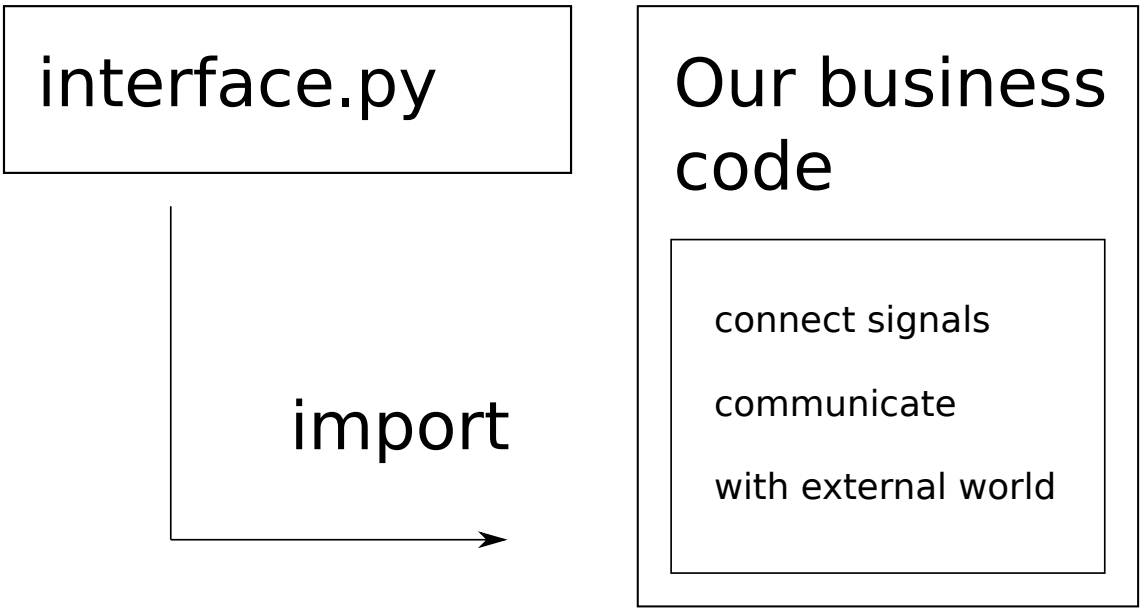

Figure 23: Software architecture

As you can see, interface.py, that is the code listed in Code 1 is imported in "our business code". The name of the business code could be whatever you want, for example app.py. That code is responsible of launch the interface, connect signals or callbacks to each button and, in general, controls. And finally connect with external world.

Now, that you have an idea of the architecture, we are ready for starting coding the controller of business code. First, go to a terminal (in desktop you can find a shortcut) and open it. Navigate to the directory where you save the files. To move to a directory you can use "cd" command. If you need to show files in a directory use "ls" command. When you are finally in the directory where files are located, create a new file called app.py. In order to create a file and open with an editor, you can use following instruction: "code app.py".

When a visual code window (Figure 24) appears you are ready to code. In order to make it easier you can copy and paste the code needed to make the interface works. The code is located in a shared folder and you will have it written in a blackboard. The file is called "helpCode.py"

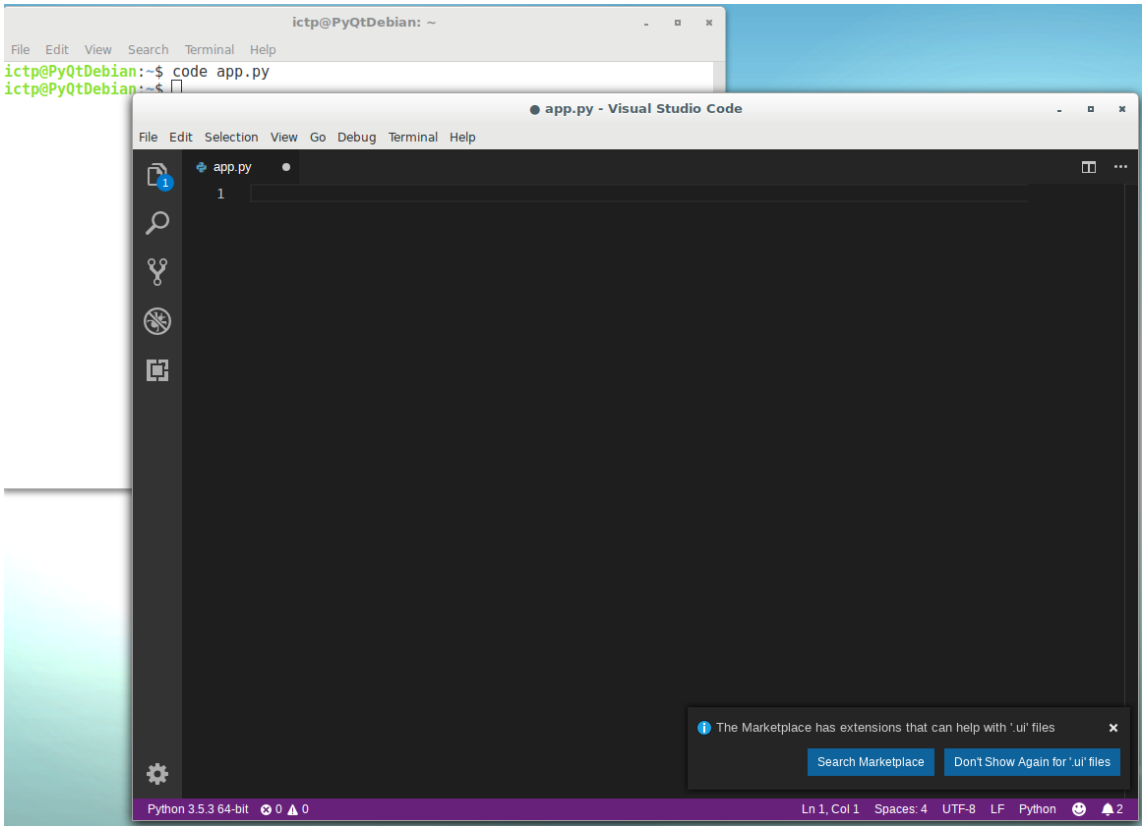

Figure 24: Window editor

In the following paragraphs the code we will explained. As you read the code you will see some "?" in that lines you will need to complete it with no more than 4 lines.

The first part to start with, it is the class called ZedboardController. That class is responsible of opening a serial connection when it is created and set the GPIOS and read the temperature from that serial port. In Code 2 you can see that class.

Source Code 2: Zed controller code

```
1  class ZedboardController:
2      def __init__(self, com_port):
3          self.com_port = com_port
4          self.transceiver = serial.Serial(com_port, baudrate=115200)
5
6      def setGPIOS(self, value):
7          str_value = format(value, '03d')
8          self.transceiver.write('sd{}\n'.format(str_value).encode())
9          self.transceiver.flushOutput()
10
11     def read_temperature_sensor(self, num_samples):
12         to_return = []
```

```
13          for i in range(num_samples):
14              self.transceiver.write("gd\n".encode())
15              self.transceiver.flushOutput()
16              data = self.transceiver.read(6)
17              to_return.append(float(data.decode()))
18
19          return to_return
```

ZedboardController is already done but you can read it in order to understand how it is programmed. Basically, it receives a name of the serial port and then it creates a new "transceiver" with PySerial library, as you can see in line 4. Then, it has two functions, setGPIOS that sends "sdXXX" in order to set leds to represent XXX number. And read_temperature_sensor that read a data from a serial transceiver and then converts it to float number.

Now we need to write the code of the controller. The controller can be shown in Code 3

Source Code 3: Controller code

```
1  class Controller:
2      def __init__(self, interface):
3          self.interface = interface
4          self.zedboard = ZedboardController('/dev/ttyACM0')
5          self.connect_signals_to_methods()
6          self.dataY = np.zeros(500)
7          self.line = self.interface.graphComponent.plot()
8
9      def connect_signals_to_methods(self):
10         self.interface.sendButton.clicked.connect(self.led_button_clicked)
11         # ?? connect start button clicked signal
12         # ?? connect stop button clicked signal
13
14     def led_button_clicked(self):
15         leds = [self.interface.led8_2,
16                 self.interface.led7_2,
17                 self.interface.led6_2,
18                 self.interface.led5_2,
19                 self.interface.led4_2,
20                 self.interface.led3_2,
21                 self.interface.led2_2,
22                 self.interface.led1_2]
23         position = 0
24         result = 0
25         for led in leds:
26             if led.isChecked():
27                         # Calculate result of binary representation of leds
28                         # for example leds in off, off, off, off, off, off, one,one should have 3 as result
29                         # Save the variable called result
30
31             position += 1
32
33         self.interface.ledInformation.append('<h1>Leds changed: {}</h1>'.format(result))
34
```

```
35          self.zedboard.setGPIOS(result)
36
37      def start_button_clicked(self):
38          self.interface.status.setStyleSheet("background-color: green")
39          self.timer = QtCore.QTimer()
40          self.timer.timeout.connect(self.timeout_reached)
41          self.timer.start(33)
42
43      def stop_button_clicked(self):
44          if self.timer:
45              self.interface.status.setStyleSheet("background-color: red")
46              self.timer.stop()
47
48      def timeout_reached(self):
49          received = self.zedboard.read_temperature_sensor(1)
50          self.dataY[:-1] = self.dataY[1:]
51          self.dataY[-1] = received[0]
52          self.line.setData(self.dataY)
53          self.interface.lcd.display(received[0])
```

The controller class has the responsability to connect each control with the functionality expected to each one. The first method, "__init__", is the constructor. That method is called each time a controller is created. In this method we create a ZedboardController that we have explained before, we connect each control with a function to be used when an action is performed (connect_signals_to_methods()) and we prepare the plot in order to later send information.

In the method "connect_signals_to_methods" you must write two lines of code in order to connect other button signals. As an example "sendButton" has been already connected to a "led_button_clicked" function. In that function we check each value of each led "RadioButton". At the end, we should have a number represented in the binary form of the leds. In this code you should complete code inside if in order to get the number represented in leds.

Finally, there are two main buttons one to start a timer that, when ready, it will get data from Zedboard and another one to stop that button.

To complete the project, we should start the interface and install the controller to it. To do that, in python, there is a magical variable called "__main__". these variables will have the value "__main__" if the code is called from a command line or an icon. In other cases, it will have the value of the file. For example, when we import in first lines the interface, "__name__" will have the value "interface" but, if we call the file directly from a command line, it will have a "__main__" in its variable.

In Code 4 you can see the main function executed when "__name__" is equals to "__main__"

Source Code 4: Main code

```
1  def main():
2      qt_app = QtGui.QApplication([])
3      main_window = QtGui.QWidget()
4      my_interface = myui.Ui_Dialog()
5      my_interface.setupUi(main_window)
6      c = Controller(my_interface)
7      main_window.show()
```

```
8       qt_app.exec_()
9
10
11  if __name__ == '__main__':
12      main()
```

The function "main" could have any name but, normally main is used. In line 11, when the interpreter goes line by line reading the code it checks if the magic variable "name" has the value "main" true (only when it is called from a command line or icon), then it will execute the code inside the "if". In this case that code is main function that is above "if".

Main function creates a "QtGui" application and then it creates a main window in which we will insert our interface. As you can see, in line 4, we are calling the class created by the compiler when we use "pyuic4". It is important to check if names are the same. Finally, that class has a method called "setupUI" that is responsible for preparing each control in their position. Then, in line 6 we create a new controller with the class designed before and we start showing the interface (line 7). In order to make it works, it has been explained before in theoretical classes, the main loop has to be called in order to start listening to control signals.

6 Testing the design

Once you have completed all the pieces of code mark with "??", you are ready for testing the design in the board. In order to test it, you should minimize the virtual machine and go to the opened SDK.

Before programming the PS with the c code explained in Section 2 it is needed to connect the temperature sensor to the board.

The board has several PMOD connectors, you must connect the temperature sensor in the PMOD A as shown in Figure 25

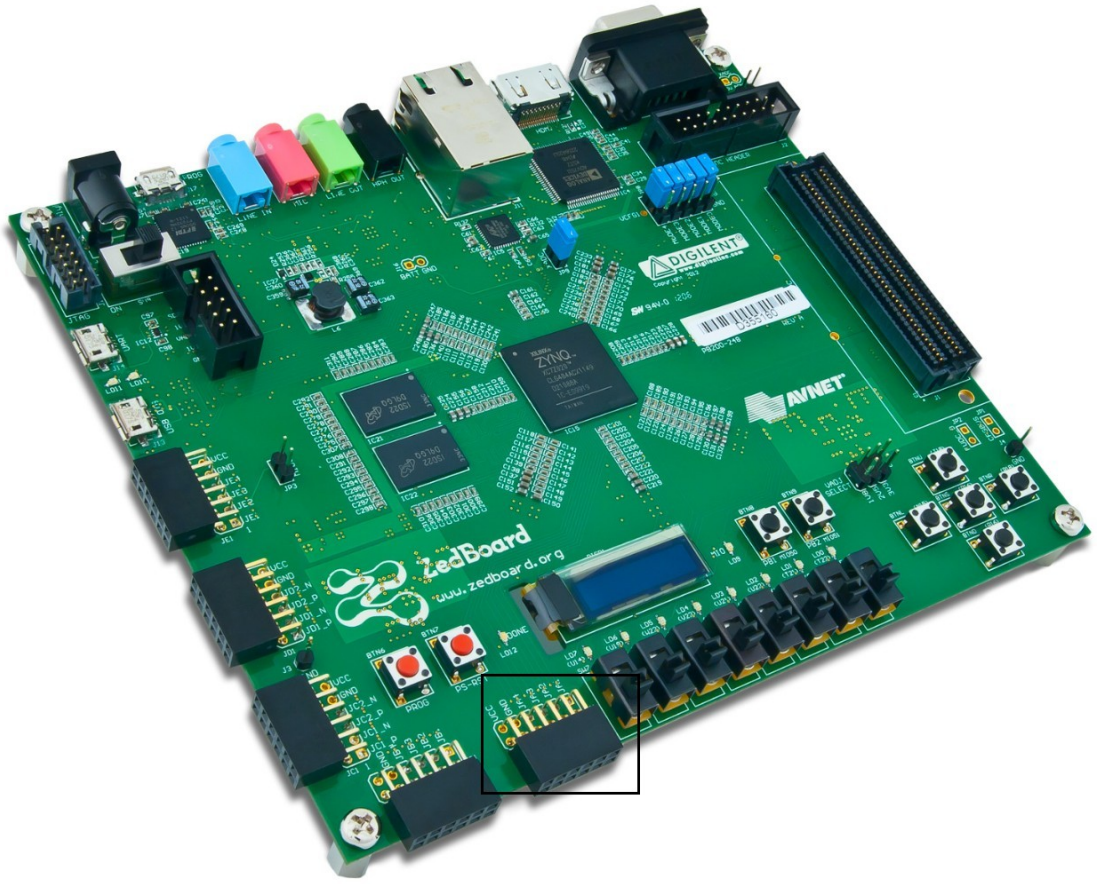

Figure 25: Connect to the upper row

Once you have connect the sensor, you are ready to program the FPGA and start the C program in PS. In Figure 26 you can see where is located the program button. Click it and then click in program button located in modal dialog as depicted in Figure 27.

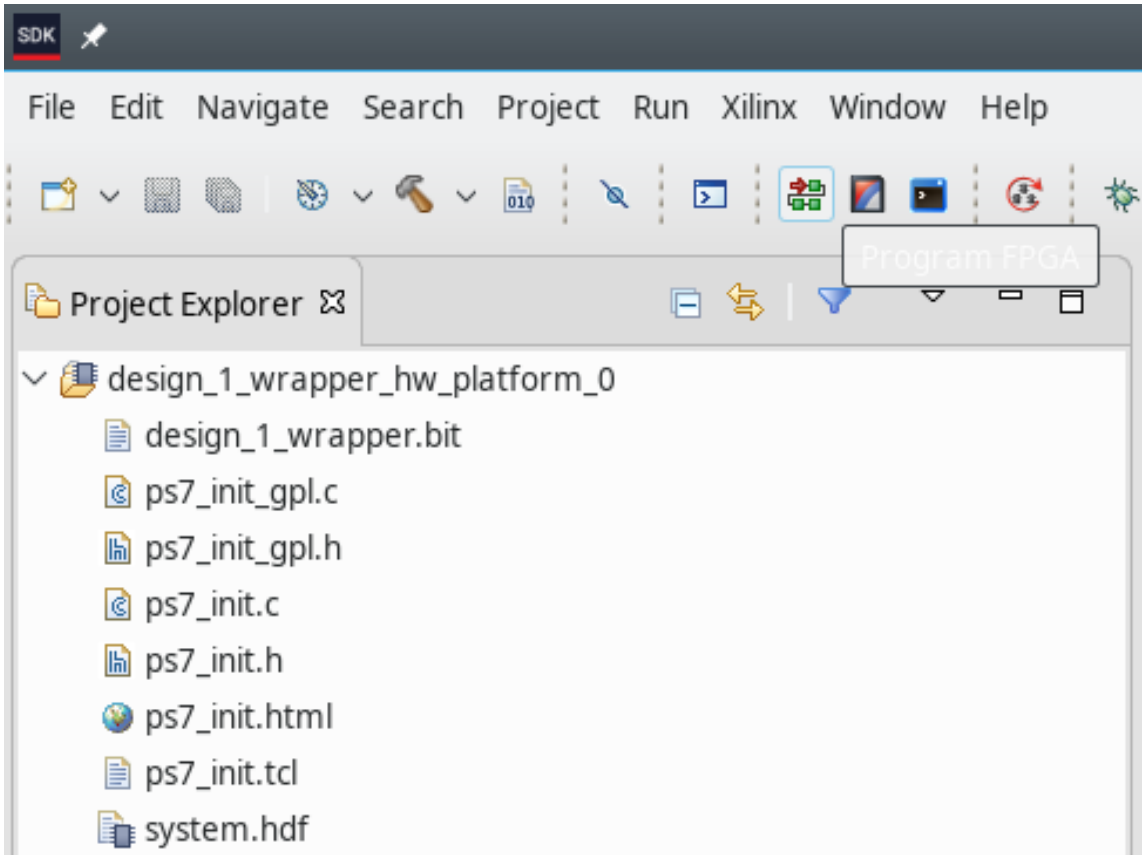

Figure 26: Green button to program FPGA

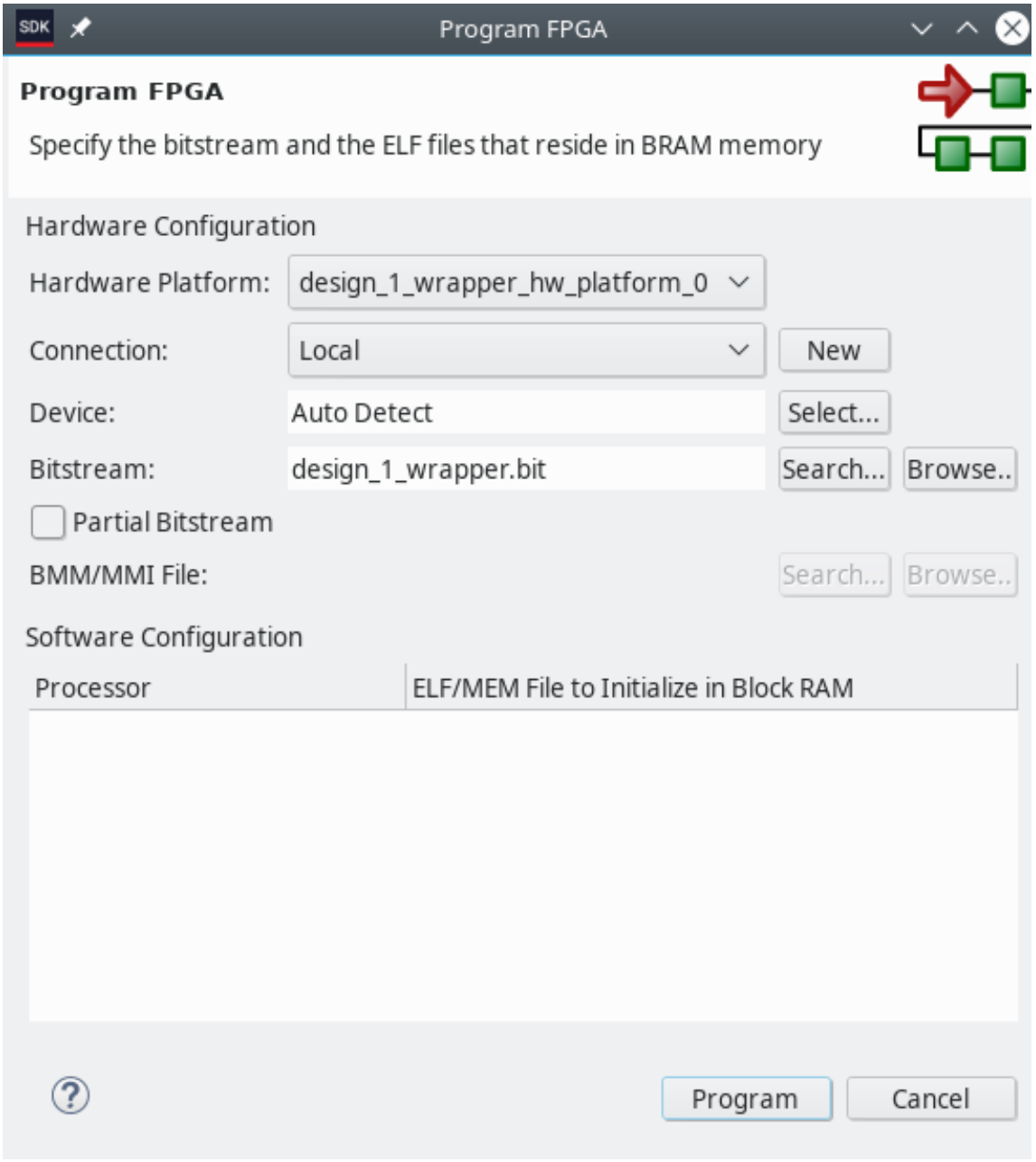

Figure 27: Modal dialog of programming FPGA procedure

When FPGA has been completed a blue led should be turned on, now you are ready to program the ARM core. To start a ARM program, you should open main.c and later click on the green button over the toolbar (Figure 28).

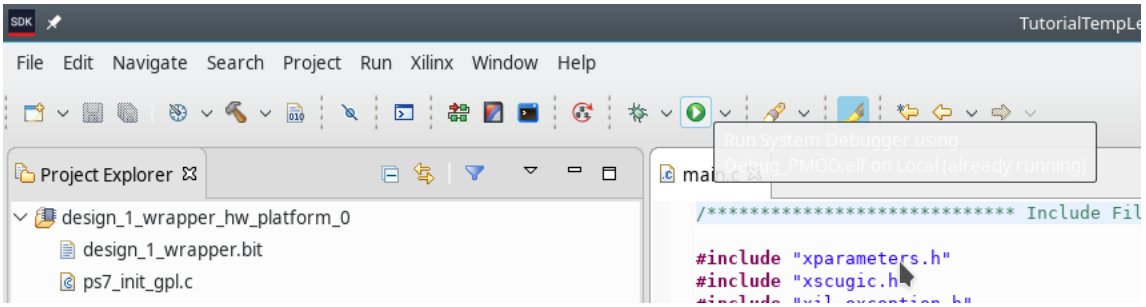

Figure 28: Green button to program the FPGA

In order to connect the serial port from the real computer to the virtual machine we have to tell virtualbox to do that. Go to menu Devices, USB and the select 2012 cypress... device (Figure 29).

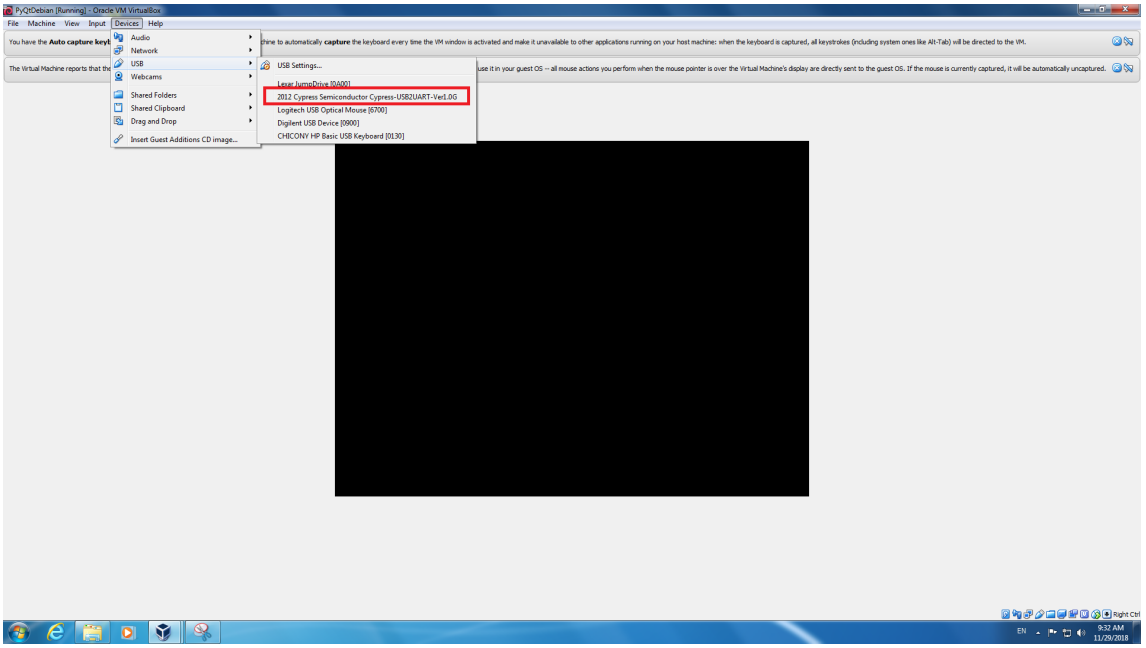

Figure 29: Connect usb to virtualbox

Now all in the board is ready, and you have connect it to the virtual machine, move to the virtual machine and open a new terminal. Go to the folder where you have the app.py file and execute it with "python3 app.py". Now, you should be ready to test the interface. In order to change the temperature you can touch the IC from the sensor.

7 Optional exercises

7.1 Add controls to select a serial device and baudrate (easy)

In this exercise, you should add two controls to select baudrate and address to serial device. You can use any kind of control.

- Design new controls in QtDesigner
- Recompile with pyuic4 your new .ui interface
- Modify your app.py to add different signals of your controls and store values
- Use these values when you create a serial device (controller)

7.2 Use layouts to make tabs responsive

If you try to resize the window you will see that tabs are not resizing properly. In order to obtain that behavior you should use layouts. Go to Qt documentation and read about layouts. Later, try to put in practice.

- Open QtDesigner and add layouts
- Recompile python code from ui file.

7.3 Modify how pyqtgraph looks (easy)

In this exercise, you can change how graphs in pyqtgraph look like. In order to do that, you should go to documentation and modify all the parameters that you want. For example, you can use antialliasing, change background color...

- Go to documentation of pyqtgraph `http://www.pyqtgraph.org/documentation/config_options.html`
- Find where you can modify those parameters
- Modify them

7.4 Modify how data is sent

Now, data is sent in a ascii way. In this exercise you can modify how data is sent. You should modify Zedboard class and SDK.

- Check how the different parts are communicating now
- Identify which parts should be changed
- Use binary form
- Modify SDK
- Relaunch PS program
- Modify ZedboardController class.