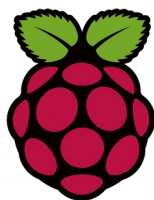


GPIO Control

Raspberry Pi 3



Autors:

Sebastian Mecheș

Daniel-Domițian Iuonac

Base statement:

Control of GPIO pins of a development board Arduino or Raspberry PI from a web interface (2 students).

Characteristics:

- The web interface will contain a select option for inputs / outputs (GPIO) that are active on the development board. The GPIO pins unused shall be powered-down.
- The web interface will allow the selection of the use mode of each active pin of development board: ON / OFF (0 or 1 logic) or the possibility to generate PWM.
- A secure login method for the web interface should be implemented, for example Google VPN Authenticator
- The state of the user's current settings will be saved either in the EEPROM memory of a microcontroller or on an SD card so that the system restarts from the last known state when the power is interrupted.

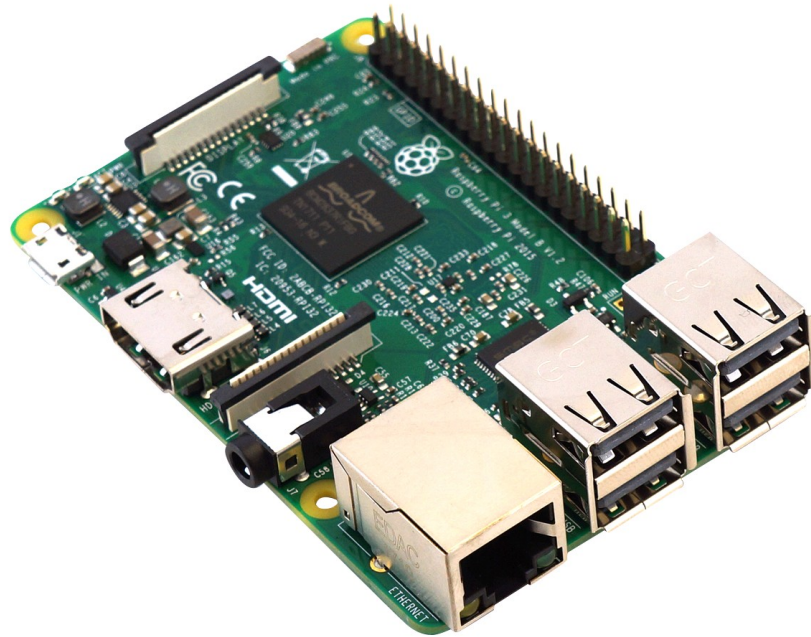
Adapted Statement:

Control of GPIO pins of a development board Arduino or Raspberry PI from a interface (2 students).

Characteristics:

- The interface will contain a select option for inputs / outputs (GPIO) that are active on the development board. The GPIO pins unused shall be powered-down.
- The interface will allow the selection of the use mode of each active pin of development board: ON / OFF (0 or 1 logic) or the possibility to generate PWM.
- The state of the user's current settings will be saved on an SD card so that the system restarts from the last known state when the power is interrupted.

Raspberry Pi 3 Board



Raspberry Pi 3 Specifications [1]

SoC: Broadcom BCM2837

CPU: 4× ARM Cortex-A53, 1.2GHz

GPU: Broadcom VideoCore IV

RAM: 1GB LPDDR2 (900 MHz)

Networking: 10/100 Ethernet, 2.4GHz 802.11n wireless

Bluetooth: Bluetooth 4.1 Classic, Bluetooth Low Energy

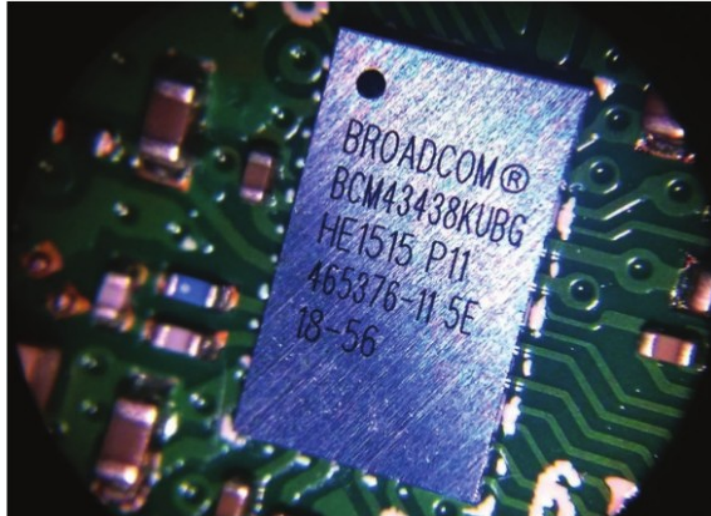
Storage: microSD

GPIO: 40-pin header, populated

Ports: HDMI, 3.5mm analogue audio-video jack, 4× USB 2.0, Ethernet, Camera Serial Interface (CSI), Display Serial Interface (DSI)

Wireless radio

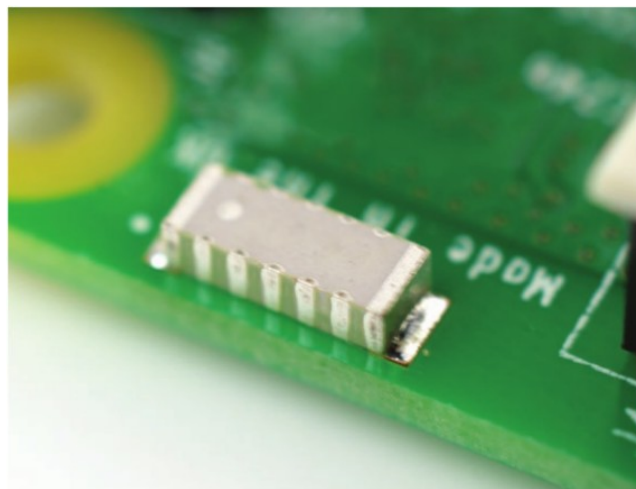
So small, its markings can only be properly seen through a microscope or magnifying glass, the Broadcom BCM43438 chip provides 2.4GHz 802.11n wireless LAN, Bluetooth Low Energy, and Bluetooth 4.1 Classic radio support. Cleverly built directly onto the board to keep costs down, rather than the more common fully qualified module approach, its only unused feature is a disconnected FM radio receiver.



Wireless radio

Antenna

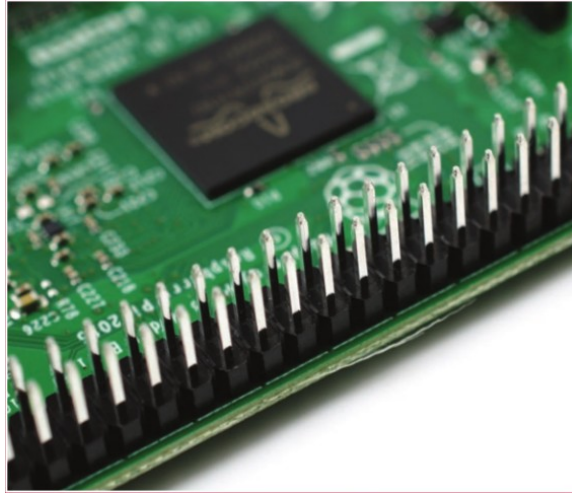
There's no need to connect an external antenna to the Raspberry Pi 3. Its radios are connected to this chip antenna soldered directly to the board, in order to keep the size of the device to a minimum. Despite its diminutive stature, this antenna should be more than capable of picking up wireless LAN and Bluetooth signals – even through walls.



Antena

GPIO

The Raspberry Pi 3 features the same 40-pin general-purpose input-output (GPIO) header as all the Pis going back to the Model B+ and Model A+. Any existing GPIO hardware will work without modification; the only change is a switch to which UART is exposed on the GPIO's pins, but that's handled internally by the operating system.



GPIO

SoC

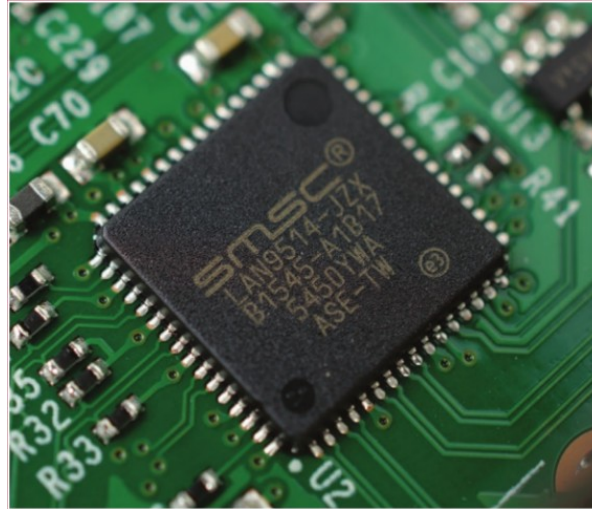
Built specifically for the new Pi 3, the Broadcom BCM2837 system-on-chip (SoC) includes four high-performance ARM Cortex-A53 processing cores running at 1.2GHz with 32kB Level 1 and 512kB Level 2 cache memory, a VideoCore IV graphics processor, and is linked to a 1GB LPDDR2 memory module on the rear of the board.



SOC

USB chip

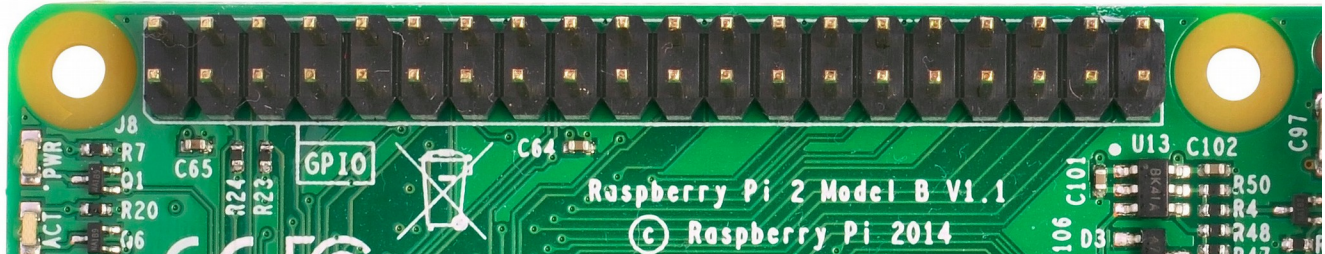
The Raspberry Pi 3 shares the same SMSC LAN9514 chip as its predecessor, the Raspberry Pi 2, adding 10/100 Ethernet connectivity and four USB channels to the board. As before, the SMSC chip connects to the SoC via a single USB channel, acting as a USB-to-Ethernet adaptor and USB hub.



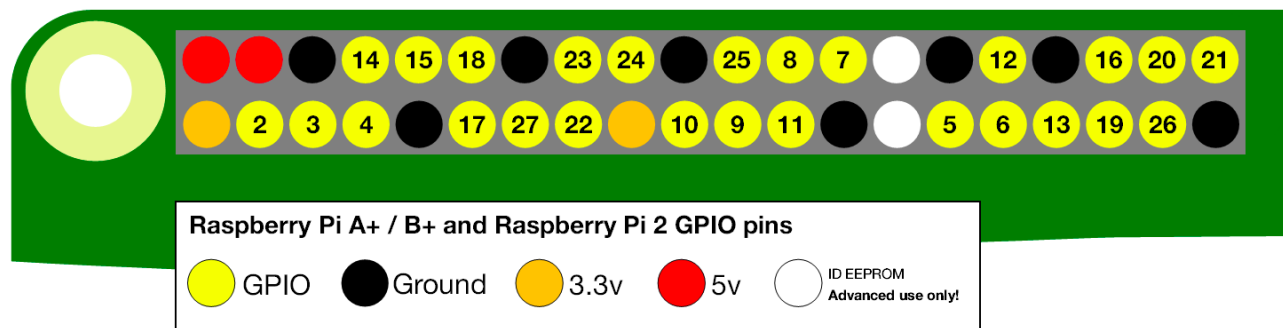
USB chip

GPIO^{[2][3]}

A powerful feature of the Raspberry Pi is the row of GPIO (general-purpose input/output) pins along the top edge of the board. A 40-pin GPIO header is found on all current Raspberry Pi boards (unpopulated on Pi Zero and Pi Zero W). Prior to the Pi 1 Model B+ (2014), boards comprised a shorter 26-pin header.



Any of the GPIO pins can be designated (in software) as an input or output pin and used for a wide range of purposes.



Note: the numbering of the GPIO pins is not in numerical order; GPIO pins 0 and 1 are present on the board (physical pins 27 and 28) but are reserved for advanced use (see below).

Voltages

Two 5V pins and two 3V3 pins are present on the board, as well as a number of ground pins (0V), which are unconfigurable. The remaining pins are all general purpose 3V3 pins, meaning outputs are set to 3V3 and inputs are 3V3-tolerant.

Outputs

A GPIO pin designated as an output pin can be set to high (3V3) or low (0V).

Inputs

A GPIO pin designated as an input pin can be read as high (3V3) or low (0V). This is made easier with the use of internal pull-up or pull-down resistors. Pins GPIO2 and GPIO3 have fixed pull-up resistors, but for other pins this can be configured in software.

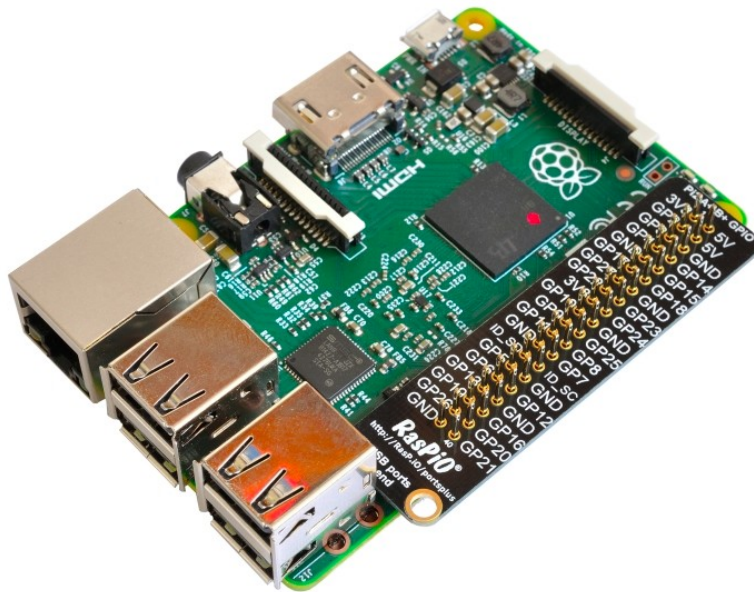
More

As well as simple input and output devices, the GPIO pins can be used with a variety of alternative functions, some are available on all pins, others on specific pins.

- PWM (pulse-width modulation)
 - Software PWM available on all pins
 - Hardware PWM available on GPIO12, GPIO13, GPIO18, GPIO19
- SPI
 - SPI0: MOSI (GPIO10); MISO (GPIO9); SCLK (GPIO11); CE0 (GPIO8), CE1 (GPIO7)
 - SPI1: MOSI (GPIO20); MISO (GPIO19); SCLK (GPIO21); CE0 (GPIO18); CE1 (GPIO17); CE2 (GPIO16)
- I2C
 - Data: (GPIO2); Clock (GPIO3)
 - EEPROM Data: (GPIO0); EEPROM Clock (GPIO1)
- Serial
 - TX (GPIO14); RX (GPIO15)

GPIO pinout

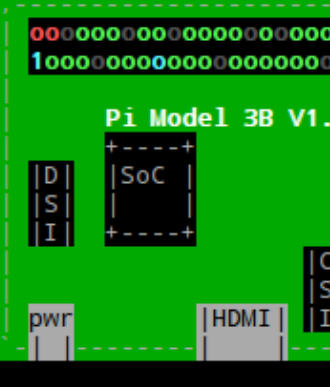
It's important to be aware of which pin is which. Some people use pin labels (like the RasPiO Portsplus PCB, or the printable Raspberry Leaf).




```

pi@raspberrypi:~ $ pinout

```



```

Revision      : a02082
SoC           : BCM2837
RAM          : 1024Mb
Storage      : MicroSD
USB ports    : 4 (excluding power)
Ethernet ports : 1
Wi-fi       : True
Bluetooth   : True
Camera ports (CSI) : 1
Display ports (DSI): 1

J8:
  3V3 (1) (2) 5V
  GPIO2 (3) (4) 5V
  GPIO3 (5) (6) GND
  GPIO4 (7) (8) GPIO14
  GND (9) (10) GPIO15
  GPIO17 (11) (12) GPIO18
  GPIO27 (13) (14) GND
  GPIO22 (15) (16) GPIO23
  3V3 (17) (18) GPIO24
  GPIO10 (19) (20) GND
  GPIO9 (21) (22) GPIO25
  GPIO11 (23) (24) GPIO8
  GND (25) (26) GPIO7
  GPIO0 (27) (28) GPIO1
  GPIO5 (29) (30) GND
  GPIO6 (31) (32) GPIO12
  GPIO13 (33) (34) GND
  GPIO19 (35) (36) GPIO16
  GPIO26 (37) (38) GPIO20
  GND (39) (40) GPIO21

For further information, please refer to https://pinout.xyz/
pi@raspberrypi:~ $

```

Programming with GPIO

It is possible to control GPIO pins using a number of programming languages and tools:

- GPIO with Scratch 1.4
- GPIO with Scratch 2
- GPIO with Python

Warning: while connecting up simple components to the GPIO pins is perfectly safe, it's important to be careful how you wire things up. LEDs should have resistors to limit the current passing through them. Do not use 5V for 3V3 components. Do not connect motors directly to the GPIO pins, instead use an H-bridge circuit or a motor controller board.

The Qt Framework [4][5]

Qt (pronounced "cute") is a free and open-source widget toolkit for creating graphical user interfaces as well as cross-platform applications that run on various software and hardware platforms such as Linux, Windows, macOS, Android or embedded systems with little or no change in the underlying codebase while still being a native application with native capabilities and speed. Qt is currently being developed by The Qt Company, a publicly listed company, and the Qt Project under open-source governance, involving individual developers and organizations working to advance Qt. Qt is available under both commercial licenses and open source GPL 2.0, GPL 3.0, and LGPL 3.0 licenses.

What is PyQt5?

PyQt5 is a comprehensive set of Python bindings for Qt v5. It is implemented as more than 35 extension modules and enables Python to be used as an alternative application development language to C++ on all supported platforms including iOS and Android.

PyQt5 may also be embedded in C++ based applications to allow users of those applications to configure or enhance the functionality of those applications.

Application Code [8]

The source code of the application is available on github repository [8].

The current application uses the Qt5 framework and is written in Python 3. The controls of GPIO pins of raspberry is made with Rpi.GPIO[6][7] python library.

The application is using MVC pattern and is structured in 4 modules: **app.py**, **controllers.py**, **models.py**, **data.py**. The start module is **main.py** and contains the next code:

```
if __name__ == '__main__':
    # import and start the Application
    import sys
    from package.app import Application
    application = Application(sys.argv)
    sys.exit(application.exec_())
```

The **app.py** module is started from the main.py module and contains the Application class:

```
import os
from PyQt5.QtGui import QIcon
from PyQt5.QtWidgets import QApplication

from db import save_configuration, load_configuration
from package.controllers import MainWindow

import RPi.GPIO as GPIO
GPIO.setmode(GPIO.BCM)

# Application class extends QApplication from QtWidgets

class Application(QApplication):
    def __init__(self, args):
        super(Application, self).__init__(args)
        # Set some properties
        self.setApplicationName('Raspberry Pi - G.P.I.O Control')
        self.setApplicationVersion('0.1')
        self.icon = QIcon(os.getcwd() + "/icons/rpi.png")
        self.setWindowIcon(self.icon)

        # On exiting application save the configuration
        self.aboutToQuit.connect(save_configuration)

        # Load the configuration
        load_configuration()

        # Create and show the main window
        self.main_window = MainWindow()
        self.main_window.show()
```

The **controllers.py** module contains all the ui controllers and the logic behind.

```
from RPi import GPIO
import os
from PyQt5 import QtWidgets, QtGui
from PyQt5.QtCore import QModelIndex, Qt

from db import PIN_LIST
from package.models import PinModel
from package.ui.digital_input import Ui_DigitalInput
from package.ui.digital_out import Ui_DigitalOutput
from package.ui.license import Ui_License
from package.ui.main_window import Ui_MainWindow
from package.ui.pulse_width_modulation import Ui_PulseWidthModulation

# DigitalInputController is a QWidget and controls the digital input UI Form
class DigitalInputController(QtWidgets.QWidget):
    def __init__(self, parent=None, flags=Qt.Widget):
        super(DigitalInputController, self).__init__(parent, flags=flags)
        self.ui = Ui_DigitalInput()
        self.ui.setupUi(self)
        self.current_model_index = None

        # Connect signal for pressing Read button
        self.ui.pushButton_read.clicked.connect(self.on_read_input)
```

```

"""
    on_read_input
    - get current pin
    - read input value
    - set the value to line edit in UI
"""
def on_read_input(self, checked):
    pin = PIN_LIST[self.current_model_index.row()]
    if pin.enable:
        value = GPIO.input(pin.id)
        self.ui.lineEdit_value.setText(str(value))

def set_selection(self, current):
    # Update current index
    self.current_model_index = current

# DigitalOutputController controls the UI form for digital output
class DigitalOutputController(QtWidgets.QWidget):
    def __init__(self, model: PinModel, parent=None, flags=Qt.Widget):
        super(DigitalOutputController, self).__init__(parent, flags=flags)
        self.ui = Ui_DigitalOutput()
        self.ui.setupUi(self)
        self.current_model_index = None
        self.model = model
        self.pin = PIN_LIST[0]

        # Connect slider and spinbox signals to slots
        self.ui.slider_value.valueChanged.connect(self.on_slider_value_changed)
        self.ui.spinbox_value.valueChanged.connect(self.on_spinbox_value_changed)

"""
    on_slider_value_changed
    - sets value to spinbox
    - outputs value to GPIO pin
    - save value to pin data structure
"""
def on_slider_value_changed(self, value):
    self.ui.spinbox_value.setValue(value)
    GPIO.output(self.pin.id, value)
    self.pin.value = value

# similar to on_slider_value_changed
def on_spinbox_value_changed(self, value):
    self.ui.slider_value.setValue(value)
    GPIO.output(self.pin.id, value)
    self.pin.value = value

"""
    set_selection
    - save current index
    - update current pin reference
"""
def set_selection(self, current):
    self.current_model_index = current
    self.pin = PIN_LIST[self.current_model_index.row()]

"""
    showEvent
    - when a show event is received then accept it
    - update spinbox and slider value
"""
def showEvent(self, a0: QtGui.QShowEvent):
    a0.accept()
    self.ui.slider_value.setValue(self.pin.value)
    self.ui.spinbox_value.setValue(self.pin.value)

# PulseWidthModulationController controls the UI form for PWM
class PulseWidthModulationController(QtWidgets.QWidget):
    def __init__(self, model: PinModel, parent=None, flags=Qt.Widget):
        super(PulseWidthModulationController, self).__init__(parent=parent, flags=flags)
        self.ui = Ui_PulseWidthModulation()
        self.ui.setupUi(self)
        self.model = model
        self.current_model_index = None
        self.pin = PIN_LIST[0]

        # Connect signals for sliders spinboxes and pwm start/stop button
        self.ui.slider_frequency.valueChanged.connect(self.on_slider_frequency)
        self.ui.slider_duty_cycle.valueChanged.connect(self.on_slider_duty_cycle)
        self.ui.spinbox_frequency.valueChanged.connect(self.on_spinbox_frequency)

```

```

        self.ui.spinbox_duty_cycle.valueChanged.connect(self.on_spinbox_duty_cycle)
        self.ui.button_pwm.clicked.connect(self.on_button_pwm)

    """
    on_button_pwm
    if button is pressed then
        - set Stop text
        - start PWM instance
        - changeFrequency
    else
        - set Start text
        - stop pwm instance
    """
    def on_button_pwm(self, value):
        if value:
            self.ui.button_pwm.setText("Stop")
            self.pin.pwm_instance.start(self.pin.duty_cycle)
            self.pin.pwm_instance.ChangeFrequency(self.pin.frequency)
        else:
            self.ui.button_pwm.setText("Start")
            self.pin.pwm_instance.stop()

    """
    on_spinbox_duty_cycle
    - update slider value
    - if pwm instance is not null then change duty cycle
    - save value to pin structure
    """
    def on_spinbox_duty_cycle(self, value):
        self.ui.slider_duty_cycle.setValue(value)
        if self.pin.pwm_instance and value > 0:
            self.pin.pwm_instance.ChangeDutyCycle(value)
            self.pin.duty_cycle = value

# similar to on_spinbox_duty_cycle
    def on_spinbox_frequency(self, value):
        self.ui.slider_frequency.setValue(value)
        if self.pin.pwm_instance and value > 0:
            self.pin.pwm_instance.ChangeFrequency(value)
            self.pin.frequency = value

# similar to on_spinbox_duty_cycle
    def on_slider_frequency(self, value):
        self.ui.spinbox_frequency.setValue(value)
        if self.pin.pwm_instance and value > 0:
            self.pin.pwm_instance.ChangeFrequency(value)
            self.pin.frequency = value

# similar to on_spinbox_duty_cycle
    def on_slider_duty_cycle(self, value):
        self.ui.spinbox_duty_cycle.setValue(value)
        if self.pin.pwm_instance and value > 0:
            self.pin.pwm_instance.ChangeDutyCycle(value)
            self.pin.duty_cycle = value

    def set_selection(self, current):
        self.current_model_index = current
        self.pin = PIN_LIST[self.current_model_index.row()]

# update values to sliders and spinboxes when controller shows
    def showEvent(self, a0: QtGui.QShowEvent):
        a0.accept()
        self.ui.slider_frequency.setValue(self.pin.frequency)
        self.ui.spinbox_frequency.setValue(self.pin.frequency)
        self.ui.spinbox_duty_cycle.setValue(self.pin.duty_cycle)
        self.ui.slider_duty_cycle.setValue(self.pin.duty_cycle)

# The main window controller
class MainWindow(QtWidgets.QMainWindow):
    def __init__(self, parent=None, flags=Qt.Window):
        super(MainWindow, self).__init__(parent, flags)
        self.ui = Ui_MainWindow()
        self.ui.setupUi(self)
        self.data_mapper = QtWidgets.QDataWidgetMapper()
        self.model = PinModel(PIN_LIST)
        self.current_model_index = None

        # Set the model to data mapper and to list view
        self.data_mapper.setModel(self.model)
        self.ui.listView.setModel(self.model)

        # Set mapping for data mapper

```

```

self.data_mapper.addMapping(self.ui.lineEdit_name, 0)
self.data_mapper.addMapping(self.ui.comboBox_type, 1)
self.data_mapper.addMapping(self.ui.comboBox_mode, 2)
self.data_mapper.toFirst()

# When list view's current index changes then set_selection is called
self.ui.listView.selectionModel().currentChanged.connect(self.set_selection)

# Instantiate the others controllers and add them to the layout
self.license_controller = LicenseController()
self.digital_input_controller = DigitalInputController(self)
self.digital_output_controller = DigitalOutputController(self.model, self)
self.pwm_controller = PulseWidthModulationController(self.model, self)
self.ui.layout_values.addWidget(self.digital_input_controller)
self.ui.layout_values.addWidget(self.digital_output_controller)
self.ui.layout_values.addWidget(self.pwm_controller)

# Set slots for combo boxes signals and button enable
self.ui.comboBox_type.currentTextChanged.connect(self.on_type_changed)
self.ui.comboBox_mode.currentTextChanged.connect(self.on_mode_changed)
self.ui.button_enable.clicked.connect(self.on_button_enable)

# Set slots for actions triggers signals
self.ui.action_exit.triggered.connect(QtWidgets.QApp.quit)
self.ui.action_license.triggered.connect(self.license_controller.show)
self.ui.action_about.triggered.connect(self.on_action_about)
self.ui.action_load.triggered.connect(self.on_action_load)
self.ui.action_save.triggered.connect(self.on_action_save)

def set_selection(self, current: QModelIndex, old: QModelIndex):
    self.current_model_index = current
    self.data_mapper.setCurrentModelIndex(current)

    self.pwm_controller.set_selection(current)
    self.digital_output_controller.set_selection(current)
    self.digital_input_controller.set_selection(current)
    self.on_update_ui(current.row())

def on_mode_changed(self, text: str):
    pin_type = self.ui.comboBox_type.currentText()
    self.on_update_controllers(pin_type, text)

def on_type_changed(self, text):
    pin_mode = self.ui.comboBox_mode.currentText()
    self.on_update_controllers(text, pin_mode)

def showEvent(self, a0: QtGui.QShowEvent):
    pin_type = self.ui.comboBox_type.currentText()
    pin_mode = self.ui.comboBox_mode.currentText()
    self.on_update_controllers(pin_type, pin_mode)
    self.on_update_ui(0)

# This method updates the ui values when a new pin is selected in the list view
def on_update_ui(self, index: int):
    pin = PIN_LIST[index]
    self.ui.comboBox_mode.setEnabled(not pin.enable)
    self.ui.comboBox_type.setEnabled(not pin.enable)
    self.pwm_controller.ui.button_pwm.setEnabled(pin.enable)
    self.pwm_controller.ui.slider_duty_cycle.setEnabled(pin.enable)
    self.pwm_controller.ui.slider_frequency.setEnabled(pin.enable)
    self.pwm_controller.ui.spinbox_frequency.setEnabled(pin.enable)
    self.pwm_controller.ui.spinbox_duty_cycle.setEnabled(pin.enable)
    self.digital_input_controller.ui.pushButton_read.setEnabled(pin.enable)
    self.digital_output_controller.ui.slider_value.setEnabled(pin.enable)
    self.digital_output_controller.ui.spinbox_value.setEnabled(pin.enable)
    self.ui.button_enable.setChecked(pin.enable)
    if pin.enable:
        self.ui.button_enable.setText("Disable")
    else:
        self.ui.button_enable.setText("Enable")

# This method updates the ui values and sets GPIO values also
def on_button_enable(self, value):
    self.ui.comboBox_type.setEnabled(not value)
    self.ui.comboBox_mode.setEnabled(not value)
    self.pwm_controller.ui.button_pwm.setEnabled(value)
    self.pwm_controller.ui.slider_duty_cycle.setEnabled(value)
    self.pwm_controller.ui.slider_frequency.setEnabled(value)
    self.pwm_controller.ui.spinbox_frequency.setEnabled(value)
    self.pwm_controller.ui.spinbox_duty_cycle.setEnabled(value)
    self.digital_input_controller.ui.pushButton_read.setEnabled(value)
    self.digital_output_controller.ui.slider_value.setEnabled(value)
    self.digital_output_controller.ui.spinbox_value.setEnabled(value)

```



```

pin = PIN_LIST[self.current_model_index.row()]
pin.enable = value

if value:
    self.ui.button_enable.setText("Disable")

    if pin.type.__eq__("Input") and pin.mode.__eq__("Digital"):
        GPIO.setup(pin.id, GPIO.IN)
    else:
        if pin.mode.__eq__("Digital"):
            GPIO.setup(pin.id, GPIO.OUT)
            GPIO.output(pin.id, pin.value)
        else:
            GPIO.setup(pin.id, GPIO.OUT)
            if not pin.pwm_instance:
                pin.pwm_instance = GPIO.PWM(pin.id, pin.frequency)

else:
    self.ui.button_enable.setText("Enable")
    self.pwm_controller.ui.button_pwm.setText("Start")
    self.pwm_controller.ui.button_pwm.setChecked(False)
    self.pwm_controller.ui.button_pwm.setEnabled(False)

    if pin.type.__eq__("Input") and pin.mode.__eq__("Digital"):
        GPIO.setup(pin.id, GPIO.IN)
    else:
        if pin.mode.__eq__("Digital"):
            GPIO.setup(pin.id, GPIO.OUT)
            GPIO.output(pin.id, 0)
        else:
            if pin.pwm_instance:
                pin.pwm_instance.stop()
                pin.pwm_instance = None

# This method shows or hides the controllers based on the selection made to pin mode and type
def on_update_controllers(self, pin_type: str, pin_mode: str):
    if pin_type.__eq__("Input"):
        if pin_mode.__eq__("Digital"):
            self.digital_input_controller.setVisible(True)
            self.digital_output_controller.setVisible(False)
            self.pwm_controller.setVisible(False)
        else:
            self.digital_input_controller.setVisible(False)
            self.digital_output_controller.setVisible(False)
            self.pwm_controller.setVisible(False)
    else:
        if pin_mode.__eq__("Digital"):
            self.digital_input_controller.setVisible(False)
            self.digital_output_controller.setVisible(True)
            self.pwm_controller.setVisible(False)
        else:
            self.digital_input_controller.setVisible(False)
            self.digital_output_controller.setVisible(False)
            self.pwm_controller.setVisible(True)

# About method show a message box
def on_action_about(self):
    msg = QtWidgets.QMessageBox()
    msg.setWindowTitle("Raspberry Pi GPIO Control About")
    msg.setText("This tool has been made by Daniel-Domițian Iuonac and Sebastian Mecheș.")
    msg.setStandardButtons(QtWidgets.QMessageBox.Ok)
    msg.setIcon(QtWidgets.QMessageBox.Information)
    msg.exec_()

"""
This is called when Save configuration is pressed
A FileDialog is shown and a file name must be provided
Saves the current pins configuration
"""
def on_action_save(self):
    file_dialog = QtWidgets.QFileDialog()
    file_dialog.setAcceptMode(QtWidgets.QFileDialog.AcceptSave)
    path = os.getcwd() + "/db/"

    file_path = file_dialog.getSaveFileName(QtWidgets.QWidget(), "Save G.P.I.O Configuration", path,
                                           filter="Json file (*.json)")

    filename = file_path[0]

    if filename:
        if ".json" not in filename:
            filename += ".json"
        from db import save_configuration
        save_configuration(filename)

```

```

"""
    This method is load the pins configuration when Load button is pressed
    It opens a File Dialog and lets user to select the configuration file
"""
def on_action_load(self):
    file_dialog = QtWidgets.QFileDialog()
    file_dialog.setAcceptMode(QtWidgets.QFileDialog.AcceptSave)
    path = os.getcwd() + "/db/"

    file_path = file_dialog.getOpenFileName(QtWidgets.QWidget(), "Load G.P.I.O Configuration", path,
                                            filter="Json file (*.json)")

    filename = file_path[0]

    if filename:
        if ".json" not in filename:
            filename += ".json"
        from db import load_configuration
        load_configuration(filename)
        self.model.removeRows(0, 27)
        for pin in PIN_LIST:
            self.model.insert([0, pin])

# This controller is for License Ui Form
class LicenseController(QtWidgets.QWidget):
    def __init__(self, parent=None):
        super(LicenseController, self).__init__(parent, flags=Qt.Widget)
        self.ui = Ui_License()
        self.ui.setupUi(self)
        self.setWindowModality(Qt.WindowModal)
        self.ui.button_close.clicked.connect(self.close)
        file = open("LICENSE", "r")
        text = file.read()
        file.close()
        self.ui.text.setPlainText(text)

```

The **models.py** contains the model the application uses in order to display the data in the views.

```

import typing
from PyQt5.QtCore import QAbstractTableModel, QModelIndex, Qt

"""
    This class inherits QAbstractTableModel in order to display data to List View in Main Window
    - data method is used to display data to fields in both roles (EditRole and Display Role)
    - setData method is used to set data to pin when fields values are changed from UI
    - columnCount method specifies how many attributes does a row have (pin attributes)
    - rowCount method specifies the number of pins
    - removeRows is used to remove pins from the model
    - insertRow is used to insert a pin in the model
"""

class PinModel(QAbstractTableModel):
    def __init__(self, pins_list, parent=None):
        super(PinModel, self).__init__(parent)
        self._pins_list = pins_list

    def data(self, index: QModelIndex, role: int = ...):
        if index.isValid():
            if role == Qt.EditRole or role == Qt.DisplayRole:
                return self._pins_list[index.row()].data(index.column())

    def setData(self, index: QModelIndex, value: typing.Any, role: int = ...):
        if index.isValid():
            if role == Qt.EditRole or role == Qt.DisplayRole:
                self._pins_list[index.row()].set_data(index.column(), value)
                return True
            return False

    def columnCount(self, parent=None, *args, **kwargs):
        return 6

    def rowCount(self, parent=None, *args, **kwargs):
        return self._pins_list.__len__()

    def flags(self, index: QModelIndex):
        return Qt.ItemIsEnabled | Qt.ItemIsSelectable

    def removeRows(self, p_int, p_int_1, parent=QModelIndex(), *args, **kwargs):
        self.beginRemoveRows(parent, p_int, p_int + p_int_1 - 1)

```

```

del self._pins_list[p_int: p_int+1]
self.endRemoveRows()
return True

def insertRow(self, p_int, parent=QModelIndex(), *args, **kwargs):
    self.beginInsertRows(parent, 0, 0)
    self._pins_list.insert(p_int[0], p_int[1])
    self.endInsertRows()

```

The **data.py** module contains the class Pin which is used as a data structure for each pin properties.

```

"""
    Pin class represents a pin. This class is a data structure
"""

```

```

class Pin(object):
    def __init__(self):
        self.id = 0
        self.name = ""
        self.type = "Input"
        self.mode = "Digital"
        self.frequency = 0
        self.duty_cycle = 0
        self.value = 0
        self.enable = False
        self.pwm_instance = None

    def set_data(self, column: int, value):
        if column == 0:
            self.name = value
        elif column == 1:
            self.type = value
        elif column == 2:
            self.mode = value
        elif column == 3:
            if value > 0:
                self.frequency = value
        elif column == 4:
            if value > 0:
                self.duty_cycle = value
        elif column == 5:
            self.value = value

    def data(self, column: int):
        if column == 0:
            return self.name
        elif column == 1:
            return self.type
        elif column == 2:
            return self.mode
        elif column == 3:
            return self.frequency
        elif column == 4:
            return self.duty_cycle
        elif column == 5:
            return self.value

    # Represents the pin state as a dictionary object
    def __repr__(self):
        data = {
            "id": self.id,
            "name": self.name,
            "type": self.type,
            "mode": self.mode,
            "frequency": self.frequency,
            "duty_cycle": self.duty_cycle,
            "value": self.value,
            "enable": self.enable
        }
        return data

    # Loads a dictionary to current object
    def load(self, config: dict):
        self.id = config["id"]
        self.name = config["name"]
        self.type = config["type"]
        self.mode = config["mode"]
        self.frequency = config["frequency"]
        self.duty_cycle = config["duty_cycle"]
        self.value = config["value"]
        self.enable = config["enable"]

```

References:

- [1] <https://www.raspberrypi.org/magpi/raspberry-pi-3-specs-benchmarks/>
- [2] <https://www.raspberrypi.org/documentation/usage/gpio/>
- [3] <https://www.raspberrypi.org/documentation/hardware/raspberrypi/gpio/README.md>
- [4] <https://doc.qt.io/qt-5/index.html>
- [5] <https://pypi.org/project/PyQt5/>
- [6] <https://pypi.org/project/RPi.GPIO/>
- [7] <https://sourceforge.net/p/raspberry-gpio-python/wiki/Examples/>
- [8] <https://github.com/dd-iuonac/rpi-control-gpio>