RC Car project

Prepared by

1. Merna Hany Sedky.
2. Mohammad Mohsen Abdel Fattah.
3. Mohammad Abd Allah Habib.

Requirements

Build a rpi3 based remotely controlled differential drive car that:

1. Will be controlled through SSH terminal via keyboard keys of the connected laptop
   - Up arrow -> move forward (S*1, S*1)
   - Right arrow -> rotate right (S*1, S*-1)
   - Left arrow -> rotate left (S*-1, S*1)
   - Up + Right -> curve to the right (S*1, S/C)
   - Up + Left -> curve to the left (S/C, S*1)
   - “a” & “z” keys -> used to control motor speed (S where S= 0-100, step= 10)
   - “q” & “w” keys -> used to control turn rate (C where C= 0-100, step=20)
   - Derive the inverse combinations for Down arrow (Bonus)
2. Only one controller can connect to your rpi3 car at a time, rpi3 shall refuse all new connections if a connection is already established.
3. Rpi3 shall periodically send Linear and Angular speed of the car on the control terminal.
4. Linear and angular speed should be absolute values in m/s and degree/s respectively, this implies you have to use encoders on your motors and calculate the real wheel speeds.
5. Rpi3 shall make sure that car stops if connection is lost.
6. Mount any range sensor in the car front (ultrasonic, laser, IR) so that the car shall continuously transmit to the controller the range to the nearest object.
7. Team should demonstrate the accuracy of their range finder system implemented on rpi3 in the range between 10cm to 150 cm.
8. Power supply to the car should not exceed 12V.
9. Car should fit in a box with whose LxWxH = 10cm x 10cm x 10 cm.

Milestones

1. Network configuration: Wifi hotspot, ssh with only 1 client.
2. GPIO handling:
    a. Basic GPIO operations.
    b. Motor interface.
    c. Ultrasonic interface.
3. User interface: handle connection between raspberry-pi and the user through ssh terminal.
4. Design and implement RC Car logic.
5. Design car body.
6. Integrate software program with the hardware.

1. Network configurations

To build a Linux image that supports SSH, we followed the configurations in the presentation slides. After that, our goal was to enable Wi-Fi on the raspberry-pi.

We followed the guide in this link[1]. The guide is divided into 2 parts, the first part explains how to configure Wi-Fi as a client on the raspberry-pi, we managed to build this image and connect to a Wi-Fi hotspot on a laptop.

The second part of the guide explains how to configure the raspberry-pi to create a Wi-Fi hotspot, configure a DHCP server to lease IPs to clients after they connect to the network. We had a problem with this configurations, when the raspberry-pi boots, the DHCP server won't start, so clients were not able to connect to the Wi-Fi hotspot. We couldn't troubleshoot why the DHCP server was not running correctly, so as a work around, we followed this guide[2].

At first we installed both `hostapq` and `dnsmasq`. `hostapd` managed the access point creation, while `dnsmasq` was used to provide a DHCP server. The access point worked successfully, but the DHCP server of `dnsmasq` didn't work because another DHCP server was already running (the one from the previous guide, which we didn't remove). So, we removed `dnsmasq` from the image and everything worked with no errors.

The files related to network configurations are:

1. /etc/interfaces: configurations for Ethernet and wireless interfaces. The wireless interface was given a static IP address, so that each time the raspberry-pi is booted, the wireless interface will have the same address.
2. /etc/ssh/sshd_config: contains configurations for ssh, modified to grant root access.
3. /etc/dhcp/Dhcp.conf: DHCP server configurations for Wi-Fi and Ethernet interfaces.
4. /etc/wpa_supplicant: was used to configure Wi-Fi to connect to an external hotspot, it's configurations are disabled by dhcpd.cof and overridden in hostapd.conf.
5. /etc/hostapd/Hostapd.conf: Contains configurations for raspberry-pi's access point.

DHCP was configured to allow leasing only 2 IPs on the wireless interface, with 1 IP statically assigned to the wireless interface, this leaves only 1 IP to be assigned to the first client that connects to the network. By using a subnet mask of 255.255.255.252 or 192.168.2.0/30.

The wireless interface is assigned the IP address 192.168.2.1/30. The Ethernet interface was not assigned an IP address, but left to the DHCP server to assign an IP to it in the range 192.168.1.0/24.

[1] www.raspberrypi.org/documentation/configuration/wireless/access-point.md
[2] https://www.raspberrypi.org/documentation/configuration/wireless/access-point.md

2. GPIO handling

For GPIO handling, we started with Wiring-Pi and C. First, we got Wiring-Pi source code from the official website, and followed this guide[3] to cross compile Wiring-Pi for raspberry-pi. After that, we made a simple C program that uses wiring-pi to control a gpio pin to toggle a LED on and off, cross-compiled the program and copied the program to the raspberry pi.

However, the program didn't work, when it was run, it gave an error that there is no line that contains `hardware` in the contents of `/proc/cpuinfo`. The contents of `/proc/cpuinfo` at that time is in this link[4].

Following a similar issue[5] on RPI.GPIO, we concluded that the cause might be the kernel version or the architecture (32 bit architecture). So, a different kernel version might have worked, or recompiling using arm32 bit architecture. However, re-compiling for a different architecture seemed problematic, according to feedback from our colleagues. We didn't proceed any further to solve this problem. What is interesting though, is that wiring pi was working well on one of the raspberry pi boards, but it was a different board (RaspberryPi3).

The issue seemed to be in RPI.GPIO as well. So, we were left with pigpio, which has a python wrapper in buildroot. So, we built an image both libgpio, python, pigpiod, pigpio and python-pigpio.

We had a problem that pigpio daemon would run, but whenever an API is called from within a program (or any terminal) to connect to the daemon and send commands to it, the connection would fail with `socket connect failed` error.

So, as an alternative, we devised our own python wrapper around `/sys/class/gpio` to handle GPIO manipulation, due to time restrictions(I). We took into consideration to make the library with only built-in packages.

The wrapper `gpiolib.py`[6] contains methods to set GPIO pin direction, read and write to GPIO pins, generate PWM signals on any GPIO pin, updating PWM duty cycle and stopping PWM signals. The PWM signals are generated by threads, no to block the main program. But due to the usage of threads, when running multiple threads at the same time, the CPU gets overloaded, and the accuracy of the PWM timing gets thrown off, so it's not advised to generate a lot of PWM signals at the same time. However, we had acceptable results running 4 PWM signals at the same time.

The next step was interfacing with motors, and implementing a code to control a DC motor. The `motor_controller.py`[7] depends on the GPIO library. It contains methods for motor control, including rotating motor in clock wise, counter clock wise directions, stopping motor, braking and speed control.

Next was the interfacing with the ultrasonic module. We made `ultrasonic.py`[8] to interface with the ultrasonic module. We followed this guide[9] on how to interface with the ultrasonic module. The sensor readings were off by 5-10 cms.

---

[3] https://www.raspberrypi.org/documentation/configuration/wireless/access-point.md
[4] http://snippi.com/s/pdsgqm8
[5] https://github.com/raspberrypi/linux/issues/2110
[6] https://github.com/ece-mohammad/RCCar-EmbeddedLinux/blob/master/overlay/root/RCCarProject/gpiolib.py
[7] https://github.com/ece-mohammad/RCCar-EmbeddedLinux/blob/master/overlay/root/RCCarProject/motor_controller.py
[8] https://github.com/ece-mohammad/RCCar-EmbeddedLinux/blob/master/overlay/root/RCCarProject/ultrasonic.py
[9] https://pythonprogramming.net/gpio-input-raspberry-pi-tutorials/

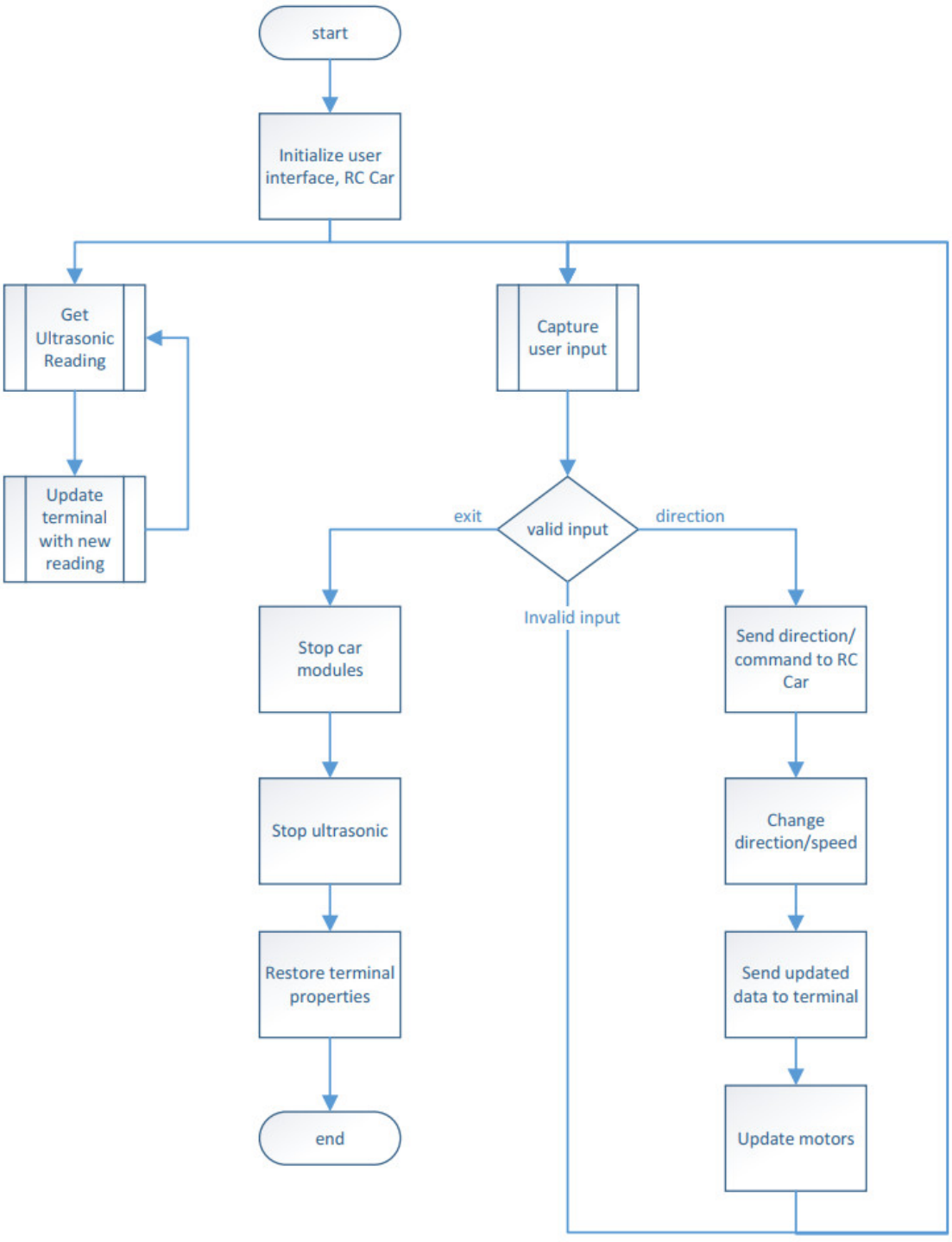
3. User Interface

A user interface that captures user input once the user logs through ssh to the raspberry-pi. Since remote ssh sessions are opened into a terminal, we chose `curses`[10] to handle user input.

The user interface[11] shows to the user how to control the car, information about the car (speed, turn rate, distance to object in front of the car). It also ignores any other key presses. The problem with this method is that, curses changes how the current terminal properties (closes echo, doesn't append line terminators to the end of each line), so if the program is interrupted, the current terminal properties are not reverted back to their initial values, the terminal must be restarted to get back to its usual state.

[10] https://docs.python.org/2/howto/curses.html
[11] https://github.com/ece-mohammad/RCCar-EmbeddedLinux/blob/master/overlay/root/RCCarProject/custom_term.py

4. RC Car Logic



5. Car Design

Due to time constraints, we didn’t design a car body and opted to use an already made car body.

6. Project Integration

That was the last step, integrating everything together and make it ready to be tested as a whole.

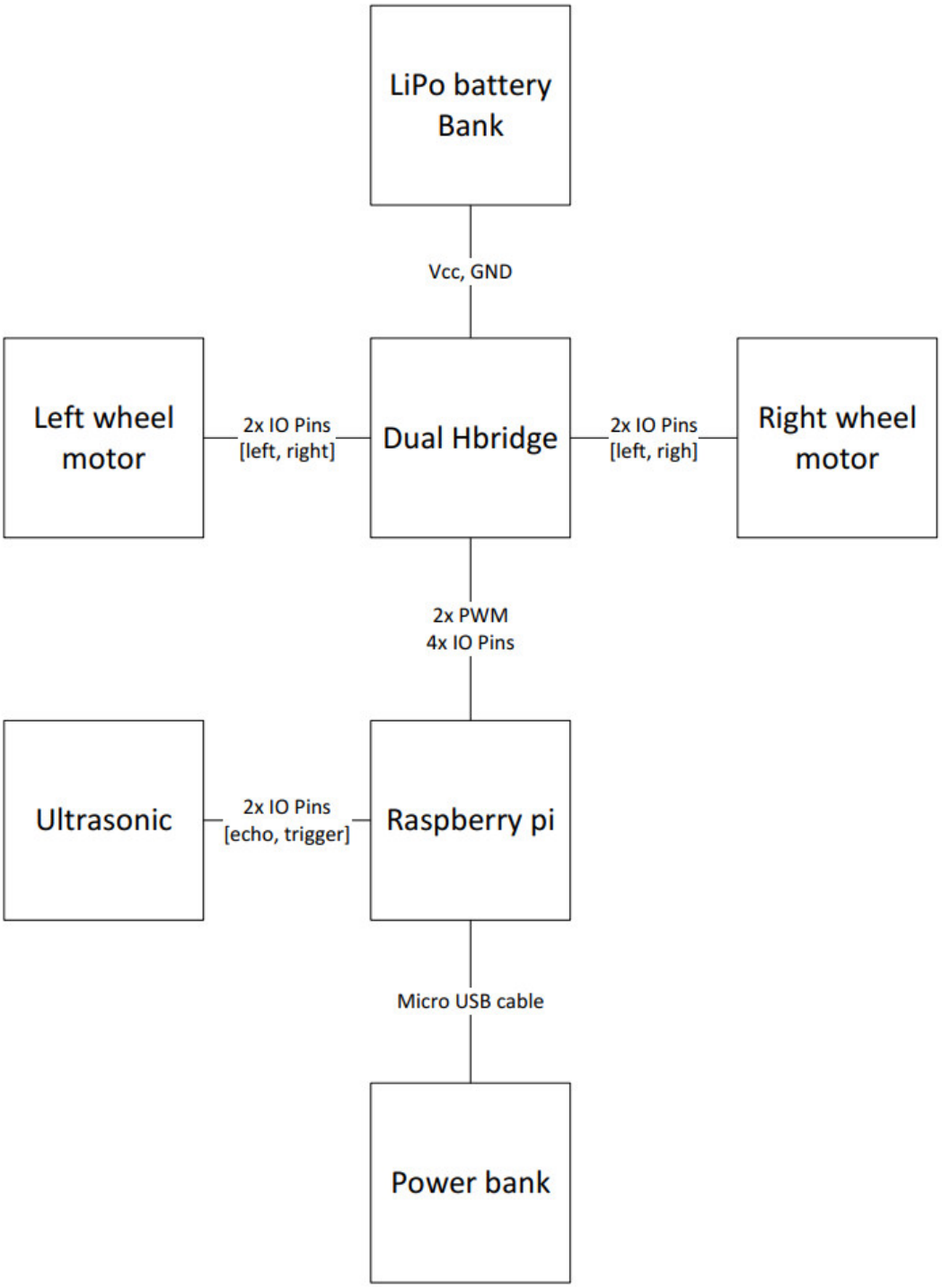
The hardware we used:

- 2 x DC motors.
- 1 x Dual H-Bridge module.
- 1 x Ultrasonic sensor.
- 1 x Power bank for powering raspberry pi.
- 3 x LiPo batteries for driving motors.

Each motor had 3 pins to control it, 2 pins to supply motor power and 1 pin to enable/disable hbridge output for that motor. The ultrasonic was powered from the raspberry pi.

The codes were copied to raspberry pi overlay folder, `/etc/profile` file was edited to run the main program after ssh login.

RC Car HW Blocks



Documentation

Project repository on Github contains the following:

1. Buildroot .config file used to build Linux image for the easbperry pi.
2. Overlay folder
3. Post build scripts
4. Documentation file.