Edgar Huckert V1.5 02-2020
Mail: edgar.huckert@huckert.com

Controlling a robot car using UDP on a Raspberry 3 Zero WLAN

Contents

Controlling a robot car using UDP on a Raspberry 3 Zero WLAN....1
Overview....1
Hardware components....1
Software components....4
Details for the server software....6
Details for the client software....6
Downloads....9
Alternative solutions and enhancements....9
Software....9
Hardware....9

Overview

This project shows how UDP over a WLAN can be used to control a small robot car with two motors. My first approach used a 433MHZ communication for the control: this proved to be however less reliable than the WLAN approach. The same is probably true for Bluetooth. The project uses a Raspberry 3 Zero WLAN ("R3ZW") board to control the motors via relays connected to GPIO signals.

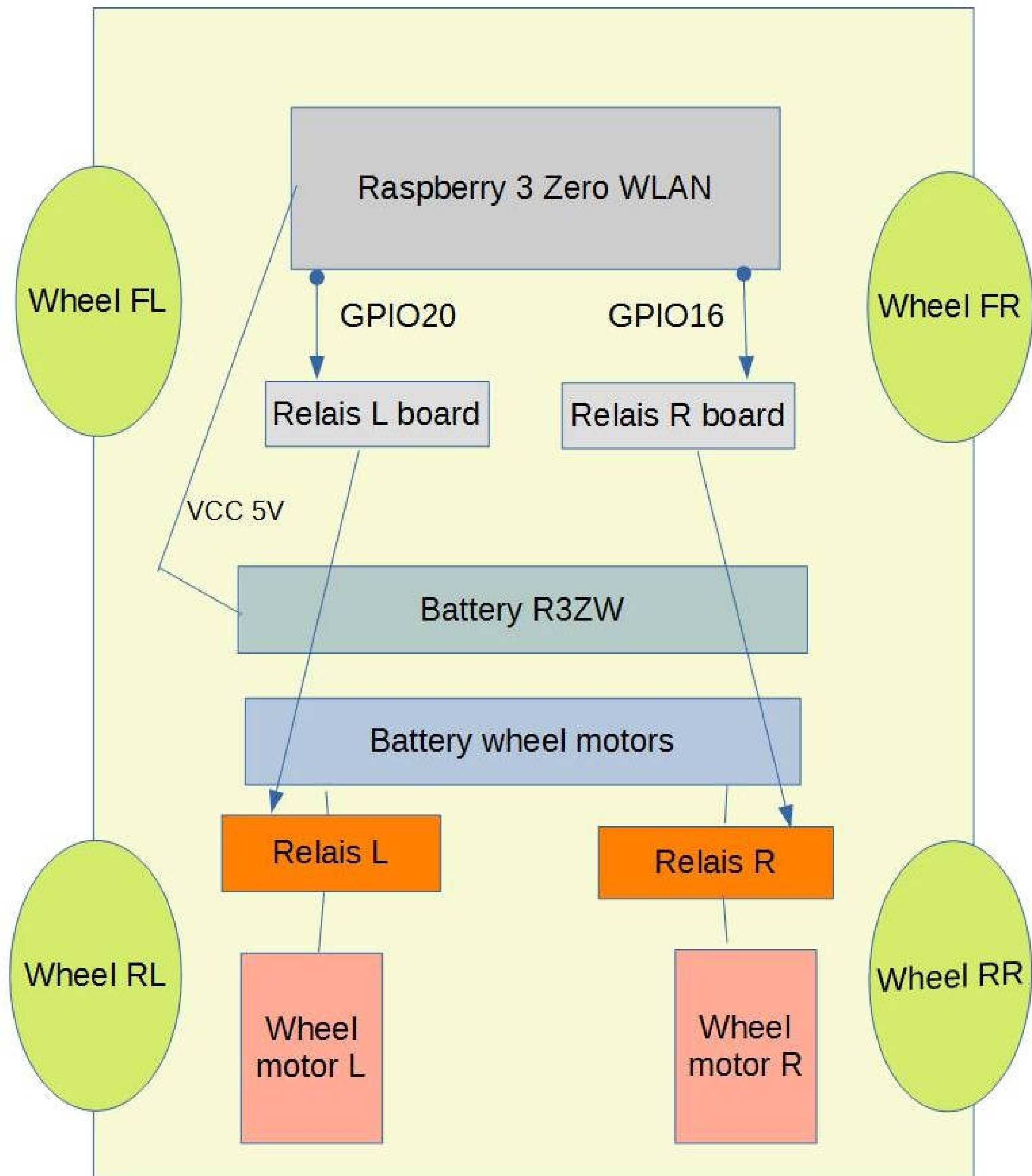
The robot car kit used here is a 4 wheel car with 4 DC driven motors. I used only the two rear wheels and the corresponding motors. The front wheels are not mounted on a separate axis. The behaviour of the robot car ressembles thus more to a military tank than a car vehicle.

The article offers downloads (source code) for simple client and server software. A portable GUI client (including a GUI exeutable for Windows) can also be dowloaded.

Hardware components

This picture shows the **hardware components**:

Controlling a robot car via Raspberry3 Zero WLAN using UDP



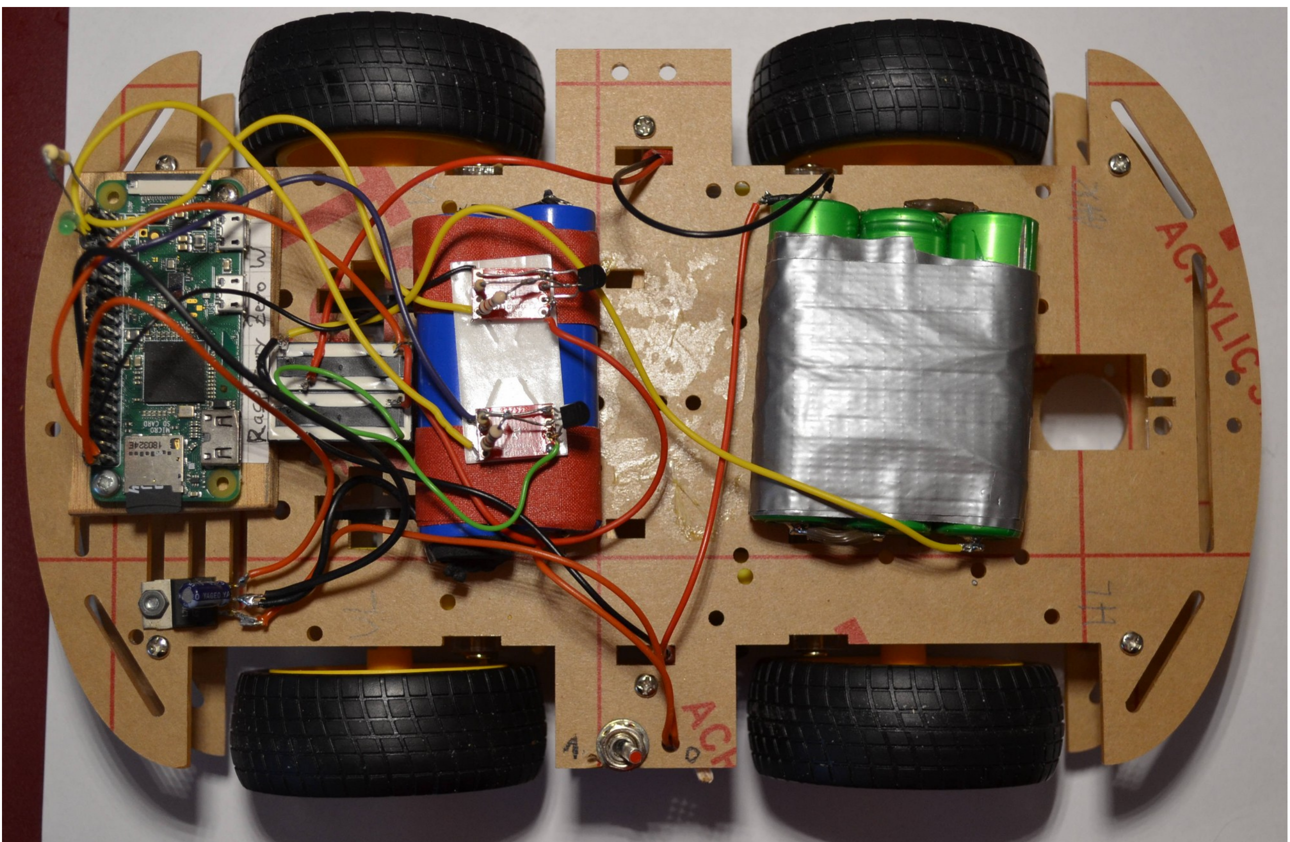
Hardware components needed:

- Raspberry 3 Zero WLAN (normal Raspberry 3 will also work)
- 4-wheel robot car: has 4 motors (only two needed)

- 2 5V relays
- 2 small DYI relay control boards using MosFet 2N700 (or similar)
- 5V regulator type LM2805 or similar
- battery for the Raspberry 3 ZW (approx. 6-12V)
- battery for the motors (5-12 V)
- switch turning on the Raspberry 3ZW 5V power

Note: this projects uses **only two of the four motors**. It is better not to mount the two front motors as they work like a brake in this project if no power is applied.

Here is a photo of my prototype (still "wild" wiring):



Component placements:

On/Off switch: on the bottom side of the picture, between two wheels. The switch is necessary to start or shutdown the R3ZW controller.

Raspberry 3 Zero WLAN: on the left (front) side

Relays: between the R3ZW and the blue battery. They just open or close the wires connecting the motors and the red battery block.

Relay boards: glued on the blue battery (two small red rectangles)

Voltage regulator: below the R3ZW on the front side, delivers 5V regulated from the blue battery.

Motors: only two DC motors for the rear wheels (right picture side). The motors are not visible.

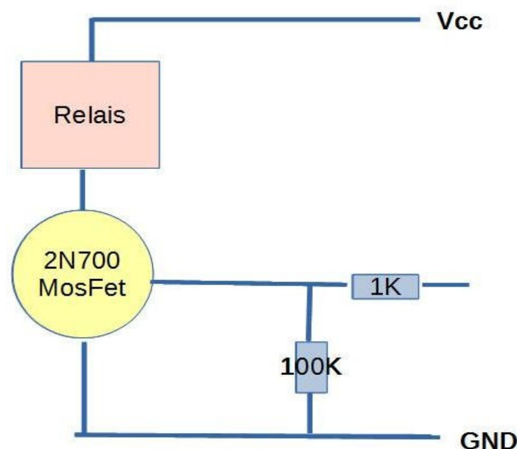
Battery blocks: blue block: for the R3ZW, green block: for the motors.

Additional notes:

The batteries are rather **accus** like those found in laptop devices. They are connected in series. The R3ZW expects exactly 5 Volt (or 3.3V). This is why the battery voltage must be regulated by a 3-pin regulator like a LM2805. My experience with a 5V power bank was very disappointing: the small power banks have internal step up regulators that often cannot deliver enough current (100-200 mA) for the R3ZW.

There is no need to regulate the power for the motors . The three green accu cells on the photo deliver ca. 11 V - but one cell less should also work.

The two DYI **relay control boards** (two small red boards above the front wheels) are extremely simple. They use per board a small MosFET 2N700 and two resistors (100K and 1 K). The R3ZW GPIO pins are connected to the 1K resistors. The scheme is here:



Commercial relay boards can also be used. It is essential that the relays operate at 5V and support some Ampères for the DC motors.

Software components

This picture shows the **software components** - a classical client server model:

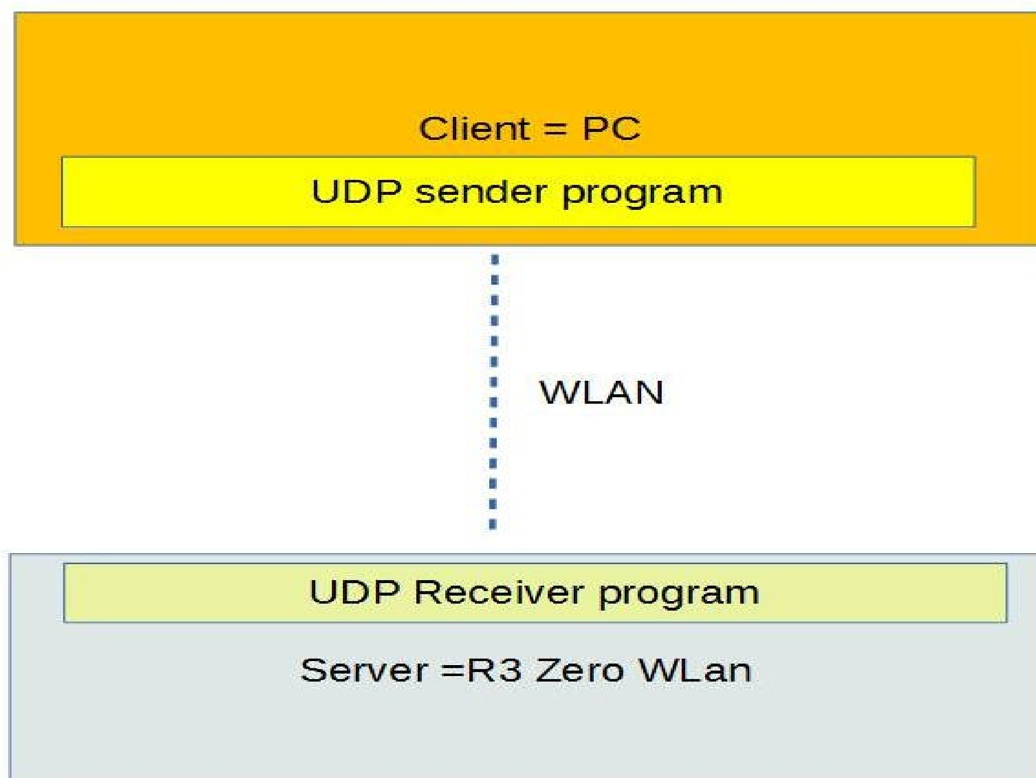
There are two software components:

1. a **server program** running on a Raspberry R3 Zero WLAN. This program accepts control string via UDP and translates them into relay controls using GPIO20 and GPIO16
2. a **client program** running an a PC (Windows or LINUX) sending simple control strings via UDP

All components are written in C or C++. Both programs can be compiled on the command line - no IDE is needed.

The **client** side may also be written in Python, Basic or any other programming language offering UDP access. There should be not problem if you use other operation systems than Windows on the client side: Mac/OSX or Unix/Linux will also work. An instruction how to compile the executable can be found in th comments at the beginning of the source code.

On the **server**/Raspberry side the **library libbcm2835.a** (or its equivalent shared version) is needed. The development package for this library (load the newest version form the Internet) offers also the **include file bcm2835.h**. An instruction how to compile the executable can be found in th comments at the beginning of the source code.



I didn't implement GUIs: bot components are simple "black window" console applications written in C++ (rather C).

Details for the server software

The server is a rather simple C++ program running under Linux in an endless loop. It performs these tasks:

- establish an UDP socket (read only)
- initialize the GPIO pins
- wait on port 9514 for incoming ASCII control strings
- parse them and change the GPIO levels accordingly

The server understands simple control strings like "10" or "11" or "00". The first character in each string is translated into a GPIO level for the right motor, the second character for the left motor. The strings must be terminated by LF.

This simple server program can easily be extended to accept larger control strings so that more GPIO pins can be controlled. This could be necessary to

- add more motors/relays
- change the direction of the robot car (inverse the DC direction)

The server program is called *udpGPIOServer*. It must be run with sudo or under user root as some library calls need extended access rights. My version uses the library *libbcm2835.a*. As the control of the GPIO pins is not time critical you could also use the well known mechanisms relying on the ***/proc*** files to change the GPIO levels.

It makes sense to start the server program in the startup sequence for the R3ZW. This can be achieved under Linux/Raspian with two alternative methods: integrate the startup in the startup control files for ***process systemd*** or integrate it in ***/etc/crontab***.

Here is a sample entry for */etc/crontab* (must be written in a single line):

```
@reboot root /home/huckert/bin/udpGPIOServer -v
1>/tmp/udpGS1.txt 2>/tmp/udpGS2.txt
```

This sample entry will start program *udpGPIOServer* at boot time and place its output in two files in directory */tmp*.

Details for the client software

Console solution

To test the robot car you can use any program that can send ASCII strings like "10" or 11" via UDP. I used one of my old UDP clients (called *udp_client)* to establish an UDP socket, read short control strings from a file and send them to the server. This simple UDP client can easily be modified to accept the control strings directly from the console instead of a a file. The actual client can be started like this:

```
udp_client -server 192.168.188.37 -port 9514 -i test.txt -ms 1000
```

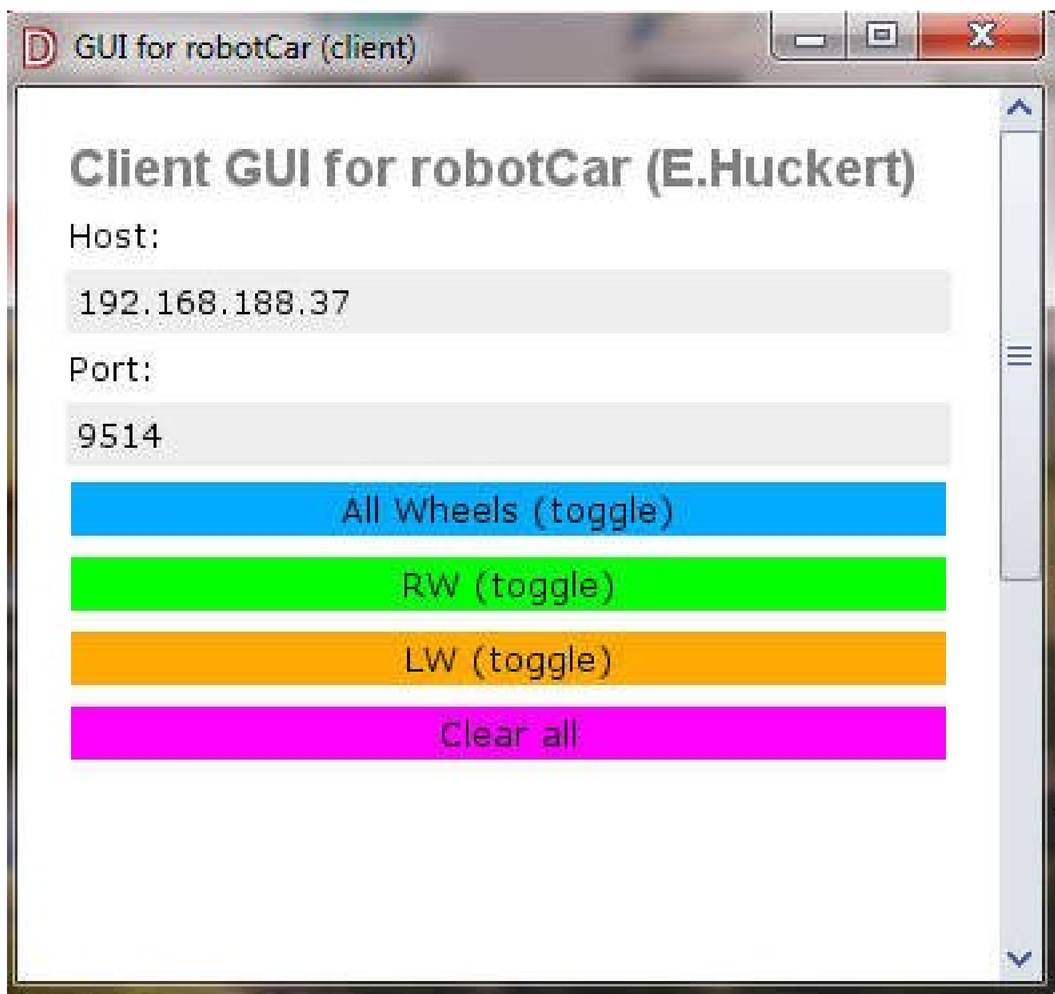
You must replace the server address (the IP address of the Raspberry) by the address in your enviroment. The port 9514 is hard coded in the server but can be changed.

I have derived from *udp_client.cpp* a second client that accepts directly control strings istead of reading them from a file. You will find this version called *udpCmdTx.cpp* in the downloads below.

GUI solution 1

I have written a very simple GUI client using programming **language D** with GUI library **dlangui** (by W.Lopatin). Using D with dlangui offers the advantage that this GUI can be compiled and linked on Windows and Linux without changes in the source code.

The command windows looks like this (screen dump):

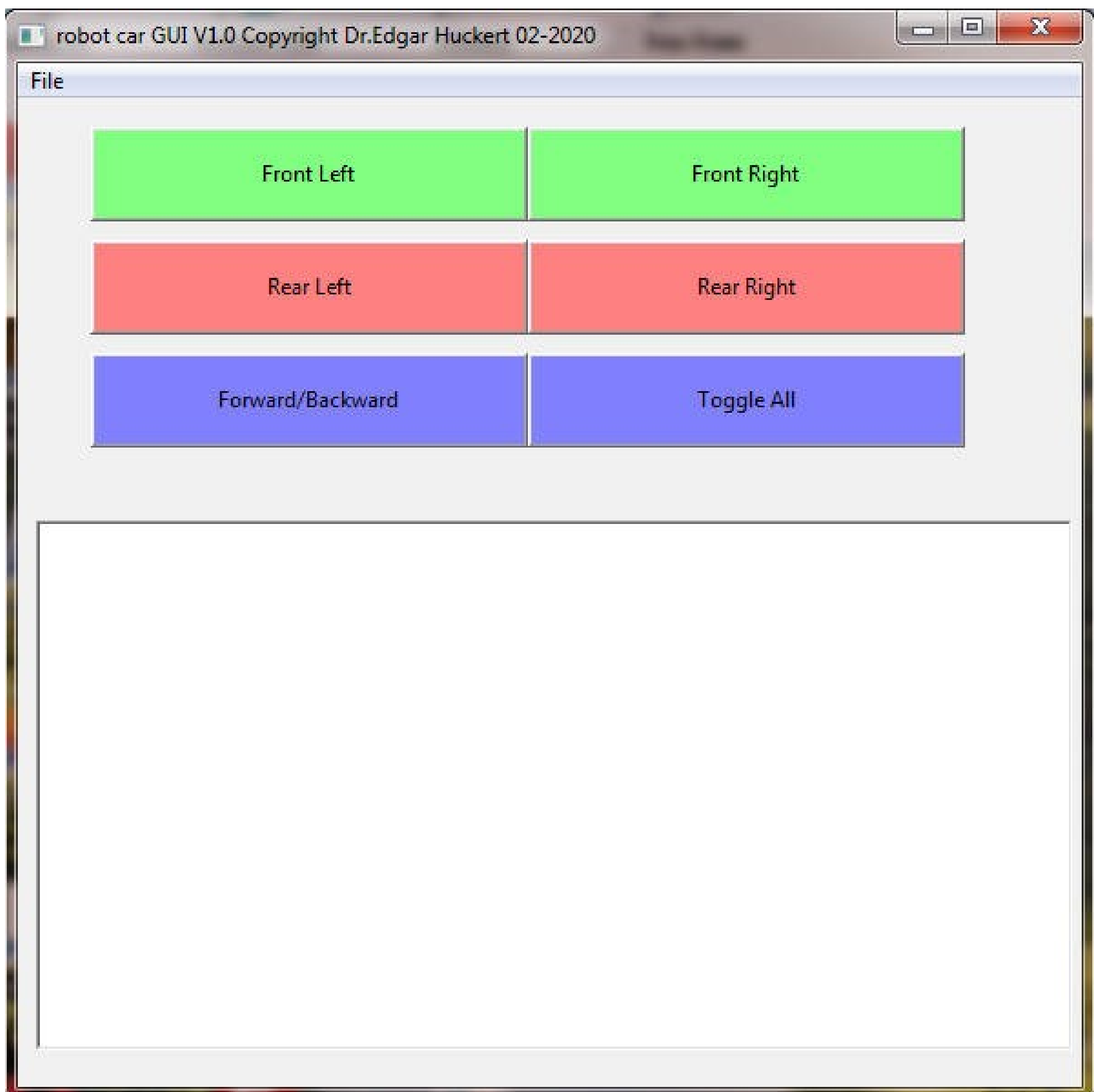


The 4 buttons are used to output strings as described above. The two entry fields can be used to enter the **IP address** and the **port number** needed for UDP.

Althound the source code for this GUI uses < 300 lines the build process (uses *dub*) may be a little bit difficult if you have no experience with D and *dlangui*.

GUI solution 2

I have written a second client in **C++ with wxWidgets**. This client is also portable between Windows and Linux. This second client is functionally more or less equivalent to the first client. The IP address and the UDP port can be specified in an **ini-file**. Here is a screen dump:



The source code, the make files, an ini-file and a Windows executable can be downloaded as specified below.

Downloads

The project specific software (sources + Windows executable for the GUI) I wrote can be downloaded from these links:

client (PC): www.huckert.com/ehuckert/programs/udp_client.cpp

www.huckert.com/ehuckert/programs/udpCmdTx.cpp

GUI 1: www.huckert.com/ehuckert/programs/robotgui.zip

GUI 2: www.huckert.com/ehuckert/programs/robotgui2.zip

server (Raspberry): www.huckert.com/ehuckert/programs/udpGPIOServer.cpp

Alternative solutions and enhancements

Software

Other software architectures can be implemented ressembling much the architecture choosen here. UDP is not necessary as it can be replaced by (implicit) WLAN TCP connections usually used in WEB applications:

1. the **client** could be a HTML mask in a **WEB browser**
2. the **server** could be a **WEB server** like Apache. The relais control logic could then be implemented via a simple CGI program.

This alternative offers a simple GUI on the client side.

Hardware

On the hardware side you could also use a processor like the ESP32 instead of the Raspberry 3ZW. This processor also has WLAN and some GPIO pins. My software for the server side must however be adapted for that processor.

Actually the robot car can only drive in forward direction. To change this the relays could be replaced by a a single **bidirectional motor driver** like the L298. Two additional GPIOs are then needed on the R3ZW side controlling the **drive direction** of the robot car. The same effect could be obtained with relays having two contacts instead of one.