# Ueberschrift

Wenwen Chen, Christian Dietz,
Jan Seeger, Rainer Schoenberger, Julian Tatsch

August 4, 2013

**Abstract**

## 1 Introduction

overview architecture bild

## 2 Simulator

After staring up the simulator, the car can be driven via the arrow keys. When pressing *C*, one can delegate the control to the built-in TCP server. The current state is shown in the upper left corner. When the control mode is switched to *TCP*, the simulator accepts commands via TCP/IP. The TCP/IP interface accepts the same commands as the servoboard. In this way, the RaspberryPi can be attached to the simulator instead of the real car. Also, the simulator does not have to run on the same machine as the control software. So for example, the RaspberryPi can be mounted on the car, but for testing purposes send the commands to the simulator instead of the servoboard.

In addition to a very simple physical driving model, the simulator also simulates the output, which would be given by our concept of a laser scanner. This output can be seen in the rectangle in the top of the simulator window. The red lines in front of the car depict the viewing angle of the camera, which is used for the laser scanner.

A screenshot of the simulator can be seen in Figure 2.

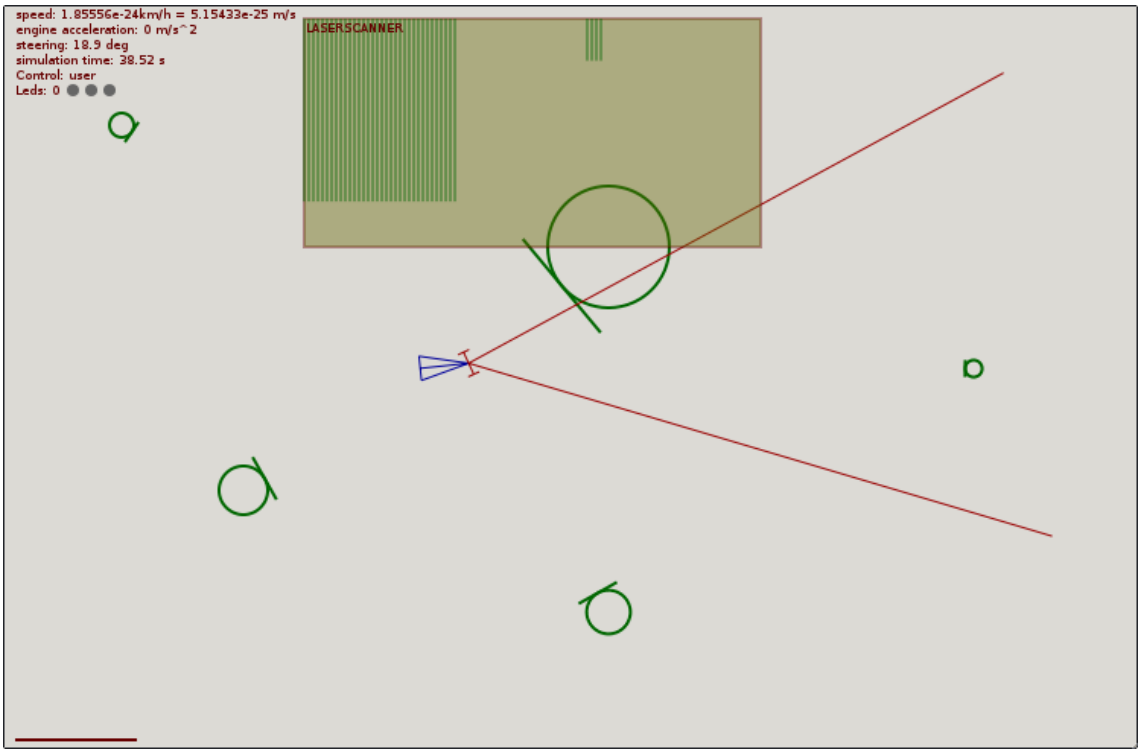


Figure 1: Simulator

# 3 RaspberryPi

The RaspberryPi is used for highlevel communication. It is connected to a wifi module via USB and opens up an access point. The RasPi is also equipped with a bluetooth module to commuticate with a Wiimote controller.

## 3.1 Software

On the RaspberryPi, there runs the raspi.exe program, which can be found in the *raspi* folder. It will start up a TCP/IP server and listen for a Wiimote controller via bluetooth. Over those interfaces, it will accept highlevel commands to control the car. Note that only one device is able to control the car at a time. There is a handover mechanism implemented in the *raspi.exe* program: If a device sends a *getperm*-command, it will take over the control

and commands from other currently connected devices are ignored.

The configuration of the raspi.exe program is done via the file *config.h*: SERVO_M defines, to which module the raspi software will send its commands. The possible values are:

- SERVO_SIM
- SERVO_BOARD
- SERVO_FPGA

If SERVO_SIM is set, *raspi.exe* will send the commands to a simulation program via TCP/IP. The IP and TCP port of the simulation program can also be configured in the *config.h* file. The *raspi.exe* will send its commands directly to the servoboard via I2C, if SERVO_M is set to SERVO_BOARD. The last option SERVO_FPGA causes *raspi.exe* to talk to the FPGA board via SPI.

If a Wiimote controller should be used to control the car, WII_ENABLED must be set to 1 in the *config.h* file. Communication via TCP/IP is always enabled.

Settings, which are more specific to one communication unit, can be configured in *servoboard.h, servosim.h* or *fpga_spi.h* respectively. Those files contain the parameters for the I2C or SPI specific communication, such as addresses, speeds or the paths to the chardevs.

## 3.2 TCP communication protocol

The *raspi.exe* TCP server accepts the following commands:

**servo set <channel> <valve>** Sets the servo which is connected on PIN <channel> to the corresponding value. <channel> must be an integer between 0 and 7. <value> is also an integer. It can have values from 0 to 8000, where 4000 would mean the zero position.

**servo led <onoff> <mask>** Sets the LEDs on the servoboard. <onoff> decides, wether the LEDs should be switched on(1) or off(0). The the three LSBs of <mask> decide, which LEDs should be switched. Only LEDs, where the bit in the mask is set to 1 will be switched on or off. For example, if one wants to enable the first and the third LED, one would send the command "servo led 1 5". Note that most of the LEDs

are by default also used by the servoboard firmware to signal various events. Once a LED has been manually set to a certain state by setting a bit to one in <mask>, this LED will not be used by the firmware anymore.

**servo getperm** This command initiates an control handover to the device, which is sending this command. From now on, *raspi.exe* will only accept control commands from this device. All other currently connected devices will be ignored.

**speedv set <speed> <steering>** This is a highlevel command, which is only accepted, if the Raspi is connected to the fpga. <speed> and <steering> are both double values.

## 3.3 Remote control devices

The car can be controlled by any device that supports TCP/IP over wifi. So for example it is possible to control the car via a notebook that is connected to the car's wifi access point. To demonstrate the remote control capabilities of the car's software, we implemented an Android client. Screenshots of this client can be seen in Figures 3.3 and 3.3.

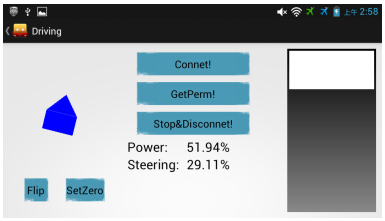


Figure 2: Android remote control client

Another option is to use a Wiimote remote to control the car. If the WII_ENABLED option is set in the *raspi.exe* configuration, the RaspberryPi software will periodically try to connect to a Wiimote controller. Button 1 and 2 need to be pressed simultaneously on the Wiimote controller to enter the discovery procedure. Once the Wiimote remote is connected to the RaspberryPi, the car can be controlled. The Wiimote has the following key assignment:

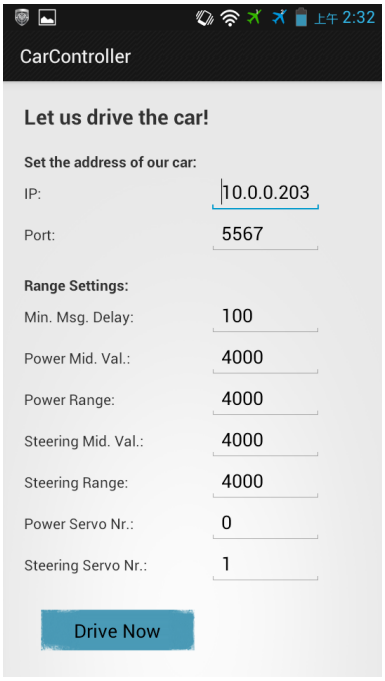


Figure 3: Android remote control client

\+ Accquire control over the car (initiate handover mechanism)

**A** Switch acceleration mode from *button* to *tilt* and vice-versa

**1** Accelerate slowly

**1+2** Accelerate

**2** Accelerate fast

**left** Accelerate backwards

**tilt left/right** Steer left/right

**tilt forward/backward** Accelerate forward/backward

# 4 Servo controller board

## 4.1 Schematics

## 4.2 Board

After the schematics have been created, the PCB board layout was designed. The components are placed on the board and connected with the corresponding copper tracks. This process is also called routing. Routing wasn't a big problem, as we had enough room on the board. Almost all tracks fitted on one layer and only a few vias and additional floating wires were needed. This made the PCB manufacturing process easier, as only a one-sided PCB could be used. Figure 4.2 shows the finished board layout, Figure 4.2 shows a 3D rendering of the servoboard and the finished board can be seen in Figure 4.2.

## 4.3 Communication protocol

The Servoboard is connected via the I2C or two-wire interface to a command unit. The command unit can either be the FPGA or the RaspberryPi.

The I2C protocol is not implemented in software, but the hardware I2C capabilities of the AVR Atmega8 microcontroller are used. This gives us faster response time and less overhead. It also unburdens the AVR CPU, as only one hardware interrupt is needed, each time a complede byte is received via I2C. Implementing I2C via software would have required many timer interrupts and might have turned out less stable.

We implemented our own communication protocol ontop of I2C. In the microcontroller's firmware, we used a simple state machine to realize this.

The communication protocol looks as follows: The Master first sends a preamble (0xff), which is used to synchronize both communication partners. This is necessary, because loosing packets or receiving corrupted data can lead to a situation, where the receiving unit ends up in an undefined or faulty state. Therefore, it might be that the communication gets stuck.

By sending a preamble, that cannot be part of the normal communication, this problem is solved: Everytime the microcontroller receives a preamble-byte, the commonication state machine goes back to its initial state, no matter what the current state is. This way, if a transmission error occurs, only the current command is lost, but further communication will still be possible.

Nevertheless, our communication protocol makes it possible to transmit 0xff inside a command using the following method: If values greater or equal than 254 are sent, they are split into two bytes. The first byte is 254 and the next one is the difference to the desired number. On the receiver's side, both values are then combined and their sum determines the acual number. For example if 255 needs to be sent, the two bytes 254 and 1 are sent.

### 4.4 Firmware

# 5 Powerboard

Very early in this project, we noticed, that supplying the various boards and modules on the car with power becomes a problem. This is why we designed a simple board to distribute the power among all components. The board also has a central on-off switch for all of the needed voltages. The schematics are very simple an can be seen in Figure 5. Figure 5 shows the finished board.

# 6 Rotary Encoders

To measure speed, we built two rotary encoders into the car. They are based on a simple and cheap light barrier with an included Schmitt trigger and a rotary encoder disc. A picture and a schematic of the light barriers can be seen in Figure 6.

There are four slots in each disc, so that we can measure the speed at the resolution of roughly a quater of a turn. The discs are mounted on shafts, which are directly connected to the differentials. We use two rotary encoders. One is connected to the front differential and one is connected to the back differential. The cardan shaft of the initial 4WD car design has been removed to make room for the rotary encoders. Because of the removed 4WD and the two rotary encoders, it is now possible to measure the speed of the front and the back wheels independently. This enables us to detect slippage, as only the back wheels are driven by the motor.

Figure 6 shows the aluminium component that has been manufactured to hold the front rotary encoder shaft in place. Figures 6 and 6 show, how the rotary encoder discs are mounted on the car.

To connect the light barriers to the FPGA, we built a sensorboard. It has the task to supply the light barriers with power and to convert the output

voltage of the light barriers to the right level. There are also two LEDs on the board, that show the current state of the light barriers. This made it easier to debug the light barriers when moving the rotary encoder discs into position. The schematics of the sensorboard can be seen in Figure 6. The board layout and a 3D rendering can be seen in Figures 6 and 6.

# 7 Conclusions

We worked hard, and achieved very little.

# 8 Future work

# References

[1] J. Y. Gil. LaTeX $2_\varepsilon$ for graduate students. manuscript, Haifa, Israel, 2002.

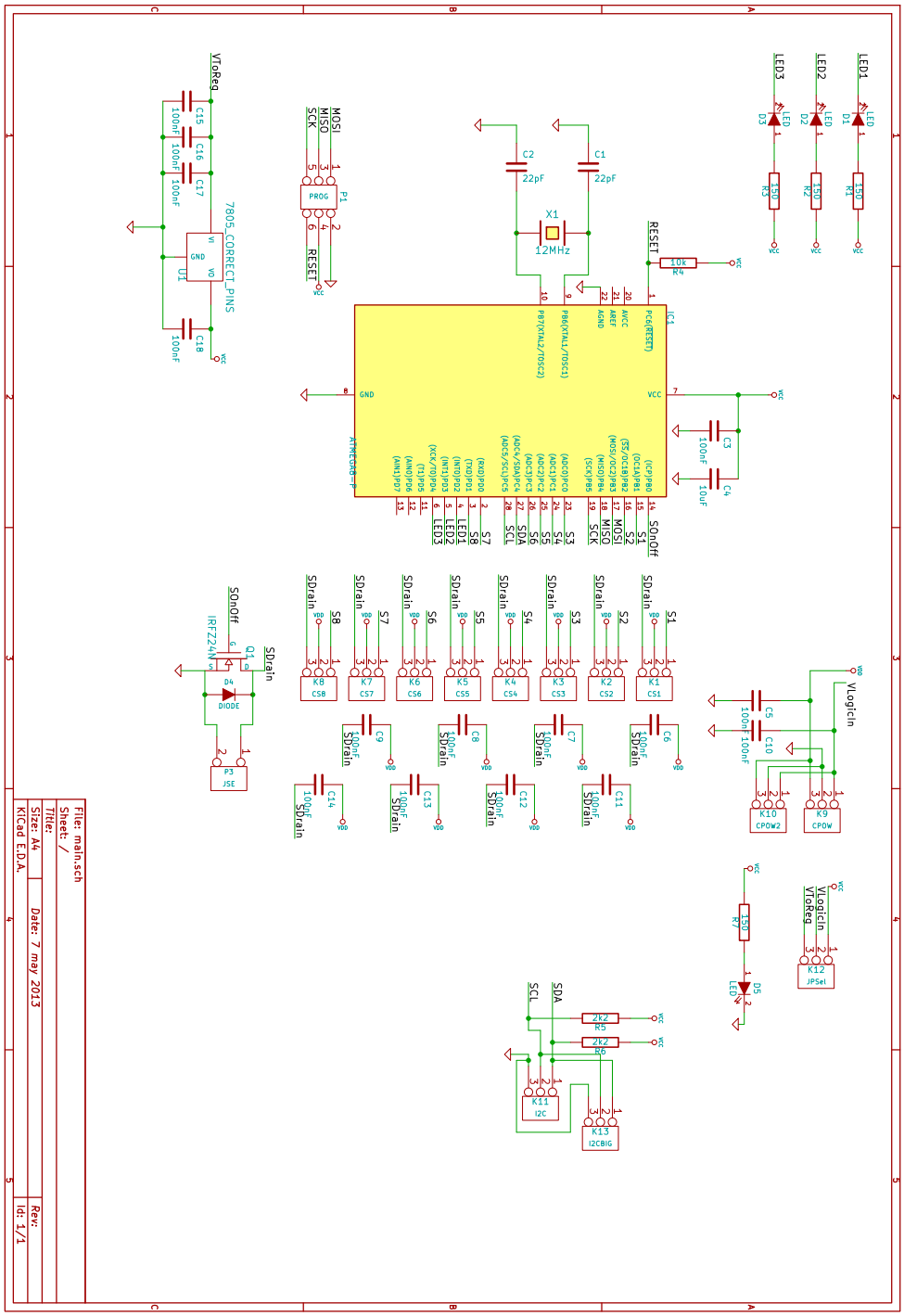




Figure 4: Servoboard schematics

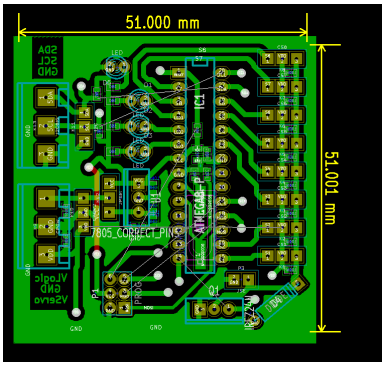


Figure 5: Servoboard

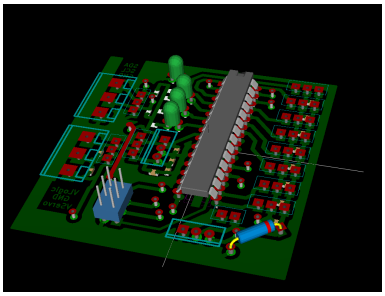

Figure 6: Servoboard

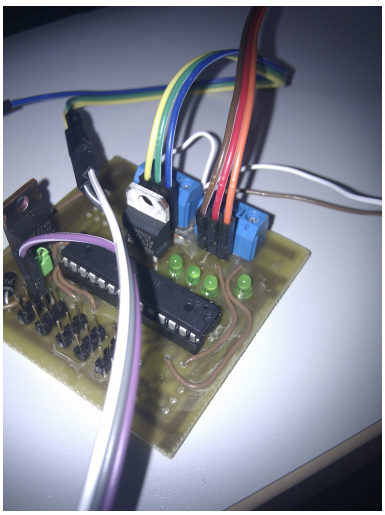

Figure 7: Servoboard

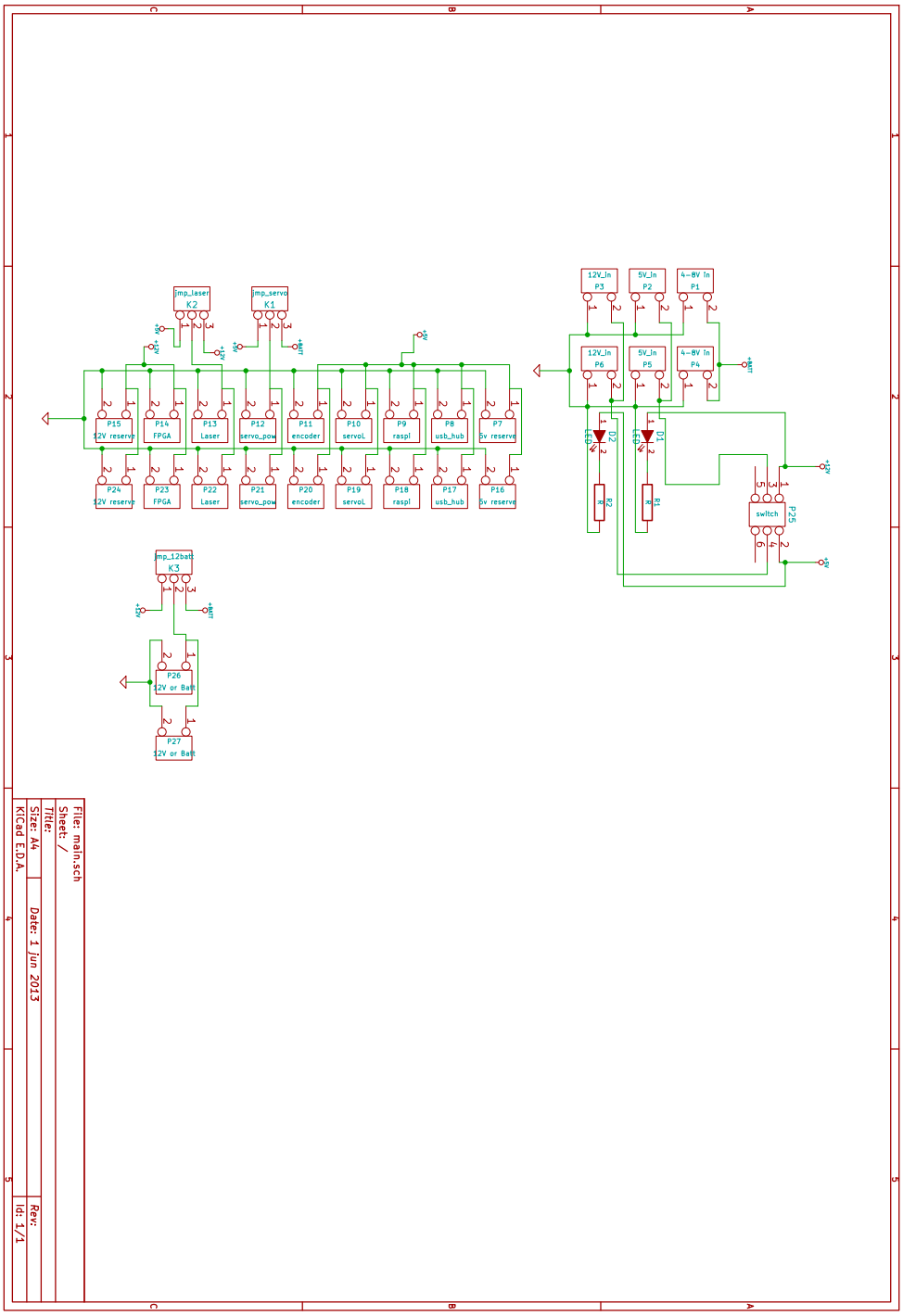




Figure 8: Powerboard schematics

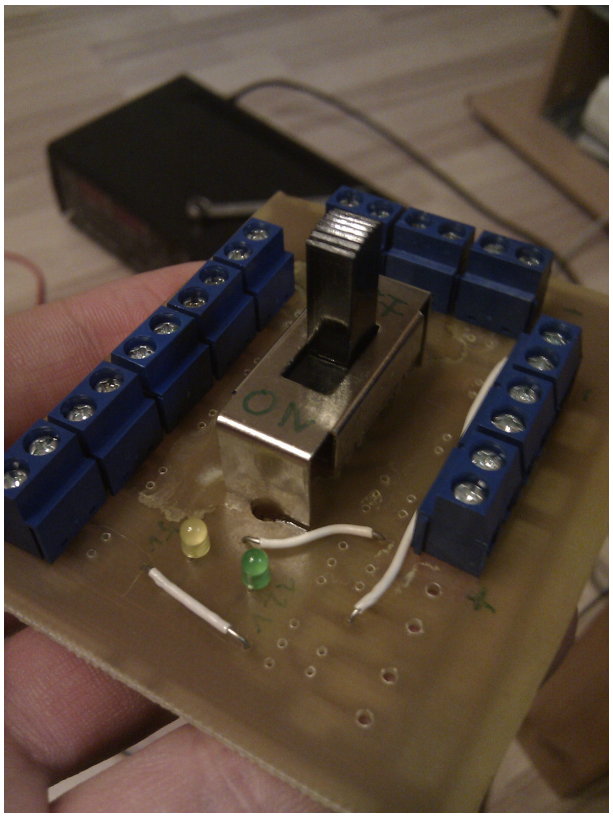

Figure 9: Powerboard

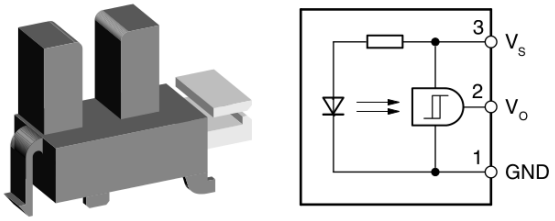


Figure 10: Light barrier

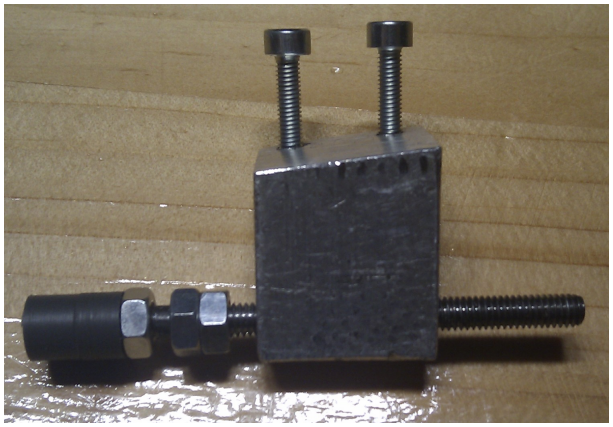

Figure 11: Encoder mounting

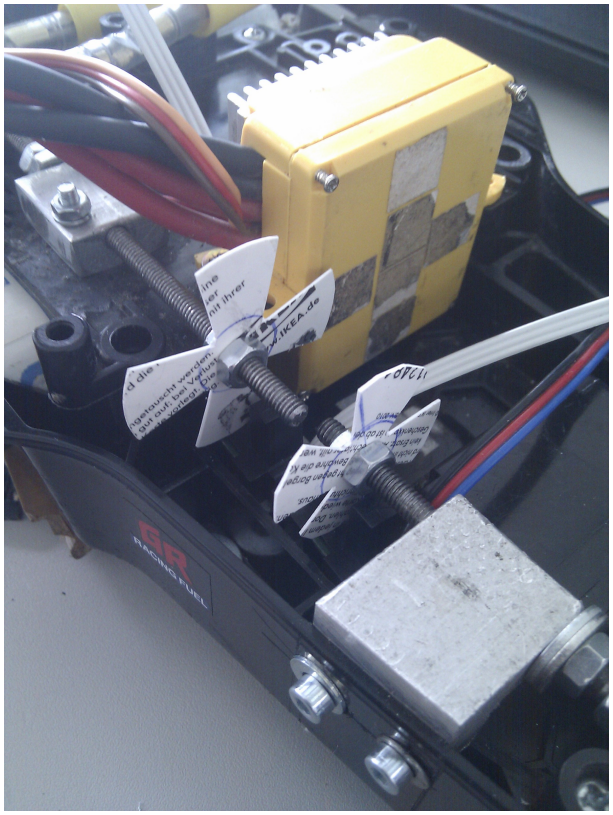

Figure 12: Encoder mounting

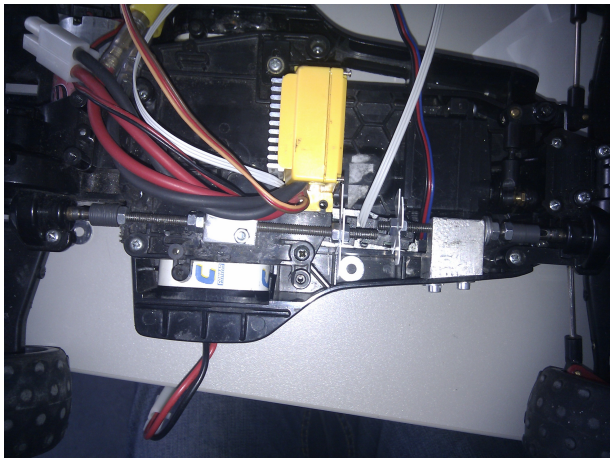

Figure 13: Encoder mounting

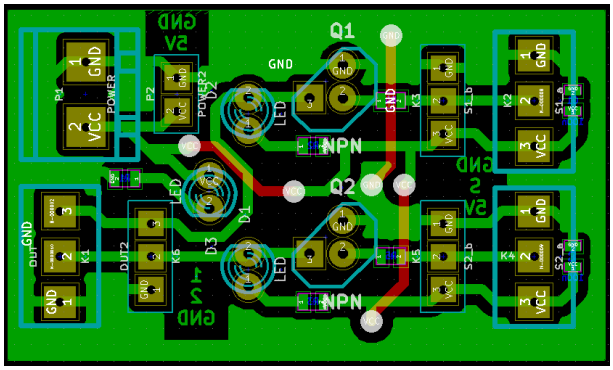


Figure 14: Sensorboard

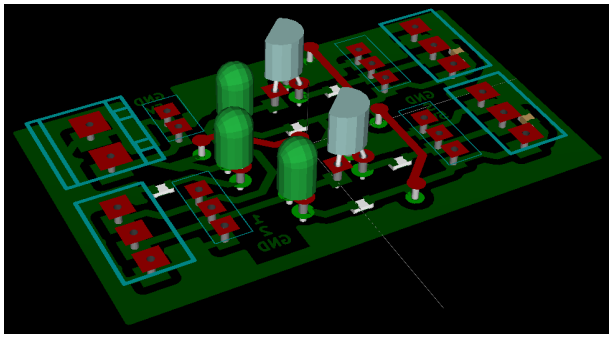

Figure 15: Sensorboard

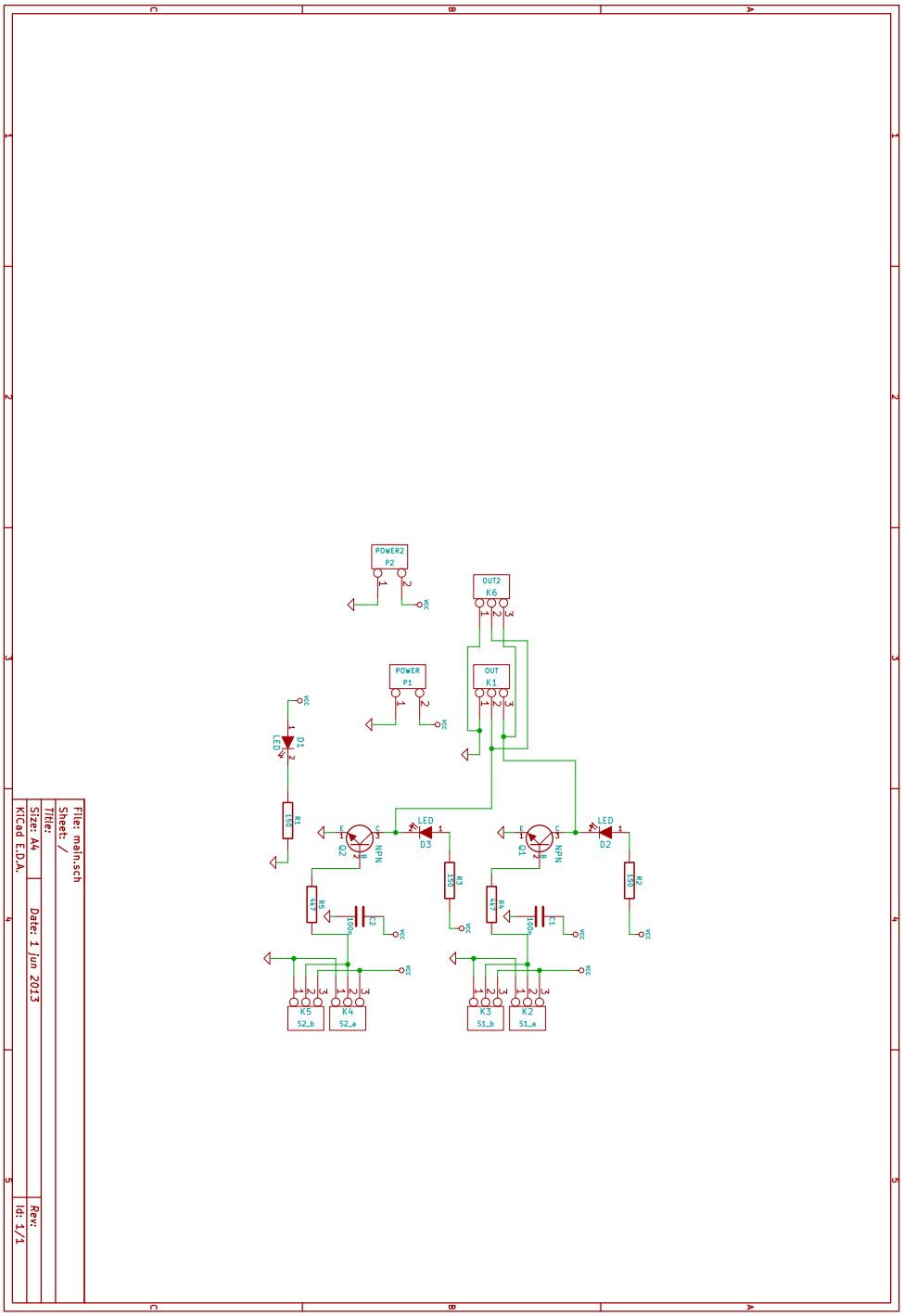




Figure 16: Sensorboard schematics