# Lab Course: Hardware/Software Co-Design with a LEGO Car

**Abdallah Attawia, Moritz Dötterl, Heiko, Berkay**
Technische Universität München, Garching, Department of Informatics
Boltzmannstr. 3, 85748 Garching, Germany

This document explains the work done on a lego car to detect and follow a line autonomously. Connection and communication between the hardware parts is explained. Furthermore, collision avoidance and line detection algorithms are thoroughly described.

## I Introduction

An autonomous car is one that can drive from point A to B independently without any human input. It can accelerate, brake and steer itself as well as sense the environment and navigate to avoid crashing. The development of autonomous vehicles is rapidly growing and a lot of big and small companies are working towards making autonomous driving a reality.
In this project a small scaled model, a lego car, along with some cheap components are used to test a real autonomous driving function, i.e. following a line on the street. This documentation starts with a brief explanation of the different hardware components. Then the cabling and communication between the components is described. Lastly, the collision avoidance technique and and the line detection and following are explained. The code, attached to this report, carries out the functions explained here and is clearly documented.

## II Hardware Parts

The hardware components used are two controllers, the DEO-Nano and the Raspberry Pi, two types of actuators, namely speed motors and servo motors, as well as two types of sensors, which are the ultrasound sensors and the USB-camera.

### II.A DEO-Nano

The DE0-Nano board is a compact-sized FPGA based development platform manufactured by terasIC. Its heart is a Cyclone IV FPGA from Altera. Furthermore it features on board memory, LEDs, Buttons, and most importantly three GPIO headers. On this FPGA we flashed the provided Nios-II image. This is an image of an

adaptable processor allowing for custom specializations such as multiple hardware UART handlers. Now it is possible to wright code for this processor in C using the hardware you desire, without any trade off. We are using four UART units to connect to three ultrasonic sensors as well as to the Raspberry Pi. Furthermore we use two PWM modules with each two channels allowing us to generate PWM signals at four pins. Those are used to control the drive motor and the steering servo.

### II.B Raspberry Pi

For this project the newly released Raspberry Pi 3 is used. Raspberry Pi is a small-sized one board computer. In this work it is used to connect the camera with the Nano-board and to run the line following code.
First, the micro SD-card for the Pi is prepared with the Raspbian Operating System installed on it. Then the Pi is boots the Raspbian and the computer is connected to the Raspberry Pi using Secure Schell (SSH) either via the Ethernet port, or using the build in Wi-Fi of the Pi 3 and the LRZ Wi-Fi network. Then the camera is connected using one of the USB ports and OpenCV is installed to run the line following code. Finally UART communication is enabled to connect the Raspberry Pi with the Nano-board. In Raspberry Pi 3 you need to disable Bluetooth in order for UART communication to work. To disable onboard Pi3 Bluetooth and restore UART over GPIOs 14 and 15, modify the file "/boot/config.txt":

- sudo nano /boot/config.txt
- Add to the end of the file:
  "dtoverlay=pi3-disable-bt"

### II.C Actuators

- Speed motors
- Servo motor

### II.D Sensors

- Ultrasound sensor
  The ultrasound sensor measures the distance from any detected object within its range. As shown in Figure 1 the ultrasound sensors detects objects by sending out ultrasound waves and receiving an echo, if an object is detected. The sensor measures the time of flight between transmitting and receiving the signal and calculates the distance according to the following formula (1).

$$L = \frac{340(m/s)x \triangle t(s)}{2} \tag{1}$$

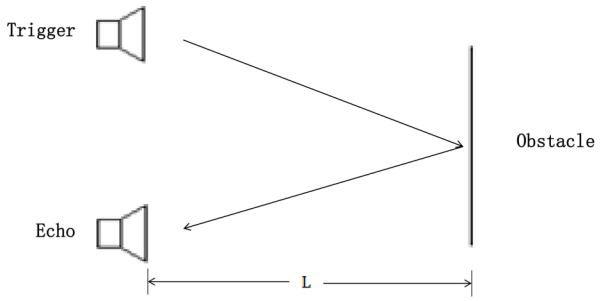


Figure 1: Theory of operation of the ultrasound sensor

In this course, the KS103 ultrasound sensor is used. It has a range up to 11m which can be specified by sending a command from the Nano-board. This sensor can operate using two digital communication protocols; I2C and Uart. It has an accuracy between $1mm$ and $10mm$

In order for the ultrasound sensor to operate, it is first initialized and set to the desired range. Like the raspberry pi, it communicates with the Nano-board using uart communication, which has to be initialized as well. The Nano-board then sends a command to the sensor to send the data using uat. The Nano-board finally receives date from the sensor. The data read from the sensor is multiplied times 170 to get the distance from detected objects in micrometer.

- USB Camera

# III Cabling

# IV Communication

# V Collision Avoidance

In order to avoid collisions, an algorithm based on the idea of the adaptive cruise control is implemented. Adaptive cruise control is a cruise control function that allows the vehicle to adapt its speed according to the traffic and to maintain a safe distance from the vehicles ahead.
In real cars, a long range radar sensor is used to detect the cars, here we use an ultrasound sensor to measure the distance from the vehicles ahead. The following algorithm is implemented to avoid the collision with other cars or objects and to adapt the speed to the surrounding traffic.

- if distance from ultrasound sensor $\leq$ a minimum set distance ($distance_{min}$)
  -car stops

- if distance from ultrasound sensor $\geq$ a maximum set distance ($distance_{max}$)
  -drive with maximum speed

- else (distance lies within $distance_{min}$ and $distance_{max}$)
  -adapt speed according to distance:

$$CurrentSpeed = MaximumSpeed * \frac{distance}{distance_{max}} \quad (2)$$

The speed of the car is set according to the aforementioned algorithm, where the current speed is the duty cycle that is assigned to the speed motors.

# VI Image processing for line detection and following

The task of the whole project was to build a Lego Car that can autonomous follow a line on the ground. For this purpose we are using an ordinary web cam on the front of the car giving us an image of the ground up to approximately 20 cm in front of the car. We developed an algorithm that detects the line in the image and calculates how the car needs to steer in order to stay on that line. The calculation is entirely done on the Raspberry Pi using the image processing framework OpenCV. The following sections explain how the image processing works and how we optimized it for the Raspberry Pi.

## VI.A Code

## VI.B Optimization

### VI.B.1 Sequential Improvements

Having this basic concept of the algorithm implemented we needed to test it of course. But our first tests where not so successful, the car lost the line quite often. To prove our concept we replaced the Pi with an ordinary laptop, using a USB UART adapter. With this setup the

algorithm turned out to be working really good, and the car stayed on the line every time, even with the driving motor running at full speed. The main difference between those two setups is the computation power, which is obviously way higher on the laptop than on the Raspberry Pi. After some performance analysis we figured out that the computation of a single frame takes around 4 ms on the laptop and 170 ms on the Raspberry Pi. We also figured out that our camera gives us 15 frames per second (FPS), so a new image is provided approximately every 66 ms. This means that we are only processing every third image on the Raspberry Pi and that is the reason for the poor performance. So there are two things to be done: first of all shorten the calculation time per image and second make sure every frame of the camera is processed and therefor really used, so we are no longer wasting information. Improving the calculation time also improves our reaction time, since thats exactly the time it takes to get an image of an event (e.g. an approaching turn) and to calculate what the car should do. So the faster the computation time is the faster the car can drive. Ensuring every frame gets calculated does not improve the reaction time, but it improves the overall throughput and therefor every 66 ms a new steering angle would be present. This also improves the feasible speed of the car.

To save computation time we disabled the software Gaussian Filter. It turned out that it can be easily replaced by changing the focus of the camera, so it produces a blurred image. That is exactly the same thing the Gaussian Filter would do, but in hardware rather than in software. It can be seen, that a sharp image without Gaussian Filter makes the algorithm very vulnerable to noise, but as soon as the focus of the camera is changed the algorithm recognizes the line without problems.

Another optimization is that the search range is restricted. Before the whole row of the image was searched for the highest gradient, now only the area of the X-Position found in the row below $\pm 80$ Pixels is searched. The highest gradient is assumed to be on the line. This assumption is reasonable, since the line always is continuous and won't change so fast. This method only works if the pixel found in the lowest row of the image is on the line, and not noise. In reality this turned out to not be a problem since if there is a correct line on the image it needs also to be in the lowest row of the image and it is identified correctly.

Furthermore only gradients grater than 25 are considered. This helps to recognize if the line has left the image to either the left or the right side of the image. In those cases it is possible that a row has no pixel on the line and therefor no gradient grater than 25. In this case the coresponding line gets neglected, saving further computation time due to less points on the line. Even more important is that this makes the algorithm less vulnerable to noise in the case that the line leaves the image to one of the sides. Without this any random noise above the point where the line leaves the image would be interpreted as the line and affect the steering calculations in a bad way.

With all these improvements we managed to bring the computation time down to approximately 100 to 110 ms per frame, which is pretty good, but still not good enough to capture all frames. But that approach only uses one core of the four cores the Raspberry Pi provides.

#### VI.B.2 Exploit Parallelism

Since it is not possible to catch every frame of the camera with just one thread on the Raspberry Pi we need to exploit the parallelism of the Pi. So we first tried to make OpenCV use OpenMP to parallelize it's computations. That seemed to work, it brought the computation time per frame down to 80 ms. But that is still not faster than the 66 ms latency of the camera. So still further improvements need to be done. But since this implicit parallelism seems not to work well together with the explicit parallelism described in the next part we had to deactivate the OpenMP version again.

So we developed our own way to make use of the four cores the Raspberry Pi has. The idea is to use multiple threads that each grab a frame, process it and propagate the resulting steering angle to the UART output and the Nano board. With one computation taking 100 to 110 ms two threads are sufficient to calculate each image. The way this works is as follows:

The main thread creates two worker threads and sleeps then. The worker threads grab the next frame generated by the camera in a synchronous way, meaning one image is always processed by only one thread. Now the thread processes the image and calculates the steering angle. After that the thread performs a check whether the calculated frame is still in the same order than the sequential stream of images. If not, e.g. the calculation of this frame took to long and the next frame has already finished, it is neglected. Usually this is not the case, since the frames all approximately take the same amount of time to be computed. Than the worker thread wakes the main thread through a condition variable and propagates the calculated steering angle. While the worker waits for the next frame of the camera the main thread generates the UART output and sends the calculated steering angle to the Nano board. The main thread starts to sleep again, while the worker thread continues to process the next image.

With this technique the worker threads alternately grab the images provided from the camera, all frames are used and no information is wasted. Since the Raspberry Pi has four cores this method could also be run with four worker threads instead of just two. In our case, there is nothing to gain by using four threads rather than two, since the camera only produces 15 FPS. So for example with three threads one thread would always be waiting, so the algorithm degenerates to the two thread version. But if a different camera with 30 FPS would be used it would be helpful to use four worker threads. This way even all the 30 FPS could all be calculated and used for the line detection. Figure 2 shows how all images of the camera can be calculated using two threads. Note that the time between two images and the time for calculating one image is on scale. For simplicity this shows only a part of the whole program and only the necessary communication. It can be seen that after each computation there is enough time for the thread to wait for the next image.

## VII Conclusion

To conclude, the car followed a closed track without getting out of line in both directions. The maximum set speed was about 85% of the maximum possible speed. With a few modifications on the camera, e.g. a higher frame rate of 30 FPS, or a wider viewing angle, or a higher computation power on the Raspberry Pi, the full motor speed could had been reached.

## Acknowledgments

## References

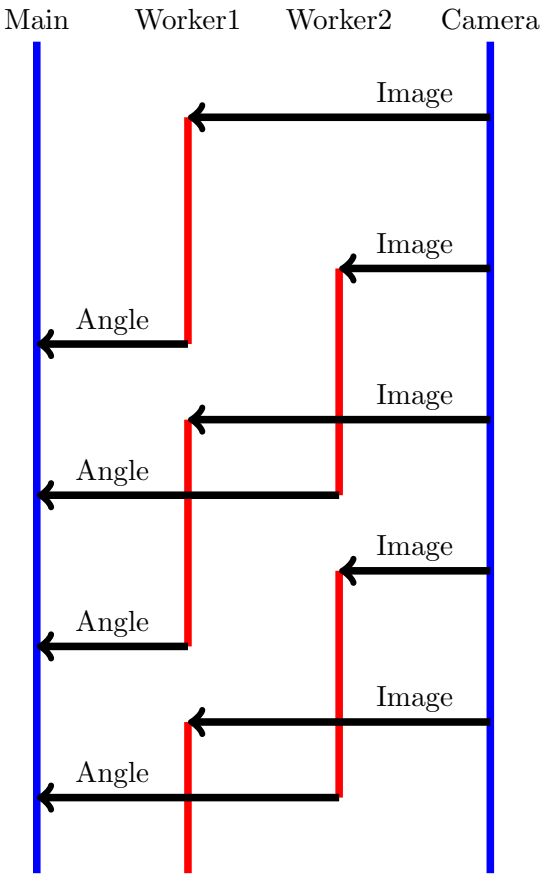


Figure 2: Multithreaded calculation