Zybo Autonomous Car

Cătălin-Constantin Bitire
catalin.bitire@gmail.com

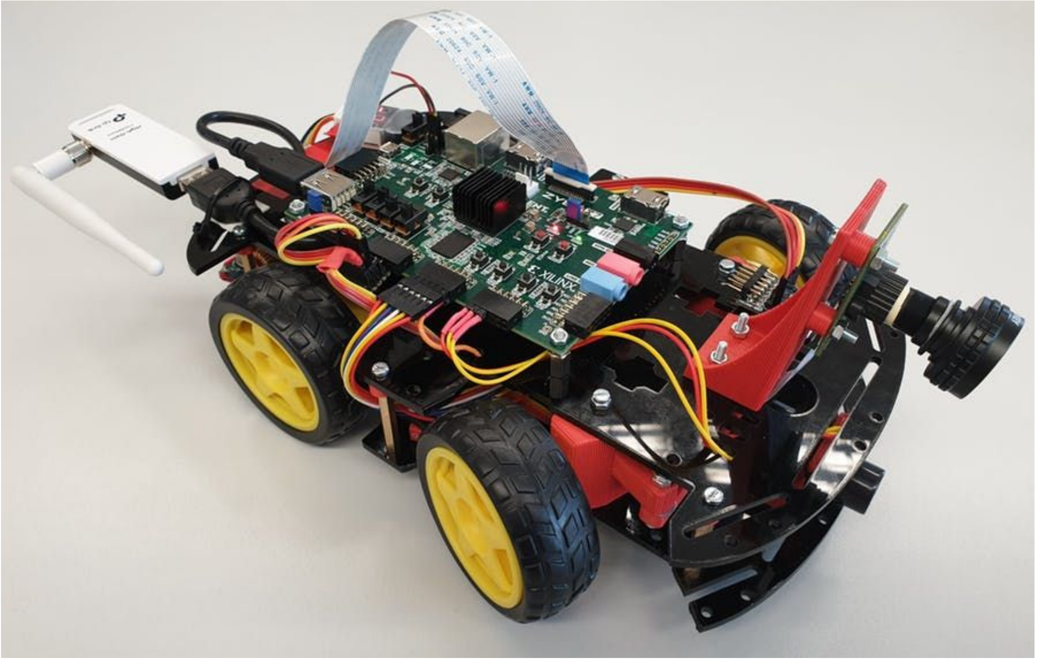
Marin-Mircea Cojocaru
mircea.cojocaru97@gmail.com

Andrei Ionașcu
andrei.ionascu.97@gmail.com

Submitted for the 2019 Digilent Design Contest Europe
05.05.2019

Advisor: Decebal Popescu

University Politehnica of Bucharest
Department of Automatic Control and Computers



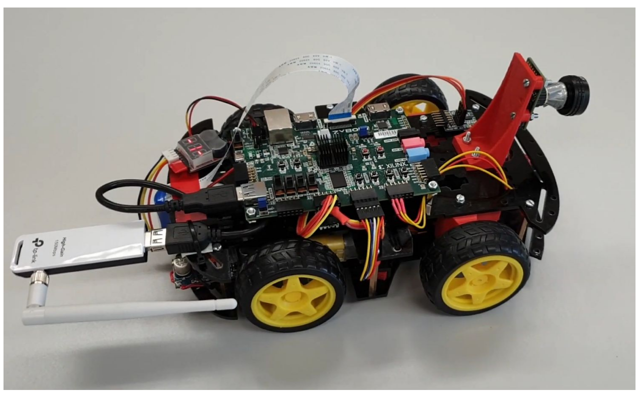
Autonomous Cars

Vehicles of the future

Product Description

An autonomous car is a vehicle that is capable of sensing its environment and moving with little or no human input. It combines a variety of sensors to perceive their surroundings, such as radar, sonar, GPS, odometry and inertial measurement units. Advanced control systems interpret sensory information to identify appropriate navigation paths, as well as obstacles and relevant signage.

Our product can be used as a base platform for a complex autonomous vehicle type of product, offering an accessible way to extend its functionality with additional components (sensors, actuators, lighting systems, etc.) based on its intended application. Our work will spare you the effort of creating the backbone of an autonomous platform, letting you focus on the custom functionalities that highlight your final product.



Features Available

- Physical platform – power and steering
- Computer vision
  - Lane detection
  - Sign detection
- Distance awareness and measurement
- Smart road signs detection
- Battery level and usage monitoring
- Remote setup and control over WiFi

Reasons to go autonomous

Green machines - optimization to ensure fuel consumption is as efficient as possible
Safer streets - potential for human error considerably reduced
Time is money - saved time with precalculated actions
Space savers - compact infrastructure

Quotes on autonomous vehicles

"Self-driving cars are the natural extension of active safety and obviously something we should do"
- Elon Musk

"My guess is that in probably 10 years it will be very unusual for cars to be built that are not fully autonomous."
- Elon Musk

Images are copyright to their respective owners[1]

[1] https://en.wikipedia.org/wiki/Waymo#/media/File:Waymo_self-driving_car_front_view.gk.jpg

Table of contents

Introduction

Abstract

The project describes an autonomous car able to drive along a lane and react to road signs, obstacles and smart RFID markers, powered by OpenCV and Embedded Linux. Our work can be used as a base platform and extended with sensors and functionality depending on its use case in a commercial application.

Objectives

We implemented an autonomous vehicle on a Zybo Z7-20 platform, using a Digilent PCAM as its main camera to capture the road and road signs ahead, with an additional array of sensors including an accelerometer, a sonar and an RFID scanner to gather additional data on its course.

As we approached the completion of our planned goals, we figured out that the project we're building can be more than just a demonstration of a lot of different components working together to create an autonomous vehicle – it can be a starting point for anyone that wishes to experiment with these two fields (i.e. computer vision and self-driven cars) to build anything that should be able to drive alongside human operated cars or along other smart vehicles, for example: autonomous delivery vehicles, fire & rescue fast response vehicles, public transport, etc. Our platform can be fitted with any number of sensors and expanded to fulfill any task, having a strong backbone.

Features-in-Brief

The project can be split into several different features which will be described in more detail further in this document. These features are:

- Physical platform – power and steering
    - VHDL drivers (Software – low-level hardware interface)
    - Electronic drivers (low-level hardware – physical actuators interface)
- Computer vision
    - Lane detection
    - Sign detection
- Distance awareness and measurement – front facing Sonar
- Smart road signs implemented as RFID cards – weather and lighting independent
- Data acquisition and logging – Accelerometer
- Battery level and usage monitoring

The main component/feature that unifies all of the above and provides a modern and relatively easy way to add additional modules is the Embedded Linux running on the Xilinx SOC, Xilinx Petalinux, highly customized and integrated with the hardware we developed, now providing an interface for programmers and developers to build applications and to integrated other hardware with our system. The specific implementation will be discussed in detail further in this document, as it is central to our project.

Project Summary

Our project is very complex, involving a great deal of both hardware design and software design, such that it will be presented in two different parts, each with its own submodules.

Hardware Design (not including the physical platform and mounting hardware)

The hardware design centers around the ZYNQ processing system and the Xilinx SOC on the Zybo board, to which we added the following modules:

- Video pipelines:
  - One general purpose video pipeline feeding from the onboard camera to the processing system’s memory via a VDMA, streaming 720p 60fps video
  - One highly specialized video pipeline that applies a set of hardware-accelerated video processing algorithms on the images from the camera, used in the Lane Detection and Lane following control modules (detailed further in this document)
- AXI4-Lite driver for the Motors and Steering, providing a way to signal and control these two centerpieces of the physical assembly
- AXI4-Lite driver for the front-facing Sonar
- I2C interfaces:
  - Camera control interface (on the CSI connector)
  - General purpose Processing System I2C bus (used with the RFID and Accelerometer components)
- AXI4-Lite interrupt manager for the RFID component (based on AXI GPIO)
- AXI4-Lite camera software-defined reset & enable (based on AXI GPIO)
- AXI4-Lite RGB led driver for debug and demonstration purposes, used to signal different events visually

Software Design

The software design centers around the Xilinx Petalinux embedded Linux distribution, to which we added and integrated the following modules:

- Custom kernel modules to provide userspace interfaces to the backing hardware modules:
  - **/dev/motors** – kernel module that manages the motors side of AXI4-Lite driver for the motors and servo listed in the hardware design section
  - **/dev/servo** – kernel module that manages the servo side of the AXI4-Lite driver for the motors and servo listed in the hardware design section
  - **/dev/video** – kernel module that provides a fast video stream access to the general-purpose video pipeline via its VDMA
  - **/dev/videoHLS** – kernel module that provides a fast video stream to the specialized, lane-detection oriented, video pipeline via its VDMA
  - **/dev/sonar** – kernel module that provides readings of the sonar’s reported distance
  - **/dev/rgbled** – kernel module that manages the debug/signaling RGB LED module

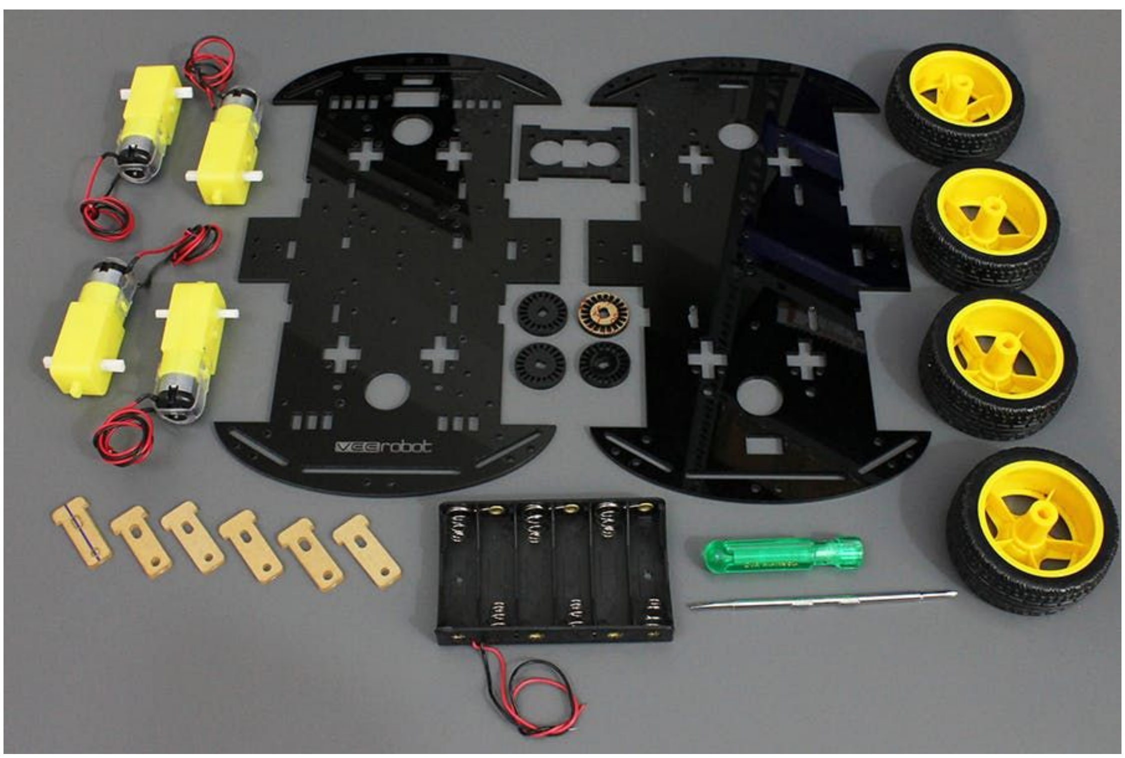
- Cross-compile ready, modified drivers for the TP-Link Wi-Fi USB Adapter based on the **RTL8188EU** chipset, optimized for performance (reduced power saving options) and made able to be integrated in the Petalinux work-flow.
- MJPG Streamer application that provides a web-based video feed off one of the video pipelines – based on the original Git repository[2] but modified to be able to be integrated with the Petalinux work-flow.
- The main control algorithm that makes use of all the sensors and modules onboard to provide the desired functionality to display relevant information to the user if prompted to do so.

Physical platform and mounting hardware

The hardware platform is built around a plastic car frame on 2 levels, with all the additional mounting hardware and supports being designed in CAD and 3D printed out of PLA plastic. The frame also includes 2 electric brushed DC motors with ample torque to power the car at a decent speed. The steering system resembles that of a go-kart (Ackerman steering), with the servo pushing one wheel hub that also transfers the motion to the second one via a pushrod. The steering system is fully 3D printed and requires minimal assembly. The frame itself requires additional holes and mounting points as per your personal needs, depending on how you want to place the different components/sensors.

A short description (images provided for reference) of all the notable components in this subsection:

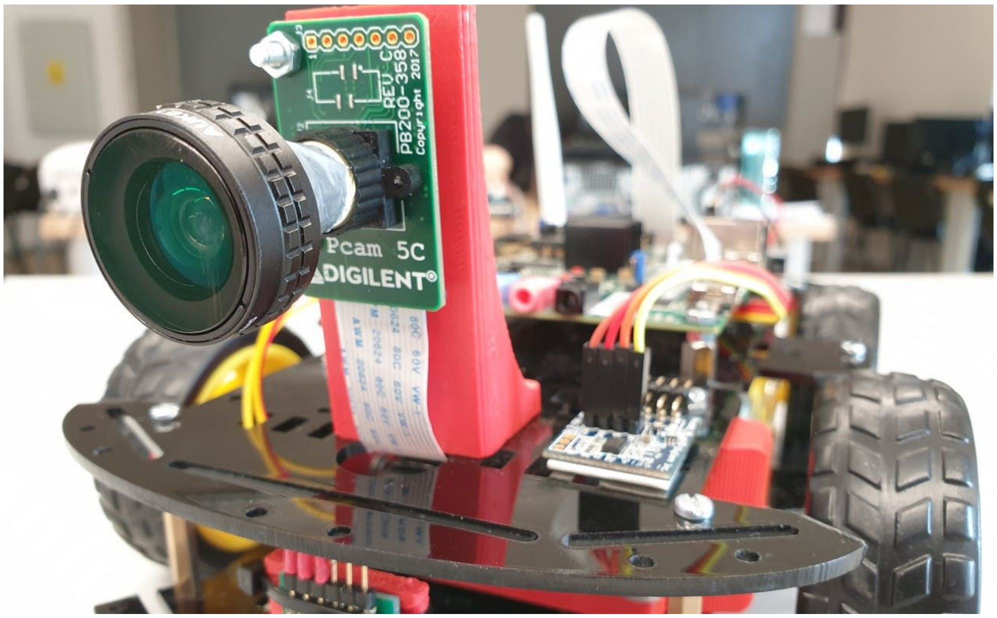
Frame, motors and steering



[2] https://github.com/jacksonliam/mjpg-streamer

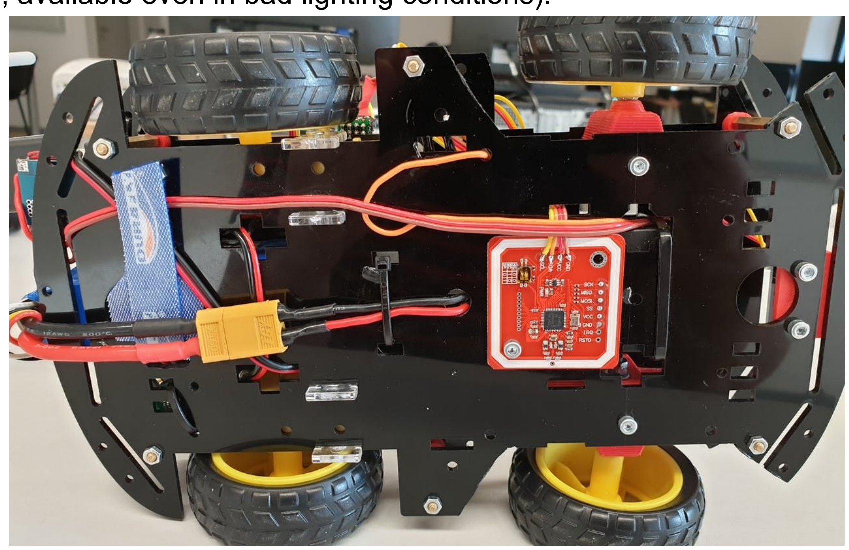
Camera and camera mount

The PCAM is mounted at the front of the vehicle, on a 3D printed stand that provides the optimal viewing angle and height. We used a fisheye lens with the original one to extend the view field, as required by the lane detection algorithm.



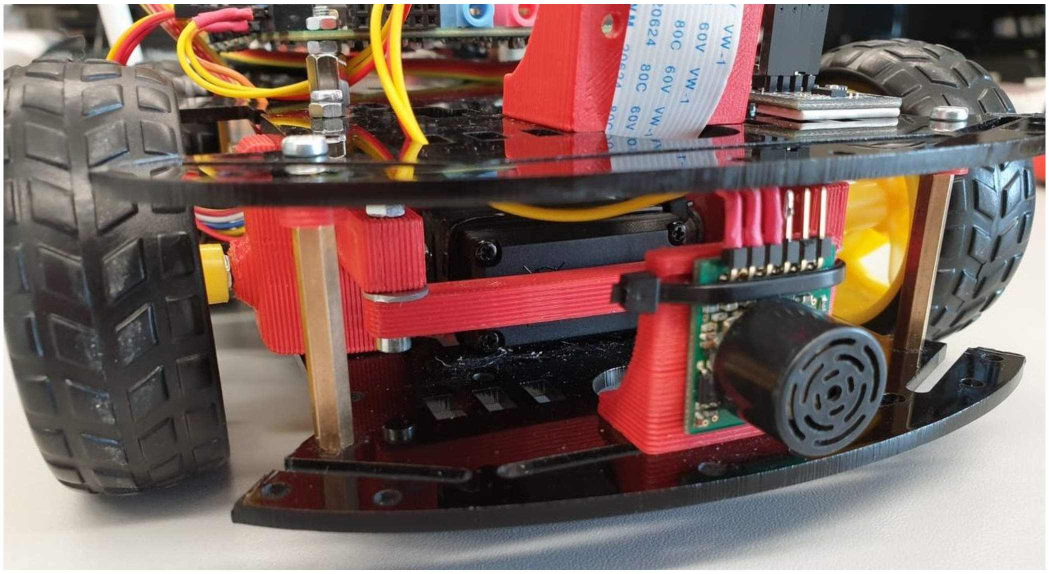
RFID Scanner

Under the vehicle, between the front wheels is an RFID scanner device connected via I2C to the main board that is used to detect and read RFID tags/cards/stickers on the ground to provide info to the car about the road, such as speed limits and road signs (both as a failsafe for the camera and as a secondary system, available even in bad lighting conditions).



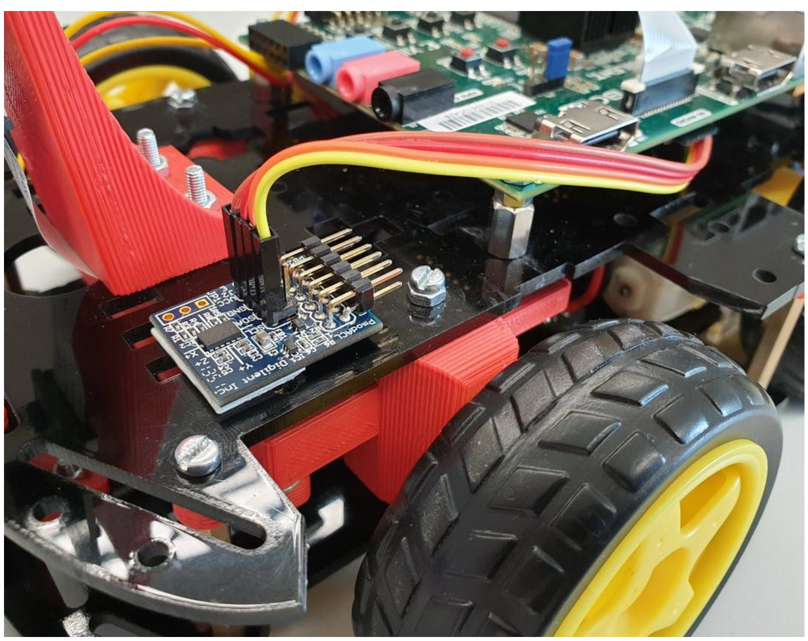
Sonar

The sonar is placed up front, and its purpose is to detect obstacles on the road and to provide a sense of distance to the processing algorithm.



Accelerometer

The accelerometer is mounted on top of the vehicle and is used to measure the current speed of the car by periodically sampling the acceleration data in 3 axes via I2C and running an algorithm on them.



Digilent Products Required

- Zybo Z7-20 board
- Pcam 5C camera
- Pmod MAXSONAR sonar
- Pmod ACL I2C accelerometer

Other Hardware Required

- TP-Link WN722N USB WiFi Adapter[3]
- High speed & high torque metal gear servo
- Pololu DRV8835 Dual DC motor driver[4]
- Pololu D24V22F5 5V Voltage regulator[5]
- 6V Voltage regulator to power the motors and the servo
- NXP PN532 board RFID Reader[6]
- 2200mAh Li-Po battery
- Plastic 4WD Car frame with DC motors - pictures above[7]

Tools Required

- 3D Printer (even small ones should be enough, as long as you do not wish to print the frame)
- Soldering Iron/Station and accessories
- Hot glue gun
- Screwdrivers, electric drill, general purpose tools for assembly and maintenance

Design Status - complete

[3] https://www.tp-link.com/us/home-networking/usb-adapter/tl-wn722n/
[4] https://www.pololu.com/product/2135
[5] https://www.pololu.com/product/2858
[6] https://www.adafruit.com/product/364 or alternative
[7] https://veerobot.com/store/MCH-CHS-RBCH-125

Background

Why This Project?

One of the main driving forces in the industry for the fields of image processing and automatic control systems is the concept of an Autonomous Vehicle capable to do everything that a human driver can do and more, like automatic navigation, better environment awareness and improved safety algorithms and systems.

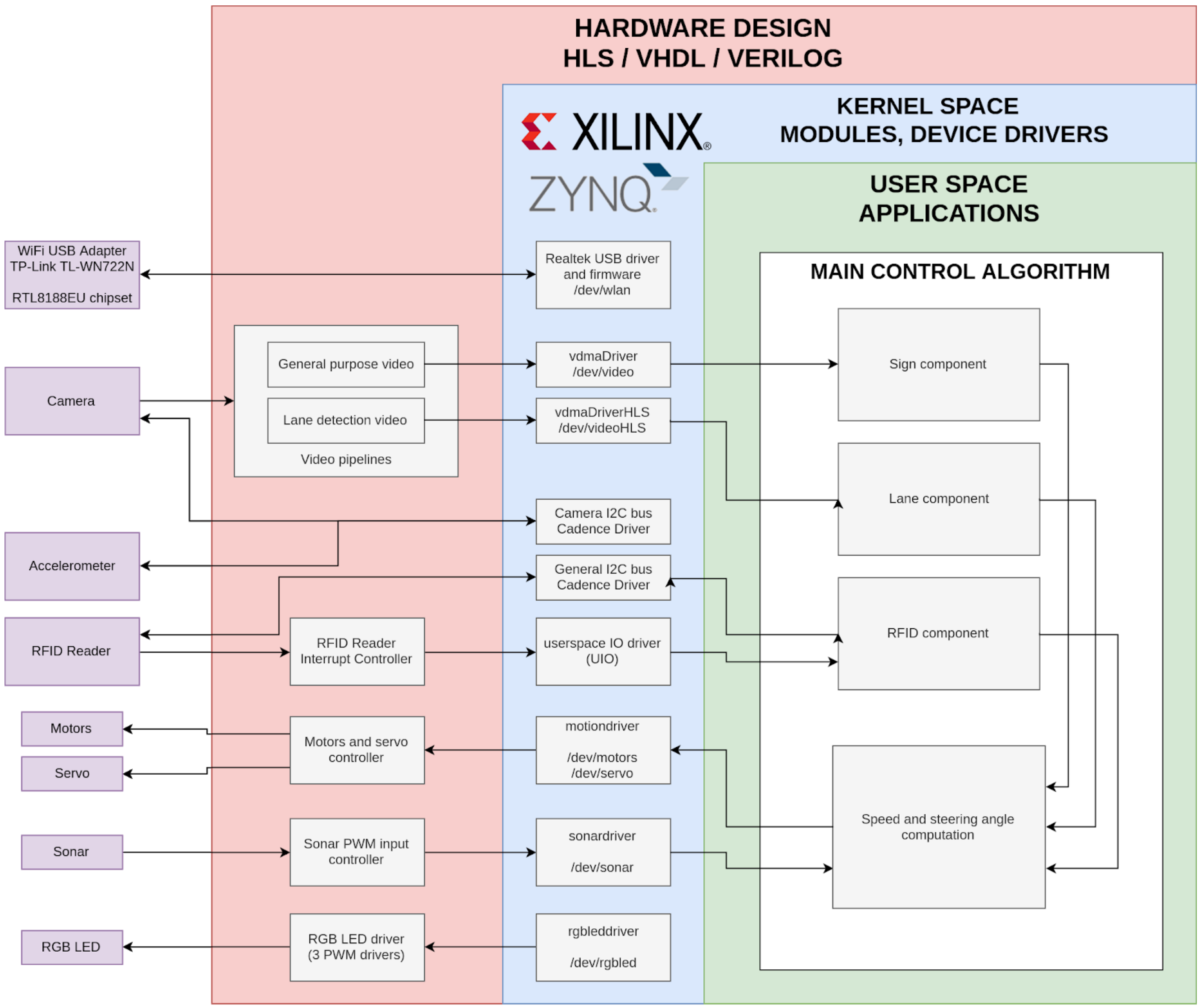
We decided to tackle the challenge of developing such a platform not because it is a completely new idea or because it was not done before, but because we saw an opportunity to experiment with new technology, expand our knowledge of both software and hardware design and implement something that will define the world in the following years as a concept.

The ‘build’ decision was based mainly on our wish to learn more about the systems in this fields (autonomous vehicles / computer vision) and to experience first-hand the development process of such an intricate project. The ‘buy’ decision is not really an option, as the only ‘platforms’ for autonomous vehicles on the market are already implemented as commercial products (eg. Tesla).

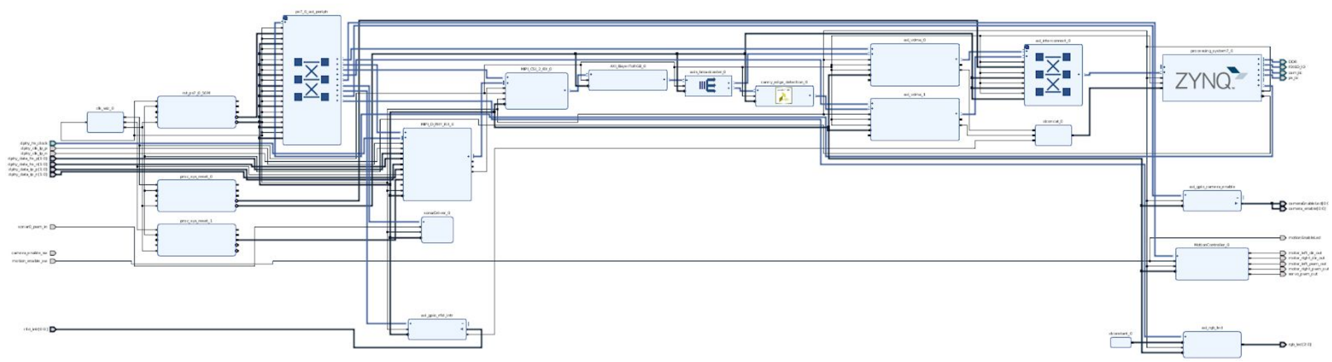
Design

Design Overview

The overall design block diagram is presented below and explained in detail, per component, further.

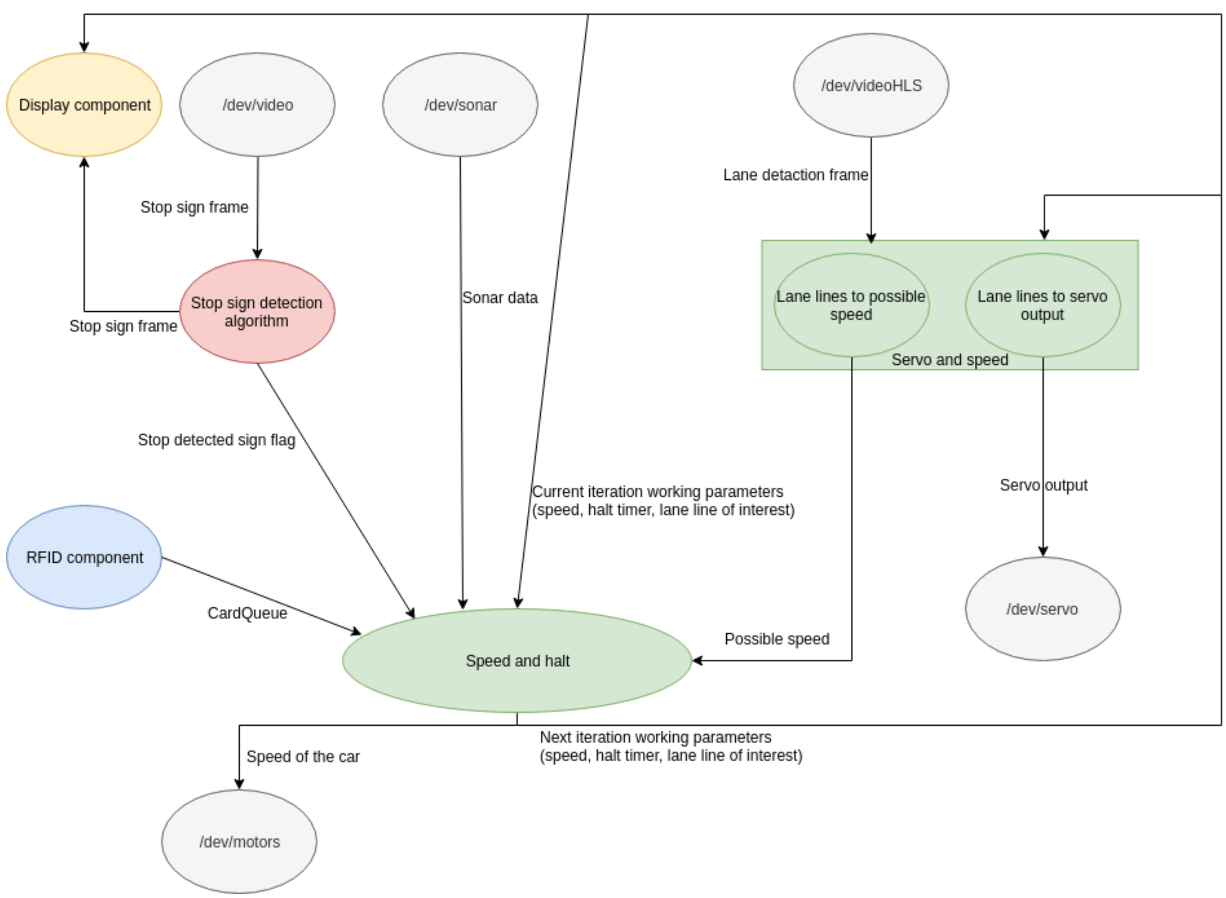


Vivado block diagram (full)



Main control algorithm and data-flow

This is the basic data-flow of the car control algorithm the three main components of it are highlighted with red green and blue colors on the following flow chart. There is also a yellow component, the display part which is the way users can see for themselves in real time the car's working parameters. There is also a calibration and configuration part to this which will be explained in the detailed design description later in this document alongside the display and main components of the algorithm.



Video pipelines and video processing - lane detection

The raw data from the camera MIPI D-PHY bus is fed into a MIPI_D_PHY_RX block that further feeds a MIPI CSI-2 Receiver, finally turning the raw data into usable 'pixels' as an AXI Stream. This Stream is in the Bayer format, so it must be converted into BGR to be usable. This last task is accomplished by the AXI_BayerToRGB block. The three mentioned blocks are based on the Digilent Pcam demo[8].

The resulting BGR AXI Stream is then fed into a AXI Broadcast block, splitting this video stream into two identical ones. One of these streams is directly buffered by an AXI VDMA (Video Direct Memory Access) and available in memory for the ZYNQ Processing System (available to Linux via the **/dev/video** node).

The other video stream is passed through a custom block that applies a set of image processing algorithms, synthesized in Vivado HLS, using the HLS Video library. The resulting grayscale image is buffered by its own VDMA and available in memory for the ZYNQ Processing System (available to Linux via the **/dev/videoHLS** node).

Sensor data acquisition and processing - RFID, Accelerometer and sonar

The three main sensors of our vehicle are presented in the image above. Both the Accelerometer and the RFID reader are I2C enabled, such that they are connected to the same bus and available to the Processing System directly. The sonar is connected via its PWM output (datasheet[9]) to a block that computes the PWM duty cycle and offers the resulting data to the Processing System via an AXI4-Lite link (further processed by its own kernel module and available as an integer on the **/dev/sonar** node).

Motors and steering control

The 'motors and steering' controller, also called internally **Motion** is a block that manages a Pololu DRV8835 dual motor driver and a servo that directly controls the front steering. The main block communicates with the Processing System via an AXI4-Lite link and internally it consists of three PWM generators with synthesis-time customizable parameters for the Resolution and Frequency. The values we chose are: Motors - 2x 16 bit, 100kHz drivers; Servo - 1x 12 bit, 50 Hz driver. The servo requires a specific frequency and duty cycle to function properly.

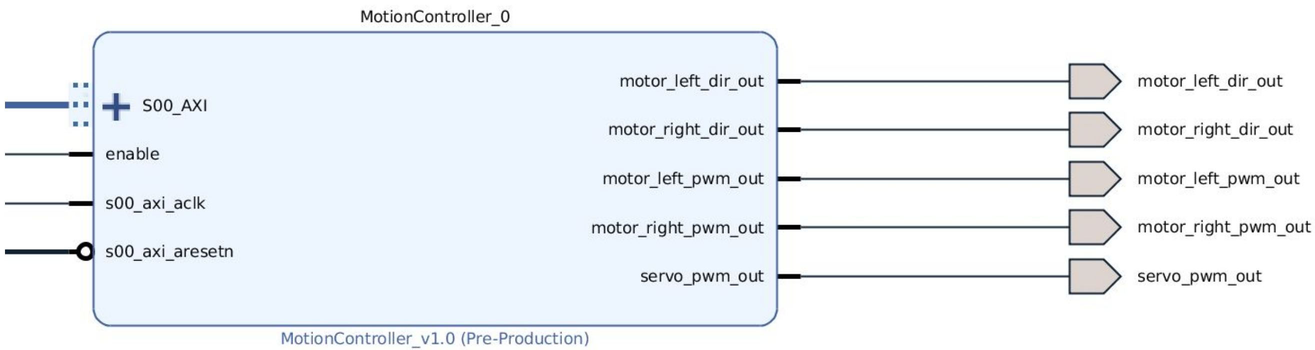
The motion component is available to the Linux OS via the **/dev/motors** and **/dev/servo** nodes.

[8] https://github.com/Digilent/Zybo-Z7-20-pcam-5c
[9] https://www.maxbotix.com/documents/LV-MaxSonar-EZ_Datasheet.pdf

Detailed Design Description

For simplicity and readability components will be described individually and cross-referenced to explain functionality as a whole.

Motors and steering control
(also called **Motion**)



The motion controller is internally based on 3 PWM drivers that signal the corresponding hardware (left motor, right motor and steering servo). The motor_left/right_dir_out outputs signal the Pololu driver the desired turning direction for the selected motor. Using an AXI4-Lite interface, it is directly connected to the Processing System's memory, and has 4 available **slave registers**. The following tables describe each of the register's purpose.

The **enable** input pin is a **safety feature**, the controller requiring both the software enable and the hardware enable inputs to be set to '1' to function. The hardware enable is tied to an on-board **switch** and is manually operated to completely prevent the vehicle from moving.

*CONTROL register **(offset 0x00)***

31	2	1	0
unused	LDIR	RDIR	EN

Name	Bit/Bits	Description
EN	0	Software enable/disable 'switch' for the motion module. **0** - Disabled (motors off, steering not accepting further input) **1** - Enabled
RDIR	1	Right motor's direction 1 - Forward 0 - Reverse
LDIR	2	Left motor's direction 1 - Forward 0 - Reverse

SERVO register ***(offset 0x04)***

31	11 0
unused	POSITION

POSITION (bits [11:0]) describes a duty cycle for the PWM driver assigned to the servo output. Care must be taken so that the actual PWM signal is valid for the used servo. Most require a 'on' time of 500 to 2500 us as a valid input signal. This limits the duty cycle of the 50 Hz signal generated here to be between 2.5% and 12.5%. That is, for a 12 bit resolution, a value between roughly 102 and 512.

Given the mechanical constraints of the steering system, values between 220 and 380 are used.

MOTORS register ***(offset 0x08)***

31	15 0
LEFT SPEED	RIGHT SPEED

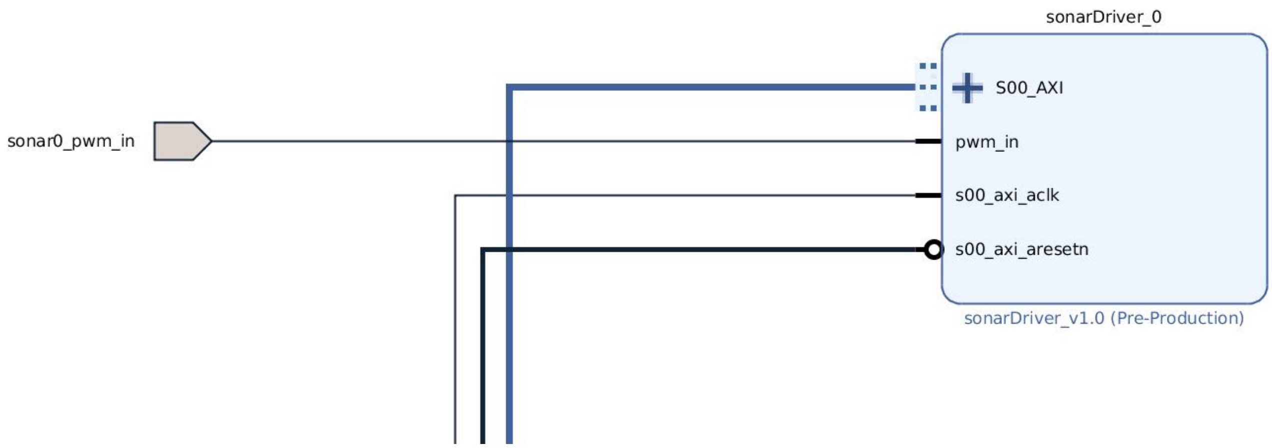
This register controls the speed of the motors, expressed as PWM duty cycle values (16 bit each). LEFT SPEED describes the left motor's power output and RIGHT SPEED the right one's.
Associated kernel module

The AXI Peripheral described above is being managed by a custom kernel device driver, implementing an easier to understand and use API of **ioctl** and **write** system calls to corresponding device nodes **/dev/motors** and **/dev/servo**.

Device node	Description and usage
/dev/motors	**ioctl** - using the parameters described in `Appendix J` (use example in `Appendix F`), users can change the motors rotation directions, and enable/disable the whole system. **write** - users can write to the device node a 32 bit integer that corresponds to the *MOTORS* register (upper 2 bytes control the left motor, lower 2 control the right one).
/dev/servo	**ioctl** - using the parameters from above, users can get the maximum and minimum enforced servo positions (described as duty cycle values for the *SERVO* register) **write** - users can write a 12 bit integer that corresponds to the *SERVO* register. The value written will be checked to ensure it is valid.

An application that presents all of the mentioned functionality of the **Motion** component and its kernel driver can be found in `Appendix F`.

Sonar



The sonar controller is based on the Digilent Pmod MAXSONAR demo project[10]. As per its datasheet, it outputs a PWM signal with an 'on' pulse width of 147 us per inch. The module counts the clock ticks for which the signal on the input (sonar0_pwm_in) is high and reports this number to the Processing System via an **AXI slave register** (**offset 0x00**).

Due to limitations, kernel space applications can not use floating point operations, so the associated device driver that provides **/dev/sonar** relays the clock count from the AXI register to userspace. The userspace application that employs the sonar needs to convert the clock count into a measure of distance, knowing the clock frequency of the Programmable Logic (AXI Clock), that is, in this implementation, 50MHz.

Userspace applications can use the **read** system call to access the data on **/dev/sonar**.

An application that uses this device and the kernel module can be found in `Appendix H`.

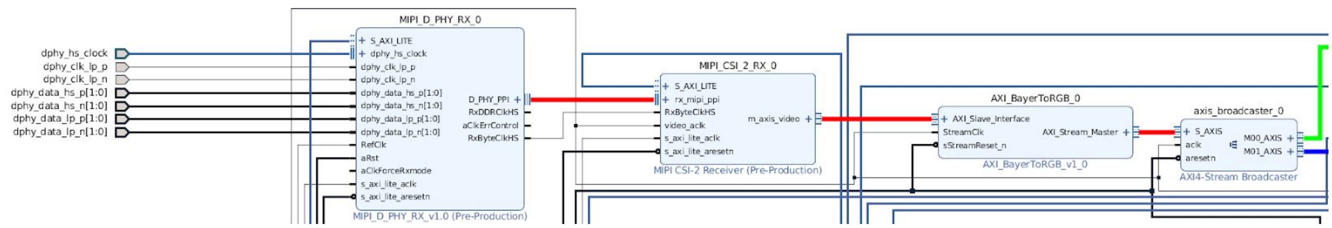
The main control algorithm's speed and steering decision component takes the data read from the sonar device node and translates it into a 'human-readable' form (centimeters or inches) and then takes action based on it, for example stopping the car if an object is close enough to the car to be considered an issue (collision avoidance).

The sonar data is taken in the speed and stop part of the control algorithm and is dealt with in the very first section of this step because it is critical to the car's safety on the road. If the car encounters an obstacle in front it will not take into consideration any stop signs or RFID cards.

[10] https://github.com/Digilent/vivado-library/tree/master/ip/Pmods/PmodMAXSONAR_v1_0

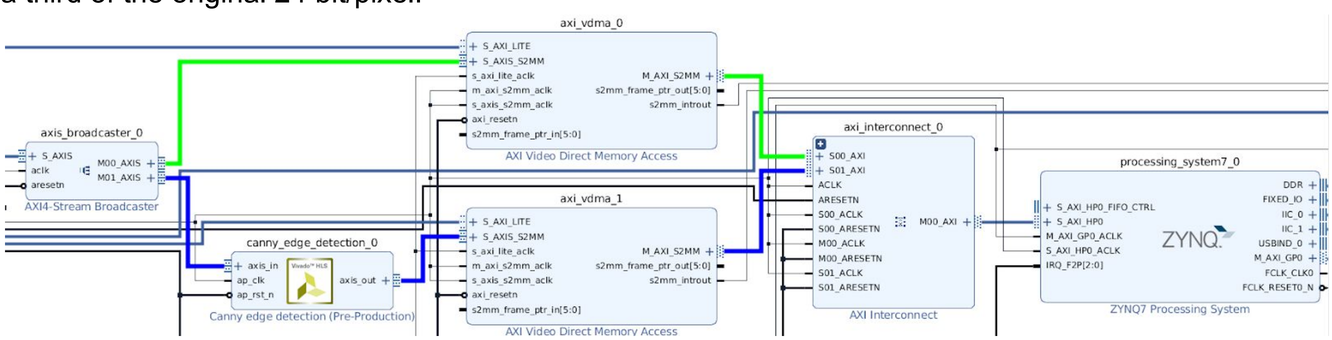
Video pipelines

The raw data coming from the Pcam 5C over the MIPI PHY lanes is interpreted and processed by the MIPI_D_PHY_RX block and the MIPI CSI-2 Receiver and the Bayer format stream is then passed through a AXI_Bayer_to_RGB block, outputting a more usable AXI Stream signal. These three blocks are from the Digilent PCAM demo project[11].



The AXI_Bayer_to_RGB was initially intended to output a RGB format with 10 bits color depth/channel, packed into a 32 bit integer. The camera, though, outputs an image of only 8 bit color depth, that leaves 2 bits/channel unused, along those 2 already wasted by the 32 bit ‘wrapper’. With some modification to the original code (`check Appendix E, lines 46-49 and 76-79 (extract)`) the output format became a 24 bit ‘wrapper’ containing 3 x 8 bit color depth channels. Thus, an improvement of 25% was achieved by removing irrelevant memory transfers. These changes are noted and detailed in `Appendix E`.

The resulting AXI Stream is split into two identical streams, to be repurposed accordingly. One of them is left untouched, 720p 60Hz, and the other one is put through a series of image processing techniques to obtain a grayscale image suitable for lane detection (see the main control algorithm). The grayscale image is also only one 8 bit channel wide, so the required memory transfers dropped to a third of the original 24 bit/pixel.



The HLS *Canny edge detection* block is based on the Github project with the same name, of the user “kyk0910”. It has been modified to change the output (and internal processing) format from 3 channel, 24 bit Grayscale to 1 channel 8 bit Grayscale. After the first step in the algorithm, which is to convert the input image to Grayscale, the whole code was changed to use the 1 channel, 8 bit Grayscale format for performance reasons. It was also brought a bit more in-line with the hls::video library (data types, structures and such). The C source code used for the synthesis is available in `Appendix B`.

[11] https://github.com/Digilent/Zybo-Z7-20-pcam-5c

These two streams are then being buffered by two AXI VDMAs (Video direct memory access) blocks, being then put into the Processing System's memory.

These two VDMAs are driven in kernel space by two modules (device drivers) that are based on Xilinx VDMA drivers, but stripped of many unused functions. The highly integrated nature of this project allows for highly customized drivers, so that a general purpose library/driver like Video4Linux was not used due to overhead concerns.

Both modules offer **ioctl** and **read** functions:

- **ioctl** - allows the user to read the image sizes (width, height, pixel size (24 bit or 8 bit)), and start/stop the video pipeline. The ioctl functionality is referenced in `Appendix K`.
- **read** - allows the user to read in a local buffer the image stored in memory from the VDMA. The read must be *width*height*pixel_length bytes* long to acquire the whole image.

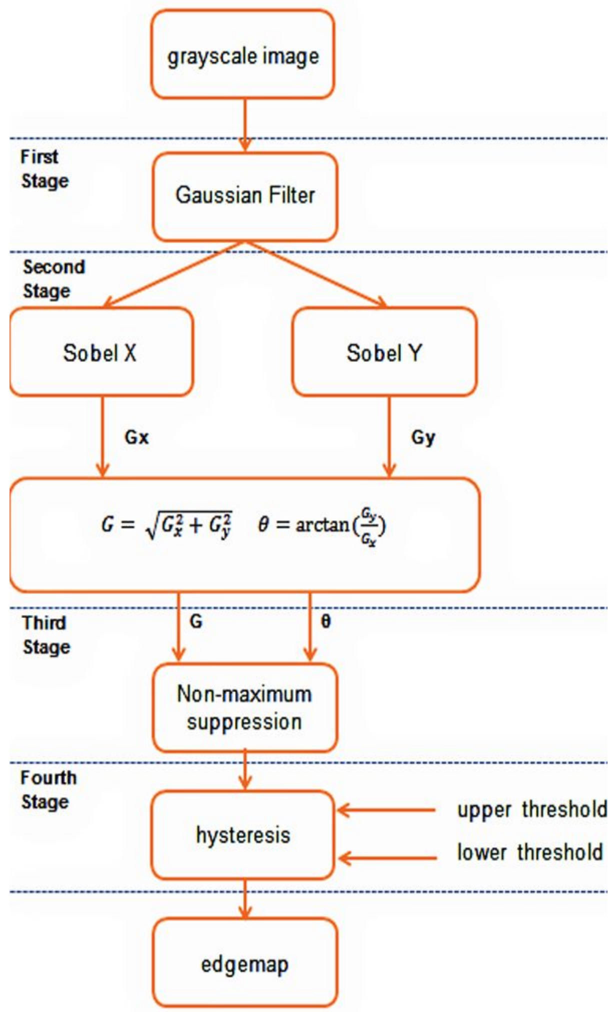
Using HLS to synthesize a *Canny edge detection* block meant that the ARM CPU does not need to do that processing via conventional OpenCV functions, now it being hardware accelerated.

Measurements showed that the Canny algorithm was the biggest resource hog in the whole program flow. As a comparison, before all the optimizations listed above, and before having this block, our implementation was stuck at something between 5-10 processed frames per second. After implementing everything mentioned here, we can safely say that we are above the camera's throughput of 60 frames per second, processing wise.

A flow chart describing the Canny edge detection algorithm can be seen on the right.

The code describing this block can be found in the References section, `Appendix B`.

An application that uses the video pipelines and kernel module to write an image to the filesystem can be found in `Appendix G`.



Note: image courtesy of www.embedded-vision.com

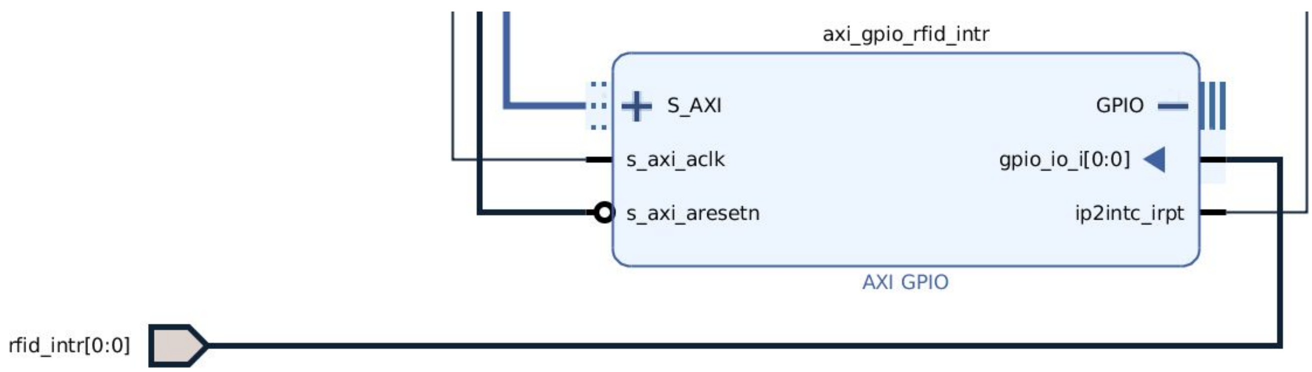
RFID card reader

The card reader placed under the vehicle is an **NXP PN532** chip with I2C, SPI and serial communication buses. We implemented the I2C method because we can reuse the physical interface lines to address other sensors (like the accelerometer and the intended power monitor).

The I2C data transactions and general communication implementation is handled by the included Cadence driver shipped with Xilinx Petalinux. The SCL (clock line) is set at 100 kHz.

Userspace applications interact with I2C devices on a given bus by using **read**, **write** and **ioctl** system calls on the corresponding device node (i.e. /dev/i2c-1). For further reference check the i2c-dev[12] library.

The **PN532** can operate by either being polled by the master device on the I2C bus until it responds with a 'data available' packet or by issuing an **interrupt** to the master to let it know new data can be **read** on the bus (for further reference check the chip's user's manual[13] and datasheet[14]). To manage such interrupts we implemented an AXI GPIO peripheral setup as an input only, 1 input, interrupt enabled block to which we connected the interrupt out of the **PN532** chip.



The block is an implementation of **AXI GPIO**[15], an IP available from Xilinx.

The device is managed by the **Universal IO**[16] kernel driver which provides a **blocking read** to its device node until an interrupt is issued. The blocking read acts as a **wait** / **sleep**, and it is not busy-waiting, wasting CPU cycles.

The main control algorithm's **RFID Component** manages the RFID device, adding all the read cards in a queue, to be processed in the future. The control application then, when available, extracts cards from the queue, compares the first data byte stored (our implementation only uses **block 4** of the 1kb card) against a database of known codes (values 0-255) and takes appropriate action (stops the car, starts the car, changes the maximum allowed speed, etc).

12 https://www.kernel.org/doc/Documentation/i2c/dev-interface
13 https://www.nxp.com/docs/en/user-guide/141520.pdf
14 https://www.nxp.com/docs/en/nxp/data-sheets/PN532_C1.pdf
15 https://www.xilinx.com/support/documentation/ip_documentation/axi_gpio/v2_0/pg144-axi-gpio.pdf
16 https://www.kernel.org/doc/html/v4.15/driver-api/uio-howto.html

The RFID part of the car control algorithm runs continuously and adds all the cards encountered on the road in a queue. The cards are popped from the queue, in the speed and stopping part of the algorithm and appropriate actions are taken for each card encountered.

The cards types are defined in the **cards.h** file. The following types of cards exist at this moment:

- SPEEDX - the cards that set a new base speed to the car (speed = X * 1000), acting like a speed limiter
- STOP - the card that halts the car for 3 seconds
- PAUSE - the card that halts the car for second
- KEEPL - the card that tells the car to only keep track of the left lane line
- KEEPR - the card that tells the car to only keep track of the right lane line
- KEEPLR - the card that tells the car to keep track of both of the lane lines

Accelerometer

The accelerometer (Digilent Pmod ACL) is an Analog Devices **ADXL345** connected via I2C to the general purpose bus (**/dev/i2c-1**) that is managed in kernel space by the included Cadence driver[17] shipped with Xilinx Petalinux. The SCL (clock line) is set at 100 kHz.

Userspace applications that require the accelerometer interact with it by using **read**, **write** and **ioctl** system calls on the corresponding device node. The data output by the accelerometer can then be used to measure the vehicle's moving characteristics. For further reference on the accelerometer usage and internal configuration protocols check the reference datasheet[18] of the **ADXL345** chip.

On startup the accelerometer is set to continuously measure acceleration data using the full 13 bit resolution over all three axes (X, Y, Z). The accelerometer is configured to read acceleration values ranging from -2G to +2G per axes. A set of correction offsets must be set before the accelerometer can read usable data. This process is called 'calibration' and it computes 3 values that must be loaded in the offset registers 0x1E for X-Offset, 0x1F for Y-Offset and 0x20 for Z-Offset, so that acceleration readings are correct, accounting for the positioning errors of the sensor on the physical platform.

The data is read from the accelerometer via the I2C bus, on registers 0x32 to 0x37. The data structure is as follows:

- 0x32 is the high byte of the X axis data, and 0x33 is the lower byte
- 0x34 is the high byte of the Y axis data, and 0x35 is the lower byte
- 0x36 is the high byte of the Z axis data, and 0x37 is the lower byte

Note that the high byte only has the 5 least significant bits set to provide a 13 bit resolution when paired with the lower byte.

To convert the sampled data into Gs (unit that is the gravitational force equivalent) the following formula is applied:

$$Gvalue = measuredValue * (Grange/2^{10}),$$

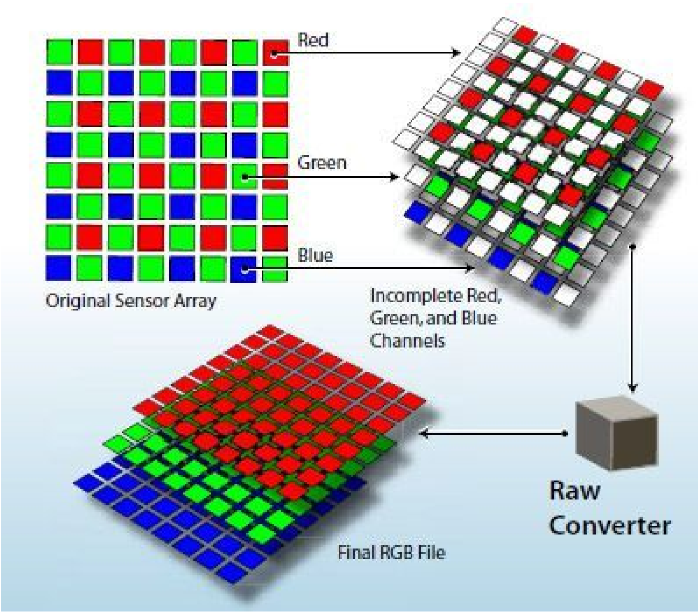
where *Grange* is the range set above, in this case 2.

[17] https://xilinx-wiki.atlassian.net/wiki/spaces/A/pages/18842160/Cadence+I2C+Driver
[18] https://www.analog.com/media/en/technical-documentation/data-sheets/ADXL345.pdf

Camera I2C configuration

The camera's onboard chip (**Omnivision OV5640**) can be configured via I2C with a great number of different settings affecting performance, image quality, lighting conditions, image size and output format. The individual parameters can be found in the chip's datasheet[19]. The camera's I2C bus (physically located on the CSI connector) is linked to the **/dev/i2c-0** device node, which is dedicated to controlling and interfacing with the camera.

For our specific application, the camera is configured to output 1280x720, 3 channel, 8 bit color depth images in an Bayer format that will be converted to RGB further down the video pipeline (see video pipeline section above). The white balance and general image quality settings chosen are set to give us the best results in the lab we used for testing.

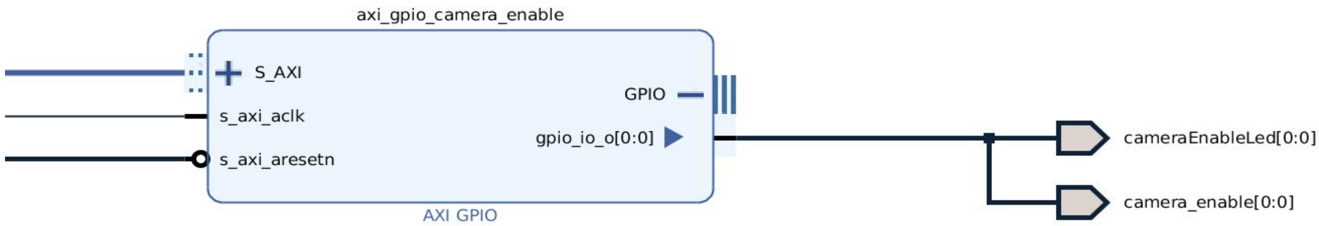


Typical image sensors capture only red, green, or blue for a pixel, and through "demosaicing" convert that data into a useful image with all three color elements for each pixel.

Image courtesy of Adobe Systems

The camera has an external enable/reset pin input which we use as a software-defined enable/reset control switch for the camera. Our design uses an **AXI GPIO** block configured as an all output, single output, default 'on' component, connected to the Processing System via the AXI4-Lite bus and managed by the **Universal IO (UIO)** kernel driver.

19 https://cdn.sparkfun.com/datasheets/Sensors/LightImaging/OV5640_datasheet.pdf



Both the camera enable block as well as the camera's I2C bus are used during boot-up by the **initcamera** application, which configures the camera with our specific settings, as well as making sure it is properly reset and enabled before attempting to do so.

RGB LED

One of the two RGB LEDs available on the Zybo board is used as a visual aid / debug output in our project. The three components are driven by a custom designed block, consisting of 3 variable frequency, variable resolution PWM outputs. For simplicity all three channels have a resolution of 8 bits and a frequency of 100 kHz. The block is connected to the Processing System via an AXI4-Lite bus. It is controlled by a single **AXI Slave register** at **offset 0x00**.

The register structure is as follows:

31	23	15	7 0
unused	BLUE	GREEN	RED

Name	Bits	Description
RED	[0-7]	8 bit value describing the PWM duty cycle of the given component 0 - 0% 255 - 100%
GREEN	[8-15]	
BLUE	[16-23]	

The associated kernel module that controls the RGB LED exposes the device node **/dev/rgbled** to userspace applications. Said applications can control the LED via **write** system calls, writing a 32 bit integer to it, that is directly mapped to the internal **AXI slave register** described above.

An application that makes use of this module can be found in `Appendix I`.

Petalinux embedded Linux distribution

The project is centered around an Embedded Linux distribution from Xilinx, Petalinux[20] 2017.4. We decided to go this route because our project includes many different modules (both hardware and software), and the Linux OS acts as a common ground between them, managing the processes and providing debug tools, persistent memory on the SD card, SSH access etc.

The Linux distribution runs a modified kernel that is adapted to our needs and provides support for the physical devices (camera, sonar, motors and servo) for the WiFi adapter via a modified USB driver based on the TP-Link driver available online[21]. Xilinx already included drivers for Zynq I2C, used to connect with the camera, accelerometer and RFID reader.

The PetaLinux Tools offers everything necessary to customize, build and deploy Embedded Linux solutions on Xilinx processing systems. PetaLinux tools eases the development of Linux-based products; all the way from system boot to execution with the following tools:

- Command-line interfaces
- Application, Device Driver & Library generators and development templates
- Bootable system Image builder
- Debug agents
- GCC tools
- Integrated QEMU Full System Simulator
- Automated tools
- Support for Xilinx System Debugger

With these tools developers can customize the boot loader, Linux kernel, or Linux applications. They can add new kernels, device drivers, applications, libraries, and boot and test software stacks on the included full system simulator (QEMU) or on physical hardware via network or JTAG.

Custom BSP Generation Tools

PetaLinux tools enable developers to synchronize the software platform with the hardware design as it gains new features and devices.

PetaLinux tools will automatically generate a custom, Linux Board Support Package including device drivers for Xilinx embedded processing IP cores, kernel and boot loader configurations. Such capability allows software engineers to focus on their value-added applications rather than low level development tasks.

Linux Configuration Tools

PetaLinux includes tools to customize the boot loader, Linux kernel, file system, libraries and system parameters.

[20] https://www.xilinx.com/products/design-tools/embedded-software/petalinux-sdk.html
[21] https://www.tp-link.com/us/support/download/tl-wn722n/#Driver

These configuration tools are fully aware of Xilinx hardware development tools and custom-hardware-specific data files so that, for example, device drivers for Xilinx embedded IP cores will be automatically built and deployed according to the engineer-specified address of that device.

Software Development Tools

PetaLinux tools integrate development templates that allow software teams to create custom device drivers, applications, libraries and BSP configurations.

Once the product's software baseline (BSP, device drivers, core applications, etc.) has been created, the PetaLinux tools enable developers to package and distribute all software components for easy installation and use across PetaLinux developers.

Main control application

The main computational part of the control algorithm is structured in 3 big parts with 2 smaller auxiliary “quality of life” parts:

- Lane component, used to position the car on the right path of the track and adjust its speed corresponding to the road.
- Sign component, used to detect stop signs.
- RFID component, used to correctly detect and store RFID cards placed in key parts of the road.
- Display auxiliary component, integrated into the lane component but completely separable from it, it is used to display relevant images to the user.
- Configuration / Calibration component is used before the 3 main parts of the car control algorithm to give the user the capability of utilizing a separate file to set important parameters without the need to recompile the program or to overwrite the configuration file to match the current road conditions.

All of these parts, except the configuration / calibration one, are ran in loops, each iteration corresponding to one frame.
The 3 main parts are running in 3 separate threads the first one (lane component one manages some shared variables and decies what to do with the data from the other 2 threads). *[Appendix A - lines 893 - 910]*

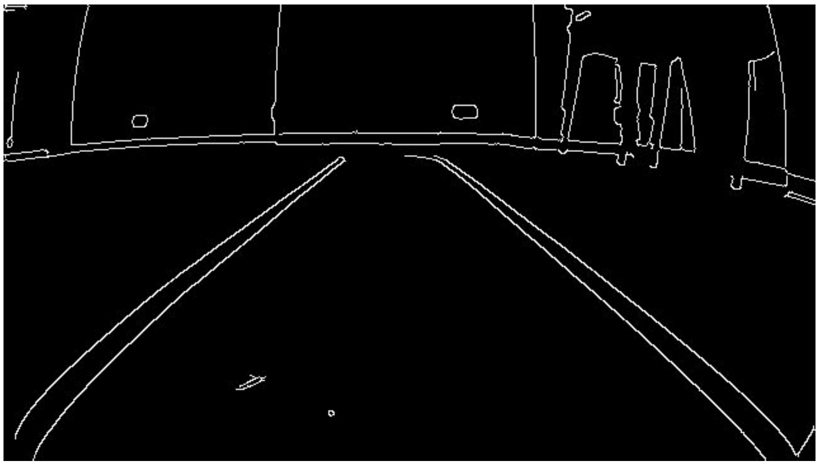
Lane Component

The lane component is the most important part of the car control algorithm because the first requirement for an autonomous car to succeed is to be able to navigate a simple, no obstacles, signs or complex instructions, road on its own. After that we can add safety and more advanced functionalities so our development platform can perform more like a real self driving car.

The first step of this component is a simple procedure of preprocessing where the image received from the `/dev/videoHLS` stream is resized. The resize factor is variable with a good one being one that maximizes the performance gain with minimal impact to the quality of the abilities of the car to correctly navigate the road.

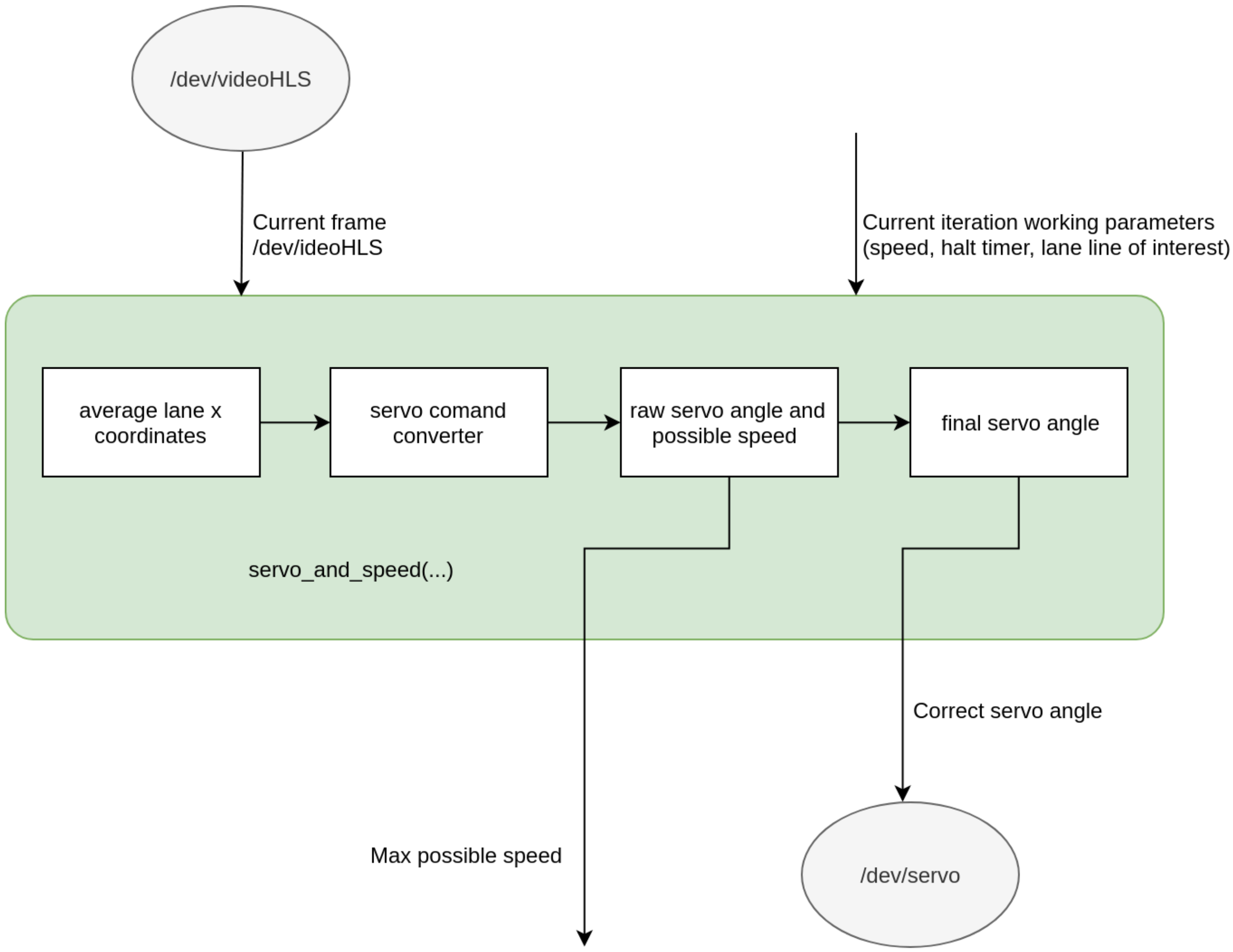
In the second step the scaled image is introduced in the OpenCV intense part of the loop where from an image of edges (what `/dev/videoHLS` outputs a Canny filtered image) is transformed into servo commands and possible speed to the road ahead. The function that does that is `servo_and_speed(...)`*[Appendix A - line 162]* which takes an input image and through several steps “converts” said image to a valid servo command and a maximum possible speed for the current road.

Detection zones are defined in the configuration file to set the the zones of interest for the car. Meaning an image like this:



will be interpreted like this:

leaving out unimportant zones of the picture like for example everything above the horizon line (need to the sign detection component but not here).



It does this through several steps:

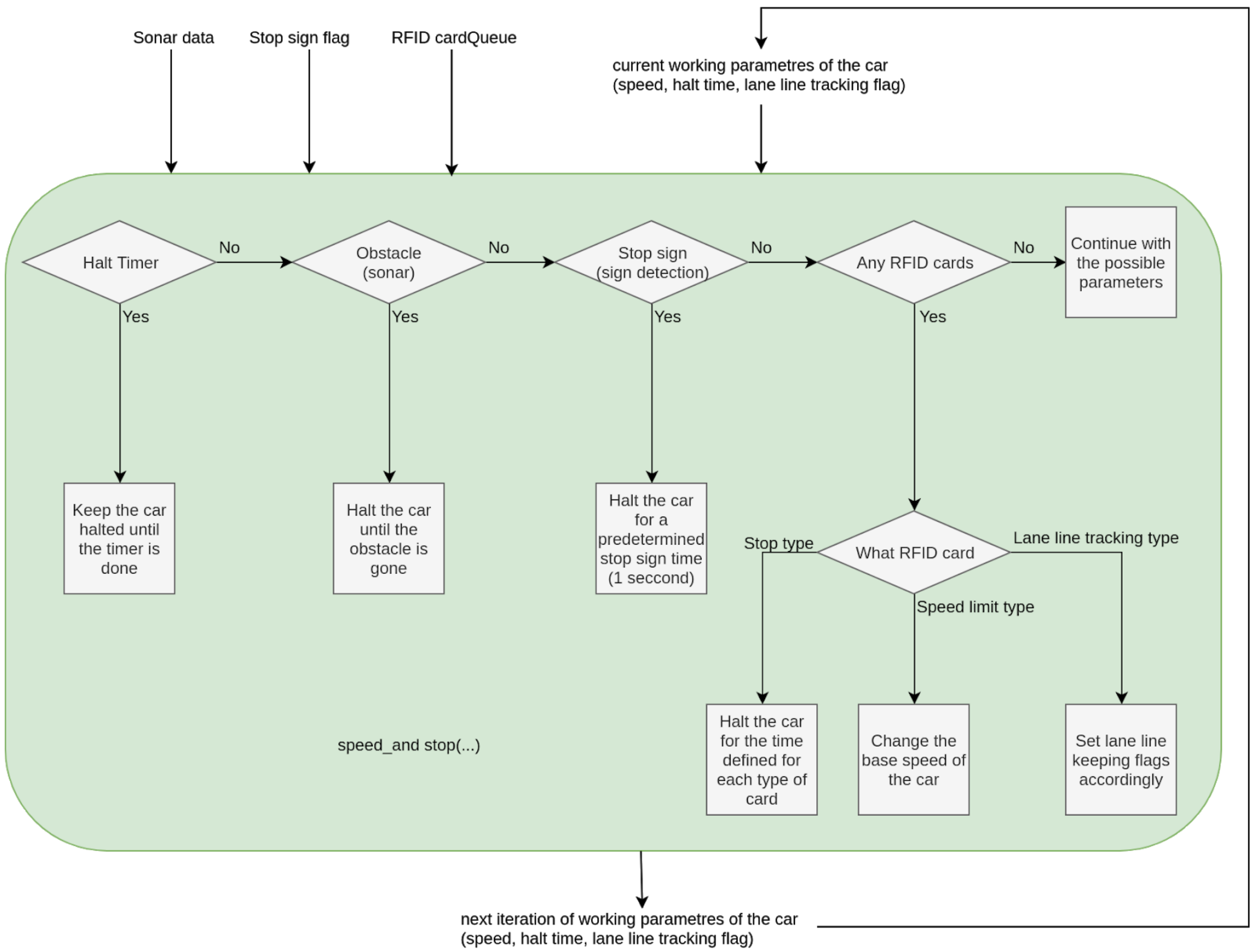
- Find the lane line position on screen with the function `average_lane_lines(...)` *[Appendix A - line 145]* which returns 6 x-axis positions on the image of the detected lines.
- The x-axis coordinates are then converted to valid servo commands with the function `map_servo_fine(...)` *[Appendix A - line 28]*. The coordinates are mapped depending on what side of the true center of the lane lines are found, this needs to happen because the detection zone is bigger between the lane lines and smaller outside of them, making this a map function with the center of the range offsetted.
- The servo values are sent to the function `choose_advanced_servo(...)` *[Appendix A - line 63]* which calculates the mean servo angle from the given commands and the maximum capable speed. The steering angle is determined just from the closest detection zones to the car, those being relevant to the most immediate road conditions to the car. The maximum possible speed is calculated with the rest of the detection zones with the following

rules: if all three detection levels identify a lane line present the current speed is gradually increased to the `base_speed * cfg.speed_up_max`, if only the bottom 2 levels of detection identify lane lines the current speed gradually reverts to `base_speed` and if only the bottom level of detection zones identify lane lines the current speed gradually decreases to `base_speed * cfg.speed_up_min`. All these speed adjustments are done in `cfg.speed_up_rate` percent steps.

- After the servo command is computed it needs to be adjusted to the current speed of the car. Like in real life when a car is driving faster it doesn't need to turn the steering wheel as hard to execute the same corner and not lose grip. This is done with the `servo_speed_adj(...)`*[Appendix A - line 131]* function.

After these steps are done we have a valid servo command that we can send to /dev/servo to turn the wheels with the desired angle. The speed of the car, on the other hand, still needs adjustments, which will be done in the `speed_and_stop(...)`*[Appendix A - line 286]* function, the third and final big step of the lane component part of the algorithm.

The speed and halt component function works according to the next flow chart:



The functions uses a timer variable that is set to a number of microseconds if the car need to stop moving. This timer is decremented at each `lane_component(...)` *[Appendix A - line 489]* end of loop with the duration of the current iteration. If the carr still has to wait for its timer to finish the function exits immediately if the car can move (no halt timer active) the function “checks” if the car can move. First with the sonar for any obstacles *[Appendix A - line 300]* then with the sign detection component for any stop signs *[Appendix A - line 320]* and if everything is clear to this point the car can look for RFID cards to execute their command (either it being a stop, speed or lane line tracking type of card) *[Appendix A - line 341]*.

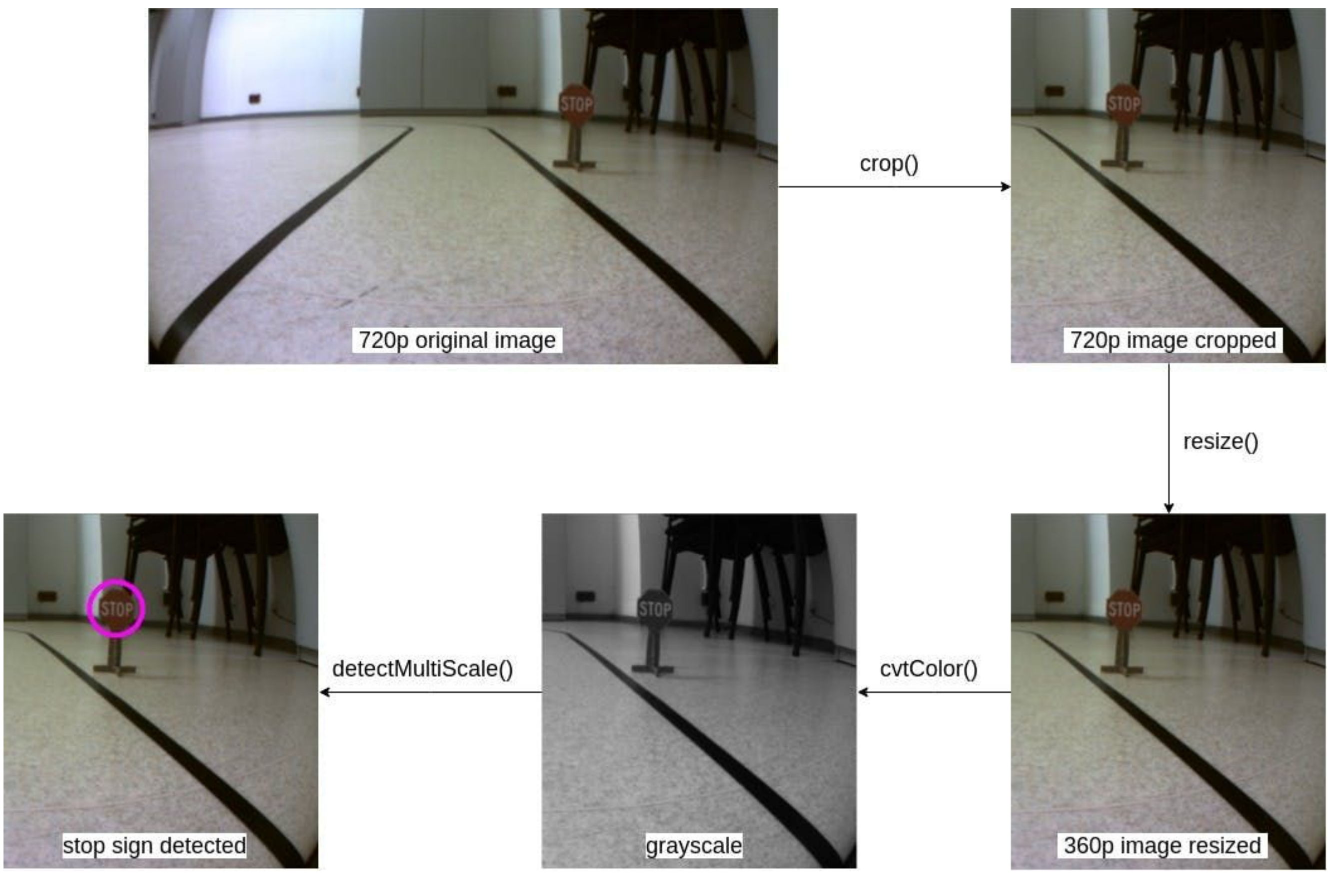
All of the mentioned features (sonar, sign detection and RFID) can be toggled on or off from the configuration file.

After the function `speed_and_stop(...)` *[Appendix A - line 286]* has finished the car has correct values to write to the motors and the control loop can move to the next iteration. *[Appendix A - lines 542 - 555]*

Sign Component

This component is responsible for sign detection through the `sign_component(...)` *[Appendix A - line 641]* function. At the moment just the stop sign detection is implemented and it works utilizing the object detection module from OpenCV. When a stop sign is detected the car executes a brief 1 second halt to demonstrate this object detection feature.

The steps of this algorithm can be described from the following flowchart:



The preprocessing for this component consists in cropping the image, leaving us with just the right part of it and resizing it to a smaller size for increased processing speed whilst keeping enough details for a correct decision. *[Appendix A - lines 658-659]*

A Haar cascade classifier is implemented and after the preprocessing step (unlike the lane component this one receives unaltered video from the camera through `/dev/video`) stop signs are detected and a shared flag is set to the current state of signs with the function `detect_and_display(...)` *[Appendix A - line 231]*. After the flag is set the loop continues to run and reset the shared flag to the according to the new frame. The sign detection loop keeps running until the lane component loop runs. We have used a pre trained Haar cascade classifier courtesy of cfizette[22].

The stop sign algorithm can be extended to others signs, the main trick being to execute the sign once you can't see the sign no more. If the car has seen the sign, because of the restricted field of view, the car doesn't "know" exactly when it is next to the sign to carry out the specific instructions. Implemented in the `speed_and_stop(...)` *[Appendix A - line 286]* function is the procedure for the car stops at the sign when in the current frame it can't "see" the sign that it has detected in the previous frame. This approach is close enough, but for sign detection to work in real life on the road there should be more cameras on the vehicle, giving it a wider (preferably 360°) field of view, or a more robust communication with the car with specific and reliable communications protocols to the car.

RFID Component

This part of the main control algorithm handles the RFID card reading part of the self driving experience. The component runs like the other 2 main components in in its own thread and it is responsible of reading correctly strategically placed rfid cards from the track.

It does so with the use of **PN532** chip which reads every card that is passed over by the vehicle. This is a more robust way of giving commands to the car than the camera and also it uses less processing power. The once a car with a specific uid has been read the car ignores the same card, so for example it will not read the stop card endlessly, as it stops right on top of it.

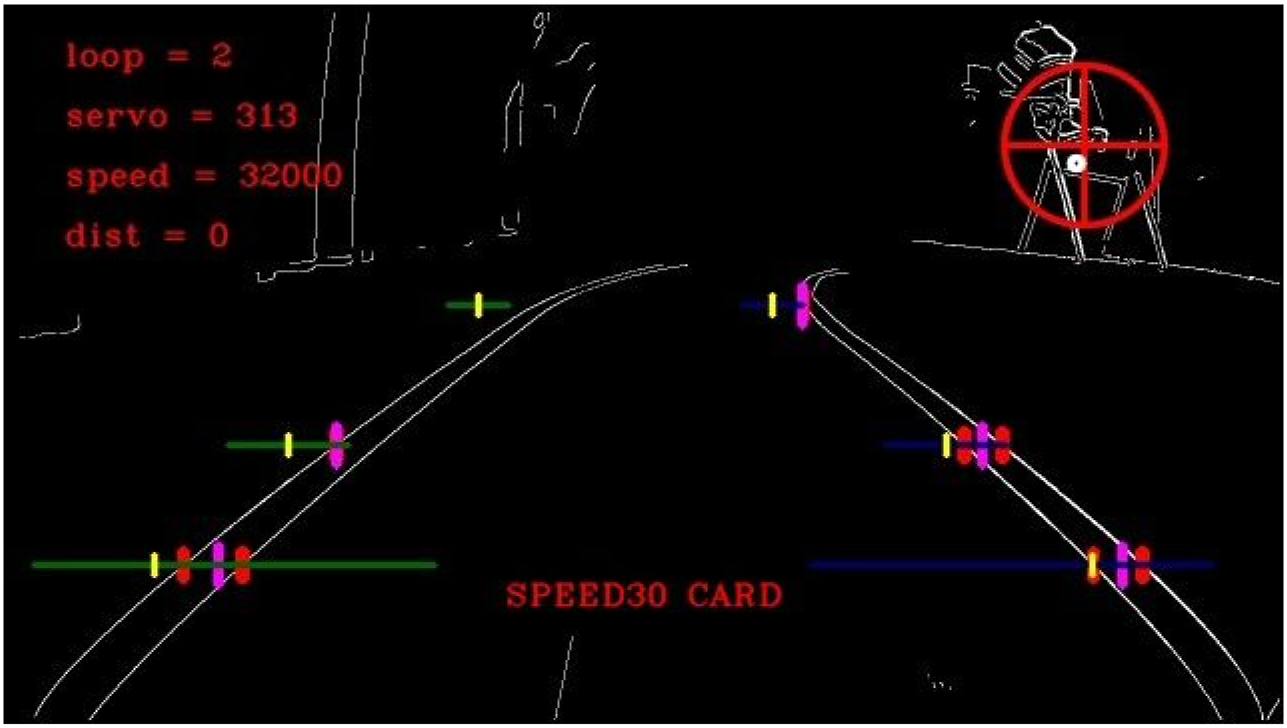
The execution of the cards is integrated in `speed_and_stop(...)` *[Appendix A - line 286]* function from the Lane component and are as follows:

- Speed changing with SPEEDX type cards, X being the value to which the speed is changed to. As mentioned in the sped adjustment part of the first component the speed is adjusted gradually, so when a higher speed card is encountered the car doesn't change to that speed instantaneously instead it changes its `base_speed` and adjusts its `curent_speed` in the following iterations.
- Halt type cards, which stop the car for a predetermined period of time.
- Lane line tracking type, which tell the car to only follow the indicated lane line of the road, so, for example at a branch in the road the car can be instructed to steer left or right.

[22] https://github.com/cfizette

Display Component

The display part of the is here to stream relevant images and text to the user so a human can understand what a computer “sees”. This part is intertwined almost everywhere in the code, most of the auxiliary functions write relevant information either to an image which is later written on the disk or print results to the console.



This image overlay has:

- In the top left corner: iteration number, servo angle, current speed and distance to object.
- In the bottom middle: RFID card display.
- In the top right corner: acceleration data.
- On the lane lines: the 6 detection zones and the points considered to be part of the lane lines, each with:
  - horizontal line showing the whole detection area
  - yellow vertical line showing the considered center of the lane line
  - red vertical lines showing intersection points between the lane line edges and the detection zones
  - magenta vertical lines showing the average intersection point on the detection zones

On a straight road the car should have the yellow line and the magenta ones as close as possible, in the above image you can observe a slight curve to the right in the road and consequently the servo angle indicates that the car should steer to the right (center of servo is 300).

If specified in the configuration file the car can stream either lane detection and car control images, or stop sign detection images (cropped image with an ellipse where the stop sign is detected).

The display component writes either the lane component relevant image or the sign component one through `cfg.draw` to `/etc/mjpg-stream.jpg` from where a tool called mjpg-streamer displays the image present in that location to `$ZYBO_IP:8080/?action=stream` in a browser.

Configuration / Calibration Component

This component is a necessary one from the perspective that not all roads are the same, and before entering an unknown road the user could adjust some parameters for the perfect experience, or the car can auto-calibrate according to the road conditions.

To calibrate the car a valid configuration file is needed meaning that it still needs user defined parameters. The user should place the car as centered as possible on a straight piece of road, facing forward. The car will calculate where the lane lines are on the straight road, indicating where they should be when it is driving itself in further iterations, and the configuration file will be overwritten.

This calibration step is optional a correct calibration can either improve the quality of the ride but a wrong one (car is not on a straight road or is not centered) can ruin it.
The parameters on which the car takes decisions are presented here (an example of a valid configuration file for 25 cm wide lane and our camera angle can be found in Appendix N):

- resize_factor - the resize factor of the video pipelines
- y_1 - Y-axis coordinate for the lowest detection zones
- y_2 - Y-axis coordinate for the middle detection zones
- y_3 - Y-axis coordinate for the highest detection zones
- line_dist_1_out - the length of the lowest detection zones outside the lane lines
- line_dist_1_in - the length of the lowest detection zones inside the lane lines
- line_dist_1(*unused) - length of the detection lowest detection zones on both sides of the lane line
- line_dist_2 - length of the detection middle detection zones on both sides of the lane line
- line_dist_3 - length of the detection highest detection zones on both sides of the lane line
- servo_fine - difference between servo commands which decides if the servo chooses the extreme value or an average of the 2 values
- min_speed - low end of the speed range, used to adjust the servo to the speed, does’t need to be the absolute lowest speed of the car
- max_speed - low end of the speed range, used to adjust the servo to the speed, does’t need to be the absolute lowest speed of the car
- min_adj_servo - high speed adjustment low end of the range
- max_adj_servo - low speed adjustment high end of the range
- servo_map_a(*unused) - one of the 4 parameters of a cubic function for fine tuning the servo speed adjustment
- servo_map_b(*unused) - one of the 4 parameters of a cubic function for fine tuning the servo speed adjustment
- servo_map_c(*unused) - one of the 4 parameters of a cubic function for fine tuning the servo speed adjustment

- servo_map_d(*unused) - one of the 4 parameters of a cubic function for fine tuning the servo speed adjustment
- speed_up_max - maximum percent of base speed increase
- speed_up_min - minimum percent of base speed decrease
- speed_up_rate - the percent of gradual speed adjustment
- sign_min - low dimension threshold for a sign to be considered valid
- sign_max - high dimension threshold for a sign to be considered valid
- sonar_dist - the distance at which the sonar triggers (lower than) or -1 for disabled sonar
- fps - maximum fps limit or -1 for unlimited fps
- sign_on - 0/1 for the sign component off/on
- rfid_on - 0/1 for the rfid component off/on
- acl_on - 0/1 for the acceleration display off/on
- draw - 1 to stream images from the lane component, 2 to stream images from the sign component
- left_mean_1 - X-axis position of the lowest left lane line detection area center
- left_mean_2 - X-axis position of the middle left lane line detection area center
- left_mean_3 - X-axis position of the highest left lane line detection area center
- right_mean_1 - X-axis position of the lowest right lane line detection area center
- right_mean_2 - X-axis position of the middle right lane line detection area center
- right_mean_3 - X-axis position of the highest right lane line detection area center

Discussion

Problems Encountered

Over the months of developing this project we encountered several problems, some of which were difficult to solve and went as far as making us question the reasoning behind the original design that ended up creating those problems.

The most relevant problems are listed and described below, along with the solutions we have found:

Power monitoring and management - Digilent Pmod PMON1[23]

The initial design had a power management component that was supposed to measure the battery's voltage and current draw, to give the vehicle's control application an idea of the current power usage and the battery capacity left. We added the PMON1 part to the I2C bus shared by the accelerometer and RFID reader and after dozens of hours of work and testing, we concluded that we can not get any usable data off of it. We attribute this problem to either the lack of technical details and usage examples in the provided datasheets from the Digilent website (hence us missing a crucial point in the setup or communication with the device) or to a faulty chip.

In the end the power management module was scrapped and in the meantime we use a Li-Po battery monitoring alarm to make sure the cell voltages are in the nominal, safe, range. In the future we could use the onboard XADC of the Zybo to measure the battery voltage via a voltage divider, or use a different, commercially available, power management and monitoring board/chip.

Steering radius too big

The initial design of the physical platform only allowed for a relatively small steering angle, because the front wheels were bumping into the corners of the plastic frame. After redesigning the 3D printed wheel hubs to extend the wheels outward to allow for more movement, it became apparent that we have to cut some of the plastic frame away. This limits the 'plug and play' aspect of our project, as several such modifications are required to completely assemble the car with our components and mounting hardware.

Reduced camera field of view

The original lens that was shipped with the Pcam 5C from Digilent has a focal length of 2.5mm, which is appropriate for general purpose Computer Vision applications, but close to unusable in our case, as we require a camera lens that can see both lines that describe a lane at the same time, even during bends and turns. We 'fixed' this issue by using a commercially available fisheye lens 'adapter' that we glued over the original lens, this way extending the field of view to something better suited for our use case.

[23] https://store.digilentinc.com/pmod-pmon1-power-monitor/

In the future we want to replace this ‘improvised’ fix (which includes some duct tape for good measure) with a proper wide angle lens that can be mounted to the M12 socket on the camera board.

USB port power issues regarding the WiFi Adapter

We encountered strange problems with the USB port of the Zybo board after we managed to make the WiFi adapter work (more on that below) - the board randomly reset with the PGOOD LED blinking once (indicating a general power issue).

After about 4 days of troubleshooting (around 30-40 hours of work) we figured out that the USB port on the Zybo Z7-20 board could not supply enough current for the WiFi Adapter. It took us this long to figure out this ‘simple’ issue because we were certain that the USB port **must** be able to provide 500mA of current (and the adapter was rated at a maximum of 250mA draw). After changing the voltage regulator (that supplies 5V to the Zybo) with a more expensive one, the problem still persisted, so it was not an issue of a noisy supply or a voltage-drop.

In the end we had to power the USB WiFi directly from the 5V Regulator, bypassing the Zybo’s internal voltage regulators and monitors (**IC26:TPS25940**), and the problem vanished.

The takeaway information is that in **future revisions** of the **Zybo Z7-20** board **Digilent** should fix this issue, maybe by tying the USB 5V line to the input 5V supply of the board **VU5V0**[24] as marked in the power schematic of the reference manual. The specification for USB 2.0 is 5V +/-5%, so 4.75-5.25V, and the Zybo input voltage is listed as 4.5V to 5.5V DC so only a minor supply constraint is needed.

Drivers for the WiFi USB adapter hard to find/use

For the typical PC user, installing a new USB peripheral is as easy as plugging it in and waiting for it to be detected and configured by the operating system. In our case, using a stripped down version of Linux, we had to source the driver from the vendor’s website. After some research it became apparent that that version of the driver was built for newer kernel versions (Petalinux 2017.4 runs kernel version 4.9) and was not suitable for our application. In the end we found a github repository of an older version of the driver that we integrated into the petalinux work-flow as a standalone kernel module. The firmware accompanying the driver/chipset was also added to petalinux to be installed under **/lib/firmware/rtlwifi**.

The Petalinux Tools distribution from Xilinx does contain a driver for the RTL8188 chipset under ‘Staging Drivers’, but for some reason it was not compatible with our USB Adapter. In the future we might try to push our driver package to the Xilinx repository, so that it is available to other developers.

[24] https://reference.digilentinc.com/reference/programmable-logic/zybo-z7/reference-manual#power_supplies

Reduced processing speed (frame rate)

Even though the camera outputs 60 images per second, our initial design did all of the OpenCV image processing on the ARM CPU that runs at a sluggish 667MHz. Because of that, we could expect processing speeds of 5 to 10 images per second, which was ok when the car was not running at a high speed - moving slower it had more time between images to adjust to the oncoming road topology.

The solution we found was to accelerate the costliest part of the image processing algorithm, the Canny Edge Detection component directly on the FPGA. Using Vivado High Level Synthesis we created a block that applies all the required transforms on the input image, so that the CPU can then use this image to decide the best course of action with the data collected and send commands to the other peripherals.

Having done so, we managed to increase the Lane Detection component's processing speed up to @70 images per second which gives us a smooth operation of the vehicle even at its maximum speed. The Sign Detection component still uses the general purpose video pipeline, as we did not want to constrain the usage of the video feeds to just these two applications. Its processing speed remains at @7 images per second, which is more than enough for its purpose.

Engineering Resources Used

The project as a whole took around 500 hours of work over a span of 4 months. A rundown of the development process (loosely approximated values) is as follows, listed in somewhat chronological order:

- Hardware research and sourcing - 2 weeks
- Overall project structure, setting goals and milestones - 1 week
- Hardware assembly, electrical work, 3D printing mounting hardware and installing the required components - 2 weeks
- Testing the motors and steering with 3rd party microcontrollers (frame adjustments, steering rework) - 1 week
- Getting used to the Petalinux build environment and workflow, trying out test applications and builds - 1 week
- Building and testing the low level VHDL drivers for the actual hardware (sonar, motors and steering) - 1 week
- Building and testing the video pipeline (initially it only had one output, the direct images from the camera in 720p resolution) - 2 weeks

The following steps are in order, but their duration can not be precisely measured, as work was being done on multiple ends of the project at the same time, while also testing and unit-testing the already built systems. Least to say it occupied the rest of the development time and it could be called 'ongoing'.

- Started work on the main control application that integrates OpenCV and everything built above. Added RFID support. At this stage, applications interacted with the low level hardware

with direct memory accesses which were not user friendly and not easily understandable/modifiable.
- Added kernel modules and abstractisations for all the low level components, to further integrate the project with the ‘Embedded Linux’ vision. After this stage, the applications interacted with the low level hardware via device nodes and kernel drivers in a more transparent way.
- Added support for WiFi and went wireless, no longer having a cable attached to the vehicle at all times. Also added some utility applications like the web server that allows developers to watch the camera feeds in real time.
- Finding ways to optimise the processing speed of the control application - added a second video pipeline with hardware accelerated image processing capabilities.

In the sense of resources used as software tools and equipment, most of the 3D work was done in Solidworks (models available as STL files on the project repository) and the tests included a combination of electrical measurement tools and logic analyzers (**Digilent Analog Discovery**).

Marketability

Our work can be used as a base platform and extended with sensors and functionality depending on its use case in a commercial application. We did not have in mind any idea to sell or monetize in any way our project, as it was built as a learning platform and an opportunity to experience new technology and a new development process.

Community Feedback

Most, if not all, of the feedback we got on our project was from other students and peers, professors and the general people we presented our work to. It is always good to hear what others have to say about your projects, and in our case we had a lot of talks about the concepts that drive an autonomous vehicle, ranging from safety to ethics.

To be specific, some of the ideas we discussed with others and got to implement are:

- We should take account of the movement speed when adjusting the steering, as higher speeds imply a longer distance traveled between two processed frames such that the steering should be controlled less aggressively to ensure a smoother road posture and less left-right wobble.
- Someone suggested that we should increase and decrease the speed gradually when able to do so (increase the speed on straight roads, decrease it before entering a sharp turn, etc) to improve the smoothness of the ride and to provide a better visual feedback of the vehicle’s activity. In a real application, this would also reduce peak currents in the electric motors resulting in longer service times.
- Add fuzzy logic in the process of mapping the detected lanes to a corresponding steering decision (that ought to bring the car back to the center of the lanes). This allowed the main application finer control over the car’s steering system such that turns are treated better.

References, thanks and credits

- The kernel modules have been developed following the suggestions and guidelines presented in the book *Linux Device Drivers, Third Edition* by Jonathan Corbet, Alessandro Rubini, and Greg Kroah-Hartman. The book can be found at https://lwn.net/Kernel/LDD3/ under the terms of the Creative Commons Attribution-ShareAlike 2.0 license.
- The Xilinx Forums has been an amazing source of information for everything Petalinux related, many thanks to the community that manages the forum.
- The Petalinux Tools 'Bible', UG1144 Petalinux Tools Reference Guide has everything anyone would need on how to use the included tools with Petalinux Tools
- Github and the developers there, made it a tiny bit easier solving weird problems with drivers, API's and code in general
- The OpenCV community for providing relevant examples and open-source code through OpenCV official documentation
- 3D printed parts can be found on the project's development repository here

Appendix A: opencv-control.cc

```
struct properties cfg;
int f_motors;
int f_servo;
int f_rfid;
int f_rgbled;
cv::CascadeClassifier stop_cascade;
std::string stop_cascade_name = "stop.xml";
std::mutex IMAGE_mutex;
std::mutex COUT_mutex;
std::mutex SPEED_mutex;
std::mutex STOP_mutex;
char lane_done = 0;
struct sigaction sigIntHandler;
char stop_sign;
struct cardQueue *c_queue = NULL;
/* ------------------------------------------------------------ */
/*** int map_servo_fine(double input, double in_min, double in_max, double mean)
**
**   Parameters:
**     input:          the average point on the detection line
**          in_min:                           the left end of the detection line
**          in_max:                           the right end of the detection line
**          mean:                             the "center" of the detection line
**
**   Return Value:
**      Individual servo command.
*/
int map_servo_fine(double input, double in_min, double in_max, double mean) {
/* the body of this function is cut from this document to save space */
}
/* ------------------------------------------------------------ */
/*** int choose_servo(int left, int right, int mean)
**
**   Parameters:
**     left:         the left side servo command
**          right:                            the right side servo command
**          mean:                             the center of the servo
**
**   Return Value:
**      Servo command.
*/
int choose_servo(int left, int right, int mean) {
  /* the body of this function is cut from this document to save space */
}
/* ------------------------------------------------------------ */
/*** std::vector<int> choose_advanced_servo(int left_1, int right_1, int left_2, int right_2, int left_3, int right_3, int mean, int current_speed, int base_speed, int lane_keep)
**
**   Parameters:
**     left_1:       the left detection line 1 average
**          right_1:                 the right detection line 1 average
**     left_2:       the left detection line 2 average
**          right_2:                 the right detection line 2 average
**     left_3:       the left detection line 3 average
**          right_3:                 the right detection line 3 average
**          mean:                             the center of the servo
**          current_speed:           the current speed of the car
**          base_speed:                       the speed set by the user
**          lane_keep:                        the lanes considered in deciding the servo output
**
```

```
**   Return Value:
**      A vector with servo command on [0] and the speed the car is capable of doing now [1]
*/
std::vector<int> choose_advanced_servo(int left_1, int right_1, int left_2, int right_2, int left_3, int right_3,
int mean, int current_speed, int base_speed, int lane_keep) {
  std::vector<int> ret;
  if (lane_keep == 1) {
    // adjust servo only with the right_lane_line
    ret.push_back(choose_servo(0, right_1, mean));
  } else if (lane_keep == -1) {
    // adjust servo only with the left_lane_line
    ret.push_back(choose_servo(left_1, 0, mean));
  } else {
    // adjust servo with both lane lines
    ret.push_back(choose_servo(left_1, right_1, mean));
  }
  if (left_1 != 0 || right_1 != 0) {
    if (left_2 != 0 && right_2 != 0) {
      if (left_3 != 0 && right_3 != 0) {
        if (current_speed < base_speed * cfg.speed_up_max) {
          ret.push_back(current_speed + base_speed * cfg.speed_up_rate);
        } else {
          ret.push_back(base_speed * cfg.speed_up_max);
        }
      } else {
        if (current_speed > base_speed) {
          ret.push_back(current_speed - base_speed * cfg.speed_up_rate);
        } else if (current_speed < base_speed) {
          ret.push_back(current_speed + base_speed * cfg.speed_up_rate);
        } else {
          ret.push_back(base_speed);
        }
      }
    } else {
      if (current_speed > base_speed * cfg.speed_up_min) {
        ret.push_back(current_speed - base_speed * cfg.speed_up_rate);
      } else {
        ret.push_back(base_speed * cfg.speed_up_min);
      }
    }
  } else {
    ret.push_back(base_speed * cfg.speed_up_min);
  }
  return ret;
}
/* -------------------------------------------------------------- */
/*** double find_avg_point_on_line(cv::Mat frame_pixels, cv::Mat frame_image, int line_y, int line_start, int
line_stop, int param)
**
**   Parameters:
**     frame_pixels:    the image from which the lane is identified
**          frame_image:              the image where the detected line is displayed
**     line_y:      the detection line y coordinate
**          line_start:                        the detection line x coordinate to the left
**     line_stop:       the detection line x coordinate to the right
**          param:                             the debug parameter
**
**   Return Value:
**      The x position of the lane line on the detection area.
*/
double find_avg_point_on_line(cv::Mat frame_pixels, cv::Mat frame_image, int line_y, int line_start, int line_stop,
int param) {
  /* the body of this function is cut from this document to save space */
}
```

```
/* ------------------------------------------------------------ */
/*** int servo_speed_adj(int servo_no_adj, int current_speed)
**
**   Parameters:
**     servo_no_adj:    the raw servo command
**          current_speed:            the current speed of the car
**
**   Return Value:
**      The speed adjusted servo command.
*/
int servo_speed_adj(int servo_no_adj, int current_speed) {
  /* the body of this function is cut from this document to save space */
}
/* ------------------------------------------------------------ */
/*** std::vector<double> average_lane_lines(cv::Mat frame_pixels, cv::Mat frame_image, int param)
**
**   Parameters:
**     frame_pixels:    the image from which the lane is identified
**          frame_image:              the image where the detected line is displayed
**          param:                            the debug parameter
**
**   Return Value:
**      A vector containing the detected coordinates from the 6 detection zones.
*/
std::vector<double> average_lane_lines(cv::Mat frame_pixels, cv::Mat frame_image, int param) {
  /* the body of this function is cut from this document to save space */
}
/* ------------------------------------------------------------ */
/*** std::vector<int> servo_and_speed(cv::Mat frame_pixels, cv::Mat frame_image, int param, int current_speed, int base_speed, int lane_keep)
**
**   Parameters:
**     frame_pixels:    the image from which the lane is identified
**          frame_image:              the image where the data is displayed
**          param:                            the debug parameter
**          current_speed:            the current speed of the car
**          base_speed:                       the speed set by the user
**          lane_keep:                        the lanes considered in deciding the servo output
**
**   Return Value:
**      A vector containing the final computations of the servo command and possible speed.
*/
std::vector<int> servo_and_speed(cv::Mat frame_pixels, cv::Mat frame_image, int param, int current_speed, int base_speed, int lane_keep) {
  int servo_no_adj;
  double left_avg_1, right_avg_1, left_avg_2, right_avg_2, left_avg_3, right_avg_3;
  std::vector<double> averages = average_lane_lines(frame_pixels, frame_image, param);
  left_avg_1 = averages[0];
  right_avg_1 = averages[1];
  left_avg_2 = averages[2];
  right_avg_2 = averages[3];
  left_avg_3 = averages[4];
  right_avg_3 = averages[5];
  int servo_left1 = 0, servo_right1 = 0, servo_left2 = 0, servo_right2 = 0, servo_left3 = 0, servo_right3 = 0;
  if (left_avg_1 != -1) {
    servo_left1 = map_servo_fine(left_avg_1, cfg.left_x_1_1, cfg.left_x_2_1, cfg.left_mean_1);
    if (left_avg_2 != -1) {
      servo_left2 = map_servo_fine(left_avg_2, cfg.left_x_1_2, cfg.left_x_2_2, cfg.left_mean_2);
      if (left_avg_3 != -1) {
        servo_left3 = map_servo_fine(left_avg_3, cfg.left_x_1_3, cfg.left_x_2_3, cfg.left_mean_3);
      }
    }
  }
  if (right_avg_1 != -1) {
```

```
    servo_right1 = map_servo_fine(right_avg_1, cfg.right_x_1_1, cfg.right_x_2_1, cfg.right_mean_1);
    if (right_avg_2 != -1) {
      servo_right2 = map_servo_fine(right_avg_2, cfg.right_x_1_2, cfg.right_x_2_2, cfg.right_mean_2);
      if (right_avg_3 != -1) {
        servo_right3 = map_servo_fine(right_avg_3, cfg.right_x_1_3, cfg.right_x_2_3, cfg.right_mean_3);
      }
    }
  }
  if (param == 1 || param == -1) {
    COUT_mutex.lock();
    std::cout << "lft  = " << servo_left1 << ", rgt = " << servo_right1 << "\n";
    COUT_mutex.unlock();
  }
  int posible_speed = 0;
  std::vector<int> srv_spd = choose_advanced_servo(servo_left1, servo_right1, servo_left2, servo_right2,
servo_left3, servo_right3, SERVO_CENTER, current_speed, base_speed, lane_keep);
  servo_no_adj = srv_spd[0];
  posible_speed = srv_spd[1];
  std::vector<int> ret;
  ret.push_back(servo_speed_adj(servo_no_adj, posible_speed));
  ret.push_back(posible_speed);
  return ret;
}
/* ------------------------------------------------------------ */
/*** void draw_accel(cv::Mat frame, float accel_x, float accel_y, int area_corner_x, int area_corner_y)
**
**   Parameters:
**     frame:       the image where the data is displayed
**          accel_x:         the x value of the acceleration
**          accel_y:         the y value of the acceleration
**          area_corner_x:   the top left corner of the draw zone
**          area_corner_y:   the top left corner of the draw zone
**
**   Return Value:
**      None.
*/
void draw_accel(cv::Mat frame, float accel_x, float accel_y, int area_corner_x, int area_corner_y) {
  /* the body of this function is cut from this document to save space */
}
/* ------------------------------------------------------------ */
/*** int detect_and_display(cv::Mat frame, int param)
**
**   Parameters:
**     frame:       the image where the data is displayed
**          param:                    the debug parameter
**
**   Return Value:
**      Stop sign flag.
*/
int detect_and_display(cv::Mat frame, int param) {
  int ret = 0;
  std::vector<cv::Rect> stop_signs;
  cv::Mat frame_gray;
  cv::cvtColor(frame, frame_gray, cv::COLOR_BGR2GRAY);
  cv::equalizeHist(frame_gray, frame_gray);
  stop_cascade.detectMultiScale(frame_gray, stop_signs, 1.1, 2, 0 | cv::CASCADE_SCALE_IMAGE, cv::Size(30, 30));
  if (param == 1 || param == -1) {
    COUT_mutex.lock();
    std::cout << "stop signs = " << stop_signs.size() << "\n";
    COUT_mutex.unlock();
  }
  for (size_t i = 0; i < stop_signs.size(); i++) {
    if (param == 1 || param == -1) {
      COUT_mutex.lock();
```

```
      std::cout << "stop " << i << " height = " << stop_signs[i].height << "\n";
      std::cout << "stop " << i << " width = " << stop_signs[i].width << "\n";
      COUT_mutex.unlock();
    }
    // draw an ellipse around the sign
    if (cfg.draw == 2 && (param == 2 || param == -1)) {
      cv::Point center(stop_signs[i].x + stop_signs[i].width / 2, stop_signs[i].y + stop_signs[i].height / 2);
      cv::ellipse(frame, center, cv::Size(stop_signs[i].width / 2, stop_signs[i].height / 2), 0, 0, 360, cv::Scalar(255, 0, 255), 4, 8, 0);
    }
    if (stop_signs[i].height >= cfg.sign_min && stop_signs[i].height <= cfg.sign_max) {
      cv::Scalar clr = cv::mean(crop(frame, stop_signs[i].x, stop_signs[i].y, stop_signs[i].x + stop_signs[i].width, stop_signs[i].y + stop_signs[i].height));
      if (clr.val[2] >= clr.val[1] && clr.val[2] >= clr.val[0]) {
        ret = 1;
      }
    }
  }
  return ret;
}
/* ------------------------------------------------------------ */
/*** std::vector<int> speed_and_stop(int param, cv::Mat frame_stream, FILE* sonar, int current_speed, int base_speed, int stop_time, int posible_speed, int lane_keep)
**
**   Parameters:
**          param:                       the debug parameter
**          frame_stream:            the image where the data is displayed
**          sonar:                       the file pointer to /dev/sonar
**          rgbled:                      the file pointer to /dev/rgbled
**          current_speed:           the current speed of the car
**          base_speed:                  the speed set by the user
**          stop_time:                   the time the car needs to be still
**          possible_speed:          THe possible speed of the car
**          lane_keep:                   the lanes considered in deciding the servo output
**
**   Return Value:
**     A vector with:
**      - [0] current speed
**      - [1] base speed
**      - [2] stop time
**      - [3] distance to object in front
**      - [4] which lane to keep
*/
std::vector<int> speed_and_stop(int param, cv::Mat frame_stream, FILE* sonar, FILE* rgbled, int current_speed, int base_speed, int stop_time, int posible_speed, int lane_keep) {
  std::vector<int> ret;
  int cur_spd = current_speed;
  int bas_spd = base_speed;
  int stp_tim = stop_time;
  int lan_kep = lane_keep;
  int dst = 0;
  int stop_sgn = 0;
  static int old_stop_sgn = 0;
  int cascade_flag = 0;
  static int color;
  if (stop_time <= 0) {
    cur_spd = posible_speed;
    stp_tim = 0;
    // Sonar part
    if (cfg.sonar_dist > 1) {
      int clk_edges;
      read(sonar->_fileno, &clk_edges, 4);
      int dist = clk_edges * CLK_TO_CM
      ;
```

```
    dst = dist;
    if (dist < cfg.sonar_dist && dist != 0) {
      cur_spd = 0;
      stp_tim = 1;
      cascade_flag = 1;
      color = SONAR_COLOR;
      write(rgbled->_fileno, &color, 4);
    } else {
      if (color == SONAR_COLOR) {
        color = 0;
        write(rgbled->_fileno, &color, 4);
      }
    }
  }
  // Sign part
  if (cascade_flag == 0 && cfg.sign_on == 1) {
    STOP_mutex.lock();
    stop_sgn = stop_sign;
    STOP_mutex.unlock();
    if (stop_sgn == 1) {
      color = STOP_COLOR;
      write(rgbled->_fileno, &color, 4);
    } else {
      if (color == STOP_COLOR) {
        color = 0;
        write(rgbled->_fileno, &color, 4);
      }
    }
    if (stop_sgn == 0 && old_stop_sgn == 1) {
      cur_spd = 0;
      stp_tim = 1000000;
      cascade_flag = 1;
    }
    old_stop_sgn = stop_sgn;
  }
  // RFID part
  if (cascade_flag == 0 && cfg.rfid_on == 1) {
    struct card * card_now = popCard(c_queue);
    if (card_now != NULL) {
      if (card_now->type < 100) {
        // speed card
        if (param == 1 || param == -1) {
          COUT_mutex.lock();
          std::cout << "SPEED" << card_now->type << " CARD" << "\n";
          COUT_mutex.unlock();
        }
        if (cfg.draw == 1 && (param == 2 || param == -1)) {
          char card_str[30];
          sprintf(card_str, "SPEED%d CARD", card_now->type);
          cv::putText(frame_stream, card_str, cvPoint(250, 300), cv::FONT_HERSHEY_COMPLEX_SMALL, 0.8, cvScalar(0, 0, 255), 1, CV_AA);
        }
        bas_spd = card_now->type * 1000;
        cascade_flag = 1;
        color = SPEED_COLOR;
        write(rgbled->_fileno, &color, 4);
      } else {
        switch (card_now->type) {
        case STOP:
          if (param == 1 || param == -1) {
            COUT_mutex.lock();
            std::cout << "STOP CARD" << "\n";
            COUT_mutex.unlock();
          }
```

```
        if (cfg.draw == 1 && (param == 2 || param == -1)) {
          cv::putText(frame_stream, "STOP CARD", cvPoint(250, 300), cv::FONT_HERSHEY_COMPLEX_SMALL, 0.8, cvScalar(0, 0, 255), 1, CV_AA);
        }
        cur_spd = 0;
        stp_tim = 3000000;
        cascade_flag = 1;
        color = HALT_COLOR;
        write(rgbled->_fileno, &color, 4);
        break;
      case PAUSE:
        if (param == 1 || param == -1) {
          COUT_mutex.lock();
          std::cout << "PAUSE CARD" << "\n";
          COUT_mutex.unlock();
        }
        if (cfg.draw == 1 && (param == 2 || param == -1)) {
          cv::putText(frame_stream, "PAUSE CARD", cvPoint(250, 300), cv::FONT_HERSHEY_COMPLEX_SMALL, 0.8, cvScalar(0, 0, 255), 1, CV_AA);
        }
        cur_spd = 0;
        stp_tim = 1000000;
        cascade_flag = 1;
        color = HALT_COLOR;
        write(rgbled->_fileno, &color, 4);
        break;
      case KEEPR:
        if (param == 1 || param == -1) {
          COUT_mutex.lock();
          std::cout << "KEEP RIGHT CARD" << "\n";
          COUT_mutex.unlock();
        }
        if (cfg.draw == 1 && (param == 2 || param == -1)) {
          cv::putText(frame_stream, "KEEP RIGHT CARD", cvPoint(250, 300), cv::FONT_HERSHEY_COMPLEX_SMALL, 0.8, cvScalar(0, 0, 255), 1, CV_AA);
        }
        if (lan_kep == 0) {
          lan_kep = 1;
        } else {
          lan_kep = 0;
        }
        cascade_flag = 1;
        break;
      case KEEPL:
        if (param == 1 || param == -1) {
          COUT_mutex.lock();
          std::cout << "KEEP LEFT CARD" << "\n";
          COUT_mutex.unlock();
        }
        if (cfg.draw == 1 && (param == 2 || param == -1)) {
          cv::putText(frame_stream, "KEEP LEFT CARD", cvPoint(250, 300), cv::FONT_HERSHEY_COMPLEX_SMALL, 0.8, cvScalar(0, 0, 255), 1, CV_AA);
        }
        if (lan_kep == 0) {
          lan_kep = -1;
        } else {
          lan_kep = 0;
        }
        cascade_flag = 1;
        break;
      case KEEPLR:
        if (param == 1 || param == -1) {
          COUT_mutex.lock();
          std::cout << "KEEP BOTH CARD" << "\n";
```

```
              COUT_mutex.unlock();
            }
            if (cfg.draw == 1 && (param == 2 || param == -1)) {
              cv::putText(frame_stream, "KEEP BOTH CARD", cvPoint(250, 300), cv::FONT_HERSHEY_COMPLEX_SMALL, 0.8, cvScalar(0, 0, 255), 1, CV_AA);
            }
            lan_kep = 0;
            cascade_flag = 1;
            break;
          }
          if (lan_kep != 0) {
            color = SINGLE_LANE_COLOR;
            write(rgbled->_fileno, &color, 4);
          } else {
            color = 0;
            write(rgbled->_fileno, &color, 4);
          }
        }
        free(card_now);
      }
    }
  }
  ret.push_back(cur_spd);
  ret.push_back(bas_spd);
  ret.push_back(stp_tim);
  ret.push_back(dst);
  ret.push_back(lan_kep);
  return ret;
}
/* -------------------------------------------------------------- */
/*** void my_handler(int s)
**
**   Parameters:
**          s:                        signal received
**
**   Return Value:
**     None.
*/
void my_handler(int s) {
  /* the body of this function is cut from this document to save space */
}
/* -------------------------------------------------------------- */
/*** void lane_component(int param, int iterations, FILE* camera, FILE* servo, FILE* motors, FILE* sonar, FILE* acl, unsigned short usr_speed, int h, int w, int l)
**
**   Parameters:
**          param:                            the debug parameter
**          iterations:                       the number of user set iterations
**          camera:                           the file pionter to /dev/videoHLS
**          servo:                            the file pointer to /dev/servo
**          motors:                           the file pointer to /dev/motors
**          sonar:                            the file pointer to /dev/sonar
**          acl:                              the file pointer to /dev/i2c-1 (accelerometer)
**          rgbled:                           the file pointer to /dev/rgbled
**          usr_speed:                        the user assigned speed
**          h:                                        height of the stream
**          w:                                        width of the stream
**          l:                                        color depth of the stream
**
**   Return Value:
**     None.
*/
void lane_component(int param, int iterations, FILE* camera, FILE* servo, FILE* motors, FILE* sonar, FILE* acl, FILE* rgbled, unsigned short usr_speed, int h, int w, int l) {
```

```
int servo_out = 300;
int old_servo_out = servo_out;
int dist = 0;
unsigned int speed;
unsigned int posible_speed;
int current_speed = usr_speed;
int base_speed = usr_speed;
int stop_time = 0;
int lane_keep = 0;
unsigned char* pixels;
pixels = (unsigned char *) malloc(h * w * l * sizeof(char));
auto start = std::chrono::high_resolution_clock::now();
auto finish = std::chrono::high_resolution_clock::now();
auto duration = std::chrono::duration_cast<std::chrono::microseconds>(finish - start);
int full_time = 0;
int loop;
try {
  for (loop = 0; loop < iterations && lane_done == 0; loop++) {
    start = std::chrono::high_resolution_clock::now();
    COUT_mutex.lock();
    std::cout << "loop=============" << loop << "\n";
    COUT_mutex.unlock();
    cv::Mat frame_1, frame, frame_stream;
    fread(pixels, 1, h * w * l, camera);
    frame_1 = cv::Mat(h, w, CV_8UC1, &pixels[0]);
    if (param != 100) {
      cv::resize(frame_1, frame, cv::Size(), cfg.resize_factor, cfg.resize_factor);
    } else {
      frame_1.copyTo(frame);
    }
    if (cfg.draw == 1 && (param == 2 || param == -1)) {
      cv::cvtColor(frame, frame_stream, cv::COLOR_GRAY2BGR);
    }
    std::vector<int> srv_spd = servo_and_speed(frame, frame_stream, param, current_speed, base_speed, lane_keep);
    servo_out = srv_spd[0];
    posible_speed = srv_spd[1];
    if (servo_out == -1) {
      servo_out = old_servo_out;
    }
    if (servo_out <= SERVO_LEFT)
      servo_out = SERVO_LEFT;
    if (servo_out >= SERVO_RIGHT)
      servo_out = SERVO_RIGHT;
    old_servo_out = servo_out;
    std::vector<int> big_speed = speed_and_stop(param, frame_stream, sonar, rgbled, current_speed, base_speed,
stop_time, posible_speed, lane_keep);
    current_speed = big_speed[0];
    base_speed = big_speed[1];
    stop_time = big_speed[2];
    dist = big_speed[3];
    lane_keep = big_speed[4];
    //speed that is written to motors
    speed = ((unsigned short) current_speed << 16) + (unsigned short) current_speed;
    int mtr_write = write(motors->_fileno, &speed, 4);
    if (mtr_write < 4) {
      COUT_mutex.lock();
      std::cerr << "Failed write to motors." << "\n";
      COUT_mutex.unlock();
      break;
    }
    int srv_write = write(servo->_fileno, &servo_out, 2);
    if (srv_write < 2) {
      COUT_mutex.lock();
      std::cerr << "Failed write to servo." << "\n";
```

```
        COUT_mutex.unlock();
        break;
      }
      if (param == 1 || param == -1) {
        COUT_mutex.lock();
        std::cout << "servo = " << servo_out << "\n";
        std::cout << "speed = " << (speed >> 16) << "\n";
        std::cout << "dist  = " << dist << "\n";
        COUT_mutex.unlock();
      }
      if (cfg.draw == 1 && (param == 2 || param == -1)) {
        cv::line(frame_stream, cv::Point(cfg.right_x_1_1, cfg.y_1), cv::Point(cfg.right_x_2_1, cfg.y_1), cv::Scalar(100, 0, 0), 2, CV_AA);
        cv::line(frame_stream, cv::Point(cfg.left_x_1_1, cfg.y_1), cv::Point(cfg.left_x_2_1, cfg.y_1), cv::Scalar(0, 100, 0), 2, CV_AA);
        cv::line(frame_stream, cv::Point(cfg.right_x_1_2, cfg.y_2), cv::Point(cfg.right_x_2_2, cfg.y_2), cv::Scalar(100, 0, 0), 2, CV_AA);
        cv::line(frame_stream, cv::Point(cfg.left_x_1_2, cfg.y_2), cv::Point(cfg.left_x_2_2, cfg.y_2), cv::Scalar(0, 100, 0), 2, CV_AA);
        cv::line(frame_stream, cv::Point(cfg.right_x_1_3, cfg.y_3), cv::Point(cfg.right_x_2_3, cfg.y_3), cv::Scalar(100, 0, 0), 2, CV_AA);
        cv::line(frame_stream, cv::Point(cfg.left_x_1_3, cfg.y_3), cv::Point(cfg.left_x_2_3, cfg.y_3), cv::Scalar(0, 100, 0), 2, CV_AA);
        cv::line(frame_stream, cv::Point(cfg.left_mean_1, cfg.y_1 - 5), cv::Point(cfg.left_mean_1, cfg.y_1 + 5), cv::Scalar(0, 255, 255), 2, CV_AA);
        cv::line(frame_stream, cv::Point(cfg.right_mean_1, cfg.y_1 - 5), cv::Point(cfg.right_mean_1, cfg.y_1 + 5), cv::Scalar(0, 255, 255), 2, CV_AA);
        cv::line(frame_stream, cv::Point(cfg.left_mean_2, cfg.y_2 - 5), cv::Point(cfg.left_mean_2, cfg.y_2 + 5), cv::Scalar(0, 255, 255), 2, CV_AA);
        cv::line(frame_stream, cv::Point(cfg.right_mean_2, cfg.y_2 - 5), cv::Point(cfg.right_mean_2, cfg.y_2 + 5), cv::Scalar(0, 255, 255), 2, CV_AA);
        cv::line(frame_stream, cv::Point(cfg.left_mean_3, cfg.y_3 - 5), cv::Point(cfg.left_mean_3, cfg.y_3 + 5), cv::Scalar(0, 255, 255), 2, CV_AA);
        cv::line(frame_stream, cv::Point(cfg.right_mean_3, cfg.y_3 - 5), cv::Point(cfg.right_mean_3, cfg.y_3 + 5), cv::Scalar(0, 255, 255), 2, CV_AA);
        char lop[20];
        sprintf(lop, "loop = %d", loop);
        cv::putText(frame_stream, lop, cvPoint(30, 30), cv::FONT_HERSHEY_COMPLEX_SMALL, 0.8, cvScalar(0, 0, 255), 1, CV_AA);
        char srv[20];
        sprintf(srv, "servo = %d", servo_out);
        cv::putText(frame_stream, srv, cvPoint(30, 60), cv::FONT_HERSHEY_COMPLEX_SMALL, 0.8, cvScalar(0, 0, 255), 1, CV_AA);
        char spd[20];
        sprintf(spd, "speed = %d", speed >> 16);
        cv::putText(frame_stream, spd, cvPoint(30, 90), cv::FONT_HERSHEY_COMPLEX_SMALL, 0.8, cvScalar(0, 0, 255), 1, CV_AA);
        char dst[20];
        sprintf(dst, "dist = %d", dist);
        cv::putText(frame_stream, dst, cvPoint(30, 120), cv::FONT_HERSHEY_COMPLEX_SMALL, 0.8, cvScalar(0, 0, 255), 1, CV_AA);
        if (cfg.acl_on == 1) {
          float accel_x, accel_y, accel_z;
          ACL_ReadAccelG(acl->_fileno, &accel_x, &accel_y, &accel_z);
          accel_x *= 9.8f;
          accel_y *= 9.8f;
          draw_accel(frame_stream, accel_x, accel_y, 500, 30);
        }
        try {
          cv::imwrite("/etc/mjpg-stream.jpg", frame_stream);
          if (param == -1) {
            char name[65];
            sprintf(name, "img_lane%d.jpg", loop);
            cv::imwrite(name, frame_stream);
```

```
        }
      } catch (std::runtime_error& ex) {
        COUT_mutex.lock();
        std::cerr << "Exception writing image: " << ex.what() << "\n";
        COUT_mutex.unlock();
      }
    }
    sigaction(SIGINT, &sigIntHandler, NULL);
    finish = std::chrono::high_resolution_clock::now();
    duration = std::chrono::duration_cast<std::chrono::microseconds>(finish - start);
    if (cfg.fps != -1 && duration.count() < cfg.loop_time) {
      usleep(cfg.loop_time - duration.count());
    }
    full_time += duration.count();
    stop_time -= duration.count();
  }
  } catch (...) {
    std::cout << "EX_T1" << "\n";
  }
  lane_done = 1;
  if (cfg.rfid_on == 1) {
    fake_interrupt();
  }
  COUT_mutex.lock();
  std::cout << "LANE time/loop = " << full_time / (double) loop << "\n";
  COUT_mutex.unlock();
}
/* ------------------------------------------------------------- */
/*** void sign_component(int param, FILE* camera, int h, int w, int l)
**
**   Parameters:
**          param:                           the debug parameter
**          camera:                          the file pionter to /dev/video
**          h:                                       height of the stream
**          w:                                       width of the stream
**          l:                                       color depth of the stream
**
**   Return Value:
**     None.
*/
void sign_component(int param, FILE* camera, int h, int w, int l) {
  auto start = std::chrono::high_resolution_clock::now();
  auto finish = std::chrono::high_resolution_clock::now();
  auto duration = std::chrono::duration_cast<std::chrono::microseconds>(finish - start);
  int sign = 0;
  int loop = 0;
  double full_time = 0;
  unsigned char* pixels;
  pixels = (unsigned char *) malloc(h * w * l * sizeof(char));
  try {
    while (lane_done == 0) {
      start = std::chrono::high_resolution_clock::now();
      cv::Mat frame;
      fread(pixels, 1, h * w * l, camera);
      frame = cv::Mat(h, w, CV_8UC3, &pixels[0]);
      if (frame.rows != 0) {
        cv::Mat region_image_sign, frame_stream;
        region_image_sign = crop(frame, frame.cols / 2, 0, frame.cols, frame.rows);
        cv::resize(region_image_sign, frame_stream, cv::Size(), cfg.resize_factor, cfg.resize_factor);
        sign = detect_and_display(frame_stream, param);
        STOP_mutex.lock();
        stop_sign = sign;
        STOP_mutex.unlock();
        if (cfg.draw == 2 && (param == 2 || param == -1)) {
```

```
          try {
            cv::imwrite("/etc/mjpg-stream.jpg", frame_stream);
            if (param == -1) {
              char name[65];
              sprintf(name, "img_sign%d.jpg", loop);
              cv::imwrite(name, frame);
            }
          } catch (std::runtime_error& ex) {
            std::cerr << "Exception writing image: " << ex.what() << "\n";
          }
        }
        finish = std::chrono::high_resolution_clock::now();
        duration = std::chrono::duration_cast<std::chrono::microseconds>(finish - start);
        if (cfg.fps != -1 && duration.count() < cfg.loop_time) {
          usleep(cfg.loop_time - duration.count());
        }
        loop++;
        full_time += duration.count();
      }
    }
  } catch (...) {
    std::cout << "EX_T2" << "\n";
  }
  COUT_mutex.lock();
  std::cout << "SIGN time/loop = " << full_time / (double) loop << "\n";
  COUT_mutex.unlock();
}
/* --------------------------------------------------------------- */
/*** void runRFID(int fd, struct cardQueue *queue)
**
**   Parameters:
**          fd:                                          the RFID file descriptor
**          queue:                              the queue in which the RFID cards are stored
**
**   Return Value:
**     None.
*/
void RFID_component(int fd, struct cardQueue *queue) {
  uint8_t success;
  uint8_t uid[6];
  uint8_t uidLength;
  uint8_t keys[6] = { 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF }; //default key
  uint8_t block = 4;
  uint8_t numCards = 0;
  try {
    while (lane_done == 0) {
#ifdef RFID_DEBUG
      printf("Waiting for card\n");
#endif
      memset(uid, 0, 6);
      success = 0;
      while (success != 1 && lane_done == 0) {
        success = readPassiveTargetID(fd, PN532_MIFARE_ISO14443A, uid, &uidLength, 1);
        if (success == 1)
          break;
      }
      if (success) {
        if (uidLength != 4)
          continue;
        success = mifareclassic_AuthenticateBlock(fd, uid, uidLength, block, 0, keys);
        if (success) {
          uint8_t data[16];
          success = mifareclassic_ReadDataBlock(fd, block, data);
          if (success) {
```

```
          struct card *card = (struct card*) malloc(sizeof(card));
          card->type = data[0];
          memcpy(&card->UID, uid, 4);
          uint8_t result = insertCard(queue, card);
          if (result == 1) {
            numCards++;
#ifdef RFID_DEBUG
            printf("Inserted card\n");
#endif
          }
        } else {
#ifdef RFID_DEBUG
          printf("Read block failed.\n");
#endif
        }
      } else {
#ifdef RFID_DEBUG
        printf("Block auth failed.\n");
#endif
      }
    }
  }
  } catch (...) {
    std::cout << "EX_T3" << "\n";
  }
}
int main(int argc, char** argv) {
  std::cout << "OpenCV version : " << CV_VERSION << "\n";
  if (argc != 4) {
    std::cerr << argv[0] << " <debug param> <number of iterations> <speed>" << "\n" << "0 - nothing" << "\n" << "1 - just text" << "\n" << "2 - just images" << "\n" << "\n" << "-1 - text and images" << "\n" << "100 - calibration mode" << "\n";
    return -1;
  }
  // get the arguments
  int param = atoi(argv[1]);
  int iterations = atoi(argv[2]);
  unsigned short usr_speed = atoi(argv[3]);
  // check parameters
  if (iterations < 0) {
    std::cerr << "Bad number of iterations." << "\n";
    return -1;
  } else if (iterations == 42) {
    iterations = 100000;
  }
  if (param == 100) {
    iterations = 1;
  }
  std::vector<FILE*> fp;
  std::string cfg_name = "prop.cfg";
  std::vector<double> calib_avg;
  // initialize the cfg
  cfg = configure(param, cfg_name, calib_avg);
  // open the different devices and check if they have opened correctly
  FILE* camera_lane = fopen("/dev/videoHLS", "rb");
  if (camera_lane < 0) {
    std::cerr << "Failed to open lane camera." << "\n";
    close_fp(fp);
    return -1;
  }
  fp.push_back(camera_lane);
  FILE* camera_sign = fopen("/dev/video", "rb");
  if (camera_sign < 0) {
    std::cerr << "Failed to open sign camera." << "\n";
```

```
    close_fp(fp);
    return -1;
  }
  fp.push_back(camera_sign);
  FILE* servo = fopen("/dev/servo", "r+b");
  if (servo < 0) {
    std::cerr << "Failed to open servo." << "\n";
    close_fp(fp);
    return -1;
  }
  fp.push_back(servo);
  FILE* motors = fopen("/dev/motors", "r+b");
  if (motors < 0) {
    std::cerr << "Failed to open motors." << "\n";
    close_fp(fp);
    return -1;
  }
  fp.push_back(motors);
  FILE* sonar;
  if (cfg.sonar_dist > 1) {
    sonar = fopen("/dev/sonar", "rb");
    if (sonar < 0) {
      std::cerr << "Failed to open sonar." << "\n";
      close_fp(fp);
      return -1;
    }
    fp.push_back(sonar);
  }
  FILE* rgbled = fopen("/dev/rgbled", "r+b");
  if (rgbled < 0) {
    std::cerr << "Failed to open rgbled." << "\n";
    close_fp(fp);
    return -1;
  }
  fp.push_back(rgbled);
  FILE* acl;
  if (cfg.acl_on == 1) {
    acl = fopen("/dev/i2c-1", "r+b");
    if (acl < 0) {
      std::cerr << "Failed to open acl." << "\n";
      close_fp(fp);
      return -1;
    }
    fp.push_back(acl);
    if (ACL_Init(acl->_fileno) < 0) {
      std::cerr << "Failed to initialize acl." << "\n";
      close_fp(fp);
      return -1;
    }
  }
  int rfid = -1;
  if (cfg.rfid_on == 1) {
    rfid = initRFID();
    createCardQueue(&c_queue);
    if (rfid < 0) {
      std::cerr << "Failed to open RFID." << "\n";
      close_fp(fp);
      return -1;
    }
    f_rfid = rfid;
  }
  f_motors = motors->_fileno;
  f_servo = servo->_fileno;
  f_rgbled = rgbled->_fileno;
```

```
// initialize motors direction (the car is always going forward)
unsigned int left_dir = 1;
unsigned int right_dir = left_dir;
ioctl(motors->_fileno, MOTION_IOCTSETDIR, ((left_dir & 1) << 1) + (right_dir & 1));
// enable motors
unsigned int enable = 1;
ioctl(motors->_fileno, MOTION_IOCTSETENABLE, enable);
// center wheels
int stock_servo_out = SERVO_CENTER;
write(servo->_fileno, &stock_servo_out, 2);
// set starting speed to 0
int stock_speed = 0;
write(motors->_fileno, &stock_speed, 4);
// turn off the led
int color = 0;
write(rgbled->_fileno, &color, 4);
// load Haar cascade
if (!stop_cascade.load(stop_cascade_name)) {
  std::cerr << "Failed to load stop sign cascade" << "\n";
  close_fp(fp);
  if (cfg.rfid_on == 1)
    closeRFID(rfid);
  return -1;
};
// set SIG_INT handler
memset(&sigIntHandler, 0, sizeof(sigIntHandler));
sigIntHandler.sa_flags = SA_RESETHAND;
sigIntHandler.sa_handler = my_handler;
sigaction(SIGINT, &sigIntHandler, NULL);
// get canfiguration data from the 2 video streams
int h_lane = ioctl(camera_lane->_fileno, CHARVIDEO_IOCQHEIGHT);
int w_lane = ioctl(camera_lane->_fileno, CHARVIDEO_IOCQWIDTH);
int l_lane = ioctl(camera_lane->_fileno, CHARVIDEO_IOCQPIXELLEN);
int h_sign = ioctl(camera_sign->_fileno, CHARVIDEO_IOCQHEIGHT);
int w_sign = ioctl(camera_sign->_fileno, CHARVIDEO_IOCQWIDTH);
int l_sign = ioctl(camera_sign->_fileno, CHARVIDEO_IOCQPIXELLEN);
if (param != 100) {
  // the main part of the algorithm, where the threads are started if needed
  std::thread t1, t2, t3;
  t1 = std::thread(lane_component, param, iterations, camera_lane, servo, motors, sonar, acl, rgbled, usr_speed,
h_lane, w_lane, l_lane);
  if (cfg.sign_on == 1) {
    t2 = std::thread(sign_component, param, camera_sign, h_sign, w_sign, l_sign);
  }
  if (cfg.rfid_on == 1) {
    t3 = std::thread(RFID_component, rfid, c_queue);
  }
  t1.join();
  std::cout << "T1 joined" << "\n";
  if (cfg.sign_on == 1) {
    t2.join();
    std::cout << "T2 joined" << "\n";
  }
  if (cfg.rfid_on == 1) {
    t3.join();
    std::cout << "T3 joined" << "\n";
  }
} else {
  // the calibration part of the algorithm overwrites the means of the
  // detection zones with ones calculated from the current image and the config file is changed
  unsigned char* pixels;
  pixels = (unsigned char *) malloc(h_lane * w_lane * l_lane * sizeof(char));
  fread(pixels, 1, h_lane * w_lane * l_lane, camera_lane);
  cv::Mat calib_img = cv::Mat(h_lane, w_lane, CV_8UC1, &pixels[0]);
```

```
    calib_avg = average_lane_lines(calib_img, calib_img, param);
    cfg = configure(param, cfg_name, calib_avg);
  }
  // the end of the algorithm, threads are joined, and for good
  // measure, the servo and motors are reverted to their default positions
  stock_servo_out = SERVO_CENTER;
  write(servo->_fileno, &stock_servo_out, 2);
  stock_speed = 0;
  write(motors->_fileno, &stock_speed, 4);
  color = 0;
  write(rgbled->_fileno, &color, 4);
  close_fp(fp);
  if (cfg.rfid_on == 1) {
    closeRFID(rfid);
    freeCardQueue(c_queue);
  }
  return 0;
}
```

Appendix B: canny_edge_detection.cpp

```
// MIT License. 2019 Yuya Kudo. https://github.com/kyk0910
#include "canny_edge_detection.h"
using namespace hls;
using namespace hlsimproc;

uint8_t fifo1[MAX_WIDTH * MAX_HEIGHT];
uint8_t fifo2[MAX_WIDTH * MAX_HEIGHT];
GradPix fifo3[MAX_WIDTH * MAX_HEIGHT];
uint8_t fifo4[MAX_WIDTH * MAX_HEIGHT];
uint8_t fifo5[MAX_WIDTH * MAX_HEIGHT];
uint8_t fifo6[MAX_WIDTH * MAX_HEIGHT];
uint8_t fifo7[MAX_WIDTH * MAX_HEIGHT];

// Top Function
void canny_edge_detection(hls::stream<ap_axiu<24,1,1,1> >& axis_in,
hls::stream<ap_axiu<8,1,1,1> >& axis_out) {
    // interface directive
    #pragma HLS INTERFACE axis port=axis_in
    #pragma HLS INTERFACE axis port=axis_out
    #pragma HLS INTERFACE ap_ctrl_none port=return
    // pipeline directive
    #pragma HLS DATAFLOW
    // FIFO directive
    #pragma HLS STREAM variable=fifo1 depth=1 dim=1
    #pragma HLS STREAM variable=fifo2 depth=1 dim=1
    #pragma HLS STREAM variable=fifo3 depth=1 dim=1
    #pragma HLS STREAM variable=fifo4 depth=1 dim=1
    #pragma HLS STREAM variable=fifo5 depth=1 dim=1
    #pragma HLS STREAM variable=fifo6 depth=1 dim=1
    #pragma HLS STREAM variable=fifo7 depth=1 dim=1

    // AXI4-Stream -> GrayScale image
    HlsImProc::AXIS2GrayArray<MAX_WIDTH, MAX_HEIGHT>(axis_in, fifo1);
    // exe gaussian blur
    HlsImProc::GaussianBlur<MAX_WIDTH, MAX_HEIGHT>(fifo1, fifo2);
    // exe sobel filter
    HlsImProc::Sobel<MAX_WIDTH, MAX_HEIGHT>(fifo2, fifo3);
    // exe non-maximum suppression
    HlsImProc::NonMaxSuppression<MAX_WIDTH, MAX_HEIGHT>(fifo3, fifo4);
    // exe zero padding at boundary pixel
    const uint32_t PADDING_SIZE = 5;
    HlsImProc::ZeroPadding<MAX_WIDTH, MAX_HEIGHT>(fifo4, fifo5, PADDING_SIZE);
    // exe hysteresis threshold
    HlsImProc::HystThreshold<MAX_WIDTH, MAX_HEIGHT>(fifo5, fifo6, CANNY_HTHR,
CANNY_LTHR);
    // exe comparison operation at neighboring pixels
    HlsImProc::HystThresholdComp<MAX_WIDTH, MAX_HEIGHT>(fifo6, fifo7);
    // GrayScale image -> AXI4-Stream
    HlsImProc::Gray_Array2_AXIS<MAX_WIDTH, MAX_HEIGHT>(fifo7, axis_out);
}
```

Appendix C: PWM_Driver.vhd

Note: this file is relevant to Appendix D, which makes use of this module.

```
--------------------------------------------------------------------------------
-- Company:
-- Engineer: Catalin Bitire
--
-- Create Date: 12/24/2018 01:23:59 PM
-- Design Name:
-- Module Name: PWM_Driver - Behavioral
-- Project Name:
-- Target Devices:
-- Tool Versions:
-- Description: Single channel, customizable PWM driver. Set sys_clk to clk's frequency in Hz
--
-- Dependencies:
--
-- Revision:
-- Revision 0.01 - File Created
-- Additional Comments:
--
--------------------------------------------------------------------------------
LIBRARY IEEE;
USE IEEE.STD_LOGIC_1164.ALL;
USE ieee.std_logic_unsigned.ALL;
ENTITY PWM_Driver IS
    GENERIC (
        sys_clk : INTEGER := 50_000_000; --system clock frequency in Hz
        pwm_freq : INTEGER := 100_000; --PWM switching frequency in Hz
        bits_resolution : INTEGER := 16 --bits of resolution setting the duty cycle
    );
    PORT (
        duty : IN STD_LOGIC_VECTOR(bits_resolution - 1 DOWNTO 0);
        clk : IN STD_LOGIC;
        pwm_out : OUT STD_LOGIC;
        enable : IN STD_LOGIC --asynchronous reset
    );
END PWM_Driver;
ARCHITECTURE Behavioral OF PWM_Driver IS
    CONSTANT period : INTEGER := sys_clk/pwm_freq;
    SIGNAL count : INTEGER RANGE 0 TO period - 1 := 0;
    SIGNAL half_duty : INTEGER RANGE 0 TO period/2 := 0;
    SIGNAL half_duty_new : INTEGER RANGE 0 TO period/2 := 0;
    SIGNAL pwm_out_buf : std_logic := '0';
    SIGNAL disabled : std_logic := '1';
BEGIN
    PROCESS (clk, enable, disabled)
    BEGIN
        IF (enable = '0' AND pwm_out_buf = '0') THEN
            disabled <= '1';
        END IF;
        IF (enable = '1' AND disabled = '1' AND pwm_out_buf = '0') THEN
            disabled <= '0';
        END IF;
        IF (disabled = '1') THEN --asynchronous reset
            count <= 0; --clear counter
            pwm_out <= '0'; --clear pwm outputs
            pwm_out_buf <= '0';
```

```
        ELSIF (clk'EVENT AND clk = '1') THEN --rising system clock edge
            half_duty_new <= conv_integer(duty) * period/(2 ** bits_resolution)/2; --determine clocks in 1/2 duty cycle
            IF (count = period - 1) THEN --end of period reached
                count <= 0; --reset counter
                half_duty <= half_duty_new; --set most recent duty cycle value
            ELSE --end of period not reached
                count <= count + 1; --increment counter
            END IF;
            IF (count = half_duty) THEN --phase's falling edge reached
                pwm_out_buf <= '0';
                pwm_out <= '0'; --deassert the pwm output
            ELSIF (count = period - half_duty) THEN --phase's rising edge reached
                pwm_out_buf <= '1';
                pwm_out <= '1'; --assert the pwm output
            END IF;
        END IF;
    END PROCESS;
END Behavioral;
```

Appendix D: MotionController_v1_0_S00_AXI.vhd

```
----------------------------------------------------------------------------------
-- Company:
-- Engineer: Catalin Bitire
--
-- Create Date: 12/24/2018 01:00:00 PM
-- Design Name:
-- Module Name:
-- Project Name:
-- Target Devices:
-- Tool Versions:
-- Description: AXI module that controls a dual-motor driver and a servo motor
-- using 3 PWM drivers
--
-- Dependencies:
--
-- Revision:
-- Revision 0.01 - File Created
-- Additional Comments:
--
----------------------------------------------------------------------------------
LIBRARY ieee;
USE ieee.std_logic_1164.ALL;
--use ieee.numeric_std.all;
USE ieee.std_logic_unsigned.ALL;
USE ieee.std_logic_arith.ALL;
ENTITY MotionController_v1_0_S00_AXI IS
    GENERIC (
        -- Users to add parameters here
        servo_bits_resolution : INTEGER := 12; -- max 12
        motor_bits_resolution : INTEGER := 16; -- max 16
        sys_clk_frequency : INTEGER := 50_000_000; -- set this to the axi clk freq
        motor_pwm_frequency : INTEGER := 100_000;
        -- User parameters ends
        -- Do not modify the parameters beyond this line
        -- Width of S_AXI data bus
        C_S_AXI_DATA_WIDTH : INTEGER := 32;
        -- Width of S_AXI address bus
        C_S_AXI_ADDR_WIDTH : INTEGER := 4
    );
    PORT (
        -- Users to add ports here
        enable : IN std_logic;
        motor_left_dir_out : OUT std_logic;
        motor_right_dir_out : OUT std_logic;
        motor_left_pwm_out : OUT std_logic;
        motor_right_pwm_out : OUT std_logic;
        servo_pwm_out : OUT std_logic;
```

```
        -- User ports ends
        -- Do not modify the ports beyond this line

--------------------------------------
-- AXI4 signals declared here, cut from this extract to save space
--------------------------------------
   );
END MotionController_v1_0_S00_AXI;
ARCHITECTURE arch_imp OF MotionController_v1_0_S00_AXI IS
    COMPONENT PWM_Driver IS
        GENERIC (
            sys_clk : INTEGER := 50_000_000; --system clock frequency in Hz
            pwm_freq : INTEGER := 100_000; --PWM switching frequency in Hz
            bits_resolution : INTEGER := 16 --bits of resolution setting the duty cycle
        );
        PORT (
            duty : IN STD_LOGIC_VECTOR(bits_resolution - 1 DOWNTO 0);
            clk : IN STD_LOGIC;
            pwm_out : OUT STD_LOGIC;
            enable : IN STD_LOGIC --asynchronous reset
        );
    END COMPONENT PWM_Driver;


--------------------------------------
-- AXI4 communication processes here, cut from this extract to save space
--------------------------------------
   -- Add user logic here
    -- slv_reg0 = control register
    -- slv_reg1 = servo register
    -- slv_reg2 = motor speed register
    software_enable <= slv_reg0(0); -- least significant bit of 'control' register
    -- both the software enable and the physical switch must be on to enable the module. Safety feature
    module_enable <= software_enable AND enable;
    --bits 1 and 2 of the 'control' register set the motor directions
    motor_right_dir_out <= slv_reg0(1);
    motor_left_dir_out <= slv_reg0(2);
    servo_input <= slv_reg1(servo_bits_resolution - 1 DOWNTO 0);
    -- force the servo output to be in the acceptable range for a servo motor
    -- pulse time between 500us and 2500us
    servo_position <= conv_std_logic_vector(minServoDuty, servo_bits_resolution) WHEN conv_integer(servo_input) < minServoDuty ELSE
                      conv_std_logic_vector(maxServoDuty, servo_bits_resolution) WHEN conv_integer(servo_input) > maxServoDuty ELSE
                      servo_input;
    motor_right_speed <= slv_reg2(motor_bits_resolution - 1 DOWNTO 0);
    motor_left_speed <= slv_reg2((2 * motor_bits_resolution) - 1 DOWNTO motor_bits_resolution);
```

```
    --define the three PWM_Drivers used in the design
    servo_driver : PWM_Driver
        GENERIC MAP(
            sys_clk => sys_clk_frequency, pwm_freq => 50,
                        bits_resolution => servo_bits_resolution
        )
            PORT MAP(
                    duty => servo_position,
                    clk => S_AXI_ACLK,
                    pwm_out => servo_pwm_out,
                    enable => module_enable
        );
        motor_left_driver : PWM_Driver
            GENERIC MAP(
                sys_clk => sys_clk_frequency,
                pwm_freq => motor_pwm_frequency,
                bits_resolution => motor_bits_resolution
            )
                PORT MAP(
                    duty => motor_left_speed,
                    clk => S_AXI_ACLK,
                    pwm_out => motor_left_pwm_out,
                    enable => module_enable
        );
        motor_right_driver : PWM_Driver
            GENERIC MAP(
                sys_clk => sys_clk_frequency,
                pwm_freq => motor_pwm_frequency,
                bits_resolution => motor_bits_resolution
            )
                PORT MAP(
                    duty => motor_right_speed,
                    clk => S_AXI_ACLK,
                    pwm_out => motor_right_pwm_out,
                    enable => module_enable
        );
        -- User logic ends
END arch_imp;
```

Appendix E: AXI_BayerToRGB.vhd

```
-------------------------------------------------------------------------------
--
-- File: AXI_BayerToRGB.vhd
-- Author: Ioan Catuna
-- Original Project: AXI Bayer to RGB Image Conversion
-- Date: 15 December 2017
--
-------------------------------------------------------------------------------
-- MIT License
-- Copyright (c) 2017 Digilent
-------------------------------------------------------------------------------
-- Component specifications:
-- Input sample format: Bayer (single color)
-- Input sample size: 10 bits
-- Input sample count: 4 at a time
-- Output sample format: RGB
-- Output sample size: 32 bits (10 bits per color) + 2 unused bits
-- Output sample count: 1 at a time
-- Maximum resolution: 2048 x  pixels;
-- Input-to-output latency: 4 StreamClk cycles.
--
-------------------------------------------------------------------------------
-------------------------------------------------------------------------------
-- Revision: 1
-- Project: Zybo Autonomous Car
-- Author: Catalin Bitire
-- Changes: Modified the output format to be inline with the Digilent Pcam 5C
-- output format (which is 3 color channels, 8 bit color depth), new
-- format is BGR 24 bits (8 bits per color). Using the original
-- design for a 3x8bit stream would mean that 8 bits are always unused,
-- accounting for a 25% useless data transfers overhead.
--
-- Depends on your choice of camera/input video data parameters. This setup works
-- on our project's implementation.
-------------------------------------------------------------------------------

LIBRARY IEEE;
USE IEEE.STD_LOGIC_1164.ALL;
USE IEEE.NUMERIC_STD.ALL;
LIBRARY UNISIM;
USE UNISIM.VComponents.ALL;
ENTITY AXI_BayerToRGB IS
    GENERIC (
        kAXI_InputDataWidth : INTEGER := 40;
        kBayerWidth : INTEGER := 10;
        -- Revision 1
        -- changed OutpudDataWidth from 32 (2 unused, 10 bit color depth, 3 channels)
        -- to 24 (3 channels of 8 bit color depth)
        kAXI_OutputDataWidth : INTEGER := 24;
        kMaxSamplesPerClock : INTEGER := 4
    );
    PORT (
        StreamClk : IN STD_LOGIC;
        sStreamReset_n : IN STD_LOGIC;
        s_axis_video_tready : OUT STD_LOGIC;
        s_axis_video_tdata : IN STD_LOGIC_VECTOR(kAXI_InputDataWidth - 1 DOWNTO 0);
        s_axis_video_tvalid : IN STD_LOGIC;
```

```
        s_axis_video_tuser : IN STD_LOGIC;
        s_axis_video_tlast : IN STD_LOGIC;
        m_axis_video_tready : IN STD_LOGIC;
        m_axis_video_tdata : OUT STD_LOGIC_VECTOR(kAXI_OutputDataWidth - 1 DOWNTO 0);
        m_axis_video_tvalid : OUT STD_LOGIC;
        m_axis_video_tuser : OUT STD_LOGIC;
        m_axis_video_tlast : OUT STD_LOGIC
    );
END AXI_BayerToRGB;
ARCHITECTURE rtl OF AXI_BayerToRGB IS

    --------------------------------------
    -- AXI stream communication processes here, cut from this extract to save space
    -- See source file on the github repository for further reference
    --------------------------------------

    -- Revision 1
    -- Changed below line to modify the output format size and color depth
    m_axis_video_tdata <= std_logic_vector(sAXIMasterRed(9 DOWNTO 2)) &
        std_logic_vector(sAXIMasterGreen(10 DOWNTO 3)) &
        std_logic_vector(sAXIMasterBlue(9 DOWNTO 2));
END rtl;
```

Appendix F: motion.c

```
/**
 * This source file can be used as a good example of the API
 * in use for /dev/motors, /dev/servo and /dev/sonar.
 *
 * Compile and run the application with no parameters to get
 * the usage information.
 *
 * Examples:
 *     ./motion <mode> ...
 *
 *     Using the motors and sonar:
 *     ./motion 0 <direction> <speed> <run time> <stop distance>
 *
 *     Using the servo:
 *     (steering input is between 220 and 380 unless changed in the future)
 *     ./motion 1 <steering input>
 */
#include <stdio.h>
#include <stdint.h>
#include <stdlib.h>
#include <poll.h>
#include <fcntl.h>
#include <errno.h>
#include <unistd.h>
#include <sys/mman.h>
#include <linux/ioctl.h>
#include <motiondriver/linux/motion_ioctl.h>

#define SERVO_LEFT 220
#define SERVO_RIGHT 380
#define SERVO_CENTER (SERVO_LEFT + (SERVO_RIGHT-SERVO_LEFT)/2)

#define CLK_FREQ 50000000.0f // FCLK0 frequency not found in xparameters.h
const double clk_to_cm=(((1000000.0f/CLK_FREQ)*2.54f)/147.0f);

int main(int argc, char *argv[]) {
    if (argc<2) {
        printf("Usage: motion option: 0=motors, 1=steering\n");
        return 0;
    }
    int motors, servo, sonar;
    motors = open("/dev/motors", O_WRONLY);
    if (motors < 1) {
        fprintf(stderr,"Can't open motors.\n");
        return -1;
    }
    servo = open("/dev/servo", O_WRONLY);
    if (servo < 1) {
        fprintf(stderr,"Can't open servo.\n");
        return -1;
    }
    sonar = open("/dev/sonar", O_RDONLY);
    if(!sonar) {
        printf("Can't open sonar\n");
        return -1;
    }

```

```
    unsigned int clk_edges;
    double dist;
    unsigned int enable = 1;
    unsigned int leftDir = 1;
    unsigned int rightDir = 1;
    unsigned short leftSpeed = 0;
    unsigned short rightSpeed = 0;
    unsigned stopDist = 0;
    unsigned option = strtoul(argv[1],NULL,0); //0 for motors, 1 for steering

    ioctl(motors, MOTION_IOCTSETENABLE, enable);
    if (option==0) { // motion 0 dir speed time
        if (argc<5) {
            printf("Usage:motion 0 dir speed time [stopdist]\n");
            printf("Speed is u16 [0, 65535]. Time is in us. Stopdist is in cm.\n");
        } else {
            leftDir = strtoul(argv[2],NULL,0);
            rightDir = leftDir;
            leftSpeed = strtoul(argv[3],NULL,0);
            rightSpeed = leftSpeed;
            unsigned int sleeptime = strtoul(argv[4],NULL,0);
            if(argv[5]!=NULL)
                stopDist = strtoul(argv[5], NULL, 0);
            unsigned int speed = (leftSpeed<<16)+rightSpeed;
            write(motors, &speed, 4);
            ioctl(motors, MOTION_IOCTSETDIR, ((leftDir&1)<<1)+(rightDir&1));

            for (int i=0; i<sleeptime/1000; i++) {
                read(sonar, &clk_edges, 4);
                dist = clk_edges * clk_to_cm;
                if (dist<stopDist)
                    break;
                usleep(1000);
            }
            speed = 0;
            write(motors, &speed, 4);
        }
    } else if(option==1) { // motion 1 position
        if (argc<3) {
            printf("Usage:motion 1 position\n");
            printf("Servo limits are [%d, %d]\n", SERVO_LEFT, SERVO_RIGHT);
        } else {
                unsigned newServo = strtoul(argv[2],NULL,0);
                if (newServo<SERVO_LEFT)
                    newServo = SERVO_LEFT+5;
                else if(newServo > SERVO_RIGHT)
                    newServo = SERVO_RIGHT-5;
                write(servo, &newServo, 2);
                usleep(100000);
                //*((unsigned *)(ptr + CONTROL_REG_OFFSET)) = 0;
                unsigned int speed = 0;
                write(motors, &speed, 4);
            }
        }
        close(motors);
        close(servo);
        close(sonar);
        return 0;
}
```

Appendix G: videotest.c

```c
/**
 * This application can be used as a reference for the /dev/video and /dev/videoHLS
 * device nodes usage and API.
 * It uses all the available function calls supported (open, read, ioctl, close).
 * The program reads the image form the given video pipeline and saves it into a
 * .ppm image format (uncompressed image).
 *
 * Usage:
 *        ./videotest <device name> <output_image_name>
 * Where device name can be /dev/video or /dev/videoHLS
 */
#include <stdio.h>
#include <stdint.h>
#include <stdlib.h>
#include <poll.h>
#include <fcntl.h>
#include <errno.h>
#include <unistd.h>
#include <sys/mman.h>
#include <sys/ioctl.h>

#include <vdmadriver/linux/charvideo_ioctl.h>

int main(int argc, char *argv[]) {
    if (argc < 3) {
        printf("Usage ./%s /dev/videoX out_name\n", argv[0]);
        return -1;
    }

    int fd;
    fd = open(argv[1], O_RDONLY);
    if (fd < 0) {
        printf("Can't open %s\n", argv[1]);
        return -1;
    }

    //Print the VDMA's status to kernel log
    ioctl(fd, CHARVIDEO_IOCSTATUS);

    //Get the image sizes from the video driver
    int h, w, l;
    h = ioctl(fd, CHARVIDEO_IOCQHEIGHT);
    w = ioctl(fd, CHARVIDEO_IOCQWIDTH);
    l = ioctl(fd, CHARVIDEO_IOCQPIXELLEN);

    unsigned char buf[h * w * l];
    read(fd, buf, w * h * l);
    close(fd);

    char filename[100];
    sprintf(filename, "/home/root/%s.ppm", argv[2]);
    FILE *outimg = fopen(filename, "wt");

    //if the pixel length is only 1 byte, then the image is grayscale (ppm format 5)
    if (l == 1) {
        fprintf(outimg, "P5\n%d %d\n%d\n", w, h, 255);
    }
```

```
    else {
        fprintf(outimg, "P6\n%d %d\n%d\n", w, h, 255);
    }
    printf("Opened %s\n", filename);
    //The images are stored in the VDMAs in the BGR format so it must be
    //changed to RGB for human understandable images
    if (l != 1) { //BGR to RGB
        for (int i = 0; i < w * h * l; i += 3) {
            uint8_t aux = buf[i + 2];
            buf[i + 2] = buf[i];
            buf[i] = aux;
        }
    }
    fwrite(buf, 1, w * h * l, outimg);
    fclose(outimg);
    return 0;
}
```

Appendix H: sonarTest.c

```
/**
 * This application can be used as a reference for the /dev/sonar device
 * API.
 *
 * Usage is:
 * ./sonarTest
 *
 * The application prints the value read from the sonar through the kernel
 * device driver once every 100ms. The actual distance is computed
 * based on the CLK_FREQ define that represents the VHDL module's clock
 * frequency (in this case 50MHz)
 */
#include <stdio.h>
#include <stdint.h>
#include <stdlib.h>
#include <poll.h>
#include <fcntl.h>
#include <errno.h>
#include <unistd.h>
#include <sys/mman.h>

#define CLK_FREQ 50000000.0f // FCLK0 frequency not found in xparameters.h

//147us is 2.54 cm
//CLK_usPeriod is (CLK_usPERIOD*2.54)/147

const double clk_to_cm = (((1000000.0f / CLK_FREQ) * 2.54f) / 147.0f);

int main(void) {
    int fd;

    fd = open("/dev/sonar", O_RDONLY);
    if (!fd) {
        printf("Can't open sonar\n");
        return -1;
    }
    printf("const %g\n", clk_to_cm);

    unsigned int clk_edges;
    double dist;

    while (1) {

        read(fd, &clk_edges, 4);

        dist = clk_edges * clk_to_cm;

        printf("%x, dist (cm) = %g\n", clk_edges, dist);
        usleep(100000);
    }

    return 0;
}
```

Appendix I: rgbledtest.c

```c
/**
 * This application can be used as a reference to the /dev/rgbled
 * device node API. Usage is:
 *
 * ./rgbledtest 0xXXBBGGRR
 * where XXBBGGRR is a 32bit value represented in HEX, like 00FF33CD
 *
 * ./rgbledtest B G R
 * where B, G and R are 8 bit values (between 0-255).
 *
 * B - blue LED duty cycle (0-0%, 255-100%)
 * G - green LED duty cycle (0-0%, 255-100%)
 * R - red LED duty cycle (0-0%, 255-100%)
 */
#include <stdio.h>
#include <unistd.h>
#include <stdlib.h>
#include <fcntl.h>

int main(int argc, char *argv[]) {
    if (argc < 2)
        printf("Usage: ./%s 0xXXBBGGRR\t or\t ./%s B G R\n", argv[0], argv[0]);

    unsigned int output = 0;

    if (argc == 2) {
        output = strtoul(argv[1], NULL, 16);
    } else if (argc == 4) {
        unsigned char r = 0, g = 0, b = 0;
        b = atoi(argv[1]);
        g = atoi(argv[2]);
        r = atoi(argv[3]);
        output = (b << 16) | (g << 8) | r;
    } else {
        printf("Invalid input. See usage.\n");
        return -1;
    }

    int fd = open("/dev/rgbled", O_WRONLY);
    if (fd < 0) {
        printf("Can't open /dev/rgbled\n");
        return -1;
    }

    write(fd, &output, 4);
    close(fd);
}
```

Appendix J: motion_ioctl.h

```
#ifndef MOTION_IOCTL_H_
#define MOTION_IOCTL_H_

#include <linux/ioctl.h>

#define MOTION_IOC_MAGIC  '9'

#define MOTION_IOCTSETENABLE    _IO(MOTION_IOC_MAGIC, 0)
#define MOTION_IOCTSETDIR     _IO(MOTION_IOC_MAGIC, 1)
#define MOTION_IOCQSERVOLEFT _IOR(MOTION_IOC_MAGIC,  2, int)
#define MOTION_IOCQSERVORIGHT _IOR(MOTION_IOC_MAGIC,  3, int)

#define MOTION_IOC_MAXNR 3

#endif
```

Appendix K: charvideo_ioctl.h

```
#ifndef CCHARVIDEO_IOCTL_H_
#define CCHARVIDEO_IOCTL_H_

#include <linux/ioctl.h>

#define CHARVIDEO_IOC_MAGIC  '8'

#define CHARVIDEO_IOCHALT    _IO(CHARVIDEO_IOC_MAGIC, 0)
#define CHARVIDEO_IOCSTART    _IO(CHARVIDEO_IOC_MAGIC, 1)
#define CHARVIDEO_IOCSTATUS    _IO(CHARVIDEO_IOC_MAGIC, 2)

#define CHARVIDEO_IOCQHEIGHT _IOR(CHARVIDEO_IOC_MAGIC,  3, int)
#define CHARVIDEO_IOCQWIDTH _IOR(CHARVIDEO_IOC_MAGIC,  4, int)
#define CHARVIDEO_IOCQPIXELLEN _IOR(CHARVIDEO_IOC_MAGIC,  5, int)
#define CHARVIDEO_IOCQBUFSIZE _IOR(CHARVIDEO_IOC_MAGIC,  6, int)

#define CHARVIDEO_IOC_MAXNR 6

#endif
```

Appendix L: motors.h

This file is an extract from the kernel module driver for the motion controller, presenting the write and ioctl system call handlers for the motors component.

```
/**
 * Handles writes to the file descriptors managed by this device node.
 * It verifies the input and then writes the value to the physical device's
 * register.
 */
ssize_t motors_write(struct file *filp, const char __user *buf, size_t count,
                     loff_t *f_pos)
{
    struct motors_dev *dev = filp->private_data;
    void __iomem *base_addr = dev->lp->base_addr;
    uint32_t value = 0;

    if (count>sizeof(uint32_t)) { // can only write 4 bytes
        printk(KERN_WARNING "Exceded count on write\n");
        return -1;
    }

    if (copy_from_user(&value, buf, count)) {
        return -EFAULT;
    }

    reg_write(base_addr, MOTOR_REG_OFFSET, value);

    *f_pos += count;
    return count;
}

/**
 * Handles ioctl calls on the files managed by this device node.
 * Can be used to set the direction of the motors and the software
 * enable bit.
 */
long motors_ioctl(struct file *fp, unsigned int cmd, unsigned long arg) {
    struct motors_dev *dev = fp->private_data;
    void __iomem *base_addr = dev->lp->base_addr;
    int err = 0, tmp;
    int retval = 0;

    uint8_t state = reg_read(base_addr, CONTROL_REG_OFFSET) & 1; //get the enable bit
    uint8_t dir = reg_read(base_addr, CONTROL_REG_OFFSET) & 0b110; // get the direction bits

    if (_IOC_TYPE(cmd) != MOTION_IOC_MAGIC)
        return -ENOTTY;
    if (_IOC_NR(cmd) > MOTION_IOC_MAXNR)
        return -ENOTTY;

    if (_IOC_DIR(cmd) & _IOC_READ)
        err = !access_ok(VERIFY_WRITE, (void __user *)arg, _IOC_SIZE(cmd));
    else if (_IOC_DIR(cmd) & _IOC_WRITE)
        err = !access_ok(VERIFY_READ, (void __user *)arg, _IOC_SIZE(cmd));
    if (err)
        return -EFAULT;
```

```

    switch (cmd) {

    case MOTION_IOCTSETENABLE: // 1 bit value for the enable
        if (! capable (CAP_SYS_ADMIN))
            return -EPERM;
        if (arg > 1)
            return -ENOTTY;
        //printk("Set enable arg:%d, dir:%d, result: %d\n", arg, dir, arg+dir);
        reg_write(base_addr, CONTROL_REG_OFFSET, (uint32_t) (arg + dir));
        break;


    case MOTION_IOCTSETDIR:  // 2 bit value for the 2 motors (maxval 3);
        if (! capable (CAP_SYS_ADMIN))
            return -EPERM;
        if (arg > 3)
            return -ENOTTY;
        //printk("Set dir arg:%d, state:%d, result: %d\n", arg, dir, ((arg<<1) + state));
        reg_write(base_addr, CONTROL_REG_OFFSET, (uint32_t) ((arg<<1) + state));
        break;

    default:
        return -ENOTTY;
    }
    return retval;
}
```

Appendix M: servo.h

This file is an extract from the kernel module driver for the motion controller, presenting the write and ioctl system call handlers for the servo component.

```c
/**
 * Handles writes to the file descriptors managed by this device node.
 * It verifies the input and then writes the value to the physical device's
 * register.
 */
ssize_t servo_write(struct file *filp, const char __user *buf, size_t count,
                    loff_t *f_pos)
{
    struct servo_dev *dev = filp->private_data;
    void __iomem *base_addr = dev->lp->base_addr;

    uint16_t value = 0;

    if (count>sizeof(uint16_t)) // can only write 2 bytes
        return -1;

    if (copy_from_user(&value, buf, count)) {
        return -EFAULT;
    }

    //printk("Got value %d\n", value);

    //limit the value to 12bit
    if (value>4096)
        return -1;

    //limit the value to the current steering config
    //TODO: add ioctl to get the defaults
    if(value<SERVO_LEFT || value>SERVO_RIGHT) {
        printk(KERN_WARNING "Requested servo value %d is not in the accepted range of [%d, %d]\n", value, SERVO_LEFT, SERVO_RIGHT);
        return -1;
    }

    reg_write(base_addr, SERVO_REG_OFFSET, value);

    *f_pos += count;
    return count;
}

/**
 * Handles ioctl calls on the files managed by this device node.
 * Can be used to get the maximum and minimum servo values accepted
 * by the physical driver (defined at the start of this source file)
 */
long servo_ioctl(struct file *fp, unsigned int cmd, unsigned long arg) {
    struct servo_dev *dev = fp->private_data;
    void __iomem *base_addr = dev->lp->base_addr;
    int err = 0, tmp;
    int retval = 0;


    if (_IOC_TYPE(cmd) != MOTION_IOC_MAGIC)
```

```
        return -ENOTTY;
    if (_IOC_NR(cmd) > MOTION_IOC_MAXNR)
        return -ENOTTY;

    if (_IOC_DIR(cmd) & _IOC_READ)
        err = !access_ok(VERIFY_WRITE, (void __user *)arg, _IOC_SIZE(cmd));
    else if (_IOC_DIR(cmd) & _IOC_WRITE)
        err = !access_ok(VERIFY_READ, (void __user *)arg, _IOC_SIZE(cmd));
    if (err)
        return -EFAULT;

    switch (cmd) {

    case MOTION_IOCQSERVOLEFT:
        return SERVO_LEFT;

    case MOTION_IOCQSERVORIGHT:
        return SERVO_RIGHT;

    default:
        return -ENOTTY;
    }
    return retval;
}
```

Appendix N: prop.cfg

The configuration file used in our demos. This can also be modified to acomodate your settings, read the detailed description of the configuration component to understand each parameter.

```
resize_factor=0.5
y_1=560
y_2=440
y_3=300
line_dist_1_out=120
line_dist_1_in=280
line_dist_1=120
line_dist_2=60
line_dist_3=30
servo_fine=15
min_speed=15000
max_speed=40000
min_adj_servo=1.0
max_adj_servo=0.5
servo_map_a=1
servo_map_b=1
servo_map_c=1
servo_map_d=1
speed_up_max=1.25
speed_up_min=0.75
speed_up_rate=0.03
sign_min=55
sign_max=75
sonar_dist=-1
fps=-1
sign_on=1
rfid_on=1
acl_on=1
draw=1
### LAST 6 LINES ARE USED TO DETERMINE THE MIDDLE OF THE LANE LINES
### THESE CAN BE CHANGED AUTOMATICALLY USING THE CALIBRATION MODE
### THE "$" MARKS THE BEGINNING OF THE SAID CALIBRATION CONSTRAINTS
$
left_mean_1=150
left_mean_2=285
left_mean_3=474
right_mean_1=1088
right_mean_2=943
right_mean_3=768
```