Technische Universiteit
Eindhoven
University of Technology

5LIA0: Embedded Visual Control

Department of Electrical Engineering

T.J.H. Vogels, 0871231
t.j.h.vogels@student.tue.nl

H.A.H. van Schaaijk, 0873871
h.a.h.v.schaaijk@student.tue.nl

Michael Kruger, 0846882
m.g.kruger@student.tue.nl

Solar car

July 2016

Table of contents

I. INTRODUCTION

The goal of this project consists of two parts, the first being mandatory and equal to every group while the second part is a free choice of each group. The first part of the project is to design software for a small solar car so that it is able to autonomously drive a track consisting solely of road markings and traffic signs using a camera mounted to a Raspberry Pi. The objectives of the project are not only to complete a track during a demo, but to do it as energy efficiently as possible.

As a secondary assignment this group chose to add a turret with a laser that, while driving, automatically aims at the signs and shoots them when close enough.

The report has the following structure; first an overview of the system architecture is shown, addressing the distribution of tasks over different parts of the system. This is followed by the explanation of the steps taken to extract information about the lines and traffic signs. Information about these lines and signs alone is not enough, and therefore in the next chapter an explanation is given about how the system acts upon this information. This acting upon should be handled by actuators and the Arduino controlling them. The Raspberry Pi and the Arduino therefore need to communicate and the actuators be controlled, which is addressed in the consecutive chapters. The following chapters will handle the secondary assignment, the mechanics and the outcome of the demo. Finally a conclusion and future work is given.

II. *SYSTEM ARCHITECTURE*

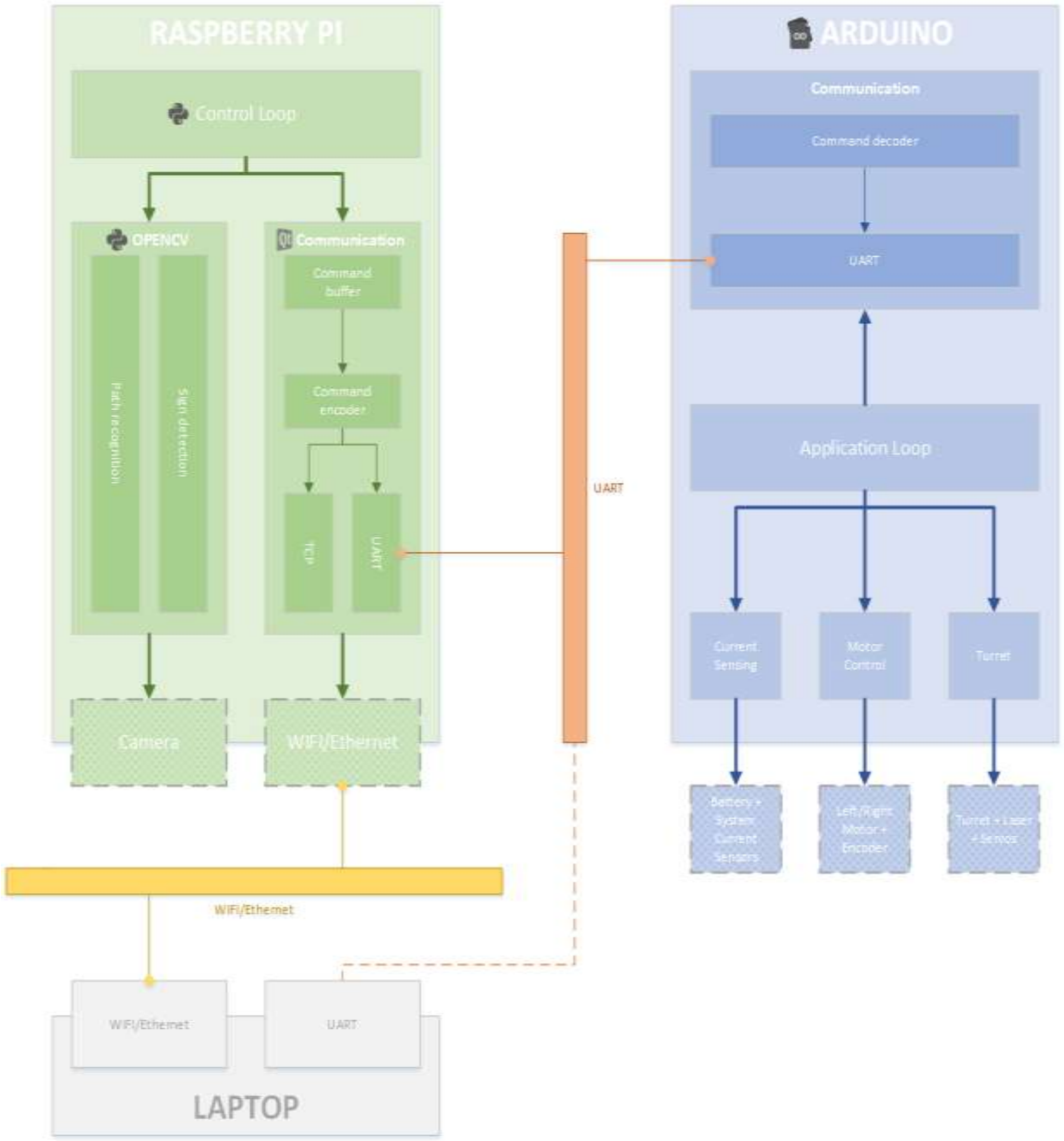


Figure 1: System Architecture

The system's architecture shows a distribution of work between the Raspberry and the Arduino in which with the Raspberry serves as the master. When the system is operating autonomous, the laptop is used for logging and seeing state-transitions.

2.1. *Raspberry Pi*

The main functionality of the Raspberry consists of three parts namely: Communication, OpenCV and Control. Each of these takes care of a specific section for the operation of the system.

- Communication: The communications paths available on the Raspberry are a UART connection to the Arduino and the WiFi/Ethernet connection with the laptop. The WiFi/Ethernet connection is mainly used to shell into the Raspberry or switch over to manual control. The UART communication channel is used to control the Arduino, which in turn performs motor control. This is elaborated upon in the Communication chapter.
- OpenCV is an open-source library dedicated to computer vision. Input images are manipulated by this library to support information to the Controlloop; more on this in the Vision chapter.
- Control: The control-loop serves as the main control of the system. It takes information from the OpenCV block and acts on this information by instructing the communication block on what commands to send to the motor control. This is described in the Control chapter

2.2. *Arduino*

The main responsibility of the Arduino is to control the parts of the system that require a quick response time or need direct hardware control. As such the Arduino is responsible for the following:

- Communication: The Arduino is responsible for reading and decoding the commands that are sent from the Raspberry Pi over UART. The communication is a master-slave configuration where the Arduino is the slave. The data in these commands are used to set the set points for the motors, retrieving data and to perform actions. The Arduino responds with the requested data or an acknowledgement.
- Motor Control: The motor control reads the encoder pulses and controls the motor power based on the encoder values and the set point.
- Turret Control: The turret control is responsible for controlling the servos that aim the turret, the laser and the firing of the “missiles”
- Current Sensing: The current sensing reads and maintains a rolling average of the battery and system current.

2.3. *Laptop*

In order to control and monitor the behaviour of the solar car, a PC interface has been made. This interface can emulate and sniff the commands from the control loop over UART or TCP. It consists of the following:

- Communication: The communication in the laptop can either encode and send commands to the Arduino directly over UART or send commands over TCP to the RPC running on the Raspberry PI.
- Debugging: The console output and video screens showing the output of the image processing is available via the debugging. This also logs output to a file.
- UI: The user interface presents an easy and visual way to control the car with sliders and buttons.

III. VISION

The possibility to act upon visual cues is provided by combining the Raspberry Pi camera and the OpenCV library. This computer vision library supports many computational efficient algorithms used throughout the field of computer vision. For the implementation of this project, these algorithms are mainly used for path –and sign detection. The goal of the vision-aspects of the system is to provide the control block with information regarding the signs and paths and their location.

3.1. THRESHOLDING

Detection of both path –and sign detection will rely, to an extent, on colour-thresholding. This is a problem in itself, as any change in lighting influences these thresholds resulting in a sub-optimal detection. It is therefore critical that thresholding is done properly before autonomous driving is engaged.

Changes in lighting however are unavoidable, especially with the auto-white balance (awb) function of the camera enabled; continually changing the way the camera perceives colour. It was therefore decided that the awb is disabled, resulting in a more equal colour perception. This makes it mandatory to recalibrate the system for each new scenario in which the lighting is slightly different for optimal result.

3.2. PATH DETECTION

For the path detection the approach shown below is used.

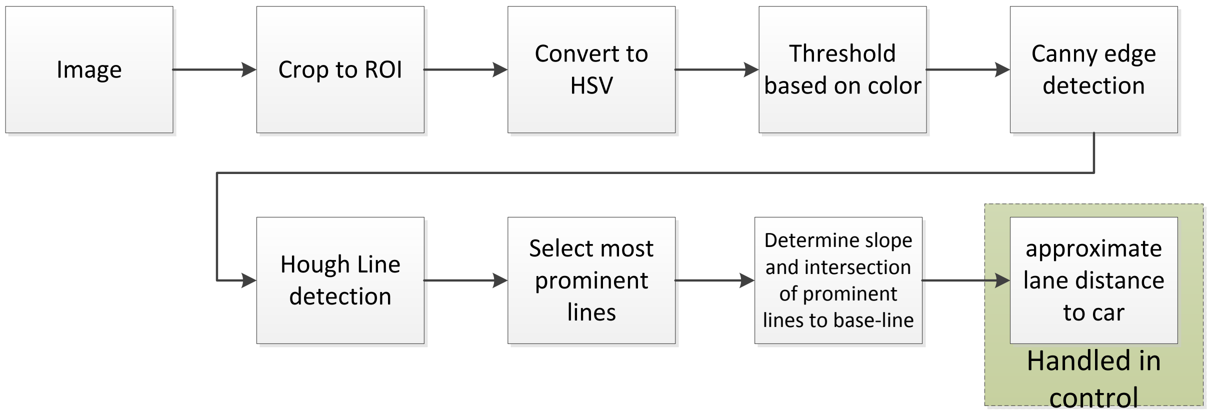


Image
Crop to ROI
Convert to HSV
Threshold based on color
Canny edge detection
Hough Line detection
Select most prominent lines
Determine slope and intersection of prominent lines to base-line
approximate lane distance to car
Handled in control

Figure 2: Path detection pipeline

3.2.1. Image manipulation and detection

The image fetched is a 480x640 image. The choice for this resolution is that it is believed to be a high enough resolution to make detection of the path and signs possible, while still be low enough to achieve a proper framerate. The region of interest (ROI) for the path detection is approximately the bottom half of the original image. This area is therefore cropped while the remainder of the image is used for the sign detection.

Following this, the image is converted to a different colour-space. This HSV colour-space is more suitable for colour-thresholding as it divides colours in Hue, Saturation and Value rather than Red Green Blue (RGB).

After this colour thresholding is done which provides a binary image in which the lanes are depicted white, while the rest of the image is depicted black. This makes further filtering possible and the result is shown in Figure 3.

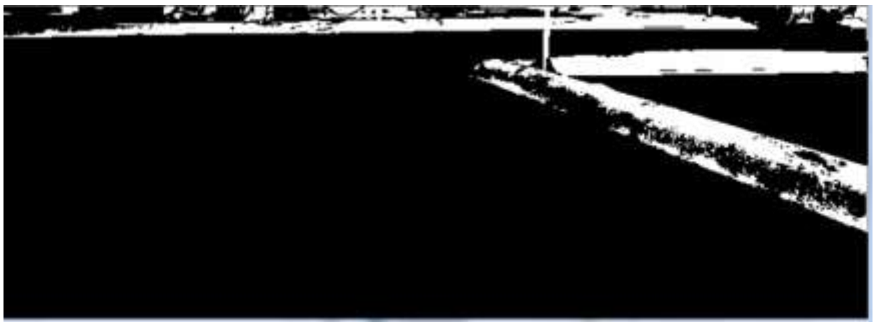


Figure 3: Thresholded lane image (Binary)

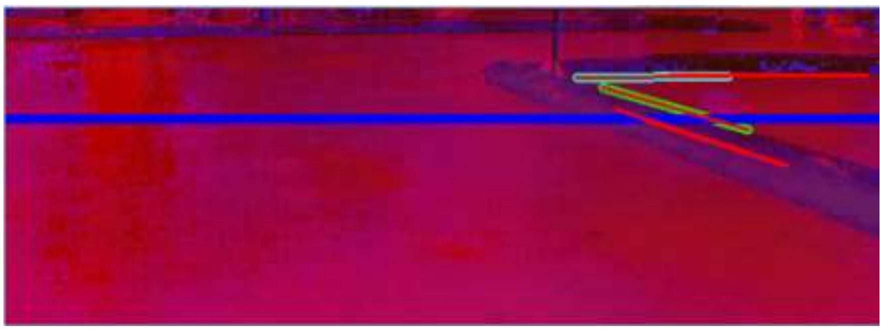


Figure 4: Line depiction on HSV image

A Canny-filter[1] is then applied to reduce the white to contours.

After this a HoughLine-filter[2] is applied to find lines within set parameters within the image. A choice was made here to set a relative low threshold for finding lines, resulting in a high amount of found lines. This was done to ensure that a line is sufficiently well detected. This is then followed by a set of heuristics which filter the found lines to find the lines specifically interesting for the system. These prominent lines depict either a left, right or horizontal lane. This is shown in **Error! Not a valid bookmark self-reference.**.

3.2.2. Heuristics

For this implementation the used heuristics are relatively simple: the first lines from the middle of the image which adhere a certain angle-constraint, are considered prominent lines and are coloured differently (in figure 5: the green(right) and white(horizontal) line).

Another heuristic about grouping lines together and thus determining the 'strength' of a line was considered, but ultimately dropped as it did not have the expected improvement when taking shadows into account. The used heuristics perform adequate without need of extra computing power for i.e. combining lines.

Finally the intersection of the prominent lines with the blue line (Figure 4) is calculated by taking the slope of the lines into account. This (virtual) intersection point is a measure for the distance of the lanes to the car and is further used by the control loop. Specifics about the control with regard to the location of the detected lines are described in the Control chapter.

3.2.3. Performance

The Hough-line detection is the heaviest operation necessary for the path detection. Effort was therefore made to find an alternative for this operation to reduce computation power. An alternative

[1] http://docs.opencv.org/2.4/doc/tutorials/imgproc/imgtrans/canny_detector/canny_detector.html
[2] http://docs.opencv.org/2.4/doc/tutorials/imgproc/imgtrans/hough_lines/hough_lines.html?highlight=hough

which finds contours[3] of large objects to either side of the image rather than a multitude of specific lines was considered. Due to timing constraints this was only investigated to some extent as the priority of the project was with delivering something working as intended before energy costs were taken into account. It is possible that this contour-approach may yield acceptable results while reducing energy consumption and increase performance and thus needs further investigation.

3.3. SIGN DETECTION

For the sign detection the approach shown below is used.

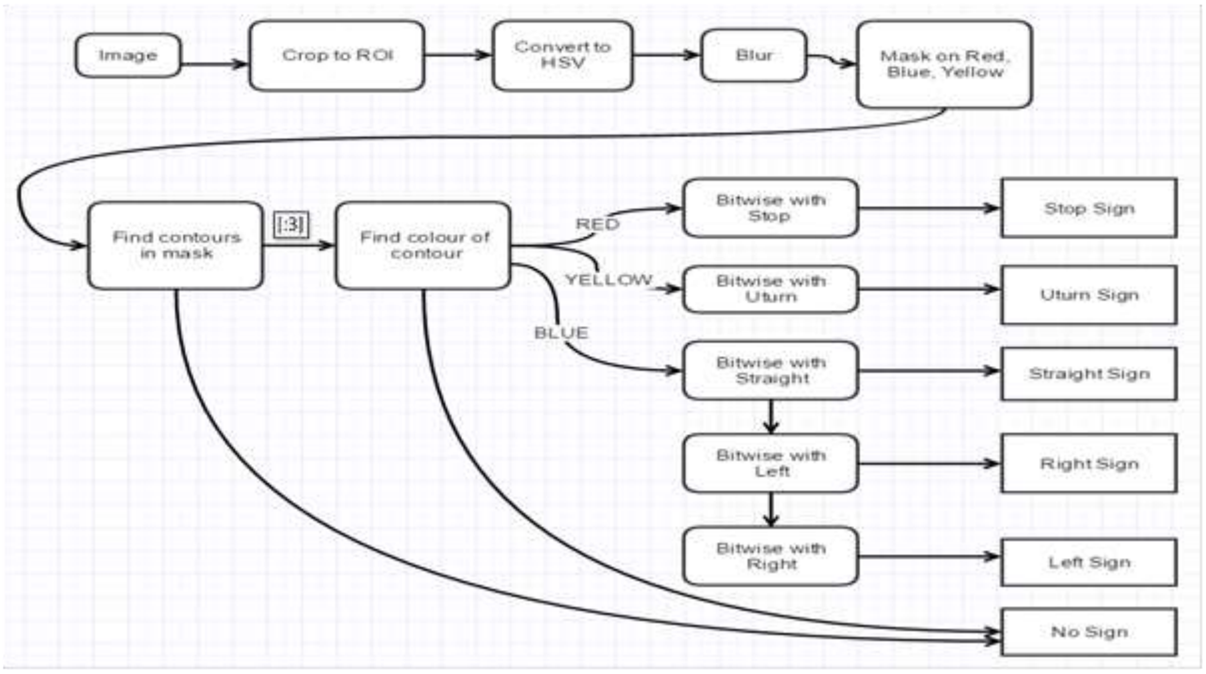


Figure 5: Sign detection pipeline

3.3.1. Segmentation

For the image fetching, cropping and conversion to the HSV colour-space a similar reasoning as with the path-detection is used. This cropped HSV image will be used to extract three different binary masks by use of colour-thresholding. Each mask represents the colour of a specific sign; Blue, Red and Yellow. These masks have certain degrees of dilations and erosions applied, specific for that mask. This is done to have optimal recognition before merging into a single binary mask displaying all three colours as indicated in Figure 6.

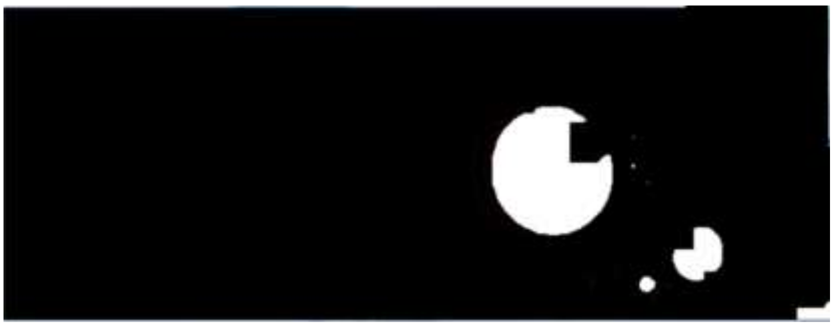


Figure 6: Binary mask

[3]http://docs.opencv.org/2.4/modules/imgproc/doc/structural_analysis_and_shape_descriptors.html?highlight=contour#cv2.findContours

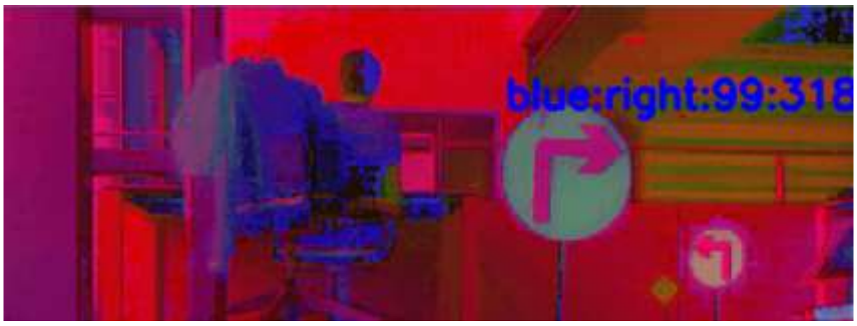


Figure 7: HSV image

On this binary mask, the OpenCV `FindContours`[4] function is applied which returns the contours of the shapes found in the mask. As the biggest contours are the most likely to be signs, only the biggest three contours will be further checked for being signs.

3.3.2. Feature extraction

Each contour is checked for its colour by comparing it to the original three masks. As the sign is most likely to be blue, this is checked first before checking the other masks. As information about the colour now is available, it is nonsensical to check the contours against all possible signs. For the five signs provided in this assignment this is not much of an issue, however, should the library of available signs increase, this pre-check increases in usefulness.

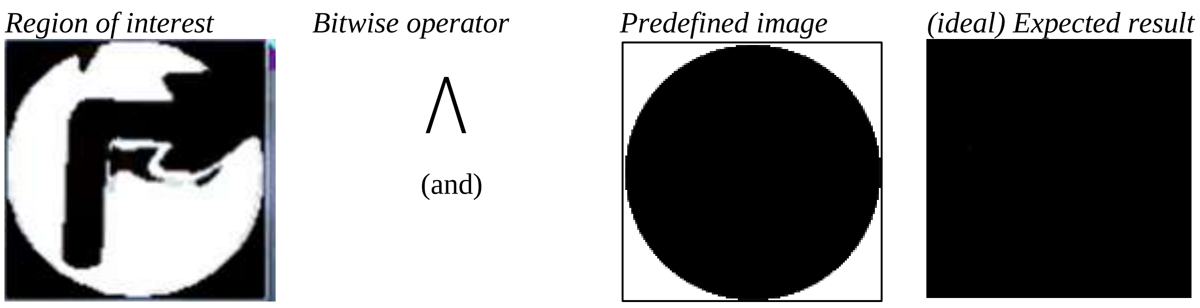
The contours are extracted from the original *undilated* mask by defining a region of interest around the contour. This is made possible by using the OpenCV `BoundingRect`[5] function; note that these regions of interest are binary images of variable size, depending on the size of the sign in the original image.

3.3.3. Detection

For the detection of the specific signs in the region of interests, the regions of interests are bitwise compared to predefined images in the memory. The result of this comparison indicates which type of sign is displayed in the region of interest. This is done as follows:

Firstly, the region of interest is resized to be 100x100 pixels, this size is chosen as this is approximately the size of a sign when it is to be reacted upon. Additionally it is large enough to still be able to detect the different signs, while small enough to have acceptable computational latency

Based on the colour and extracted region of interest a check is done whether the ROI fits the projected shape (circle, hexagonal or diamond). An example is shown



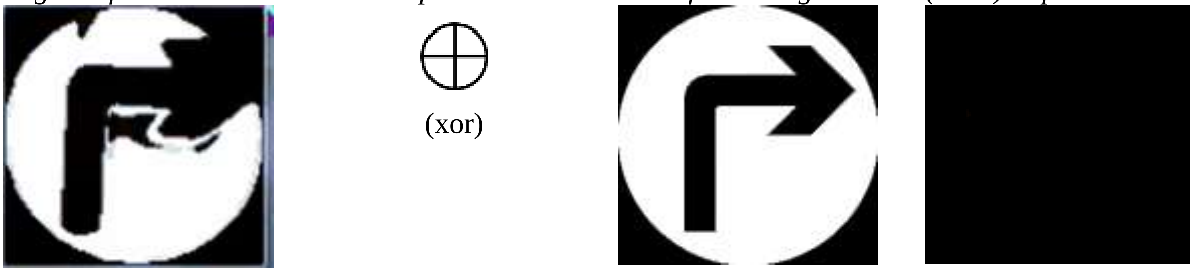
[4] http://docs.opencv.org/2.4/doc/tutorials/imgproc/shapedescriptors/find_contours/find_contours.html

[5] http://docs.opencv.org/2.4/doc/tutorials/imgproc/shapedescriptors/bounding_rects_circles/bounding_rects_circles.html

The amount of white pixels is an indication for how closely it resembles the expected shape. The lower the amount of white pixels, the better the fit, thus by setting an upper bound of accepted white pixels, it is made sure the shape is in line with a sign.

As we are now relatively sure the shape within the region of interest is in accordance with the sign, we repeat this process, but now the area within the contour is used to check whether we are not simply seeing i.e. blue circles. Again, this is shown by example.

Region of interest *Bitwise operator* *Predefined image* *(ideal) Expected result*



It is obvious that the ideal result rarely will be met as the sign may be slightly rotated or the mask isn't showing the region of interest completely as expected. This is fine as the result only has to meet the expected result to some degree. Again, the mismatch is indicated by the amount of white pixels. By setting an upper bound it is made sure that the system is relatively certain a sign is found.

The first check could in theory be omitted as the second check has shape information in it as well, but this reduces the certainty of the detection as the area where white pixels may occur due to mismatching essentially doubles, making it more difficult to set accepting bounds without losing accuracy.

As an example; think of a blue rectangle with a vertical lane in the middle. If we xor this with the straight sign, almost everything would be black except for the corners. Now, as the corners are not the major part of the images this is hard to correct for using only setting a threshold. Hence the idea is that if we know the shape of the image is right, we can be more precise about what is inside it.

3.3.4. Tweaking

It was found that this upper bound was frequently violated when small signs were detected. It is believed that this is due to the resizing of the image. When a small region of interest is stretched to be 100x100 pixels, also its error-margin (in the form of white pixels) is increased. This was solved by distinguishing between these small and larger signs:

- For the larger signs the method described above is applied: should the amount of white pixels be below a certain value, the system is confident it found that specific sign.
- For smaller signs, the region of interest was compared with all predefined images; the result which had the least amount of white pixels was selected as the sign found.

This latter approach worked as intended, but may give rise to more frequent false positives. This is of no issue however as the sign will be properly detected when it gets closer. It merely makes sure that when the field of view of the camera is insufficient (when swerving due to lines which are far apart), the system already did see the sign with an acceptable degree of certainty.

Furthermore, it may occur that the system detects multiple signs in a frame. It is then decided that the sign with the largest radius has priority over smaller signs. This makes sense as the radius is directly related to the distance from the sign, given that each sign has approximately the same size.

IV. CONTROL

The sign and path detection provides an output of which signs and lines were detected as well as their position. In order to be able to follow a path correctly with this output, a control loop has to be implemented to determine when and how to react to the path and signs.

In order to control the car the speed of the left and right motor is controlled. We chose this method of control as it allows us to keep the Arduino code simple and eliminate state behaviour in the Arduino. This also makes our motor control idempotent allowing us to resend set points. We therefore do not have to keep track of distances or angles. Instead we reasoned that we will be able to do corrections by changing the motor speeds and that at the right speed this can be done quick enough to correct.

The main and default state of our control loop is the follow path state. In this state the car will attempt to stay in the lanes. This is the default state since path following is applicable in most cases and even if a sign is missed may still result in following the path correctly. The main state machine responsible for the control is given in Figure 8.

In order to handle state transitions correctly we maintain a counter for each possible sign. When a sign is detected in a frame the counter for that sign is incremented and all others are decremented by a fraction. When the counter reaches the valid threshold a state transition is allowed. This prevents false positives from triggering an incorrect state transition. Transitions using the sign counters can take place in the follow path, turn left and turn right states. After the action is performed and the exit transition takes place the counters are reset.

We allow transitions to other states in the turn left and turn right state, because we do not have clear exit state transitions for these states where we know for sure when the turn was completed.

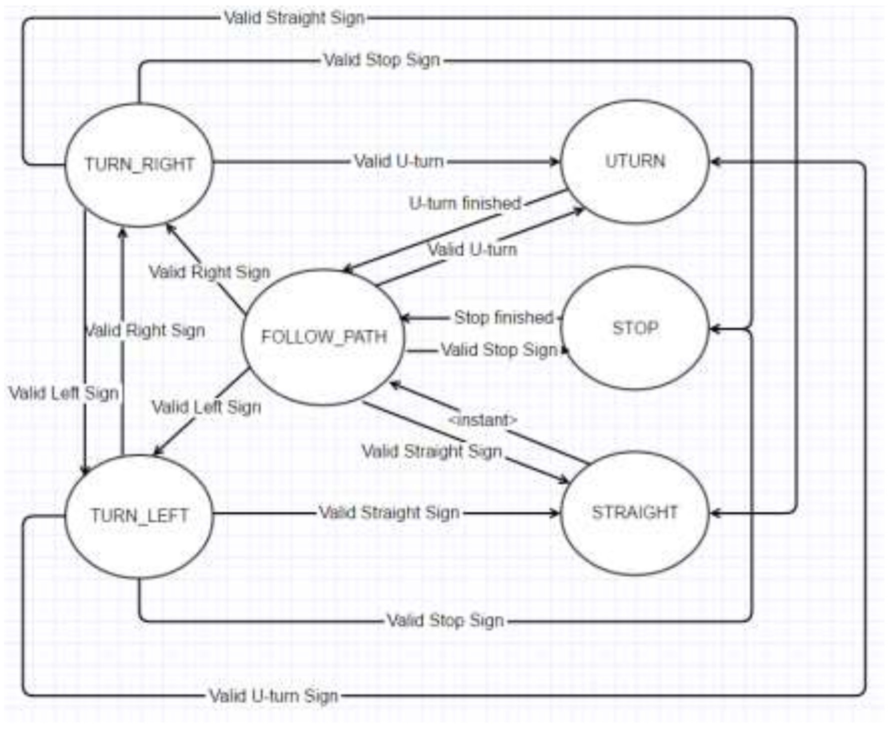


Figure 8 Main Control State Machine

4.1. PATH FOLLOWING

Following a path mainly depends on the vertical lines detected to the left and the right of the solar car. The goal of path following is simply to avoid crossing over the lanes. Our control tries to estimate the distance of the lanes from the car and keep it within acceptable margins. We considered other method such as determining the angle, or intersection points of lanes, but dropped this for this simpler solution. This solution also allows us to control the car when detecting both or only one of the lines. The description of path following is given below:

Vertical lane detection:

The lines that are considered to be our vertical lane lines are determined as follows:

1. The outcome of our vision processing is a set of lines both horizontal and vertical.
2. From this set of lines the first line to the left and the right from the middle of the image is chosen.
3. For these lines their angle and intersection point with a horizontal line (blue in Figure 9) drawn near the bottom of the image is calculated.
4. Lines where the angle indicates that the wrong line was chosen, for instance its angle is conflicting, are dropped. Similarly if the intersection is too large indicating a horizontal line it is also dropped.

Example output:

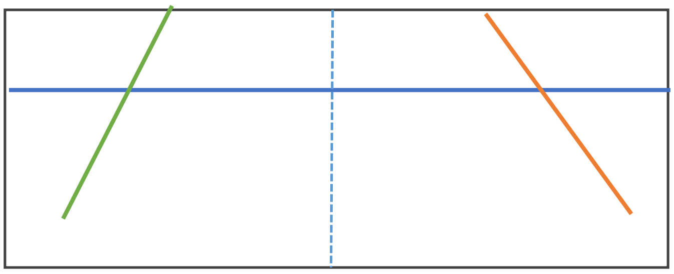


Figure 9: Example vertical lane detection

cmd output:

{“LeftLine”:[x1,y1,x2,y2,intersection,angle],”RightLine”:[x1,y1,x2,y2,intersection,angle]}

Based on the result of this our path following algorithm distinguishes 4 different cases and follows the lanes accordingly:

Both Lanes:

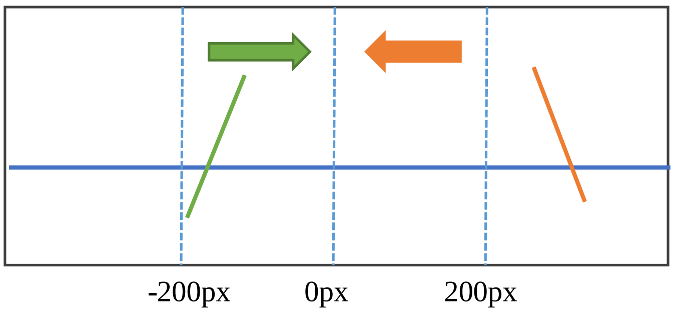


Figure 10: Both lanes control (correcting to right)

When both lanes are detected, correcting to the left or right is done when either of the lines are too close to the middle. This is the case when the intersection point is less than 200px from the centre. When this happens the car will correct in the opposite direction in order to bring the intersection point further than 200px from the centre. When both lines are further than 200px from the centre the car will move straight forward. The speed of the correction turning is determined by how close the intersection is to the middle. We do not consider the case when both lines are less than 200px from the centre, this indicates that the path is too narrow for the car to cross and therefore we neglect it. In the example above, the car will correct to the right in order to move the left lane out of the 200px zone.

Single Lane:

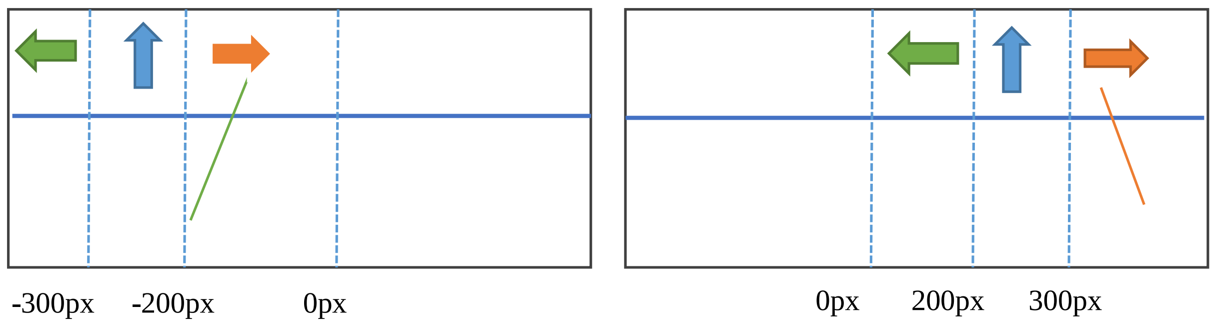


Figure 11 Left lane control (correcting to right) and right lane control (correcting to right)

The next two cases is when only the left or right line is detectable. When only one lane is detected correcting to the left or right is done when line is too close or too far to the middle. This is the case when the intersection point is less than 200px or greater than 300px from the centre. When the intersection is less than 200px the car will correct to move the line towards 200px. For the left this means turning right and for the right this means turning left. When the distance is greater than 300px the car will correct to move the line towards 300px. For the left this means turning left and for the right this means turning right. When the line is between 200-300px the car will move straight. The speed of the correction turning is determined by how far the line is from this boundary. In the example above (Figure 11 left), the car will correct to the right in order to move the left lane out of the 200px zone. And in example on the right the car will correct to the right to bring the right lane to the 300px boundary.

No Lanes:

When we do not detect any lanes the car will simply continue moving straight. Our assumption is that we are still within the lanes even though we do not see them. As such, we will eventually detect a lane when we come close to crossing it.

Horizontal Lane detection:

Vertical lane detection is sufficient to follow a path that does not contain very sharp 90 degree turns or junctions. In order to negotiate these we also use horizontal lines to determine if the car is approaching a lane head on. Deciding what to do in this case is non-trivial, turning right or left might mean we are going in the wrong direction. In this case we base our decision on the last detected sign.

4.2. Sign Reaction

With the basic path following logic implemented, it is necessary to react to the signs. In order to do this the expected behaviour is defined for each of the signs. The behaviour is relatively simple and does not cover all possible cases. This behaviour is described in Figure 12 and in detail below.

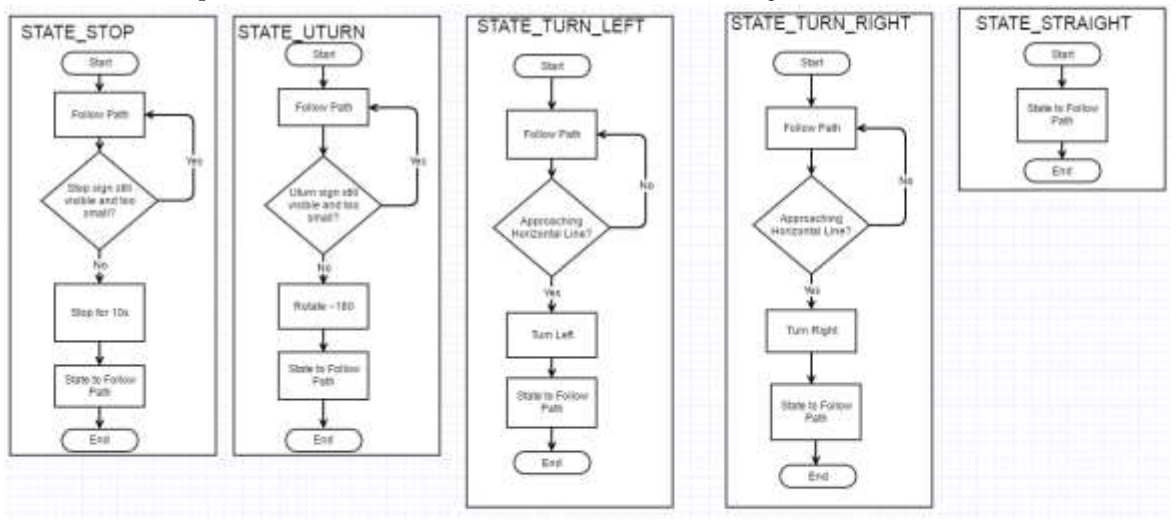


Figure 12 Flowcharts for reacting to signs

	To handle the turning of left and right the car will continue to follow the path until confronted with a horizontal line. When this occurs the car will turn in the direction indicated by the sign. It does this until no horizontal line is visible and will continue following the path. The car will stay in this state until another sign is encountered. This is because we cannot be sure when a turn is complete, for instance if no horizontal line is encountered. This strategy works in most cases but will not work if approaching a 4-way intersection where no horizontal line will be encountered. Our planned approach for this was to detect 3 horizontal lines one for the right, centre and left. In this case the car will turn when detecting the horizontal line in the direction of the turn. Due to time constraints we did not implement this.
	To handle the straight sign we simple transition back to following path. Since this sign indicates that we should keep following the path.
	When reacting to the stop sign the car will continue to follow the path until the stop sign is no longer visible. Or is visible, but too big indicating that the car is close to the stop sign. The car will stop for 10 seconds and then transition back to the following path state.
	When reacting to the U-turn sign the car will continue to follow the path until the U-turn sign is no longer visible. Or is visible, but too big indicating that the car is close to the U-turn sign. The car will then move the motors in opposite directions for 6 seconds. This is approximately enough time to do a 180 degree turn, but does not have to be exact and anything from 120-240 degrees is enough since the car will go back to following path and can correct for a partial or over turn. After this the state transitions to following path.

V. COMMUNICATION

The communications on the robot consist of three parts. The most important part is the communication from the Raspberry pi to the Arduino over UART. Then there is the DBus which lets programs communicate to each other and the JSON RPC which is accessible over TCP for manual control and logging.

In the image below an overview is shown of the communication model. This model consists of an `InterfaceCollection`, which is a class that manages all available interfaces and the way the interfaces are connected together, and the interfaces itself.

All the interfaces work in asynchronous way, meaning that whenever a function is called the program does not stall until a reply is received. To achieve this, the signal and slot mechanism[6] of Qt is used.

[6] http://doc.qt.io/qt-4.8/signalsandslots.html

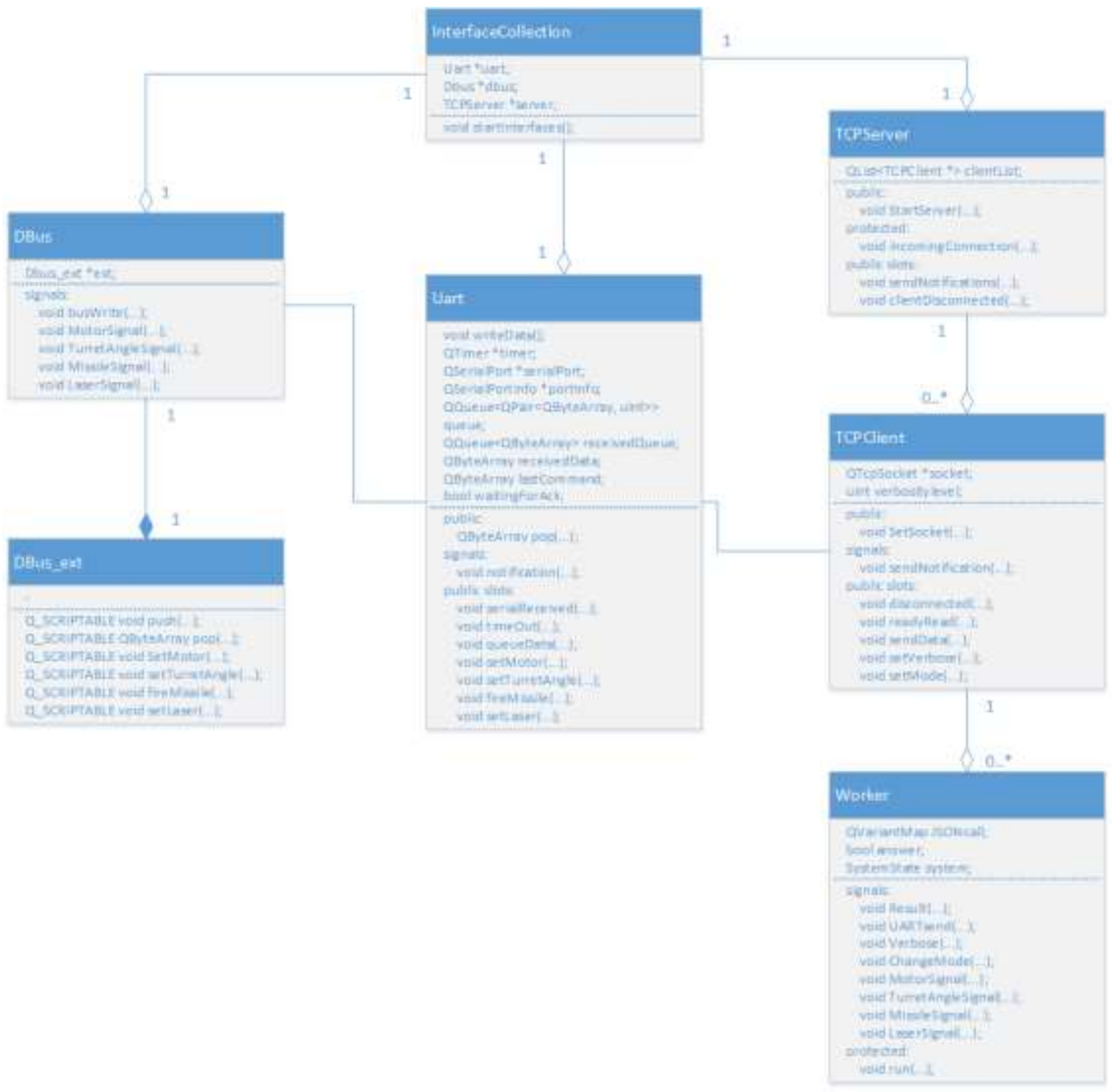


Figure 13 Communication model

5.1. UART

In Figure 13 the class Uart is shown. The most important properties of this class are the `QTimer`, the `queue`, `receivedData` and `waitingForAck`. The structure of the packets, which can be found in appendix A, consists among others of start and stop bytes and a simple checksum.

Although the UART interface is fully asynchronous, commands are still only send when the previous command has been acknowledged by the Arduino. This is to make sure the limited buffers of the Arduino are not overflown, and thus packages are not dropped. The asynchronous part of this class is that it is always possible to add new commands to the queue even though a reply has not yet been received.

The timer (`QTimer`) is there to make sure that all of this does not lead to a deadlock. Every time a new command is send, and thus the system is waiting for an acknowledgment, a timer is started which triggers a reset of the buffers when timed out. The timer is stopped automatically whenever data is received before the time-out, but is restarted if the received data does not (yet) contain a valid [Appendix A] packet.

The received data comes in as chucks which are asynchronously received. These chucks of data are concatenated in `receivedData` until a valid packet is found in it. When such a packet is found it is removed from `receivedData` and added to the output queue.

As mentioned before the commands are queued before being send. For manual control however a lot of commands can be sent for setting the angle of the turret and the speeds of the motors. Therefore the queue takes both the data packet and the command ID [B], which makes it possible to replace commands in the queue with the same command ID. Old commands are replaced instead of just being deleted from the queue to make sure that once a command with a certain ID is put in to the queue, there will be a point at which a command with that ID will come out, even though the data might have changed.

5.2. DBUS

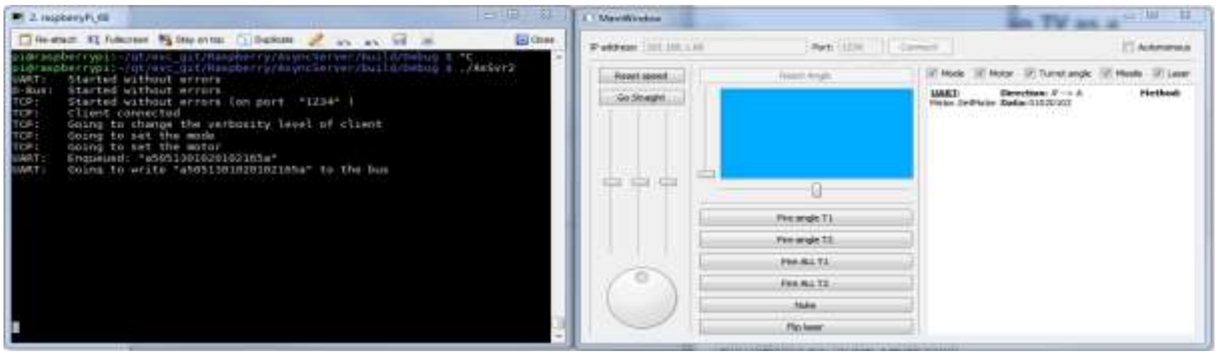
D-Bus[7] is a message bus system, a simple way for applications to talk to one another. In addition to inter-process communication. In Figure 13 the classes `DBus` and `DBus_ext` are shown. Of these classes `DBus_ext` is the most interesting as this is the class that is externally available.

This D-Bus is implemented so that this communication interface keeps totally independent from the control loop and other parts of the system. This should make it possible to write parts of the system in different programming languages; as long as a language is used that can connect to the D-Bus interface, it can call all the functions that are made available by this class.

Even though D-Bus method calls are normally synchronous, the implementation is modelled in such a way that it immediately returns. This is achieved by letting the `DBus_ext` class do nothing else than emitting a signal via a queued connection[8] of its parent class, `DBus`, and return after that. This is to ensure that whenever some other program calls a D-Bus method, the calling program does not wait for a long time even though the method does not return anything. For example, when using the `push(…)` method to send raw data over UART, it makes no sense to wait until the data is actually send.

While the D-Bus implementation works as-is, it does not fully come to justice. The initial plan was to implement a bigger D-Bus API that would, using D-Bus signals[9], allow to run different parts of the system in different programs that are all connected to the D-Bus. However, due to time constraints and the control loop being fully implemented in Python anyway this is dropped.

5.3. TCP



```
pi@raspberrypi:~/qt/esc_git/Raspberry/AsyncServer/build/Debug $ ^C
pi@raspberrypi:~/qt/esc_git/Raspberry/AsyncServer/build/Debug $ ./AsSvr2
UART:    Started without errors
D-Bus:   Started without errors
TCP:     Started without errors (on port  "1234" )
TCP:     Client connected
TCP:     Going to change the verbosity level of client
TCP:     Going to set the mode
TCP:     Going to set the motor
UART:    Enqueued: "a5051301020102105a"
UART:    Going to write "a5051301020102105a" to the bus
```

Figure 14 Left the debugging of the server, right the client GUI.

The TCP interface is mainly for manual control and logging. In Figure 13 the classes `TCPServer`, `TCPClient` and `Worker` are shown. These three classes handle the TCP connection and provide a fully compliant JSON RPC 2.0[10] API. See appendix C for the full RPC specification and some examples. The TCP interface allows for debugging in a more lightweight fashion than for example a SSH tunnel.

[7] https://www.freedesktop.org/wiki/Software/dbus/
[8] http://doc.qt.io/qt-5/qt.html#ConnectionType-enum
[9] https://dbus.freedesktop.org/doc/dbus-tutorial.html#signalprocedure
[10] http://www.jsonrpc.org/specification

As requesting state information via TCP is difficult as the communication program does not track the state and should thus communicate either to the main control loop, which is constantly busy, or to the Arduino over a relative slow UART connection, it is chosen to not make interfaces to request this information. Instead a publish subscribe model is chosen, where clients can subscribe to certain subjects and the server will then send notifications when something has happened regarding that subject.

As a result of this publish subscribe model the interface shown in **Error! Reference source not found.** will always be up to date, assuming the client is connected to the server and is subscribed. It does not matter if some other client is controlling the car manually or by the control loop, notification will always be sent. Note that in the image the data that is enqueued for UART is also visible in the log window in the client GUI.

To keep the TCP communication asynchronous, the server only handles incoming connections. When a client connects, the server creates a new `TCPClient` to handle the communication. Now when the client sends data, a new `Worker` is created, which runs in a different thread to keep everything asynchronous and non-blocking again, by `TCPClient` to handle the request.

VI. ARDUINO

The Arduino is responsible for the control of the motors, turret and measuring of the current. In order to do this the Arduino decodes commands from the Raspberry Pi and encodes commands as responses. The software architecture in the Arduino is described below:

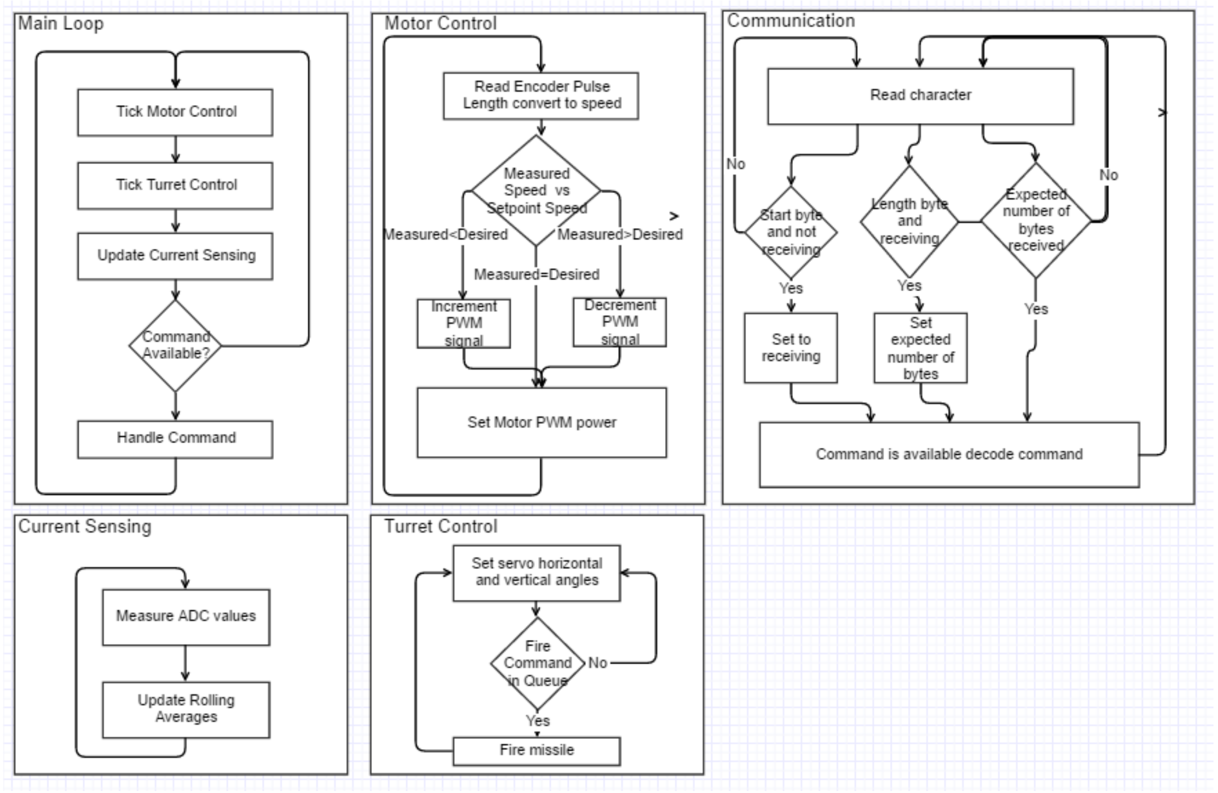


Figure 15 Arduino Software Architecture

- Main Loop: The main loop is written in an AFAP (As Fast As Possible) fashion. For each iteration of the loop the motor control and current control is ticked and the average of the current sensing is updated. At the end of the iteration it is checked if a command is queued and is handled if that is the case. For all commands a setting is either set or retrieved from the respective motor, turret or current sensing modules. This acts as a flag that is handled in the next iteration. This method was chosen to make it easy to add additional modules and to keep the handling of the commands consistent and quick.

- Motor Control: To control the motors the encoders are used to determine the rotation speed. To do this the pulse length is measured and converted to the speed. This speed is compared to the desired speed set point. If the actual speed is less than the desired speed the PWM duty cycle is increased. If the actual speed is more than the desired speed the PWM duty cycle is decreased. We chose this simple method of control since we had trouble using the PID controller and were unable to get smooth speed control. We also reasoned that since our main loop iteration time is less than 20ms we will be able to react quick enough to not need a more complex motor controller.
- Communication: The communication module is responsible for receiving and decoding commands as well as replying with data. Since the protocol defines a start byte the communication waits for this start byte and then reads the number of bytes specified by the length byte that follows. The resulting data package is decoded to get the command id and data. The checksum is checked to see if any errors were introduced during transmission and the command is dropped if that's the case. The communication is also responsible for encoding data and sending replies.
- Current Sensing: The current sensing module measures the ADC values in each iteration and converts it to the current. These currents are added to a rolling average that eliminates current spikes. This code is not the code provided by group 1.
- Turret Control: The turret control is responsible for maintaining the aim, turning the laser on and off and firing the missiles. To maintain the aim set point angles are received as commands and the servos are set to these angles. Similarly commands are available for turning the laser on and off and firing missiles. Since firing a missile takes 100ms, a queue was created to fire missiles in succession. To maintain the timing of 100ms a non-blocking timer is used to ensure the loop continues to iterate.

VII. SECONDARY ASSIGNMENT

The secondary assignment that was chosen is a mounted turret that can auto aim and fire “missiles” at the signs.

To control the aiming of the turret a pan-tilt mechanism was designed. This mechanism is controlled by two servos. A laser is added to give an indication of the target the turret is aiming at. The missile launchers are produced by WLtoys and are intended for quadcopters. These missiles work using a spring loaded mechanism. A small motor rotates to push a pin on the mechanism and fire a missile, by controlling the time the motor is on a missile can be fired.

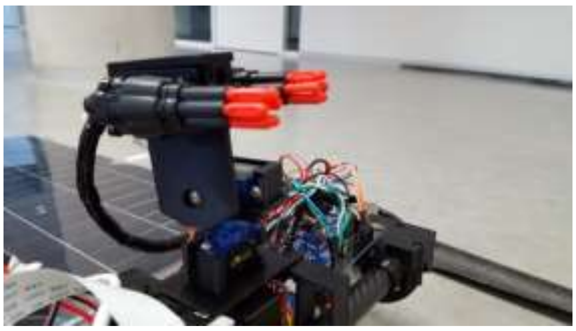


Figure 16 Turret mounted on car

The finite state machine for the control of the turret is given in Figure 17. Whenever a sign is detected the turret will aim at it using the x, y coordinate of the sign in the image. In order to do this the angles were calibrated by pointing the laser at the edges of the camera’s field of view at a distance of about 1m. These angles where then mapped to the x, y coordinates. Since the distance of the sign is not considered aiming will be less accurate if the distance is too far or close. We chose 1m since this is about the range that the “missiles” could fire. When aiming the laser will be turned on, if no sign is detected in the current frame the laser is turned off. When the car switches state and a sign is detected that is close enough, a missile will be fired.

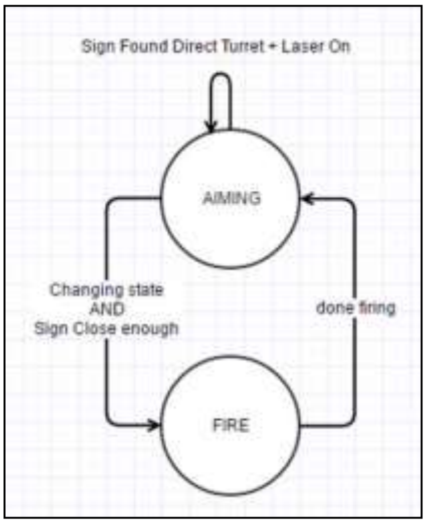


Figure 17 Turret control FSM

VIII. MECHANICS

The mechanics that were designed and printed includes the mechanics for the turret and a modified back wheel to improve the car handling.

8.1. TURRET

In order to aim our turret we designed and printed a pan-tilt mechanism that can hold the turrets and laser. We also printed mounts to mount the mechanism to the rods of the car. The mechanics are illustrated below:

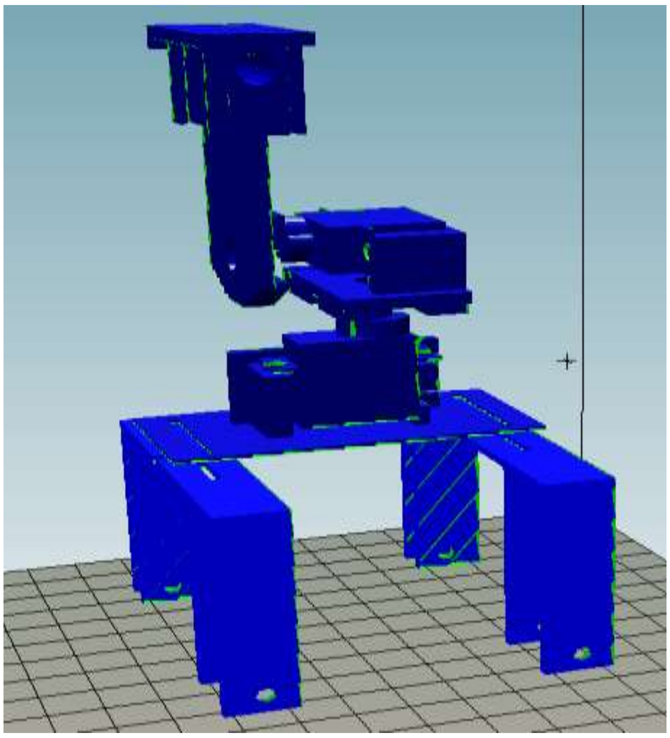


Figure 18 Turret Assembly

8.2. BACK WHEEL

Since the car had issues with slow turning and the back wheels locking we decided to modify our car to fix this. Even though we were able to correct for this in software, we felt that it would be better since on a narrow track this may result in the car crossing lanes. We designed and printed a mount for a single back wheel. We printed our group number in the mount for easy identification of our car.

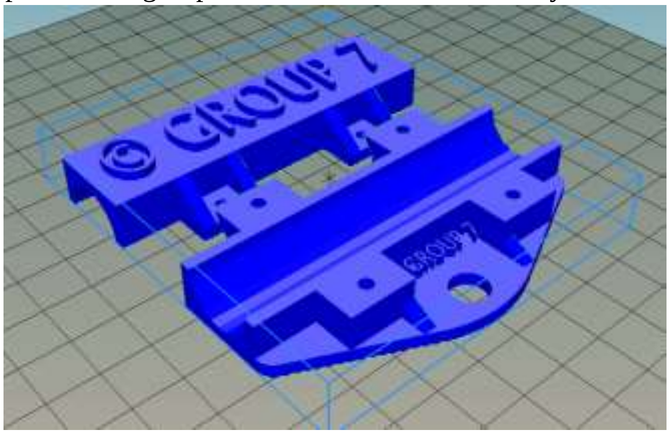


Figure 19 Back wheel mount

IX. DEMONSTRATION

During the demonstration on 1st July the solar car performed two runs on the test track. The results below are for the final test run. This run was completed in 1 minute and 35 seconds. We consider the final run a success since the course was completed without any human intervention and without the front wheels touching the lines. During the run all the signs were recognized and reacted upon.

9.1. Power Consumption

Our focus for this project was not on reducing the power consumption, but rather completing the course successfully. We reasoned that completing the course efficiently and with fewer corrections will reduce the current consumption. This is one of the reasons why a slower motor speed was chosen. Note that these power consumption measurements also include the current consumed by the turret which consists of the two servos, two motors to fire missiles and a laser.

The current consumed by the complete system is given in Figure 20. The current measuring software maintains a rolling average over the last second in order to average out large spikes. The software used for these measurements is not the same software as provided by group one, but uses the correct current scaling.

In total 110 data points were taken, one data point per second for the run. Some interesting observations can be made, for instance at the U-turn there is a large current spike, which could be due to the reversing of the one motor in order to make the turn. Also the left turn saw an increase of current, due to the relatively tight corner and the several corrections in turning that the car made. As expected, the lowest current consumption can be seen after the car stopped. The average consumed current was 106 mA with a power consumption of 1.28 W.

In Figure 21 the current generated by the solar panel is given. The average current provided was 76 mA. This is surprising high given that the test was performed inside on a cloudy day.

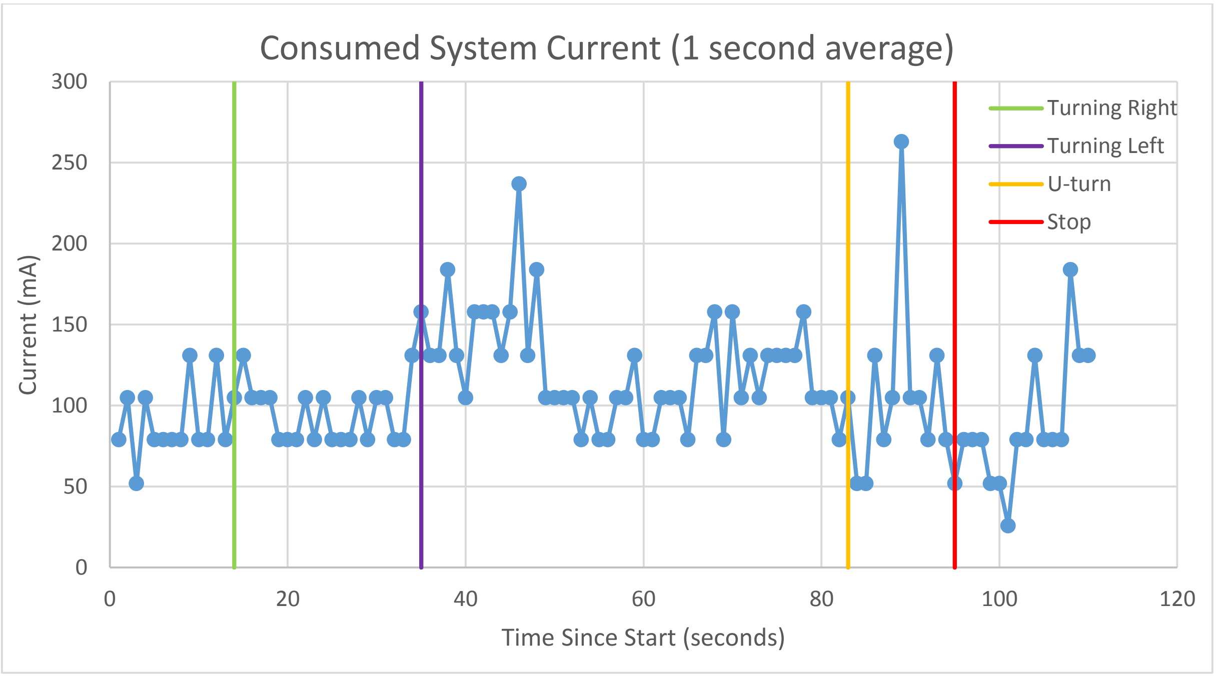


Figure 20 System Power Consumption during run

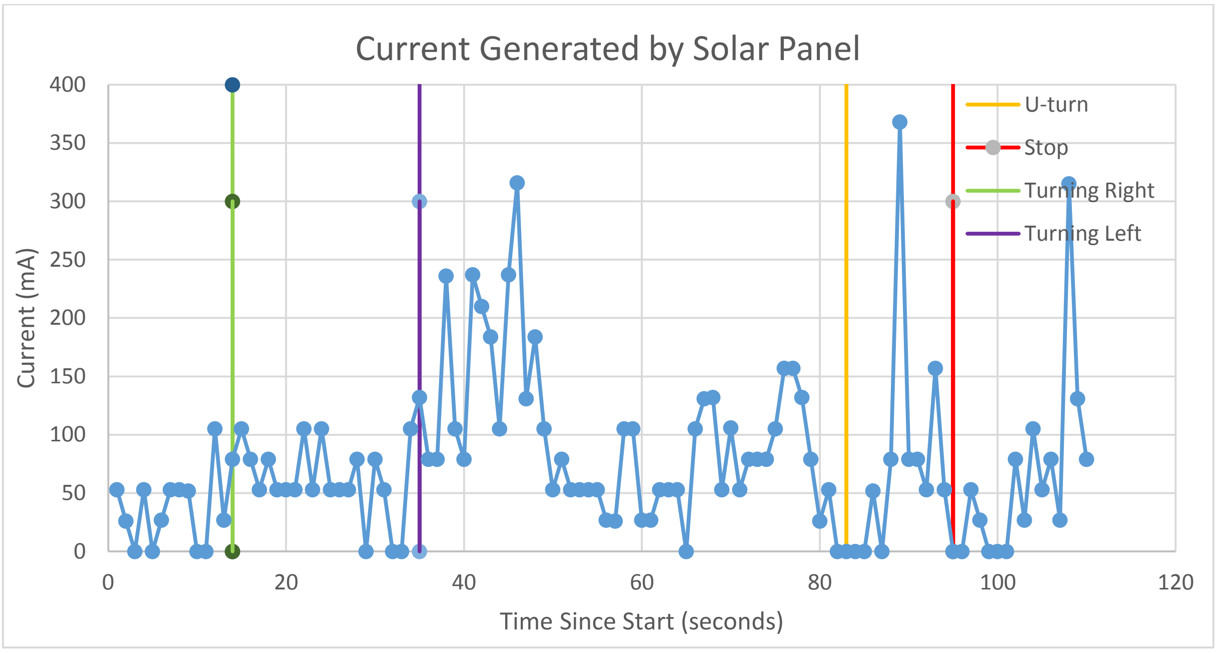


Figure 21 Current Generated by Solar Panel

X. CONCLUSION

Even though there is enough room for improvement (see chapter 0 below), the project could still be seen as successful given that the solar car, as opposed to all competing cars, autonomously finished the demo track without touching the lines while acting upon every sign (see chapter IX).

The detection of lanes and traffic signs works very reliably when calibrated correctly, but reliability drops fast when the light of the environment changes. Heuristics can obviate some of these problems for line detection, but does not work that well for sign detection. After all, intermittent lines are still lines, while traffic signs rely on the contour and therefore do not work when wrongly cut.

The computational heavy Hough line detection for lines and blurring for the sign detection make the performance and therefore framerate drop. Alternative methods are investigated but not implemented due to time constraints.

The information that is extracted from the vision is handled in the Control Loop. This Control Loop is a finite state machine which has following lines as its default state which increases the reliability as the car will still stay within the lines even though a sign might be missed.

To change the state traffic signs have to meet a certain threshold until it is seen as a valid sign. This is again to increase the reliability as false positives can be eliminated.

A robust and fully asynchronous communication mechanism is implemented to let different parts of the system and the outside world communicate.

An interface for UART, D-Bus and communication over TCP is implemented. The communication between the Raspberry Pi and the Arduino uses UART. To make this reliable a checksum is added while timers prevent deadlocks. A D-Bus is implemented for IPC and to be independent from programming languages. The TCP handles the communication with the outside world to remote control and log the system in a way that is computationally not intensive.

The Arduino handles all the control of the actuators and sensors a relative simple way. There is an AFAP main loop which reads UART commands and handles those when available. The motor control uses encoders to check if the actual speed agrees with the desired speed and increments or decrements its setpoint based on this.

As a secondary assignment it is chosen to add a turret with a laser to the car which aims at the signs and shoots at them when close enough. The turrets are bought, while the structural parts are 3d printed by members of the team.

Another change is made to the car as well using 3d printed parts. The original design of the car had two back wheels which bumped to the side-lines to often and are therefore replaced with a single back wheel.

We, members of Group 7, have worked together very tightly with a lot of full day meetings. A meeting took place at least every lecture day. When the deadline was approaching multiple late night meetings took place until the closing of TU/e, these involved heavy testing and debugging. All of our members collectively went well over time budget, but agree it was worth every minute and are proud to be the only group to finish the test track completely autonomous.

XI. FUTURE WORK

There are several issues with the current implementation which can be improved in future iterations. These issues are mostly performance and efficiency related.

- **Lower image quality.** For the current implementation an image size of 640x480 is used. This size was chosen to make sure the image quality was high enough to be able to extract the necessary information. It seems the case however that this may also be the case for lower resolutions. Even if this makes the path –and sign detection more difficult, the gain in performance from lowering the resolution will be significant. Due to timing constraints this is not implemented.
- **GPU pre-processing using OpenGL (ES):** It was encouraged to implement GPU support to achieve a higher performance. This makes sense as a high amount of processing power for detecting lines and signs is in the pre-processing of the images. Most of these steps, like blurring and filtering, lend themselves for great optimisation using parallel processing. The Raspberry Pi has a very capable GPU which could be used for these parallel task, but there unfortunately is no official, supported way of using it for parallel computing. The Raspberry however does support OpenGL (ES) which makes it possible to use the GPU, although it is a bit cumbersome. Quite some time was invested in investigating this, but lack of documentation and examples lead to dropping this for the time being, given that even with a low framerate, an acceptable result can be met.
- **Parallelization:** Python offers methods for parallelization by use of the "PP"-module[11]. This would make a significant increase in throughput possible as the path –and sign detection would be ran in parallel instead of sequential, given that both functions have an approximately equal runtime. Additional overhead could result in a slightly higher energy consumption however. This is not fully implemented due to timing constraints.
- **Porting to low level language:** Python is chosen as the language for the path and sign detection as it is ideal for prototyping since it is a high level language and does not need compilation. This however comes at the price of possibly both speed and power consumption. The D-Bus (chapter 5.2) makes it possible to use a different kind of language for each part of the system, and thus allowing the use of multiple languages that is tailored to each task. It should of course be noted that the D-Bus has overhead itself, and that it might be even more interesting to port everything to a low level language.
- **Auto-calibration**: Each change of environment/lighting makes colours observed differently. This results in a sub-optimal colour-mask; even with auto-white balancing disabled. Therefore frequent calibration is necessary. It would be worthwhile to investigate whether this may be done automatically or use the auto-white balance function for recalibration of the colour thresholds.
- **Avoiding computational heavy functions**: Computational heavy functions like blurring and Hough functions are used while more lightweight methods exist. For example; path detection currently is based on a HoughLine detection method. A method based on contours was investigated as mentioned in the path detection chapter, but ultimately dropped due to timing constraints. Given time, this implementation based on contours may function adequately while reducing the computation time of the path detection significantly.
- **Intersections:** The current implementation relies on horizontal lines appearing when a left –or right sign is detected. While this works for the given path, this will not work as expected when taking 4 way intersections into account. A different, more accurate path detection may be necessary to deal with these type of situations.

[11] http://www.parallelpython.com/

- **Narrow field of view**: The given system has a relatively narrow field of view making it possible that a sign is missed when swerving through the lanes. This issue may become less troublesome when the previous matters are attended to and a higher framerate is achieved. Additionally a greater field of view can be obtained by installing a fish-eye or wide-angle lens.

XII. WORK DISTRIBUTION

The workload of the project was evenly distributed. For the sake of simplicity is the project divided in three main subjects namely Vision, Communication and Control. Each of the group members had a main subject while participating to some extent with each of the main parts of the other members. A close co-operation was possible due to frequent meetings and working concurrently on different parts of the system while continually helping each other out.

Who	Main Focus	Tasks completed
Harm van Schaaijk	Communication	• Communication D-Bus and Commands encoding/decoding • UI remote control (on Laptop) • Open CV environment setup • Vision (Sign/Path Detection) • Mechanics back wheel • Car Assembly • Communication Protocol/System Architecture • GPU pre-processing using OpenGL(ES) (not completed)
Michael Kruger	Control	• Mechanics turret • Arduino code (motor/turret/communication/current sensors) • Python Project framework • Sign Reaction/Path following/Turret Control State Machines • Car Assembly • Communication Protocol/System Architecture • Printing Assistant
Tom Vogels	Vision	• Vision (Sign/Path Detection) • Vision (conversion to distances/angles) • Vision thresholding/calibration • Open CV environment setup • Sign Reaction/Path following/Turret Control Strategies • Communication Protocol/System Architecture • Car Assembly • GPU pre-processing using OpenGL(ES) (not completed)

APPENDICES

A. UART Packet Structure:

Start Byte	Length	Command ID	Data	Checksum	Stop Byte
0x5A	1 Byte	1 Byte	Fixed length	1 Byte	0xA5

- **Start Byte:** Every packet starts with this byte.
- **Length:** Gives the length in Bytes of the **Command ID + Data.**
- **Command ID:** Defines which command should be executed. (see …)
- **Data:** Has a fixed length for each **Command ID.**
- **Checksum:** Is a XOR of **Command ID**, and each Byte in **Data.**
- **Stop Byte** Every packet stops with this byte.

B. Communication Arduino to Raspberry PI

Raspberry PI is the master and initiates all communication. Arduino is the slave and responds to commands. Commands in red were planned, but not implemented.

Command ID	**Parameters**	**Description**
Status info:		
0x01	-	Error status (TBD)
0x02	-	Get Command queue length (ready to accept commands)
0x03	-	Reset all peripherals
0x04	-	Test sequence
Motor Control:		
0x11	DIR SPEED	Set left motor speed and direction
0x12	DIR SPEED	Set right motor speed and direction
0x13	DIR SPEED DIR SPEED	Set the speed and direction for both motors simultaneously
Power Information:		
0x21	-	Get battery current
0x22	-	Get system current
Turret Control:		
0x31	ANGLE	Set turret horizontal angle [0, 180]
0x32	ANGLE	Set turret vertical angle [0, 90]
0x33	-	Fire missile from turret 1
0x34	-	Fire missile from turret 2
0x35	-	Fire all missiles turret 1
0x36	-	Fire all missiles turret 2
0x37	-	Fire all missiles both turret
0x38	ON/OFF	Turn laser on [0x01] /off [0x00]
0x39	ANGLE ANGLE	Set angles for both horizontal and vertical angles simultaneously
0x41-0xFF		RESERVED
Configdata		
0x41	Mode	Change control mode (autotarget+manualdrive vs autodrive)
0x42	MaxSpeed	set the maximal speed of the system
0x43	E_Mode	Set energy mode of the system

C. JSON RPC METHODS

RPC methods

“JSONRPC.Version” will return the version of the JSON RPC. It does not need any parameters. The answer contains the version in the result field

Example:

```
→       { "jsonrpc":"2.0",
          "id":1,
          "method":"JSONRPC.Version" }
←       { "jsonrpc":"2.0",
          "id":1,
          "result":"0.1" }
```

“JSONRPC.GetMethods” returns the methods that are supported by the current version of the RPC. It does not require any parameters. The server’s response will contain all supported methods in an array.

Example:

```
→       { "jsonrpc":"2.0",
          "id":2,
          "method":"JSONRPC.GetMethods" }
←       { "jsonrpc":"2.0",
          "id":2,
          "result":[
            "JSONRPC.GetVersion",
            "JSONRPC.GetMethods",
            "System.SetMode",
            "..." ] }
```

System Control Methods

“System.SetMode” sets the operating mode of the system. The server will respond that the setting was successful.

Params: 1. System.SetMode `( “autonomous”, “manual” )`

Example:

```
→       { "jsonrpc":"2.0",
          "id":3,
          "method":"System.SetMode",
          "params":{
            "mode":"autonomous"} }
←       { "jsonrpc":"2.0",
          "id":3,
          "result":"OK" }
```

“System.SetVerbose” sets the verbosity level. The system sends out notifications about various parts of the system. With this function it is possible to subscribe to each of these parts.

Params: 1. System.VerbosityList

(“mode”,”motorSpeed”,”turretAngle”,”turretmissile”,”laser”)

Example:

```
→      { "jsonrpc":"2.0",
         "id":4,
         "method":"System.SetVerbose",
         "params":[
           "mode",
           "motorSpeed",
           "turretAngle" ] }
←      { "jsonrpc":"2.0",
         "id":4,
         "result":"OK" }
```

“System.GetMode” gets the operating mode of the system. The method takes no parameters and the server will respond with a `System.OperatingMode (“autonomous”, “manual”).`

Example:

```
→      { "jsonrpc":"2.0",
         "id":5,
         "method":"System.GetMode"}
←      { "jsonrpc":"2.0",
         "id":5,
         "result":"autonomous" }
```

“System.SendUART” sends a string of data over the bus. The server will respond that the setting was successful.

Params: 1. System.uartDataType `( “hex”, “string” )`

Example:

```
→      { "jsonrpc":"2.0",
         "id":6,
         "method":"System.SendUART",
         "params":{
           "dataType":"hex"
           "data":"5AFFA5" } }
←      { "jsonrpc":"2.0",
         "id":6,
         "result":"OK" }
```

Motor Control Methods

“Motor.SetMotor” sets the speed of one or two motors. The motors can be set by setting the speeds of the motors individually. If an optional parameter is not supplied, the system will not change the speed of that motor. Note that this method only works when in automatic mode.

Params:	1. Optional.Motor.Speed“left”	`( [-255,255] )`
	2. Optional.Motor.Speed“right”	`( [-255,255] )`

Example 1:

```
→       { "jsonrpc":"2.0",
          "id":7,
          "method":"Motor.SetMotor",
          "params":{
            "left":"-6",
            "right":50 } }
←       { "jsonrpc":"2.0",
          "id":7,
          "result":"OK" }
```

Example 2:

```
→       { "jsonrpc":"2.0",
          "id":8,
          "method":"Motor.SetMotor",
          "params":{
            "type":"speed",
            "left":"50" } }
←       { "jsonrpc":"2.0",
          "id":8,
          "result":"OK" }
```

Turret Control Methods

“Turret.SetAngle” sets the horizontal and vertical angle of the turret. The angle can bet set for one or both the horizontal and vertical angle. If an optional parameter is not supplied, the system will not change the angle of that motor.

Params:	1. Optional.Turret.Angle	“horizontal”	`( [0,180] )`
	2. Optional.Turret.VAngle	“vertical”	`( [0,90] )`

Example:

```
→       { "jsonrpc":"2.0",
          "id":10,
          "method":"Turret.SetAngle",
          "params":{
            "hoirzontal":"-6",
            "vertical":50 } }
←       { "jsonrpc":"2.0",
          "id":10,
          "result":"OK" }
```

“Turret.FireMissile” tries to fire a single or all missiles from one or both turrets, dependent on the parameters.

Returns:	1. Turret.TurretID	“turret”	`( 1,2,12 )`
	2. Turret.amount	“amount”	`( “one”, “all” )`

Example:

```
→      { "jsonrpc":"2.0",
         "id":12,
         "method":"Turret.FireMissile"
       "params":{
           "turret":1,
           "amount":"one" } }
←      { "jsonrpc":"2.0",
         "id":12,
         "result": "OK" }
```

“`Turret.SetLaser`” turns the laser on or off. The angle can bet set for one or both the horizontal and vertical angle. If an optional parameter is not supplied, the system will not change the angle of that motor.

Params: 1. bool `( true, false )`

Example:

```
→      { "jsonrpc":"2.0",
         "id":13,
         "method":"Turret.SetLaser",
         "params":{
           "on":true } }
←      { "jsonrpc":"2.0",
         "id":13,
         "result":"OK" }
```