AutoTux Final Report

Max Enelund
Jerker Ersare
Thorsteinn D. Jörundsson
Jonas Kahler
Dennis Karlberg
Niklas le Comte
Marco Trifance
Ivo Vryashkov

BSc. Software Engineering and Management,
Department of Applied IT,
Gothenburg University

May 2016

Contents

Chapter 1

Project Organization and Milestone Planning

1.1 Project Organization[TDJ]

At the start of project we conducted a few meetings in order to visualize and plan the tasks at hand. A draft of the system architecture was created which in turn was used to delegate tasks between the groups. Parts of this draft was later revised based on feedback we received as well as realizations made down the line.

The team was structured into task-oriented groups in the following manner:

STM32	Jerker Ersare Thorsteinn D. Jörundsson
Decision Maker, Overtaking and Parking	Niklas Le Comte Marco Trifance
Proxy and Serial	Jonas Kahler Ivo Vryashkov
Image Processing and Lane Following	Max Enelund Dennis Karlberg

As the project progressed and tasks were completed, group members participated in other efforts where applicable and as needed. Among those, Jerker aided in lane-following, Ivo conducted unit tests and Jonas created a configuration tool used in development and vehicle testing.

1.2 Milestones and Successfully Completed Tasks[TEAM]

We always delivered on time except the sensor recording diagram (April 12), due to problems with hardware. We received a new ultrasonic sensor the day before the presentation and did not manage to make recordings and diagrams in time.

Until Feburary 2	Installed and learned about OD Made diagrams for presentation
February 02	**Simulation Environment**
Until February 16	Discussed concept and architecture Prepared report
February 16	**Concept &Architecture**
March 24	Discussed the team structure, goals, tasks and responsibilities
March 25	Received the car, basic Arduino sketch for controlling ESC and servo
March 28	Sensor readings through Arduino to measure sensor ranges and angles
March 29	**Revised Concept & Architecture**
April 1	STM32 reading IR sensor values
April 4	STM32 reading US sensor values Basic connection between boards
April 6	Basic readings from the Open DaVINCI simulations to own components Camera fully working, saving images to SharedMemory.

April 8	STM32 basic version of wheel encoder reading Basic reading and writing from/to the usb serial connection Basic parser for the data packets Basic lane-following working
April 11	Basic parking spot detection with sensors
April 12	**Basic lane-following (simulation)** STM32 PWM output Stable communication between the boards Basic parallel parking done without noise
April 13	STM32 output based on received packets First integration between LaneFollower and Decision Maker Camera handling rewritten to use OpenCV2 API.
April 14	STM32 RC mode LaneFollower using new algorithm for lane-following
April 15	Improved data packets parser and communication between boards
April 18	Smoother lane-following implemented on the car
April 19	**Lane-following (car)** Serial connection running at 15 Hz The parking handles noise data
April 22	Initial stop-line handling
April 25	STM32 LED driver and basic light logic Overtaker handles noise data
April 26	**Parking & Overtaking (Simulation)**
April 27	CxxTest introduced for the serial module
April 28	First working ATConfigurator version Robust stop-line detection

April 29	STM32 physical lights switch Google testing framework introduced Robust communication and data exchange between the two boards Parking on the real car is okey
April 31	Edge Detection changed to use Canny
May 3	**Revised lane-following & basic parking/overtaking** Refactoring and improvement of the serial module
May 4	STM32 headlights stronger in low surrounding light Data quality check added in LaneFollower Image optimizations, reduced the memory requirements
May 5	Lane follower execution time measured and optimized
May 6	ATConfigurator can send out selected state Overtaker refactored to work on car
May 9	STM32 ESC calibration mode Tests completed for the serial module
May 10	Lane follower road offset (car keeps to the right to keep markings in camera view) ATConfigurator shows ASCII representation of the current webcam image Car is working with new logic for the parking inside of the spot
May 11	**Final presentation**
May 13	Serial connection running at 20 Hz

Chapter 2

Algorithmic Aspects

2.1 General[NLC]

To create an autonomous miniature vehicle that could complete scenarios such as lane following, overtaking and parking there had to be a place where these parts were integrated. That integration point came to be the Decision Maker. The communication between the Lane Follower and Decision Maker is done using OpenDaVINCI's containers and broadcast conferences.

2.2 Decision Maker[MT]

The Decision Maker (DM) is a time-triggered component implementing the car's state machine and the main logic for controlling the vehicle. Below we list the states included in the DM state-machine and provide a brief description of their intended behaviors:

Lane Following: the vehicle follows the track and stops temporarily when stop lines are detected. Since control values are exclusively produced by the Lane Follower, this state assumes a scenario where no obstacles are placed on the track.
Driving: the vehicle follows the lane and overtakes obstacles that are placed on the track. Control values are produced by the Lane Follower and the Overtaker object included in the DM.
Parking: the vehicle follows the track constantly looking for a parking spot to perform a parking maneuver. Control values are produced by the Lane Follower and the Parker object included in the DM. This state assumes a scenario with no obstacles on the actual lanes.
Resume: this state is supposed to resume the car from a parking spot. It isn't fully developed and tested yet.
The current state in the DM state-machine can be switched in the Configurator Tool.

To reproduce the behaviors described above, the DM alternates between the control values shared by the Lane Follower and those produced by the Overtaker and Parker objects. While the Lane Follower is implemented as time-triggered component running parallel to the DM, the Overtaker and Parker are two instances of classes providing control values via public methods that are called by the DM according to the current state in its state-machine.

In addition to selecting and sharing the desired vehicle control values, the DM is also responsible for:

- communicating information about which lane the vehicle is currently traveling (relevant for overtaking maneuvers in the *driving* state).
- share light control values.

2.3 Lane Follower[DK]

The responsibility of the Lane Follower component is essentially to process and compute a desired steering wheel angle based on a picture sent through from the Proxy component. The processing and lane detection part is explained in the **Algorithm Fundamentals** section. This part will simply explain the lane-following logic, with the assumption that the image processing and lane-detection steps are already done.

Lane Following: The lane-following is based on values extrapolated during the lane-detection phase. During that phase we measured the distance from the center of the image to the left and right lines, these distances are measured in pixels. If the car is completely centered, these values should be the same. The distance to the lines when the car is perfectly centered is the value we call "distance". The objective of the LaneFollower is to try to keep both values as close to this distance as possible.
When the car is in the right lane, the logic prefers to look at the right line, only when there are no markings on the right-hand side of the road do we consider the left line markings. Here we have an example of the algorithm we use to calculate the desired adjustment:

e - Desired correction
x - Distance to right lane marking
y - Pixels to the center of the image

e = ((x - y) - distance) / distance

This is what the algorithm looks like when we are following the right line, when we are following the left line the algorithm looks a bit different, essentially it is just reversed. It is also worth mentioning that we purposely decided to offset the entire perception of the image in order to make the car stay more to the right in the lane. This helps in both parking and overtaking, by keeping to the right we are closer to the objects considered in the

parking and overtaking, thereby making sure they do not unexpectedly get out of sight. It also improves the lane-following in sharp right turns where we noticed a recurring behaviour of the right lane markings getting out of sight for the camera. Altering the perception of the image seemed like the easiest way to adjust the position of the vehicle without fiddling with the algorithm.
Additionally we also looked into the ideas of *PID Control*. The general idea of this control method is to alter the desired correction based on a few different gains. The one gain we found most useful was the *Proportional Gain*. The proportional gain is simply multiplied with the desired correction, if the proportional gain is over 1, desired steering is increased. If it is below 1, it is decreased. This proved useful when trying to smooth out the handling in curves. The proportional gain together with the aforementioned *distance* turned out to be the two values we used to tweak the LaneFollower for smooth and stable lane-following. The other two gains *Integral Gain* and *Derivative Gain*, were used when computing the final desired steering however, we did not attempt to experiment a lot with these gains seeing as we achieved steady lane-following using only proportional gain. In the final product we left these very close to 0. The integral gain would potentially be more useful on a more imperfect track. The derivative gain could have also been useful if the lane-following played a bigger part in overtaking maneuvers, but since most of the lane-switching was hardcoded in our solution, the conditions were already perfect enough when the control was handed back to the LaneFollower, that we did not have to consider the derivative gain. For further information about the *PID Control* refer to the Algorithmic References section.

Overtaking: The only connection we ended up having in our implementation between the overtaker and LaneFollower was whether the LaneFollower should use left- or right-lane handling. This was done by sending a value from the overtaker to the LaneFollower that simply notified the LaneFollower if the overtaker had switched lane. Another idea would have been to have the LaneFollower actively checking if it has passed a lane marking and using that as an indication whether the car is in the right or left lane. However, we ended up using the first method since we figured that the only point at which the car actually switches to the left lane is when it is overtaking, making the first solution seem more appropriate in our implementation. In our solution though, the overtaker completely takes over during the duration of the lane switches, making any other information except which lane we were in unnecessary for the LaneFollower.

Speed limitations: Even during pure lanefollowing with no other algorithm such as overtaking or parking running, the speed of the car in our current implementation is quite limited. Due to limited testing time and other components such as the parker and overtaker needing to be tested thoroughly, we did not budget our time well enough to properly optimize the LaneFollower at higher speeds. We ended up opting for a slower speed with a smoother ride, seeing as we needed to drive at a slower speed to properly test the other components. We found that one of the main problems with traveling at a higher speed could have been the time it took for the data to travel through the components. The time it took for the camera to capture the image, the Proxy forwarding it to the LaneFollower and the LaneFollower suggesting a steering angle, was too long for it to be accurate by the time it reached low-level board when driving at higher speeds. Looking back at it now, one simple solution that might have solved that for us could have been to simply move the point at which we do the detection a bit further ahead. It would require a bit of testing but we are quite certain that by applying this change, we could have found a happy medium for a higher speed.

2.4 Overtaking[MT]

The logic for overtaking maneuvers is implemented in the Overtaker class. The basic idea behind the overtaker is that it should constantly perform some *obstacle detection* operations while the car is controlled by the LaneFollower. Whenever an overtaking maneuver is initiated, the overtaker takes control of the vehicle to perform the necessary maneuvers and, once these are completed, it releases the control back to the LaneFollower.
The overtaker contains its own state-machine and uses sensors and wheel encoder values to handle transitions between states. The sensors that are relevant for the overtaker are the front-forward and front-right ultrasonic sensors as well as the front-right and the rear-right infrared sensors.
Figure 2.1 below displays the overtaker state-machine and indicates the operations that are executed in each state and the conditions that trigger transitions. Operations performed in each state are labeled as the functions implemented in the Overtaker.cpp module (for the sake of readability we decided to omit the function parameters). Orange boxes denote states where the overtaker controls the vehicle directly, while blue boxes indicate states where the overtaker activity is limited to understanding the surroundings and checking the conditions that trigger the transition to the next state. Below we provide a brief description of the operations performed in each state together with the conditions that trigger transitions from a state to another.

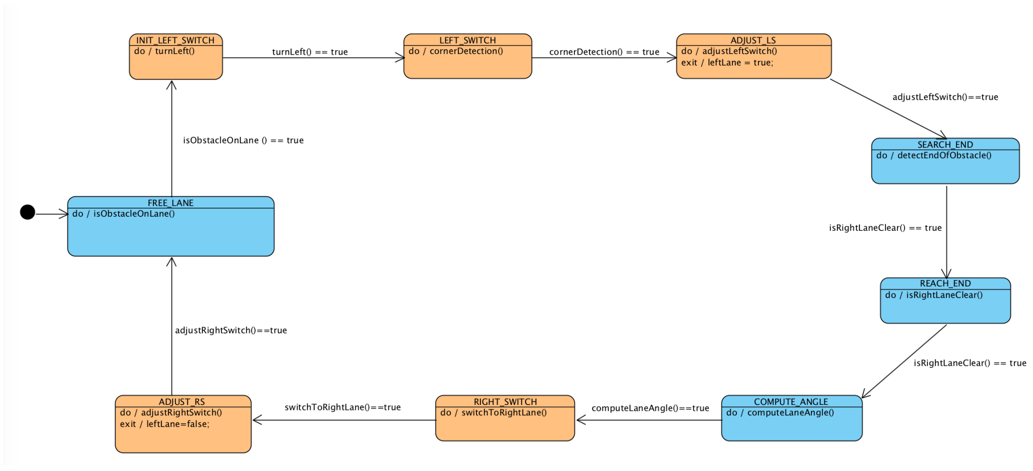


Figure 2.1: Overtaker FSM

Free-Lane: we use the front-center ultrasonic sensor to look for obstacles to overtake. If an obstacle is detected within a distance below a fixed threshold (7.0 in the simulator), the overtaker takes control of the car and transitions to *Init Left Switch*.

Init Left Switch: the vehicle steers left 30°and travels a path of fixed length (3.0 in the simulator) before moving on to the next state. The reason why we introduced this state is simply to move the car enough to make the front-right ultrasonic sensor point at the back of the obstacle that is being overtaken so that the corner detection algorithm can be performed correctly in the following state. We are aware that this state could lead to undesired behavior, especially if the length of the path is not estimated correctly. More in details, by using this algorithm we are "hoping" that the front-right sensor is pointing at the obstacle when the state machine transitions to the next state. A viable and probably more solid option would be that of using the gyro sensor to determine the target heading that will point the front-right ultrasonic sensor at the obstacle. Due to time constraints, we could not implement this logic before the final presentation, and therefore we opted for this quick fix instead.

Left Switch: the vehicle keeps steering left while using the front-right ultrasonic sensor to detect when the car has passed by the rear-left corner of the obstacle. Once the corner is detected, the FSM transitions to the next state.

Adjust Left Switch: this state is responsible for completing the left-switch maneuver by steering right approximately 12°and traveling for a fixed-length path (3.6 in the simulator). The rationale behind this state is to move the vehicle in a position that is likely to improve the camera's ability understand the lane and produce the desired steering control values in the next state. After the adjustment maneuver is completed, the overtaker sets the current lane to *left* and releases the control back to the LaneFollower.

Search End: at this point the vehicle is controlled by the LaneFollower instructions. This allows the vehicle to correctly follow the lane while overtaking on curves. Our implementation of the Lane Follower aims to keep the car closer to right side of the lane when traveling on the left lane. By doing so, we ensure the right side infrared sensors not to lose sight of the obstacle that is being overtaken, therefore allowing the overtaker to detect the end of the obstacle. As soon as the front-right ultrasonic sensor stops detecting the obstacle while the infrared sensors still detect it, the state-machine transitions to the next state.

Reach End: the overtaker uses the right side sensors to determine when the vehicle has reached the end of obstacle. When all three right side sensors (ultrasonic front-right and infrared front-right and rear-right) stop detecting the obstacle, the overtaker FSM moves on to the *Compute Angle* state.

Compute Angle: this state is responsible for detecting curves on the area of the track where the vehicle is going to perform the switch to the right lane. We do so by computing an average of the recommended steering angle from the LaneFollower over 5 frames and by comparing this value with some fixed thresholds (below -0.13 radians for a left turn, above 0.1 for a right turn). This estimation allows us to adjust the values of the desired steering angle and the length of the path to travel depending on whether the right switch maneuver will be performed on a left turn, right turn or on a straight lane.

Right Switch: during the right-switch the vehicle steers right and travels over a fixed-length path before transitioning to the next state. The values for steering angle and path length are set accordingly to the results from the computations performed in the previous state.

Adjust Right Switch: this state is responsible for adjusting the car on the lane before to return the vehicle controls to the LaneFollower. We do so by steering left approximately 30°and travel for a fixed-length path (2.0 in the simulator). Once the vehicle has traveled enough, the overtaker releases the control of the vehicle, updates information on the current lane and transitions to the initial state *Free-Lane*.

In order to avoid transitions to be triggered by noise in sensors values, we require the conditions that are based on sensor values to hold for 3 consecutive frames before transitioning to next state. However, it is worth saying that sensor readings used by the overtaker are already smoothed via moving averages as explained later in the Environment Detection section on.

Figure 2.2 displays the values returned by the sensors during an execution of an overtaking maneuver run in the simulator. The recording for this simulation are provided in file overtaking_noise.rec that was submitted together with this document. The horizontal black dotted line denotes the threshold used by the ultrasonic front-forward sensor to trigger the transition from *Free-Lane* to *Init Left-Switch*. Thus, the transitions described above take place in the area of the diagram to the right of the intersection between the black dotted line and the orange line. The init left-switch state moves the car in a position that allows the front-right ultrasonic sensor to "see" the obstacle (red line) so that corner detection can be performed in the left-switch state. The switch between left switch and adjust left-switch takes place immediately after the local minimum displayed by the red line, which represents the distance between the front-right us sensor and the rear-right corner of the obstacle. The following readings display the states adjust left-switch and search-end. The transition from search-end to reach-end is triggered as soon as the red line signals -1, while reach-end to compute-angle is triggered as soon as the rear-right infrared sensor (green line) reads -1. The final two states right switch and adjust right switch only use the wheel encoder and therefore cannot be described in terms of sensors values. The logic described above is based on some assumptions that work perfectly in the simulator environment but might generate wrong behavior if applied to a real-world scenario. For example, the *corner detection* algorithm requires some sort of regularity in the shape of the obstacle. Irregularities such as a non-flat rear surface are likely to lead to a wrong detection of the corner and finally have the car to initiate the adjust maneuver too early or too late. Another assumption is that obstacles should never be positioned in a way that makes them trespass the broken line that separates the two lanes. If we were to test the overtaker in a scenario that does not meet this requirement, we would probably see the vehicle collide with the obstacle during the Search-End or Reach-End state. This is due to the fact that the LaneFollower is controlling the car during those states, while the overtaker is using the sensors only to detect the end of the obstacle. Introducing adjusting maneuvers based on

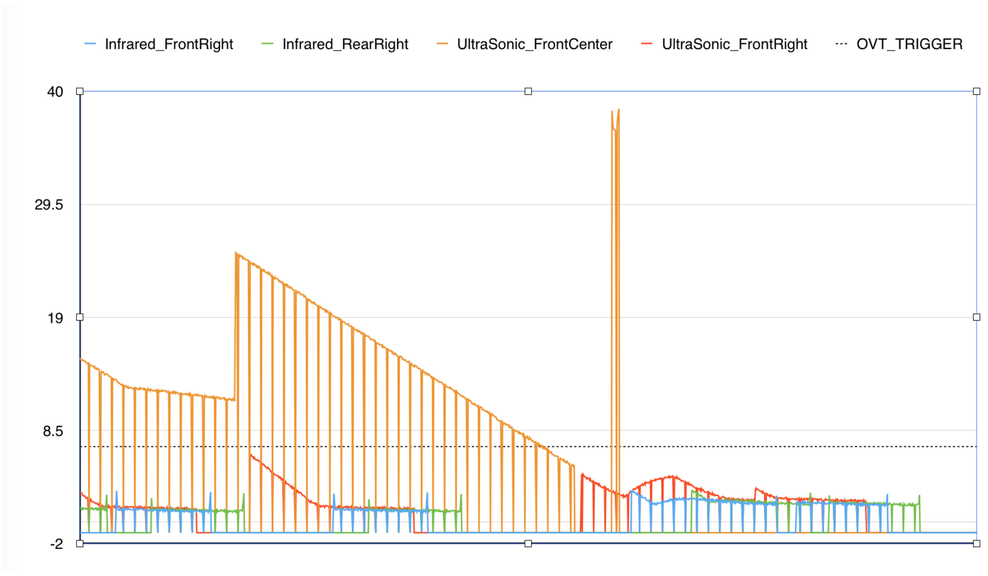


Figure 2.2: Overtaker FSM

sensor values would definitely allow to relax this assumption, but at the cost of compromising the estimations performed during the *Compute Angle* state.

2.5 Parking[NLC]

To realize a parallel parking scenario for an autonomous miniature vehicle there had to be a state machine. The implementation of the logic is inside the Parking class and the final state machine can be viewed below. It consists of five different identified states; searching, measuring, parking, parked and resume. The two states searching and parking both consists of sub-states. The Parking has public methods which the DM is using to know where in the parking it is, e.g. has found a parking spot, or if it is reversing or not (for the lights on the car). The reasoning about the parallel parking was that the car should always be able to park within a certain amount of distance between two objects. From that it should never fail the parking. Also to make it parallel inside the spot it has to travel roughly the same distance with each steering wheel angle to get it straight inside the parking spot. With this understanding the parking scenario was created as written below. To handle the noise data for both the simulation and from the real world there is a counter implemented that checks that there isn't any off data from the sensors making any state changes within the logic. This changed on some parts of the code later in the project due to always getting average values

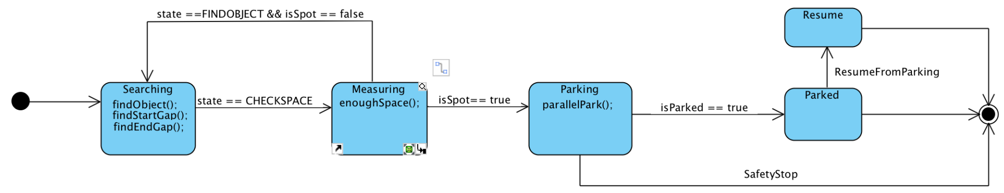


Figure 2.3: Parking FSM

from the STM, and the code was more focused on the real car instead of the simulation (which gave noise data on every sensor at the same time).

Searching: To be able to judge if there exists a place for the car to park it needs to make use of at least one or two sensors. This parking is using the rear right infrared sensor and the wheel encoder for the detection of a parking spot. The rear right infrared sensor was chosen because then the car will end up in a good start position with only a small adjustment. The wheel encoder plays a huge part for the parking scenario, with its help it is possible to measure lengths, e.g. the size of a gap. During the searching state the car should still be controlled by the LaneFollower for staying in a good position on the track and keep the car on the lane, and when a spot has been found the parking will take over in DM. To determine if an object are detected by the sensors is that the values from the sensors will decrease, and the opposite goes for not seeing anything (in OpenDaVINCI simulation seeing nothing is -1 from the sensors). With that understanding it is possible to detect an object and when it ends. From when a gap is detected (end of an object) the current traveled distance can be stored for measuring the size of the gap. The next step is to find if there is another object coming up ahead or that the distance from the back object is long enough for the car to park (only if the car hasn't been inside a curve where it in that case makes it go back to the first state). This can be done by the same logic as for detecting the first object and then stores the current traveled distance from where the gap ends. This felt like the most reliable way of detecting gaps between object.

Measuring: This state's purpose is to check if the gap is big enough to park in. The stored value for the end of the gap minus the stored value for the beginning of the gap will give the size of the gap. This size can then be used to compare if the size is bigger than a limit set for what the car can park inside. If it isn't big enough it transitions back to the searching state, otherwise it continues to the parking state.

Parking: Within this state the parking routine is executed. Every step of these sub-states returns vehicle control data for the DM to control the car. The parking state is making use of the ultrasonic front sensor, the infrared rear back sensor and the wheel encoder to execute the parallel parking maneuver. The ultrasonic sensor is relevant because there will be an object in front of the car and the car should not hit it, the same reasoning goes for the infrared sensor but for the back object. The wheel encoder is used to determine how much the car has traveled inside of the sub-states, so that it is possible to get the car straight inside the parking spot as mentioned above. The distance is used for the state switching inside the different states. The sub-states are: *phase0, phase1, phase2, phase3, phase4* and *safety stop.*

Phase0: This sub-state's goal is to take the car into a good position before starting the parking. This is important because to be able to get into the parking spot with a good angle while reversing it has to get into a good position to the front object.
Phase1: This sub-state's goal is to reverse around the back left corner of the front object. The car will turn the wheel angle 0.5 radians and reverse the set amount of distance. It will then get a good angle to get into the parking spot.
Phase2: This sub-state's goal is to reverse straight back into the spot to adjust the depth of the parking. This is mainly used inside the OpenDaVINCI simulation due to that the car is a further out and to get it into the middle of the spot in a nice position it has to back straight for a bit. It was not needed on the real car because the objects were next to the road and it got into a nice position with only *phase1* and *phase3*.
Phase3: This sub-state's goal is to get parallel inside the parking spot. It starts turning the wheel angle to -0.5 radians while it is reversing. As stated above the car is using the wheel encoder for the distance to know when it has straighten up inside the parking spot. It is in this state it's making the most use of the ultrasonic front sensor and the infrared rear back sensor. It is using them to not hit any of the objects it is parking in between and also be able to park within smaller spaces. When it gets too close to the back object it is switching to go forward instead with the wheel angle 0.5 radians. If it hasn't finished the distance yet and gets too close to the front object it's starts reversing again with -0.5 radians. All this is repeated until the car has traveled the set distance to get parallel. One problem we discovered within this state was that for the real car there is a delay with the sensor readings.

It reacted on the different triggers too late so that the car collided with the objects. To fix this we increased the threshold to be longer and it had more time to process the information, that fixed the issue but could have made it so that it can't get into a certain size of gaps.
Phase4: This sub-state's goal is to get into the middle of the parking spot with good distance from the back and front objects. It is still depending on the front ultrasonic sensor and the infrared rear sensor and here the wheel encoder isn't used anymore for the parking. It has only the wheel angle 0 radians set. This state has two different paths to go depending on if there was an object in the front or not, but they are similar in their functionality. They are checking which of the sensor readings is the bigger one and then adjust the position based on that. To know if it is inside the middle of the parking spot there is a comparison between the sensors and an acceptance range that the difference should be less than.
Safety stop: This sub-state's goal is to stop the car if there is an object to close to the car. This is a safety mechanism that should make the car not to drive into objects when objects are too close to the car. It is not active when the car itself is inside the parking spot because it is already using the sensors to switch directions.

Parked: This state is when the car is done with the parking. The parking routine is completed and won't do anything anymore. This state is connected to the DM via a public function that returns a boolean for if the car is parked. The state has traveled into a do nothing state within the DM and it's waiting for a new input (resume) or to turn it off.

Resume: This state is for taking the car out from the parking spot and then continue to the lane following. This was only an extra feature to make the car be more as in the real world where it can continue to drive even if it has been parked.

We don't know how the realization of the parallel parking could have been done in a different way that we made it. The only thing that comes to mind is to have used the Gyro instead of the distances inside the spot in phase3 to make it to be parallel inside the parking spot. The Gyro wasn't connected because we felt that using the distances was good enough to use for completing the task.

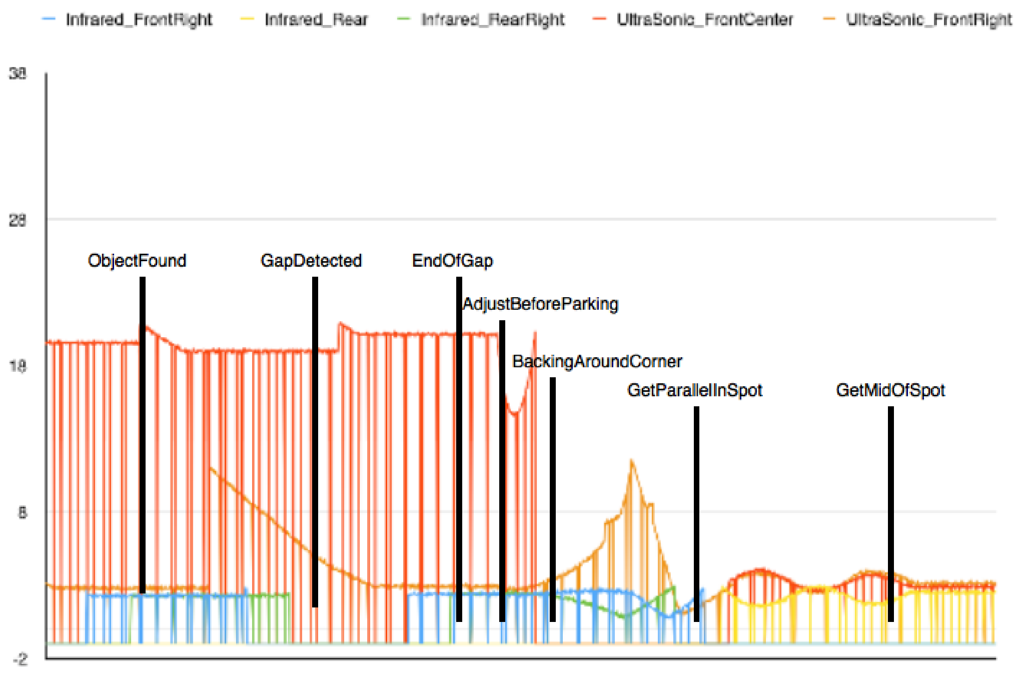


Figure 2.4: Every stage in the parking can be seen in the sensor value diagram.

Chapter 3

Algorithmic Fundamentals

3.1 Environment Detection

3.1.1 Image Processing[ME]

The first step towards a desired steering angle is processing the image. The ideal result of the image processing would be an image with perfectly applied edge detection, where all the edges are one colour and the rest is another. We decided to turn all edges white and everything else black.
We started out by using OpenCV to convert our colorspace image into grayscale. By using the *threshold* function from OpenCV it was easy to filter out a certain spectrum of brightness in the image and drawing contours around the remaining edges was done with a mixture of *findContours* and *drawContours*.
This resulted in an image that was only black with white contours around objects, but due to how the *threshold* function was built it was rather hard to achieve a consistent good quality image because of the different lightning conditions in the room. To counter this problem we tried to create an adaptive threshold using the light sensor on the front facing ultrasonic sensor, this did increase the quality of the image. But while testing this we stumbled upon OpenCV's *Canny* function.

Canny is a function specifically made for edge detection, it uses the *Canny86* algorithm (See Algorithmic References for more information about *Canny*). Using this we get an acceptable image to work with at almost all times. We noticed that the algorithm itself might be a bit resource heavy but not so heavy that it would pose as a problem, especially not when we only analyze half the image as explained in section 4: Implementation Details.

3.1.2 Lane Detection[DK]

When the image is processed with clear edges it is time to apply lane detection logic. Every iteration we loop through pixels horizontally and vertically, looking for white or light-grey pixels. For the left and right lane markings, the pixel we start looking from is vertically a pixel very close to the bottom of the picture, we refer to it as our control scanline. Horizontally, we start from the middle of the picture. We then loop through the pixels to the left and right of the starting point, looking for white pixels. The number of pixels looped through before finding the desired pixel is then stored and used as a distance measurement when applying the lane following.

The stop-line detection works the same way as the detection of the left and right lane markings. When looking for stop-lines however, we have two separate starting points, they are offset +/- 50 pixels to the left and right of the starting point we used in the lane marking detection. We then loop through pixels on the vertically instead of horizontally. The reason we check the stopline at two separate locations is to simply increase robustness, more specifically in this case these two distances are compared to check whether the line detected is more or less horizontal. Additional robustness in this detection is achieved by checking this for a few iterations before actually treating it as a stop-line. If these robustness checks are passed, we forward the distance to the stop-line to the DecisionMaker, in which we adjust the speed appropriately. Before this is done however, we ran a few tests and concluded that there was no point in sending any stop-line distances if the line was to far away. Infact, we ended up only sending the distance if it was low enough that the DecisionMaker would consider stopping. This was not the initial idea, at first we wanted to send farther distances and have a certain distance at which the car slows down up until it reaches the threshold of stopping. However, seeing as we had some issues getting the car to run at higher speeds with our current implementation, we felt that there was no purpose in slowing down due to the car's speed already being low enough to simply stop. The car also stopped very smoothly when setting the speed to 0.

Furthermore, when no lane markings are found on either side during the lane-detection, a flag we call quality is set to false. What this means is that the data from the LaneFollower is not to be trusted when it comes to decision making. The intention was that this value were to be handled in the DecisionMaker, potentially lowering the speed and perhaps even setting the steering wheel angle to 0. We unfortunately did not have time to handle this in the DecisionMaker although it was handled in the Lane Follower.

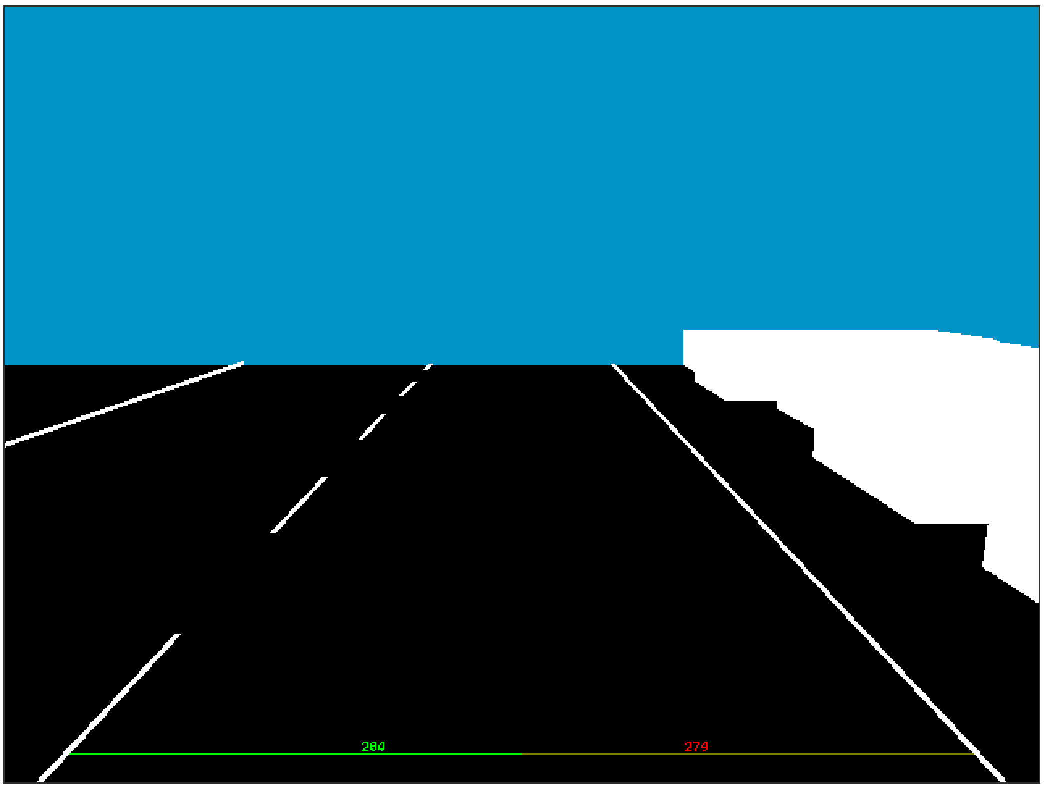


Figure 3.1: LaneFollower in the simulation

3.1.3 Ultrasonic Sensors[JE]

The ultrasonic sensors use the IC protocol for communication. As discussed further in section 6: Hardware-Software Integration, this link seems sensitive to fluctuations in voltage, and although we reduced this sensitivity by mounting external pull-up resistors connected to the I^2C bus, the sensors can still respond with unexpected 0's occasionally instead of the current distance measured. Therefore, we discard any readings of 0 until it occurs at least three consecutive times.

All values are averaged with a circular buffer, currently having the size of 4 elements. These mechanisms may reduce reaction time slightly for sudden obstacles, but we considered the reduction in noise to be worth this, taking into account which challenges the miniature car was expected to handle.

Values are capped to 90 cm (anything above this will still be sent as 90), because values above this will likely be unusable, noisy fluctuations not directly interesting to the current state of the car.

The light sensor of the front ultrasonic sensor is also read, and used to determine the strength of the headlights. Earlier we discussed the possibility of adjusting the camera image processing threshold values based on the light sensor reading, which is why the light sensor reading is also transmitted to the high-level board.

3.1.4 Infrared Sensors[JE]

The infrared sensors are read using the analog-to-digital converters of the STM32. We based our formula for translating sensor voltage levels to distance on a formula designed for Arduinos, which have a lower ADC resolution. To compensate for the difference in resolution, we simply bitshift the values.

To minimize noise, each time we read the sensor values we let the ADC read 4 samples for each sensor. Then we translate the average sampled voltage level for each sensor into a distance value, which is in turn averaged with the previous distance value.

Values are capped to 28 cm, because any values above this will be unusable, noisy fluctuations.

3.1.5 Wheel Encoder[TDJ]

The wheel encoder works by measuring wheel revolutions using infrared reflections. In a proper setup, it should be possible to not only measure distance traveled, but speed as well. However, fitting the wheel encoder properly was hard and the results were at times unpredictable.

The circumference of the wheel measured was 20.42 cm. This wheel had nine stripes on its interiors, which reflected the infrared light back to the sensor. After a meter of travel, the sensor should have measured 44 stripes. These measurements could be used to calculate both speed and distance traveled using the amount of stripes per meter.

Initially, the wheel encoder ran within its own thread. The rate of reliable results was not satisfactory with this setup, so it was setup to rely on interrupt routines which were triggered when a new value was read. This increased the reliability of results quite significantly, but was still insufficient at operating speeds with an inaccuracy of 10 - 15%

The particular wheel encoder used is clearly specified by the manufacturer as only intended for a specific wheel (different from the wheels used in this project). If we would have had more time, we would consider using for example a computer mouse to measure distance travelled.

3.2 Vehicle Control[JE]

3.2.1 Safety Mechanism

If no valid control data is received through the serial connection for a given time, the wheels are centered and the engine is stopped. This constitutes a basic safety mechanism for a broken connection or execution problems on the high-level board.

3.3 Steering Control

The steering wheel angle is forwarded agnostically from the high-level board to the servo. In the OpenDaVinci components, radians are used. In the proxy, these are converted to degrees, centered around 90 degrees to make sure we can use unsigned bytes for transmitting. Hence 90 degrees means forward, 60 means full left turn and so on. We roughly see 30 degrees or about 0.5 radians as the maximum turning angle in each direction.

3.3.1 Engine Control

For controlling the engine power, we use a set of prefixed pulsewidth values. The high-level board sends an integer number, where 0 means reverse, 1 means neutral, 2 means forward slowly, and 3 means cruise speed forward. Speeds above 1.0 in the OpenDaVINCI-based components are seen as cruise speed. The conversion occurs in the proxy component. The solution of indexed speed modes has safety benefits, and indeed we did not have the problems of unexpected engine power surges that several other groups did.

We do not use any speed regulator mechanism to adjust the engine power based on how the current speed relates to the desired speed, and therefore had to manually adjust the PWM values to use the car with low battery power. We deemed the speed measurement from the wheel encoder to be too unreliable to use as a basis for adjusting the engine power. If we would have had the time to explore more reliable ways to measure speed or distance, a speed regulator could have been helpful.

3.3.2 Light Control

Using the neopixel LED strips, we gave the car headlights, tail lights, brake lights, indicator lights, reverse light and an RC mode light (blue). The reverse light, indicator lights (flashing orange) and brake lights are controlled by the components running on the high-level board. The RC mode light is automatic. The tail lights are always lit, and the headlights have a strength that adapts to the surrounding light levels (as measured by the light sensor in the front ultrasonic sensor), which can help reduce the cameras exposure time in dark environments.

Chapter 4

Implementation Details

4.1 Low Level Board[JE]

4.1.1 Neopixel Software Driver

We wrote our own software-based bitbanging driver to interact with the neopixel LED strips. When interacting with the neopixels, it pauses non-critical ChibiOS interrupts momentarily while alternating the (high/low) digital state of the output pin according to a specific timing protocol, where a certain timing of the high and low states means 0, and another means 1. This way, a sequence of bits are sent to the light strip.

Each neopixel reads 24 bits (8 for each three colors), and then forwards any excessive bits to the next pixel. Hence, to control 16 neopixels, we need to send 24 * 16 bits. When all bits are sent, a pause of a certain time lets the neopixel strip know we are finished writing, and that the corresponding LEDs should be turned on with the desired brightness.

4.1.2 Packet Parsing and Serial Connection Details

Received bytes from the serial connection are put into a buffer. When the buffer is large enough, packet parsing is attempted. The algorithm looks from the end (most recent) of the buffer and steps backward until it finds the end delimiter of a packet. When it does, it verifies that the start delimiter and size header are also present (following the netstring standard), and then reads the values transmitted in the packet body. The last byte in the packet body is a checksum byte, which is the result of computing the XOR value of all other value bytes. If the checksum byte indicates the packet is valid, the buffer is cleared, because any older packets will not be of interest. The green LED on the STM32 board is lit up to indicate valid communication.

Whenever we tried to parse the buffer but didn't find a valid packet, the orange LED is lit on the STM32 board. This can occur either because the checksum did not match the contents, or because several bytes (enough to expect a packet) were received that did not constitute a packet.

If no valid packets are received for a while, the red LED is lit. To make sure the output buffer is not filled, which would cause the thread to block while writing and could happen if the USB connection was disconnected, we make sure not to send anything in this state. However, since the serial connection is running in a dedicated thread, any blocking state would not in itself cause safety problems.

4.2 High Level Board

4.2.1 Proxy[ME & JK & IV]

The AutoTuxProxy component extends OpenDaVINCI time-triggered module and runs in the frequency provided by the user with the flag "–freq=". It consists of four separate namespaces, each responsible for a different task - camera, proxy, serial and containerfactory.

[ME] As stated above, within this component we have the handling of the camera. In this case a regular PC web-camera. To capture images we're using a third party library - OpenCV (Open Source Computer Vision). This is a library aimed towards real-time computer vision. Using this library gives us a nice and intuitive way to communicate with the physical camera attached to the Odroid.
Running this module creates a SharedImage (OpenDaVINCI object) from defined variables in the OpenDaVINCI configuration file. We then use the OpenCV library to open (connect) to the physical camera connected to the Odroid board, and fetch (grab), then read (retrieve, decode) the grabbed image into the SharedImage memory location.
In the earlier stages of the development we stored the whole image, which we later on changed to only copy the bottom half of the grabbed image, due to the fact that we're only looking at that part when using the image in the LaneFollowing module. By doing this we essentially cut the memory needed by this module in half, if we're lucky we also saved a couple of nanoseconds when copying the image to the memory location.
How ever we did have problems with the slice consumption within this module, this was mainly because of the different lighting conditions at the test track. As a darker room would mean that the camera needed a longer exposure time this would also increase our slice consumption time to unacceptable values, this did not pose as a problem when the room was better lit up.
To counteract this problem we tried to use OpenCV to set a reasonable static exposure time that would work ok for a more wider area of lightning conditions. Although this was not as easy as it would seem as OpenCV v.2 is using *Video4Linux* API version 1 and the camera properties we needed to access is not available in versions below API v.2. Hence this is still not completely

solved.

[JK] Each proxy iteration (as soon as a time slice is free), the time-triggered module reads the newest valid packet received via USB (wrapped in a vector), rechecks for its correct size, and sends out the data to the session. The data the proxy sends out, is of type SensorBoardData and VehicleData. To generate the containers we send out, two factories are used (part of the containerfactory namespace) which return a shared pointer (to automatically get rid of these objects after they aren't used anymore).
The SBDContainer factory basically takes our internal order and puts the data from the sensors in the order OpenDaVINCI is using in its simulation. The VDContainer factory is slightly more complex in it's behavior: after setting the speed it takes all four bytes we receive from the USB for calculating the absolute travelled path and bit shifts them together to one unsigned integer value.
Note that all values for distances and speed are getting converted from centimeter based values to meter based ones (as it is used in the high-level code). The last step is to receive the most recent containers for VehicleControl and LightSystem data from the session. The data gets processed (steering wheel angle from radians to degrees, speed to four different fixed values and the light settings bit shifted into a four bit sized unsigned char) and it's checksum is generated. Finally a vector out of these values is created which replaces the buffer wrappers send buffer which will be sent out back to the STM32 via USB.

[IV] The serial module is responsible for the serial connection between the low-level board - the STM32 - and the high-level board - the Odroid. It reads from the usb sensor-board data, collected from the sensors, and writes to it control data, evaluated by the DecisionMaker component. The communication between the two is managed by a third-party library - libusb-1.0. Libusb is a lightweight library that is portable and easy to use. It supports all transfer types - control, bulk, interrupt and isochronous - and two types of transfer interfaces - synchronous and asynchronous. Furthermore, the library is thread-safe, it does not manage any threads by its own. The class implementing the calls for managing the usb connection is in SerialIOImpl.cpp. When an object of that class is created, a libusb_context is initialized by making a call to libusb_init() function and passing it a reference to the context struct. It is possible to call this function with the argument NULL , in which case a default context is created for the user. The problem that can occur by doing so is that - if there are multiple components in the application using libusb to communicate with any of the usb devices - the settings for the context might not be applicable, e.g. a part of the program needs to talk to the STM32 board but another can be talking to some other device connected to the system. Although, our program does not have that particular problem, we decided to bind a context to the device we use as a good practice. After the context is initialized, the list of available devices is obtained, and the device we are interested in is opened. To match the correct usb port, we use the STM32 vendor and product id which can be obtained by running lsusb in the terminal on a Linux-based machine. Once the device has been opened, we check if the interface we want to operate on is free, i.e. the kernel is not using it, and claim it if that is not the case. By claiming the interface of the device, we can exchange data through the usb serial connection.

The reading from and writing to the usb are performed using the synchronous API interface. The function to use to do the operation is libusb_bulk_transfer() which takes a number of parameters. In particular, the context to operate on, the usb endpoint, i.e. if it is read or write, indicated by the direction bits in the endpoint address, the buffer where to store the data, the length of the buffer, an int for the actual bytes transferred, and the timeout for the function call. The above mentioned function is blocking but since we do not have multiple components using the usb and there is a dedicated thread operating with it, we decided that is sufficient to use this API. The timeout for the call is set to 10 milliseconds for both the read and write. Libusb is also smart in a way that it does not wait for the entire buffer to be filled. In a case that less data is received than the size of the buffer, the read operation is terminated and the result is returned.
The buffer size for reading is set to size 128. We considered it to be enough with respect to the frequency we are running and the size of our data packets. For writing to the usb, the length of the buffer is equivalent to the size of the data that needs to be sent.

Parsing of the received packets is done when data is appended to the SerialBuffer object with a call to appendReceiveBuffer(), which takes a vector with unsigned chars as elements. The function traverses through all elements, starting from the back (most recent packet) and looks for the start and end delimiters of a correct packet. When one is detected, the values for the different sensors are obtained along with the checksum for this packet. The checksum is then checked against its validity and if so, the packet is considered valid and put in the internal buffer.

4.2.2 Configuration Tool[JK]

The main goal of this component was to have a tool at hand which can be used to tweak values, like the lane-follower gains or other camera setup values, on the run, without changing values in the source code or header files. This would've required recompilation every time and would therefore have been very time consuming.
An application with a GUI seemed optimal. Because we are running everything through a SSH session via the console, we decided to use GNU's ncurses library, which allowed us to create a Text-User-Interface (TUI) independent of the terminal emulator in use.

From the very beginning, we used the effc++ warnings. A lot of "Class contains pointer data members" warnings arose. Research on the internet showed a possible solution: wrapping these pointers into unique_ptrs. Unfortunately these wrapper pointer data types do not interwork with the ncurses library functions, so we ended up in removing the warnings.

To interact with the car's OpenDaVINCI session, a module was written which extends the *TimeTriggeredConferenceClientModule* (TTCCM). Depending on whatever frequency is set by running the application, the ATConfigurator will send out the setup values, as well as the selected state. To avoid lagging, a frequency of 2 Hertz is recommended.
To access this session data, a Singleton class, containing of different data members, as well as their getters and setters, was used. The TUI as well as the TTCCM is accessing this class.

To get some guidance for setting up the webcam to the correct angle, the SharedImage is fetched from the memory and transformed into an *ASCII picture*. Initially it was planned to use libcaca for doing that - a library, which even supports to display these images in color. While trying to integrate libcaca into the ATConfigurator we noticed that it spawned a separate window, instead of being in the same window as the configurator. This was fixed by using the libcaca ncurses driver, which unfortunately overwrote also our main ui window.
We ended up using an external tool to generate the picture and load it into the application. After unsuccessfully trying out img2txt (failed because of its usage of ANSI and UTF-8 chars), we switched over to jp2a, which generates a pure black and white ASCII representation of the picture.
While using this solution some small lags can be noticed: the image file is (over)written to whatever the frequency is set. Therefore it can happen that

the image is empty or corrupt, while jp2a tries to read it. We tried to avoid this by using a small buffer/tmpimage within the code, but some lags can be noticed still. Further improvements seemed not reasonable due to the late stage of the project phase.

One problem we faced while developing this application was navigating through it with arrow, return and back keys. The problem is that the values aren't as they are programmed in ncurses. Custom values were defined in the header file.

Chapter 5

Software Architecture

5.1 Low-level Board Software Architecture[JE]

The STM32 software consists of a main thread that reads sensor input values from hardware components and forwards control data to the hardware 13.8 times per second, and a serial thread that sends the sensor values to and reads control data from the Odroid via the USB connection 20 times per second. The interaction with hardware is explained further in section 6: Hardware-Software Integration.

The hardware modules all have similar function names, to simplify their use. All function names are prefixed with the source filename. We have taken care to encapsulate the internal behaviour of all modules similar to the way common in object oriented programming, so values are retrieved via *getter* functions. In our opinion, we managed to achieve a simple yet robust and easily extensible architecture with modular aspects without wasting time overgeneralising or over designing.

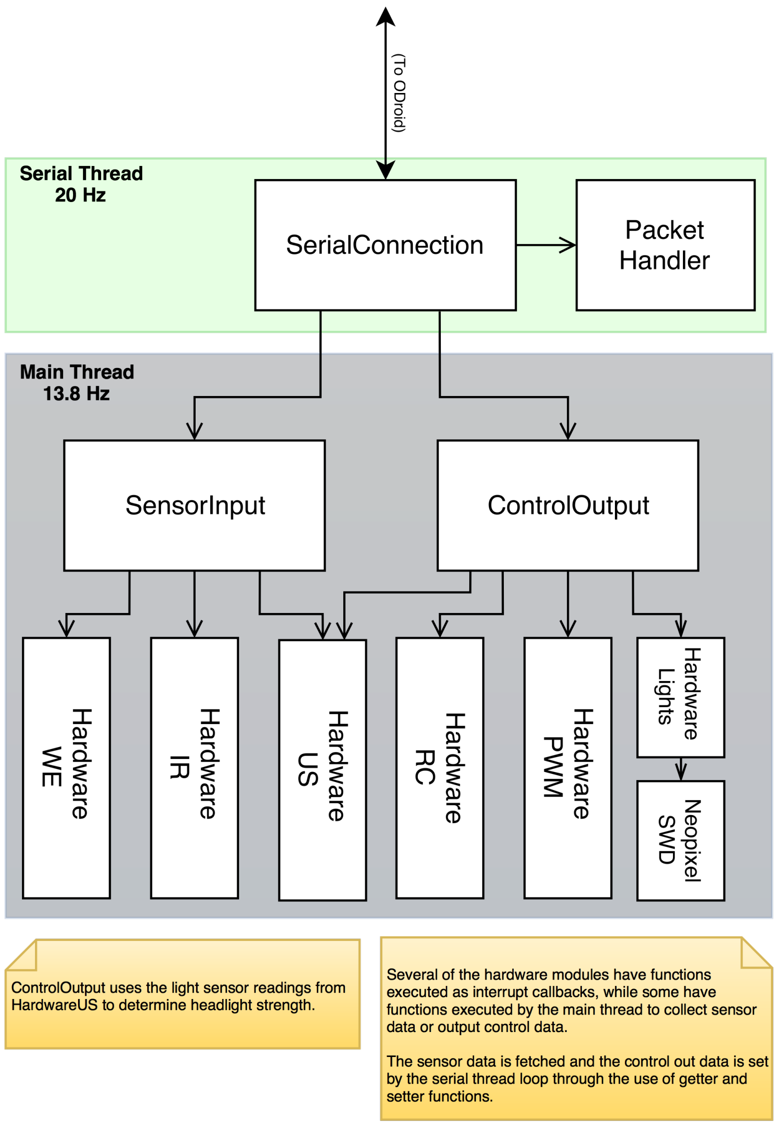

(To ODroid)
Serial Thread
20 Hz
SerialConnection
Packet Handler
Main Thread
13.8 Hz
SensorInput
ControlOutput
Hardware WE
Hardware IR
Hardware US
Hardware RC
Hardware PWM
Hardware Lights
Neopixel SWD
ControlOutput uses the light sensor readings from HardwareUS to determine headlight strength.
Several of the hardware modules have functions executed as interrupt callbacks, while some have functions executed by the main thread to collect sensor data or output control data.
The sensor data is fetched and the control out data is set by the serial thread loop through the use of getter and setter functions.

Figure 5.1: STM32 Architecture

5.2 High-level Board Software Architecture

5.2.1 Overall Architecture[JK & DK]

Initially we decided on running three different components: a proxy, a lane follower and a decision maker component.
The proxy is responsible for exchanging the data with the low-level board and putting the webcam image into the shared memory, the lane follower is interpreting the webcam image and recommends steering angles and warns about stoplines while the decision maker component finally takes and interprets this data and decides on final steering wheel angle and speed. The decision maker component also includes the logic for overtaking and parking.
We ended also up in having a fourth component running on the Odroid board: the configuration tool. This tools purpose is to configure the lane-follower, changing the main state, as well as looking at current sensor values.

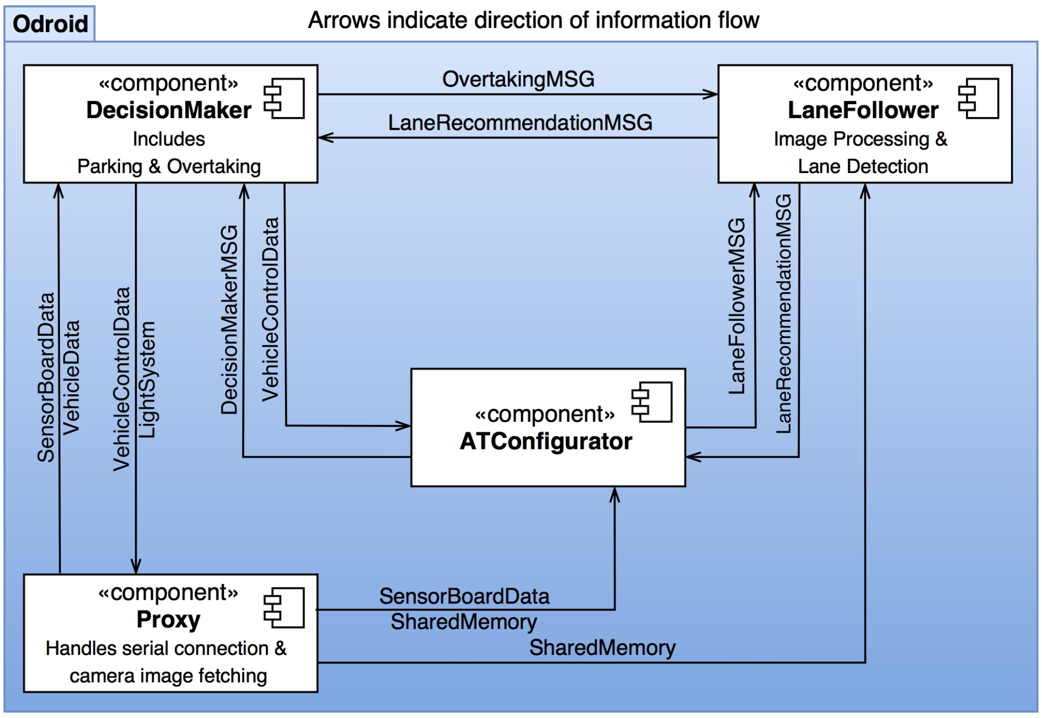


Figure 5.2: Odroid Architecture

On the high-level board the data exchange works through the OD conference. All components use the conference to share data between one another. When we started out, the communication between the DecisionMaker and the LaneFollower was one-directional. The LaneFollower simply used the shared image received from the proxy component and and sent steering recommendations to the DecisionMaker. However, due to our solution of the overtaking, we ended up having messages sent both ways between the DecisionMaker and the LaneFollower.
The ATConfigurator component both sends and receives data from the DecisionMaker and LaneFollower. As mentioned above, this component was used as a tool to change and monitor data on the fly.
The data-flow between the Proxy and the DecisionMaker component is also two-directional, the DecisionMaker retrieves sensor data from the Proxy and sends vehicle control data back to the Proxy. Even though the data circulates a lot to and from all components, it does not mean that they are dependent on one another. The LaneFollower is only dependent on the Proxy component, as it can not work without receiving an image. The DecisionMaker does not depend on any of the other components. The same goes for the Proxy component.

5.2.2 Communication Packets[IV]

We used Netstring for the structure of our data packets. This allowed us to keep our packets small and compact. The sensor-board data packet has the following outline:

12 : us1 us2 ir1 ir2 ir3 speed dis1 dis2 dis3 dis4 light checksum ,

In the above, *12* is the number of elements in the packet, *:* is the start delimiter and *,* is the end delimiter. The length header and the delimiters are written in ascii characters. Each element in the packet body is an unsigned char. The values for the different sensors, i.e. ultrasonic and infrared, are holding integer values 0-255. The speed is in centimeters per second. The dis1-4 represent the distance travelled as a four-byte integer. Light represents the light sensor reading of the surrounding brightness.

The vehicle-control data packet is as follows:

4 : speed angle lights checksum ,

The outline of the packet follows the same convention as the sensor-board data. For the speed, we used an enum for the different types, i.e. 0 is backward, 1 is stopped, 2 is forward and 3 is cruise speed. The angle values are converted from radians to degrees (centered around 90 degrees instead of 0 to stay in the unsigned range). The lights are controlled by setting the bits for the different states - bit 0 (rightmost bit) for brake, bit 1 for reverse, bit 2 for flashing left and bit 3 for flashing right.

5.2.3 Configuration Tool[JK]

The configuration tool contains of two separate threads running:
The first thread is the main thread of the application. The *TimeTriggeredConferenceClientModule* provided by OpenDaVINCI runs in that very thread.
The second thread is the TUIs main loop which is basically a while loop running at roughly 40Hz. This loop refreshes all ncurses windows in a rate which feels instant to the user as well as reading the input commands from the keyboard (which is non blocking because of the usage of ncurses no-delay mode.

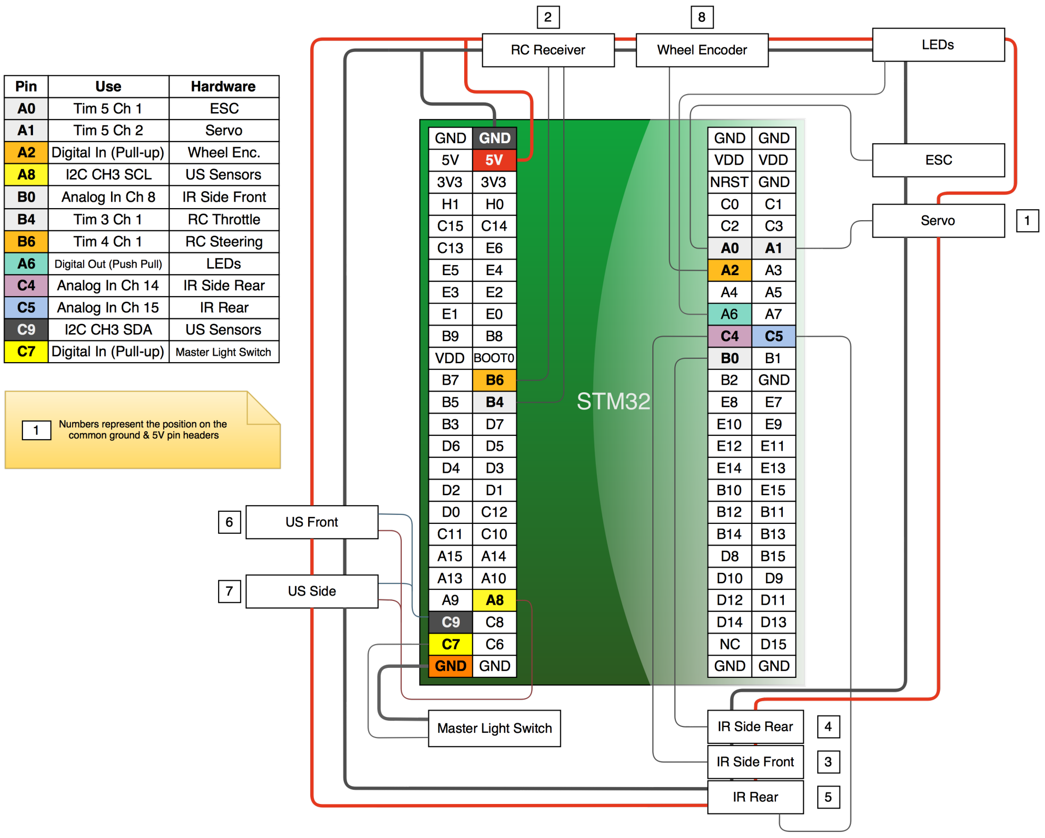
As mentioned in section 4: Implementation Details, a Singleton class is used to exchange data between the *TimeTriggeredConferenceClientModule* and the TUI. The decision for a Singleton was made to avoid passing around an object to every single of the using classes.
It is certainly not an optimal solution, but in our opinion "good enough".

Because *ncurses* is a library written in C, some decisions were made to make it fit more into our object oriented code written in C++:
Each ncurses window got wrapped into a separate object, having member functions necessary for effecting this window. Eg: each ncurses window object draws itself with the help of the *void refresh(void);* member function.

Chapter 6

Hardware and Software Integration

6.1 Low-level Board[JE & TDJ]



Pin	Use	Hardware
A0	Tim 5 Ch 1	ESC
A1	Tim 5 Ch 2	Servo
A2	Digital In (Pull-up)	Wheel Enc.
A8	I2C CH3 SCL	US Sensors
B0	Analog In Ch 8	IR Side Front
B4	Tim 3 Ch 1	RC Throttle
B6	Tim 4 Ch 1	RC Steering
A6	Digital Out (Push Pull)	LEDs
C4	Analog In Ch 14	IR Side Rear
C5	Analog In Ch 15	IR Rear
C9	I2C CH3 SDA	US Sensors
C7	Digital In (Pull-up)	Master Light Switch

Figure 6.1: Wiring Schematic (Block diagram)

[JE] It took a while for us to arrive at this wiring solution. Our decisions were limited by which STM32 pins support which features, and which features and pins can be used simultaneously. For example, one thing we learned along the way was that PWM output is most easily set up with the so called "general purpose" timers, numbered 2 to 5, on the STM32. These timers are connected to a certain set of pins. Another example of differences between pins is that some tolerate only 3.3V instead of 5V. We had to refer to the user manual of the STM32 continuously when deciding where to connect each new hardware component type.

The master light switch is an on/off switch that when turned off will cause the light controlling module to turn all lights off. This is helpful to save battery power when lights are not needed.

For most hardware interaction we use the drivers provided by ChibiOS:

- The PWM (Pulse Width Modulation) driver for generating PWM output to control the ESC and servo
- The ICU (Input Capture Unit) driver for measurement of incoming RC PWM signal
- The I^{2}C (Inter-Integrated Circuit) driver for interacting with the ultrasonic sensors
- The ADC (Analog-to-Digital Converter) driver for reading the analog values from the infrared sensors
- The EXT (External) driver to execute a callback when the wheel encoder signal goes (digital) high.

However, since the neopixel light strips doesnt use a very common communication protocol for their digital signal, there are no drivers for it provided by ChibiOS. We evaluated third-party drivers but found only one somewhat viable option, that happened to require two of the STM32s timers. Eventually we decided to write our own software based bit-banging driver instead, explained further in section 4: Implementation Details.

As discussed in the section 5: Software Architecture, some of the hardware modules for sensor input (wheel encoder, RC input) have interrupt callback functions that automatically gets executed whenever values change. Others (ultrasonic and infrared) need to initiate the measurement on the main thread, running at a frequency of 13.8 Hz. For the infrared ADC measurement, a callback is then executed automatically when the conversion is finished.

The frequency of 13.8 Hz is somewhat limited by our ultrasonic sensors, that need at least 65 ms to complete their ranging. This means a call is first made to start ranging, and a while later a call is made to fetch the measured values. It may have been possible to decrease the time needed by adjusting the range settings of the ultrasonic sensors, which could have been interesting to explore if the project would continue for a longer time.

Most hardware and software integration worked well and was straightforward. The most problematic hardware components were the ultrasonic sensors, where the I^2C communication seems sensitive to fluctuations in voltage, which can occur for example when the steering servo is used. We reduced this sensitivity by mounting external pull-up resistors connected to the I^2C bus.

Another problem was interference between the two sensors: the signal of one sensor would be perceived by the other sensors as a reflected sound, and therefore lower the measured distances of both sensors compared to if they were running alone. We handled this problem by reducing the gain of both sensors, making them send out weaker signals. This problem may not have had the same impact if the I^2C driver on ChibiOS supported broadcast mode, i.e. the possibility of sending a request to both sensors at the exact same to start their ranging process.

[TDJ] We also ran into issues with faulty hardware, most notably the front-right ultrasonic sensor and the side-rear infrared sensor. While troubleshooting the ultrasonic sensor, we noted that it seemingly worked as intended while being run on an Arduino microcontroller as opposed to the STM32.

The ultrasonic sensor was replaced with a sensor from another group which had an Arduino low-level board. The replacement sensor worked as intended, but they returned our original sensor as it became unstable after a short period of use on the Arduino. The faulty sensor was then returned to Federico and a new one was installed which worked as intended.

The faulty side-rear IR sensor received wildly fluctuating values and tiny adjustments to its wiring would cause further disruption, and would at times cause the STM32 to power off. We noted that this was still the case after replacing its wiring, leading us to believe that this was either a soldering issue or a problem with the sensor itself. Slight adjustments to the sensor could cause it to function normally, but these adjustments were temporary at best and the smallest interruptions to it would revert the sensor to its previous broken state.

A replacement sensor was acquired, and the original wiring was kept in place. The new sensor gave consistent values and did not cause any interruptions to the STM32.

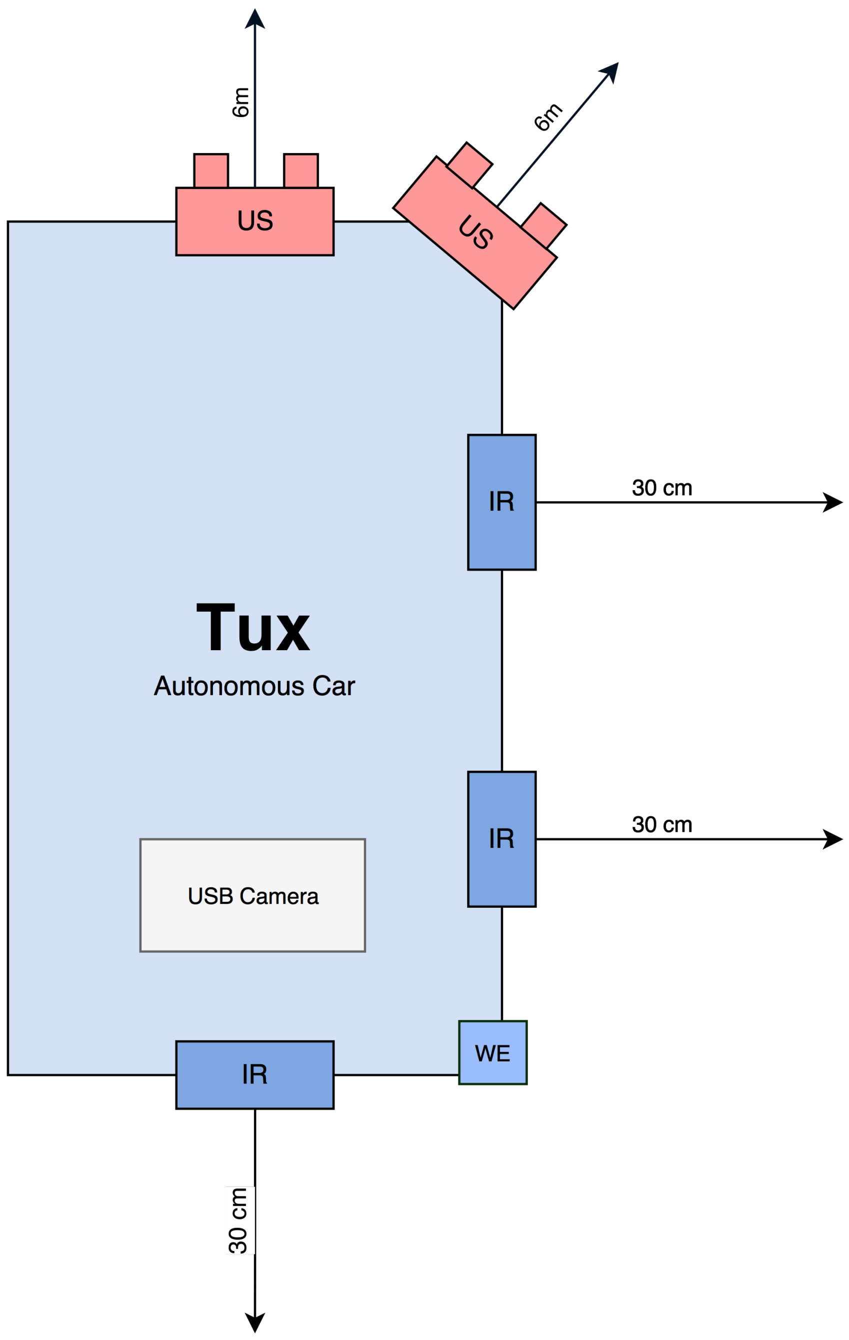


Figure 6.2: Sensor layout

Chapter 7

Test Results

7.1 Hardware Tests & Sensor Calibration[TDJ]

Early in the project, we evaluated the sensor readings and found one faulty IR sensor and one faulty US sensor as discussed previously. Once we got a new IR sensor, all IR sensors seemed to give reasonable values. The values given after applying the conversion formula (returning the distance based on the voltage) were not perfect, but had an acceptable accuracy within around +-10%.

When evaluating the averaging mechanisms applied to the US sensor values, we realized they had interference problems. We would test this by pointing the car up into the air so nothing was in front of the sensors (at least not within several meters), and still they would give values such as 70 centimeters. We studied the data sheets and lowered the sensor gain to eliminate this problem.

After confirming that distance measuring sensors were working as well as expected, we moved onto the wheel encoder. All wheel encoder test runs were done on a 1 meter test track at three varying speeds: slow, operational speed and brisk. Propulsion method varied, and was done by hand or using its motor.

As the wheel diameter was 6.5 cm, each full revolution of the wheel would offset its distance by 20.4 cm. This gave us what we thought would be a reliable method of measuring distance and, by that, speed by using wheel encoder stripes/meter. Given that there were 9 stripes marked for the wheel encoder, it should have measured 44.1 stripes in a meter. This metric seemed reliable for excruciatingly slow speeds where measurements were guaranteed and reliable, yet this was not the case at operational speeds.

We also discovered that running the wheel encoder in its own thread with a predefined sleep interval between each reading of the encoder state was not suitable, and later changed it to run using an interrupt routine which was triggered every time a new value was read. The tests confirmed our suspicion that this method was preferable and gave agreeable results at most times.

The stripes/meter value was tampered with in order to increase metric reliability. Eventually the value 41 proved to give us the most accurate result at operational speeds. This value were found by calculating the difference of measured distance and actual distance at the various speeds. Ten runs were made with each stripes / meter value in order to gauge reliability and the precision was acceptable at best. We believe that a completely different method or setup would be required to further increase the accuracy of distance measurement, as has been approached in other parts of the report. This would very likely increase the quality of maneuvers such as parking and overtaking.

7.2 Software Tests[IV]

Google Test and Google Mock were used to perform unit testing for the serial module to ensure packet parsing is correct in the serial buffer and that the serial handler behaves as expected. At first, CxxTest was considered and some unit tests were implemented in this framework. However, it was decided to switch to the Google testing framework for its simplicity and ease of use. The CxxTest environment did not provide a direct support to mock class object, unless the mock objects are standard library functions. One way to solve this problem would have been to define the mocked class functions in a separate source file and link it at compile time. Meaning that we would have one .cpp file for the 'real' and one for the 'mock' functionality. This would have resulted in more effort and time spent than estimated with Google Test and Google Mock.

SerialBufferTest.cpp
The tests focus mostly in the parsing of correct packets when data is received from the usb connection. The appendReceiveBuffer() function in the serial buffer class is tested with various fault packets, e.g. wrong delimiters, wrong checksum, etc. Along with those, some tests check the correct behaviour of the checksum() function.

SerialHandlerTest.cpp
The tests are designed to verify the expected behaviour of the readOp() and writeOp() functions of the serial handler class. For this purpose, we mock the serial io interface to simulate correct or false output when performing read and write from the serial connection. In addition, test were included for the functions trying to connect and reconnect with the serial IO.

7.3 Integration Tests[JK]

Our integration tests were mostly done manually by using the Odroid mounted on the car itself. For some parts, like the the LaneFollower interworking with the DecisionMaker, the OpenDaVINCI simulation was used. For integrating the proxy component with the STM32 board, the STM32 board was connected directly to our computers via USB to test the serial connection implementations both in the proxy and on the STM32. The STM32 was programmed to indicate any communication problems through it's LEDs, which helped to identify any communication problems throughout the project.

Most of the software integration went fine straight from the beginning. Nevertheless tweaking of different values was unavoidable (for example for the parking scenario or the overtaking). After the optimal values were found, these were pushed to to the repository as default values (TacoServ user).

While integrating other components with the DecisionMaker, we faced some issues. When the session was up and running with one supercomponent, one proxy and the configuration tool, the DecisionMaker component didn't always receive the SensorBoardData and DecisionMakerMSG as it was supposed to. Sometimes it affected both data types, sometimes just one. We ended up hard coding the state value and restarting the DecisionMaker each time until SBD values were received.
We weren't able to fix this problem. Rechecking with the teacher confirmed that we are using the receiving mechanisms in a correct way. We suspect it may have to do with processing time issues or maybe the Linux environment on our Odroid, and we would have investigated this further if we had more time.

Chapter 8

Project Retrospective

[JK] Personally I think the group size was way to big considering the scope of this project. Due to these circumstances we did some additional stuff like the ATConfigurator tool, which was an awesome experience to work at. Especially learning about ncurses will be helpful for further UNIX/Linux developing.
When it comes to the Odroid board, we noticed that it wouldve been good to have a faster micro SD card available, because of the very low read and write speeds we had to and from it.
Another improvement wouldve been a better WiFi module. Both of these limitations led to a very slow connection to Odroid (see section 9: Toolchain & Odroid). It could also be that the Odroid is already a bit outdated. A switch to an RPI v3 for the next students could be helpful.
Group wise, I would follow a better software process the next time, due to the fact that it helps with the communication and tasks splitting in the group. This time it felt sometimes a bit unstructured and improvised.
But overall: a fun project and I think a lot of our members will be up for the Carolo Cup if asked.

[IV] Things that went well include the serial connection and the use of several testing frameworks. For the usb communication, it was a good experience to work with the libusb-1.0 library. It is well developed and tested, and provides a lot of freedom to the programmer as to how to implement a usb connection. The supported synchronous and asynchronous API is fun to work with and also presents a challenge, especially the latter one. At first, the usb connection was realized using asynchronous reading and writing, and therefore manually handling the callback function from the io operations. However, due to the lack of multiple devices and a dedicated thread to the execution of the io, as well as the similarity of the two implementations, a switch to using the synchronous API was made. Therefore, reducing the complexity of our code. In addition, learning how to use CxxTest and Google test framework was a useful and valuable experience.

Things that we could have done better is to perhaps put more effort into the design of the entire system. For example, up until the very end of the project, we had a problem when starting up the different components of our application. Some components, like the LaneFollower, would start normally and receive values from the Proxy whilst others, like the DecisionMaker, would not receive any values at all. That led to multiple restarts of the failed component until it connected properly and started receiving values.

[TDJ] We felt that the group size was too large for this project. The project progressed at a very acceptable rate and deadlines were met, yet tasks were at time too saturated with manpower.
Choosing the STM32 over the Arduino opened a lot of doors, but proved to be a bit complex to get up and running.
A number of hardware components were in a poor condition when we received the car. The side-rear infrared sensor gave wildly fluctuating values, and the front right ultrasonic sensor simply did not work on the STM32 despite intensive troubleshooting.
It wouldve been interesting to work with a properly functioning wheel encoder as well. Aligning it on its axis was hard, as the wheel did not conform to the requirements set by the encoder.

[JE] In my opinion, the only really challenging part in organizational terms in this project has been how to make sure the developers of the respective behavioural components (lane following, overtaking and parking) have had enough time with the car to test their components. We were warned for this and relatively prepared but still it's hard to imagine the extent of this challenge beforehand.

It would have been interesting to have better hardware, such as better infrared sensors with longer range, a wheel encoder suited for the wheel or other way of measuring distance, and a faster camera (in terms of exposure time in darker conditions). That said, overall I'm happy to have learned a lot about embedded systems programming and interacting with hardware.

[MT] I feel like I could use some extra days to test the overtaking on the real car. Working with the simulator was fun and useful at the same time, but the transition to the real world required some major changes that could have been optimized with some additional time.

[ME] Personally I felt that the group worked very well, we managed to split the different parts of the complete project down to smaller modules and assigned these modules to smaller groups.
Although the group size was a minor problem when it came to actually implementing parts on the car, as we needed to troubleshoot while running things on the car it became hard timewise, also adding to this was the limited amount of power on time because of only having one battery.
I also felt that the Odroid board was not really up to par with the rest of the

hardware, it was sluggish and slow when it came to working on the board. If this was because of the WiFi module or something else I dont know, but being slower than a Raspberry Pi One while having better hardware I find a bit weird.
To sum, fun project! Definitely continue doing this, maybe have some extra hardware available for failures. Also, make sure that the cars keeps a consistent quality amongst groups. Several cars had problems because of poorly executed soldering, building etc.

[NLC] I feel that the group worked well in this project, but it was to big groups to work with. Sometimes it felt like some people had nothing to do at some points. Me myself had a lot of problems with the OpenDaVINCI and the virtual machine in the beginning, when code that had worked one moment ago stopped working and getting errors from the OpenDaVINCI. I reinstalled everything around five times before choosing to use another laptop with linux installed on it. After that it was fun to be able to work and create the parking scenario. The biggest group problem we had was that a lot of people wanted to test on the car on the same time and finding time for everyone wasnt easy, and that we had a limit on the battery made it even harder. For me I would have liked to have even more time for testing the parking, even though the end result was good.
But overall it was a good project that we as a group made and we can be happy about how the car came to be.

[DK] Personally I enjoyed this project a lot more than I anticipated at the start. The fact that the project was so hardware adjacent turned me off a bit seeing as I enjoy high-level programming a lot more. When I realised that we were allowed to split up the work so that people worked on completely separate parts however, I quickly jumped on to work on the high-level board, more specifically the Lane Following and Detection part. Working on this part of the project was what made it enjoyable for me.
The group co-operation worked very well. At the start of the project we split the group into smaller teams, even though we ended up co-operating a lot between teams towards the end of the project, we did stick to our initial teams throughout most of the development process. I would say that I think that the size of the group was a bit on the large side. We have worked in groups of about six members before, I think groups of that size works a bit better in terms of finding sizeable tasks for everyone in the group.
OpenDaVINCI was a big hassle at times, especially the installation process and the generation of data structures. Just the fact that we had to recompile OpenDaVINCI on all machines including the Odroid every time we wanted

to add a new data structure felt like our time was a bit wasted. I do not know if there is a better way to do this, but it can be something worth looking into for further projects like this. Maybe really stress the generated data types early in the project.
As I said though, the project was a lot more fun than I expected and I am happy with how it turned out.

Chapter 9

Toolchain & Odroid

9.1 GCC/G++[JK]

In our initial presentation we held in front of the class, we talked about possibly using Clang as our compiler, due to some research we did during our C course (where we noticed that for example software interrupts aren't as good supported in gcc as they are in clang).
Due to the fact that everybody used gcc straight from the beginning and there was not really a discussion in the group about it, we stuck with it.

9.2 CMake[JK]

As proposed by the teacher, we are using CMake. It made the whole build process easier and more comfortable. Also the support for CMake of all modern IDEs allowed each member to use the IDE/development environment of his choice.
Another big advantage using the CMakeLists.txt is the overview we get. For example is it easy to find out the level of warnings we are using in that software project and the external software components (libraries, etc) we are linking the software with.
Notice that we mark all INCLUDE_DIRECTORIES as SYSTEM, to avoid warnings coming from these external software entities.

9.3 Jenkins[JK]

To automatically build the software components each time we push changes in their code to a certain branch, we introduced Jenkins to our toolchain.
To make this possible, Jenkins is connected to our GitHub repository with a so called webhook.
To ensure that also an overall build (on the Odroid) will work, we introduced an overall build behavior based on an overall CMake file as well. The server we are running the build on reflects the same build chain as it is installed on the Odroid (except the compiler, because of the hardware architectures are different). All necessary build tools and libraries are installed at the same paths.

9.4 pmccabe[JK]

We decided to integrate pmccabe directly into our Jenkins builds. Before each build starts, we run the pmccabe command to show us the ten most complex functions of the current project (see pmccabe man page).
To see the result, we just went to a specific build and looked at the console output.

9.5 Cppcheck[ME]

Analysing the code is a good way to get an overview about coverage and hidden bugs.
This tool in particular is a static analysis tool. That means that the code will be analysed in a static way, giving us an overview about static bugs as:

- Out of bounds checking
- Memory leaks checking
- Null pointer references
- Uninitialized variables
- Invalid usage of STL (Standard Template Library)
- Unused or redundant code
- Obsolete or unsafe functions

Unfortunately this tool will not give us any information about code coverage as the program is not being actively monitored while running, hence static analysis.

9.6 SonarQube[ME]

This is a open source platform for continuous inspection of code quality.
We use this in collaboration with Jenkins and Cppcheck, every build in Jenkins will be inspected and the results will be uploaded to this platform where it presents the Cppcheck results in a human readable and pleasant way.

When looking at the results all warnings/failures from Cppcheck will be transformed into a new “Issue” and it clearly states the bug and where it can be found. All of the issues are also converted into Technical Debt according to predefined rules bundled with SonarQube. This is something while it's nice to have, we've not really been using.

9.7 Operating System[JK]

To have a lightweight operating system at hand, we decided to use a Debian 8.4 *Jessie* minimal image, available on the Odroid forums.
Most of our team members used a debian-based linux distribution before, so it seemed to be the most reasonable choice if there would be the need for them to work with the package manager, etc. The use of a minimal image made sure, that we would not have any unnecessary software, like a Desktop Environment, installed.

9.8 Connection[JK]

To connect to the Odroid, two of our team members were able to host a hotspot, which the Odroid could connect to.
This hotspot made sure, that the Odroid had an internet connection, so updating packages on it was possible, as well as pushing changes to the code made while testing and integrating could be pushed to the git repository (note that these changes were pushed with the user TacoServ).
We accessed the Odroid with an SSH connection subsequently.

Appendix A

Individual Report Contribution

ME = Max Enelund
JE = Jerker Ersare
TDJ = Thorsteinn D. Jörundsson
JK = Jonas Kahler
DK = Dennis Karlberg
NLC = Niklas le Comte
MT = Marco Trifance
IV = Ivo Vryashkov
TEAM = The whole team

LaTeXtemplate & setting by Jonas Kahler

Appendix B

Used Algorithmic Reference

B.1 Canny Edge Detector

Used for edge-detection in the Lane Follower, computes edges out of a grayscale image.
→ Canny detector on the OpenCV documentation

B.2 PID Control

Algorithm used in the Lane Follower for making the driving more stable and smooth.
→ YouTube video on the used algorithm

B.3 BoxParker.cpp

Used as initial inspiration for making of the parallel parking.
→ OpenDaVINCI Git Repo code

B.4 LaneFollower.cpp

Used to achieve a basic understanding of simple lanefollowing and lanedetection.
→ OpenDaVINCI Git Repo code

B.5 OpenCVCamera.cpp & Camera.cpp

Used as a building block for the camera operations.
→ OpenDaVINCI Git Repo code