FrostAV Development

Team Frost: Lewis Collum, Ian Shelly, Justin Marcy

Updated November 7, 2019

Contents

1 System Requirements

1.1 Problem Overview

The system to be specified requires the design and construction of an autonomous car that has the ability to navigate within a lane. The system should satisfy the tasks specified in section

1.2. Furthermore, the car must have wireless communication abilities. And, the sensor systems implemented on the car must be minimally invasive (see section 1.6).

1.2 Problem-Domain Requirements

1. Given a lane, the car must travel approximately parallel to it, such that the car stays within its boundaries consistently, and, if the car is to accidentally leave it, it promptly returns.
2. Given a corner, the car must turn, continuing from the car's current lane to the next, such that the car stays within its boundaries consistently, and, if the car is to accidentally leave it, it promptly returns.
3. Given an obstacle, the car must stop until it is moved further from the car, or it is removed from circuit boundaries.
4. Given a sign, the car must respond to the event provided by it.
5. Given a circuit, the car must complete a full loop.

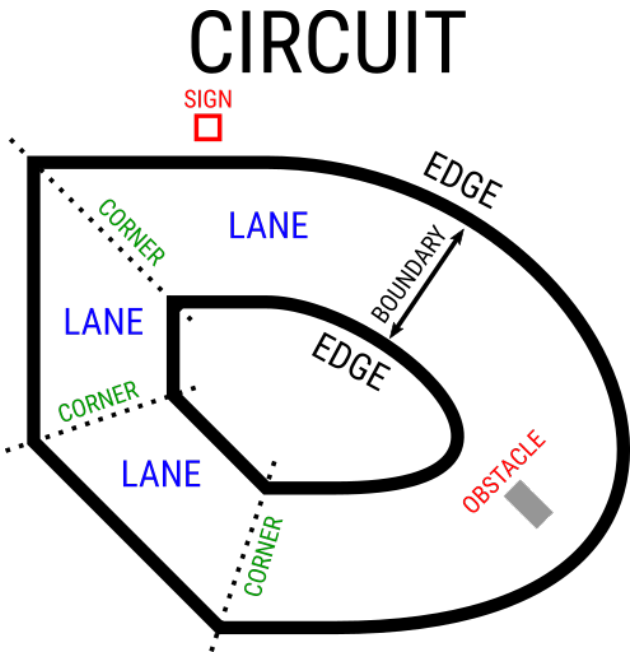


Figure 1: An example circuit which the car must navigate through. Provides visuals for the definitions in section 1.2.1.

1.2.1 Problem-Domain Definitions

Term	Definition
Corner	A sharp change in path direction that connects two lanes
Lane	A path that has a boundary
Boundary	The area between two parallel edges
Edge	A line or a curve
Circuit Boundary	All connected boundaries which define the total area of the circuit
Circuit	A closed path defined by connected lanes and corners
Obstacle	Any object that lies within circuit boundaries
Sign	A flat image mounted to a post outside of circuit boundaries

1.3 Control Requirements

1.3.1 Control System:

The Control System shall:

1. Interface with vehicle peripherals, such as, motors, servos, batteries, etc.
2. Be the only system coupled to the vehicle.
3. Have a subset of Control Modules for each peripheral in need of control.

1.3.2 Control Modules:

Each Control Module shall:

1. Encapsulate a single purpose, such that, one controller controls a single peripheral.
2. Couple to a peripheral electronically, not mechanically.
3. Be able to communicate with other controllers via a wired bus.
4. Be able to communicate with non-controllers that depend on it, via a wired bus.
5. Communicate with other controllers and non-controllers via a single shared wired bus.

1.4 Autonomy Requirements

The Autonomy System shall:

1. Not be directly coupled to the Vehicle Interface (section 1.5.1).
2. Be able to fit on the vehicle.
3. Allow the vehicle to navigate, as per section 1.2, without user interaction.
4. Be able to communicate with the Control System (section 1.3).

1.5 Interface Requirements

1.5.1 Vehicle Interface Requirements

The Vehicle Interface shall:

1. Include a way for the vehicle to be turned on and off, such that, the vehicle receives no power to the drive system or logic when off.
2. Include a way for the vehicle to have its logic turned on, while the drive system is off.
3. Provide a battery peripheral that powers the logic.
4. Provide a battery peripheral that powers the drive system.
5. The total power consumption of the logic and drive system cannot exceed the maximum capacity of the battery peripheral(s).
6. Provide a peripheral that moves the vehicle.
7. Provide a peripheral that steers the vehicle.
8. Provide an electronic interface from each peripheral.

Term	Definition
Logic	Eletronic Systems including the Control and Autonomy Systems
Drive System	The electromechanical parts on the car, such as, the motor(s)

1.5.2 Wireless Interface Requirements

The Wireless Interface shall:

1. Allow for wireless tunneling (e.g. via SSH)
2. Be able to access a server.
3. Provide bi-directional communication.

1.6 Sensor Constraints

1. The sensors to go on the vehicle must be minimally invasive such that any sensor attached to the chassis is not coupled to an existing mechanism on the chassis. For example, an encoder cannot be used since it is coupled to the vehicle drive system. Sensors such as accelerometers and cameras can be used since they can be attached to the chassis, but are independent of existing vehicle mechanisms.

1.7 Financial Constraints

1. The FrostAV team shall not exceed $300 towards parts under the superivision of Clarkson University's Department of Computer Engineering.

2 System Abstraction

2.1 Abstract System Operation Description

The Frost Autonomous Vehicle consists of a vehicle and a system for controlling the vehicle. The vehicle provides an interface mainly to move it and to steer it. Bridge Controllers handle the peripherals that make up the vehicle interface. Bridge Controllers cannot operate unless they have instructions. These instructions come from a wired-bus. Different types of modules can send instructions to the wired-bus. The most important modules are the Networker and each

individual Sensor System. Sensor Systems provide feedback for the Bridge Controllers to act upon. Furthermore, the Networker transfers information between the wired-bus and the wireless-bus. The wireless-bus allows for communication with devices that are not on the vehicle itself. Overall, the abstract system provides a map for how information is received and directed towards the vehicle's interaction with the environment.

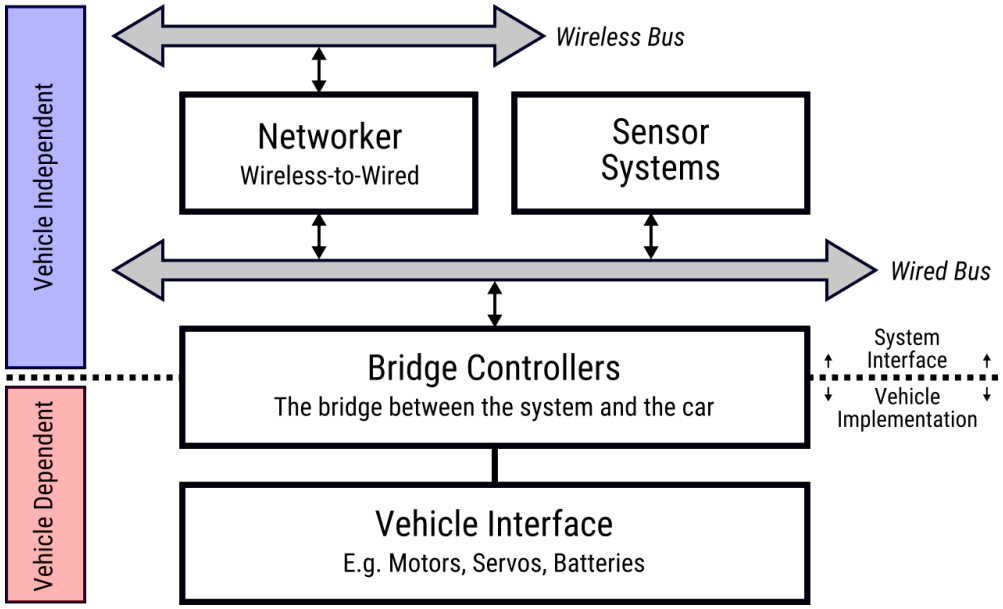


Figure 2: Fundamental module abstraction. Consists of a wired-bus for inter-communication between modules on the Frost vehicle, and a wireless-bus for communication with a server or an ssh client. Bridge Controllers act as the coupling between the vehicle and the modules that do not depend on the vehicle interface.

2.2 Bridge Controllers

"Bridge" refers to a software design pattern from the Gang of Four, which intends to "decouple an abstraction from its implementation so that the two can vary independently." As such, the Bridge Controllers will have software that implements the Bridge design pattern, so that implementation code (for a specific vehicle) is decoupled from the system interface.

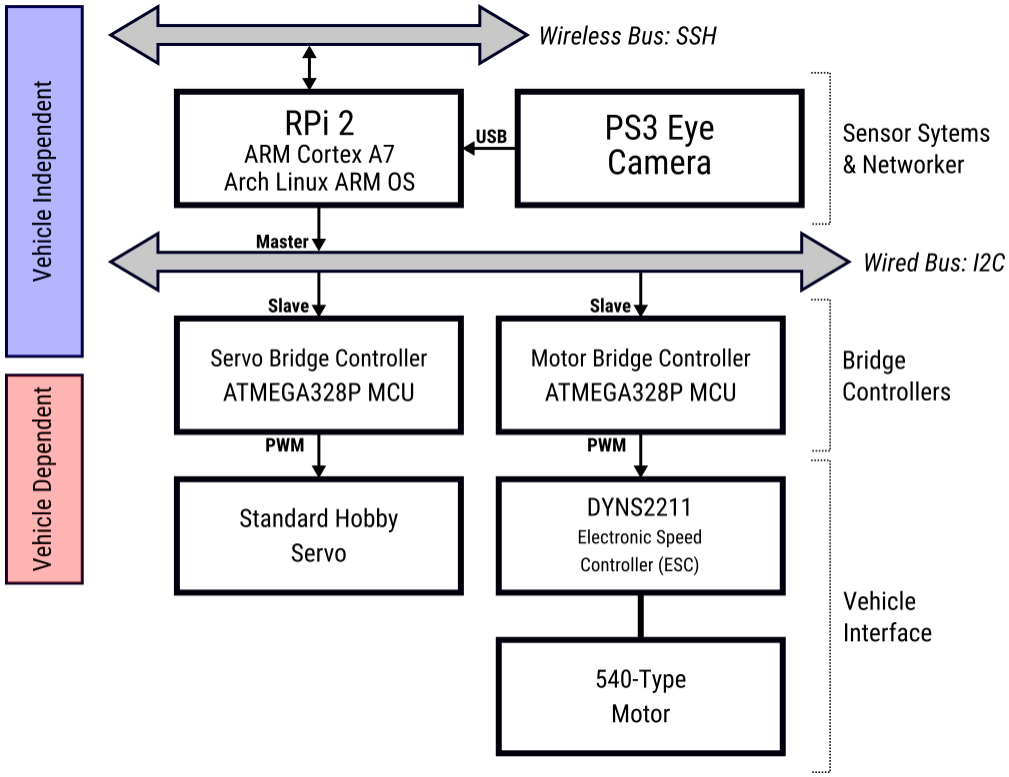
2.3 Networker

The Networker is reponsible for transfering information between the wireless-bus and the wired-bus.

2.4 Sensor Systems

Various Sensor Systems may be added to the wired-bus. An individual Sensor System should receive information from a sensor, process the information, and send the result to the wired-bus.

3 System Plan



3.1 System Operation Description

The PS3 Eye Camera provides visual information about the environment. This information will be processed with the OpenCV library on a Raspberry Pi 2 with an Arch ARM operating system. Processed camera information will then be piped through I^2C to the Bridge Controllers. The Servo Bridge Controller has a PWM output which directly drives the steering servo. The Motor Bridge Controller has a PWM output that is handled by the electronic speed controller (ESC), which drives the motor.

Overall, this system operates in a feedback loop commonly illustrated similar to the figure below.

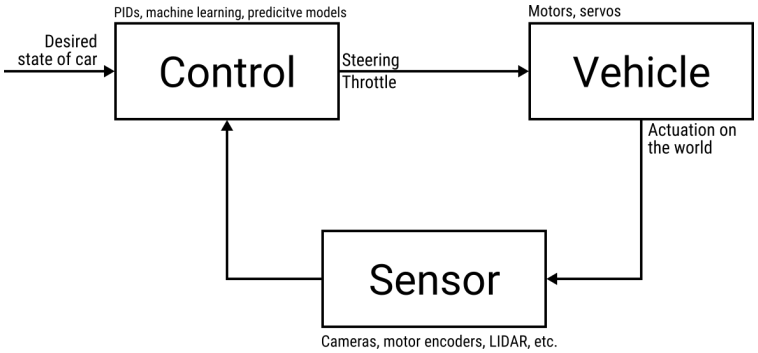


Figure 3: A simple feedback loop for an autonomous car.

3.2 Feedback Loop Flow of Information

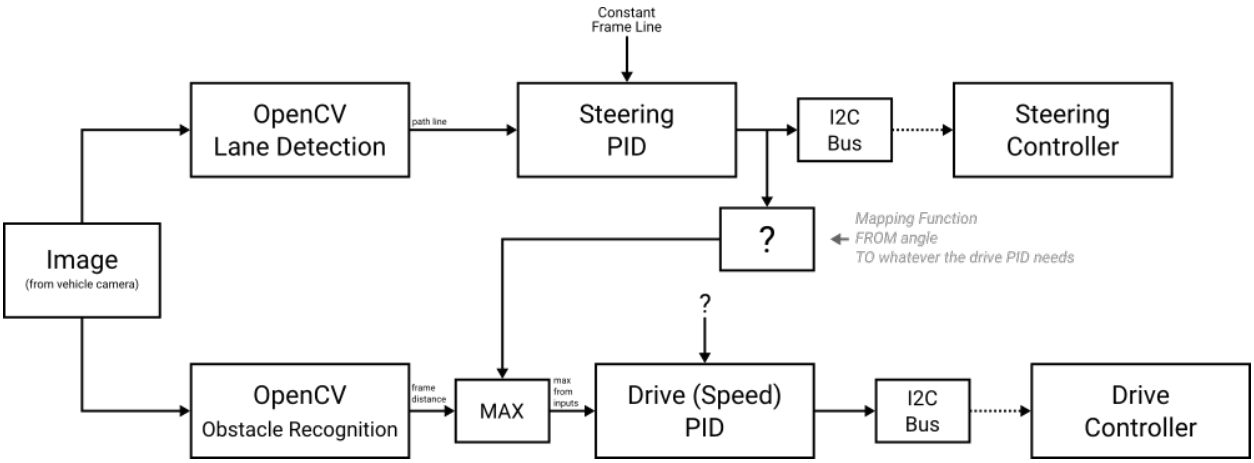


Figure 4: A concrete illustration of the feedback loop in figure 3

3.3 System Assumptions

The assumption made in this development stage is that the motor speed and servo steering angle can be controlled individually, such that motor speed and steering angle do not require direct knowledge of each other. Hence, the arrows from the wired-bus to the Servo and Motor Bridge Controllers are unidirectional. This assumption is meant to simplify the I^2C wired-bus configuration - since bi-directional communication on this bus would require arbitration.

3.4 Bus Communication

JSON will be used as the format for data for both the wireless and wired buses. JSON provides easy to understand data structure packets that can be serialized and sent as a stream. Using JSON sacrifices speed for readability and extendability of the system.

3.4.1 Wireless-Bus

In this stage the wireless-bus should allow for an SSH client to connect to it. As such, JSON packets that would normally be sent to the wireless-bus, will instead be stored locally in the system, but will be callable from an SSH client.

3.4.2 Wired-Bus

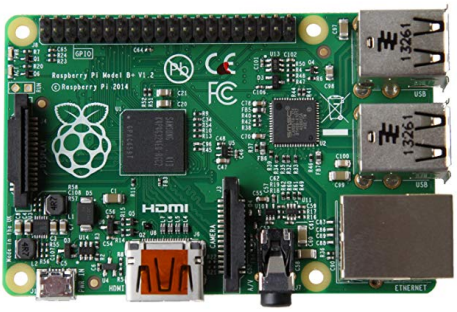
For the wired-bus, I^2C will be used. I^2C addressing will be done manually for each slave connected to the bus. Data will be sent from master to slaves via the JSON format.

3.5 Block Function Description

3.5.1 Raspberry Pi 2 (RPi2)

The RPi2 has an Arch Linux ARM operating system. It is responsible for processing camera information from the PS3 Eye. It is also responsible for being the Networker, which transfers

information between the wired and wireless buses.



3.5.2 Steering Servo & Motor Bridge Controllers (ATMEGA328P)

Processes commands from the wired-bus to control the vehicle's steering servo and motor.

3.5.3 PS3 Eye Camera

Provides images over USB, for environmental perception.

3.5.4 Electronic Speed Controller (DYNS2211)

Takes in a PWM signal and outputs an amplified and directional PWM wave to the vehicle's motor.

3.5.5 Power Supply

Takes the 12v input from the battery and steps the voltage down to 5v to power the vehicle's systems. Also provides current and voltage monitoring. This module is designed and built as part of this project (see section 6 for details).

4 Lane Detection using OpenCV

4.1 OpenCV

4.1.1 OpenCV & Reading Video

The primary software component responsible for camera perception is OpenCV. OpenCV is an open source software library that implements machine learning algorithms to provide object detection and motion tracking to the user (opencv). We will be using this library as the foundation to create lane detection, object detection, and text recognition for the customer's vehicle. Each of these three components require a specific image filtering technique before they can be utilized in an algorithm. Before we can filter an image however, opencv requires an input. The input in this design is the PS3 camera mounted at the front of the vehicle. Inputing a constant video feed is challenging because it creates a large computational load on the CPU and RAM. To reduce this, we will be sampling our video and using periodic frames for input.

```
#include<iostream>
#include <opencv2/opencv.hpp>
using namespace cv;
int main(int argc, char** argv)
{
  Mat frame;                                  //temp image variable
  VideoCapture capture;                       //Array of captured frames
  namedWindow("Window", WINDOW_AUTOSIZE);     //create viewable window/executable
  capture.open(argv[1]);                      //reads input video
  for(;;)
    {
      capture>>frame;                           //amends new frame over previous
      if(frame.empty()) break;                  //break if there is no video feed
      imshow("Window",frame);                   //display frame in window
      if(waitKey(30)>=0) break;                 //Define framerate sampling
    }
  return 0;
}
```

In the figure above, our input video feed will be piped in as an argument argv. The video frames are then sampled and stored in an array of type "VideoCapture", an object type included in the OpenCV library. The "waitKey" function allows the user to select the maximum amount of frames sampled in one second. This is subject to change depending on the camera specifications and hardware utilization. It is important to find a balance between sampling rate and vehicle speed, as these affect accuracy and safety. Since our car will be traveling slow, it is possible to lower our samples to 20 or 10 frames per second without a significant compromise. This will be determined using our software test plan.

4.2 Lane Detection

Lane detection is the primary autonomous feature of the vehicle. Through OpenCV, we can take advantage of various image-manipulation and detection algorithms to accomplish this task. The goal of this feature is to receive a piped video input and then output a horizontal difference in position to the PID controller. In order to provide a clear image to the line detection algorithm, image filtering is used to reduce the noise that naturally comes with each environment. This section will provide a detailed description of how each image process is used to improve accuracy and reliability to the design.

4.2.1 Image Masking

Image masking is perhaps the most effective way to eliminate unwanted noise. Applying a mask is a way to single out a particular color in an image while ignoring others. Since we are using blue colored masking tape as lanes, we will use a blue mask. However, the input image is in RGB format. This does not help reduce noise as effectively due to each pixel having at least some amount of blue value. By using OpenCV's image processing library, the image can be converted HSV format or Hue, Saturation, Value. In OpenCV, the values are defined as follows.

1. Hue: [0-179]
2. Saturation: [0-255]
3. Value: [0-255]

The following figures shows the conversion to HSV format using the convert color function.

```
cv::cvtColor(src, hsvFrame, cv::COLOR_BGR2HSV);
```

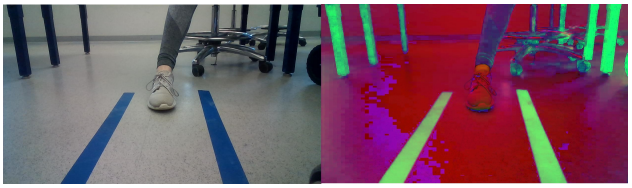


Figure 5: Source Image vs. HSV Format

"src" is the input frame while "hsvFrame" is the output. Once in HSV format, the mask values for blue can be specified. These values cannot be concrete however, as lighting conditions and noise will affect the HSV value. Instead, there must be a range of values. This range was acquired by reading the HTML color code allow the blue tape using Microsoft 3D Paint. The color codes are 6-bit hexadecimal values (0xFF, 0xFF, 0xFF) that can be converted to HSV. Recall, hue has a maximum value of 179 in OpenCV and must be calculated by multiplying 179 by the percentage value between 0x00 and 0xFF. After acquiring the minimum and maximum HSV values of the lanes from the test image, we are ready to use the mask function called "inRange." The function takes input arguments input image, minimum HSV values, maximum HSV values, and output image. The line below shows the values used for the blue mask.

```
cv::inRange(hsvFrame, cv::Scalar(100,100,100), cv::Scalar(110,255,150), newFrame);
```

The result can be seen below.

Figure 6: Mask Frame

4.2.2 Reduced Contrast

While adjusting the contrast of an image does not significantly improve the detection quality, it does help reduce discoloration. This includes effects such as strong reflections that cause artifacts throughout the image. In OpenCV, the function "convertTo()" allows the user to adjust the contrast by changing the 3rd input argument shown below.

```
newFrame.converTo(lowCont, -1, 0.5, 0);
```

Here, values below 1 reduce contrast while value above 1 increase contrast. By testing multiple values, we found that 0.5 was the darkest result without hiding the lanes. The figure below shows the chosen result.

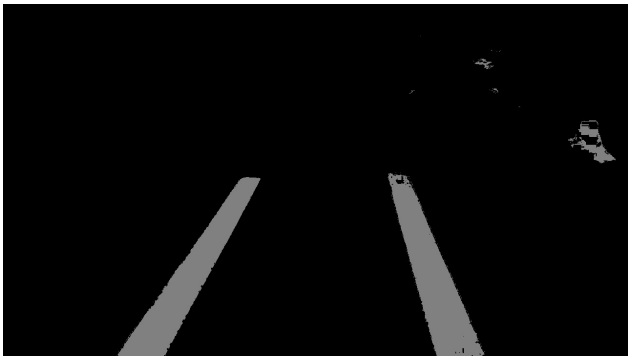


Figure 7: Low Contrast Frame

4.2.3 Gaussian Blur

A gaussian blur is a specific form of smoothing, also known as pixel averaging. It breaks the input image up into small 2D arrays, determined by the kernel size, and then multiplies each element by the gaussian function. Next, the results are summed and stored into a single pixel. Each output

pixel is essentially an average of its neighbor pixels, and thus creates a blur affect. The kernel size is chosen by the user and determines the radius and intensity of the blur. In other words, the more neighbor pixels averaged together, the more they will have a similar value.

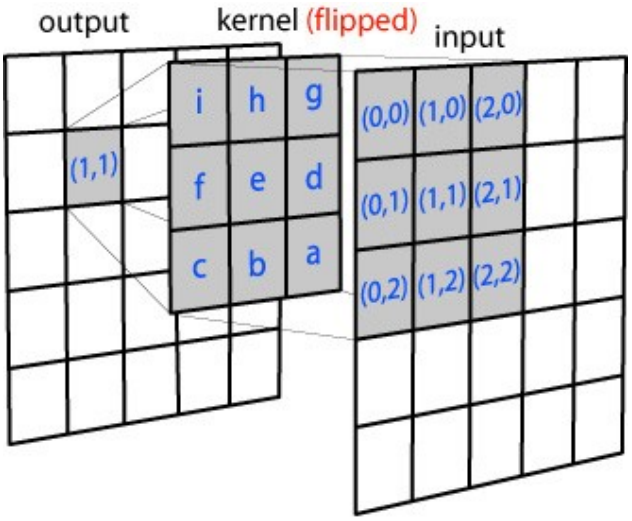


Figure 8: Gaussian Kernal Example

This process is critical to prepare for line detection, as line detection relies on consistent groupings of pixels. The gaussian blur is provided in the OpenCV image processing library and helps eliminate noise and small objects. The for-loop below shows the implemented blur. Here, we chose the blur to be applied 4 times. This is set through "MAX_KERNEL_LENGTH." The kernel size begins at 1 and is incremented by 2 since gaussian matrices must be odd (1,3,5,7...)

```
int MAX_KERNEL_LENGTH=7;
for (int i=1; i<MAX_KERNEL_LENGTH; i=i+2) {
    cv::GaussianBlur(lowCont, blr, cv::Size(i,i), 0, 0);
}
```

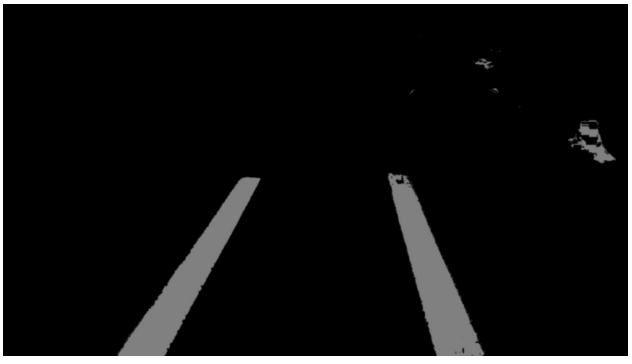


Figure 9: Gaussian Blur Frame

4.2.4 Area of Interest

Creating a region of interest is the last step before edge detection is applied. By cropping the image, much of the background noise is removed and the lane region is maintained. The best area to remove is the top half of the image, as this is where much of the enviorment noise is. However, the sides of the image should not be removed becasue the lanes might enter this lateral region during turns.

```
cv::Mat croppedFrame = blr(cv::Rect(0, blr.rows/2, blr.cols, blr.rows/2));
```

The code above shows a simple method to resize an image. The previous blurred frame invokes an OpenCV function called "rect()." The image is simply halved by dividing the horizontal rows by 2. The output of the image can be seen below.

Figure 10: Area of Interest (Bottom Half)

4.2.5 Canny Edge Detection

Canny edge detection is a specific type of line detection in OpenCV. Canny requires the image to be in two colors, since an edge is detected only near contrasting pixels. In other words, the calculation will be done where the dark pixels meet the light pixels. At each of these areas the gradient is calculated using the first derivative of the x and y coordinates. The following figure shows the equation used to calculate the gradient and angle.

$$Edge\_Gradient\ (G) = \sqrt{G_x^2 + G_y^2}$$

$$Angle\ (\theta) = \tan^{-1}\left(\frac{G_y}{G_x}\right)$$

Figure 11: Gradient & Angle Calculation

The gradient is used to calculate an angle, which allows us to store the gradient direction. Gradient direction must be perpendicular to the edge. To simply the calculation, the direction is rounded to one of four possible directions: left, right, diagonal left and diagonal right. At each pixel along the edge, the local maximum of the gradient is calculated and compared to the surrounding pixels in the same direction. Essentially, if the gradient direction is not the same as the two pixels next to it, then it receives a value of 0 and will not be counted as part of the edge.

The next component of the canny function is hysteresis thresholding. This allows the user to define a range for the gradient intensity and determines whether the pixel meets the criteria of the edge. The figure below is an example from opencv.org

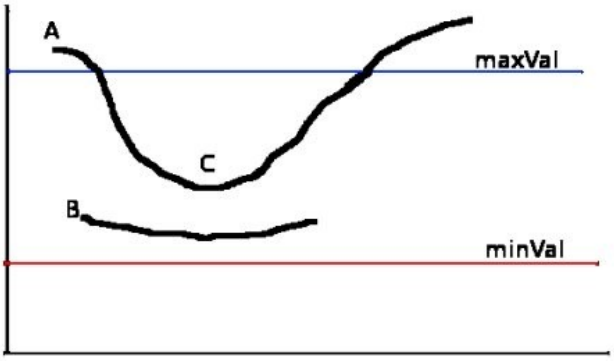


Figure 12: Min & Max Threshold (opencv.org)

Even if a certain gradient intensity for a pixel is above the maximum threshold, such as point A, it is still considered to be part of the line because it matches the gradient direction of its neighbors. Although point B falls within the min and max region, it is disconnected with its neighbors. In short, a gradient intensity must match the direction of its neighbors and cannot exceed to threshold for more than one value at a time. In other words, if there are two consecutive values outside the range then it is not counted.

```
cv::Canny(blr, edges, 50, 200, 3);
cv::cvtColor(edges, cdst, cv::COLOR_GRAY2BGR);
```

The code above shows the Canny function with input, output, minimum hysteresis threshold value, maximum hysteresis threshold value and kernel size. Similar to the gaussian kernel size, the canny kernel size is used to define the size of an array for calculating the gradient intensity with respect to its surrounding pixels. The default size is 3. The rendered edges are stored in a new frame . The output of the edge detector is shown below.

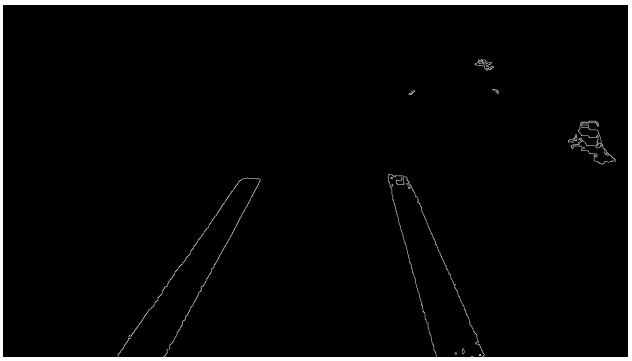


Figure 13: Detected Edges

4.2.6 Hough Transformation

The final step in lane detection is responsible for inputting the frame containing the edges, recognizing the lines, storing the values, and then redrawing them over the image. There are two methods of hough transformations, standard and probabilistic. The standard model generates a polar coordinate of each line, while the probabilistic model generates the endpoints of a line in

a vector. Another important difference is the standard model projects the line past its endpoint because it is following the calculated angle. This can cause issues if there are significant curves and turns within the lane. Because of this, we will be using the probabilistic model. Since the output of the transformation is the endpoints of a line, then the first step is to create a vector of length 4. This will hold the starting and finishing endpoints (X0, Y0, X1, Y1). OpenCV has a vector already included in their library. Next, the hough transformation function must be called.

```
std::vector<cv::Vec4i> linesP;
cv::HoughLinesP(edges, linesP, 1, CV_PI/180, 50, 50, 10 );
for( size_t i = 0; i < linesP.size(); i++ ){
    cv::Vec4i l = linesP[i];
    cv::line( cdstP, cv::Point(l[0], l[1]), cv::Point(l[2], l[3]), cv::Scalar(0,0,255), 3, cv::LINE_
}
```

Looking at the code above, function takes arguments input edges, output array of vectors, rho, theta, threshold, minimum line length, and minimum line are specifying that the polar input from the edge detection have a radius resolution of 1 pixel and angle resolution of 1 degree. The hough transformation records the number of pixels intersecting the predicted line; this is known as . When the threshold meets or exceeds the user-defined threshold value, it confirms that those pixels are part of the line and stores their position as a vector The threshold value is highly dependent on the resolution of the image, so a high threshold value would not be suitable for a low resolution image. minimum line minimum line are both very important. The minimum line length specifies the number of point considered to be a line while the minimum line gap joins other close separate lines. This is critical in removing insignificant lines caused from noise. Once the output vectors are populated, the for-loop is responsible for redrawing the red lines using the built-in The result is as follows. line().function gapand lengthLastly, linesP.in thresholdthe thetaand Rhogap. HoughLinesP()the Vec4itype

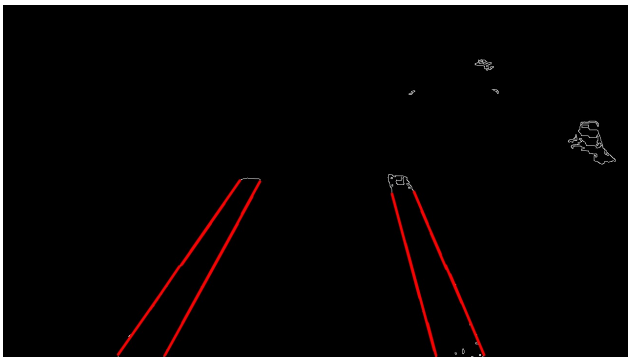


Figure 14: Probabilistic Hough Transformation

4.2.7 Output

Although visually we have achieved a successful output, the PID controller needs to receive a positional offset. To calculate an offset, we need to know the center vertiacal axis of the camera as well as the combined slopes of the left and right lanes. The goal is to find the angle projected over the center line of the camera. We can accomplish this by averaging the left and right slopes to create an artificial center slope. However, the width of the lane lines creates a problem because it generates two lines per lane, for a total of four lines. Looking at the current vector output below,

it is a mixture of left and right lane lines endpoints. There are also stray lines that did not merge with nearby lanes.

```
[justin@frostpi lane_detection]$ ./ver2
[235, 715, 318, 597]
[367, 650, 510, 391]
[796, 385, 853, 599]
[330, 718, 480, 447]
[857, 421, 896, 510]
[384, 503, 485, 358]
[824, 488, 883, 707]
[366, 530, 425, 446]
[915, 550, 985, 716]
[906, 532, 939, 609]
[489, 427, 526, 360]
[301, 622, 361, 534]
[876, 466, 908, 540]
[869, 650, 887, 719]
[841, 381, 869, 447]
[364, 532, 453, 405]
```

Figure 15: Hough Vector Outputs <X0, Y0, X1, Y1>

The endpoint vectors will allow us to calculate the left and right slopes. To determine which vectors belong to which side, we will take half the horizontal x position of our image and assign the vectors with x positions less than the center as positive slopes and the others negatives slopes. In the code below, integers averageN and averageP hold the averaged slopes of the right and left lanes. This for loop occurs directed after the hough lines are generated, and its main pupose is to reconstruct the hough lines over the image. The slope calculations are included in this loop for ease becasue each individual vector is being cycled already. This avoids multiple for loops and reduces time.

```
for ( size_t i = 0; i < linesP.size(); i++ )
    {
        cv::Vec4i l = linesP[i];
        cv::line( cdstP, cv::Point(l[0], l[1]), cv::Point(l[2], l[3]), cv::Scalar(0,0,255), 3, cv::L
        if (linesP[i][2] <= 640) {
            int slope = (linesP[i][3]-linesP[i][1])/(linesP[i][2]-linesP[i][0]);
            averageN=averageN + slope;
        }
        else {
            int slope = (linesP[i][3]-linesP[i][1])/(linesP[i][2]-linesP[i][0]);
            averageP = averageP + slope;
        }
    }
```

Once the right and left slopes are averaged, we now calculate the midpoint slope.

1. If the positive slope is greater than the absolute value of the negative slope, then the midpoint slope must be negative because the negative slope has a wider angle.
2. Otherwise, the midpoint slope is positive

```
int midslope=0;
double angle=0;
if (averageP > abs(averageN)) {
    midslope = averageN - averageP;
}
else {
    midslope = averageP - averageN;
}
if (midslope < 0) {
    angle = -((CV_PI/2) - atan(abs(midslope)));
}
else if (midslope > 0) {
    angle = (CV_PI/2) - atan(midslope);
}
else {
    angle = 0;
}
std::cout<<angle*180/CV_PI*100<<std::endl;
```

Next, based off the midpoint slope being positive, negative, or perfectly zero, we can calculate the angle from the y-axis using trigonometry. This can be seen in the code above. Lastly the angle is converted to degrees and then scaled by 100 to be understood as an integer. This is becasue the rasberry pi struggles with dynamic memory and thus doubles and floats should be avoided. The output is piped to the PID controller and then scaled back down for accuracy.

4.3 Test Plan

Test	Description	Result Type
Lane Pre-Processing Latency	What is the average time between when the frame is captured to when the hough transformation results are complete?	Numeric
	Is this reult less than 33ms?	Pass / Fail
Frame Rate Sampling vs. Vehicle Speed	Increment vehicle speed by 1 mph (1-5) and increment framerate by 10 fps (10-60) Record quality of result. Which combination of speed and framerate gave the most consistent results?	Plot
Threshold Value	Increment the threshold value by 50 starting from 0. At what value can the hough transformation no longer generate a line?	Numeric

5 Controlling the Steering Servo with a PID

5.1 Concept

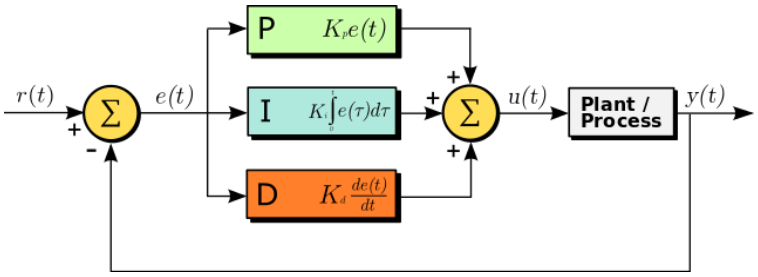


Figure 16: A block diagram of a PID controller in a feedback loop. r(t) is the desired process value or setpoint (SP), and y(t) is the measured process value (PV). *Arturo Urquizo, CC BY-SA 3.0 https://creativecommons.org/licenses/by-sa/3.0*

A PID controller can be used to correct the steering of an autonomous vehicle. Given a vehicle with the task of following a line or path, the vehicle must be able to detect the path as well as where it is in relation to the path. The difference can be taken from these two pieces of information to form the cross-track error. The cross-track error will be sent as the input to the PID — in figure 16, the cross-track error is denoted $e(t)$.

The PID has three control terms that are summed together to get a control variable output. Proportional control allows the vehicle to steer harder when it is further from the path. Derivative control induces a resistance to the pull from the proportional control; this brings the car smoothly back to the path in a timely manner while reducing the chance that the vehicle will overshoot the path due to the proportional control. Finally, the integral control corrects for external effects such as wind and terrain.

Each of these controls rely on a properly tuned PID. To tune a PID, each control term has a gain which must be adjusted until the system is stable. Such details will not be covered in this document.

5.2 Parts

Part	Purpose
Arduino Uno (AVR attmega328p)	Steering Vehicle Interface Controller that interfaces between the Pi and the vehicle servo.
Standard Hobby Servo	The same type of servo that will be to steer the vehicle.

5.3 Experimental Setup

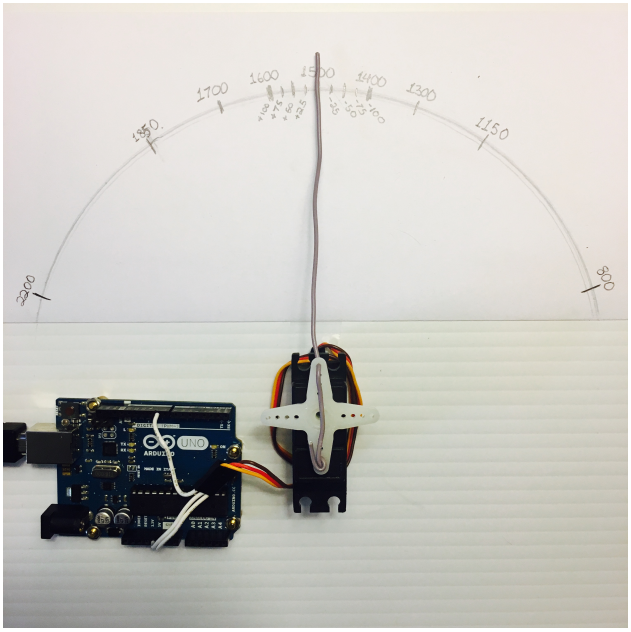


Figure 17: The Arduino provides microsecond PWM values from 800 to 2200μs to the servo. The servo reacts to an error value via a software PID controller. Since there is nothing in this setup that generates a true error value, error values must either be simulated or retrieved from a feedback sensor (such as a camera). The values along the circumference are microsecond values that correspond to the servo position; they help verify the accuracy of the PID.

5.4 Software Design

All code in this section was compiled using the avr-g++ tool, and using C++17. Test code was compiled with g++ as opposed to avr-g++.

5.4.1 USART to Command the Servo from a Terminal Emulator

In order to talk to the servo, we need to set up the USART on the AVR microcontroller. We will only show the code for a basic USART setup. Since the USART is only being used for experimental purposes only, we will not explain the code. For reference of how the code was used in our experiment, see the `main.c` file in the code appendix section 8.1.1.

`usart.hpp`:

```
#ifndef USART_HPP
#define USART_HPP

#include <avr/io.h>
#include <stdint.h>

namespace usart {
    void setup(uint32_t clockFrequency, uint16_t baud) {
        uint8_t ubrr = clockFrequency/16/baud - 1;
        UBRR0H = ubrr >> 8;
```

```
        UBRR0L = ubrr;

        //Enable Transmitter & Receiver
        UCSR0B = 1 << RXEN0 | 1 << TXEN0;
        //Frame Format: 8 data, 2 stop
        UCSR0C = 1 << USBS0 | 3 << UCSZ00;

    }

    void print(char c) {
        while (!(UCSR0A & 1<<UDRE0));
        UDR0 = c;
    }

    void print(char* string) {
        while (*string) print(*string++);
    }

    char getChar() {
        while (!(UCSR0A & 1<<RXC0));
        return UDR0;
    }
}

#endif
```

5.4.2 Servo Control

To control the position of the servo, the attmega328p has a 16-bit timer. Once we set up the timer, we can provide it a pulse duration between the accepted values of the servo (typically 1-2ms).

The code for the servo is at the register level. Ideally, one should use a hardware abstraction layer (HAL) to avoid writing production code at a register level. In our `main.cpp` file, we have the following to set up our servo.

```
#include <avr/io.h>

static constexpr uint32_t hertzToCycles(uint16_t hertz) {
    return clockFrequency/prescaler/hertz;
}

static void setupServoPwm() {
        DDRB |= 1 << PINB1; //Set pin 9 on arduino to output

        TCCR1A |=
        1 << WGM11 | //PWM Mode 14 (1/3)
        1 << COM1A0 | //Inverting Mode (1/2)
        1 << COM1A1; //Inverting Mode (2/2)

        TCCR1B |=
        1 << WGM12 | //PWM Mode 14 (2/3)
        1 << WGM13 | //PWM Mode 14 (3/3)
        1 << CS11; //Prescaler: 8

    //50Hz PWM to cycles for servo
        ICR1 = hertzToCycles(pwmFrequency)-1;
}
```

The servo uses the MCU's 16-bit Timer/Counter1 with PWM. It is connected to pin 9 on the Arduino Uno (port B, pin 1).

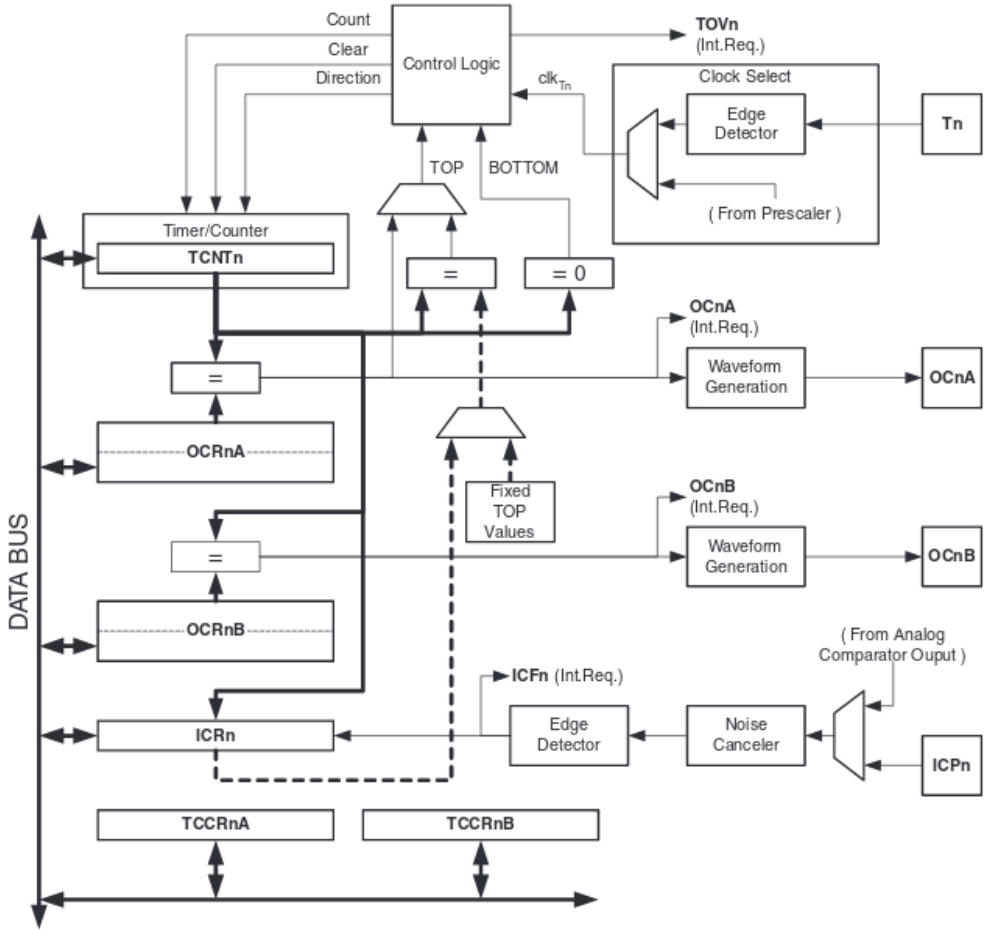


Figure 18: ICRn stores the number of cycles needed for a 50Hz PWM signal for the servo. OCRnA stores the number of cycles until the counter reaches the TOP. OCRnA is used to change the pulse width of the PWM signal.

We also need to specify that we want to use Fast PWM Mode (resets counter when it reaches the TOP) and set ICR1 as the TOP. This corresponds to `mode 14`.

Mode	WGM13	WGM12 (CTC1)	WGM11 (PWM11)	WGM10 (PWM10)	Timer/Counter Mode of Operation	TOP	Update of OCR1x at	TOV1 Flag Set on
13	1	1	0	1	(Reserved)	–	–	–
14	1	1	1	0	Fast PWM	ICR1	BOTTOM	TOP
15	1	1	1	1	Fast PWM	OCR1A	BOTTOM	TOP

Figure 19: Shows the `WGM` flags needed for mode 14.

We do this on the register level by setting the `WGM` flags from figure 19 in the `TCCRnA` and `TCCRnB`

registers (as shown in the code above).

Since the Arduino Uno uses a $16MHz$ clock, we will need a prescaler to divide the frequency such that,

$$\frac{16MHz}{50Hz \cdot \texttt{prescaler}} < 2^{16} - 1.$$

$50MHz$ is the needed frequency for the PWM signal for the servo and $2^{16} - 1$ is the maximum value that will fit in the `OCRnA` register (the size of an `int` on the attmega328p). If we set the prescaler to 8, the inequality above is true. We can do that by setting the `TCCRnB` register using the figure below. Setting this register for a prescaler of 8, looks like `TCCR1B |= 1 << CS11` (as shown in the code above).

CS12	CS11	CS10	Description
0	0	0	No clock source (Timer/Counter stopped).
0	0	1	$clk_{I/O}$/1 (No prescaling)
0	1	0	$clk_{I/O}$/8 (From prescaler)
0	1	1	$clk_{I/O}$/64 (From prescaler)
1	0	0	$clk_{I/O}$/256 (From prescaler)
1	0	1	$clk_{I/O}$/1024 (From prescaler)
1	1	0	External clock source on T1 pin. Clock on falling edge.
1	1	1	External clock source on T1 pin. Clock on rising edge.

Figure 20: We are using a prescaler of 8, which corresponds to `CS1 = 0b010`.

Finally, we can set the timer to inverted mode, meaning the pulse will occur at the end of the PWM period, as opposed to the beginning of the period. This is trivial for our application, but we still must choose either inverting or non-inverting. To set the timer to inverted we set the corresponding flags in the `TCCRnA` register, as `TCCR1A |= 1 << COM1A0 | 1 << COM1A1` (as shown in the code above).

COM1A1/COM1B1	COM1A0/COM1B0	Description
0	0	Normal port operation, OC1A/OC1B disconnected.
0	1	WGM13:0 = 14 or 15: Toggle OC1A on Compare Match, OC1B disconnected (normal port operation). For all other WGM1 settings, normal port operation, OC1A/OC1B disconnected.
1	0	Clear OC1A/OC1B on Compare Match, set OC1A/OC1B at BOTTOM (non-inverting mode)
1	1	Set OC1A/OC1B on Compare Match, clear OC1A/OC1B at BOTTOM (inverting mode)

Figure 21: We make sure to set the `COM1A1` and `COM1A0` flags in the `TCCRnA` register.

5.4.3 Clamping

One common problem is providing a servo with a PWM value outside the valid range of the servo. For example, our servo has a maximum acceptable pulse width of $2200\mu s$. If we provide our servo

with a pulse width 2300μs, it could damage the servo. We will discuss a simple method for clamping the pulse widths provided to the servo.

The `Clamp` class looks as follows:

```
#ifndef CLAMP_HPP
#define CLAMP_HPP

#include <stdint.h>

struct Bounds {
    int16_t lower;
    int16_t upper;
};

class Clamp {
    Bounds bounds;

public:
    constexpr static Clamp makeFromBounds(Bounds bounds) {
        return Clamp(bounds);
    }

    constexpr Clamp(Bounds bounds): bounds{bounds} {}

    int16_t clamp(int16_t value) {
        return (value < bounds.lower) ? bounds.lower :
            (value > bounds.upper) ? bounds.upper:
            value;
    }

    Clamp() {}
};

#endif
```

The class takes in a `Bounds` and uses the `clamp` member function to output a value between the upper and lower bounds. Since we are using this code on a microcontroller, we want an object of type `Clamp` to be initialized at compile-time. To do this we declare the constructor, and the static factory method, as `constexpr`. To ensure compile-time initialization, we must pass a constant expression or rvalue to either the static factory method or the constructor.

Here is a unit test suite, using CxxTest, for the `Clamp` class:

```
#include <cxxtest/TestSuite.h>
#include "Clamp.hpp"

class TestClamp: public CxxTest::TestSuite {
    Clamp clamp;
    Bounds bounds;

public:
    void setUp() {
        bounds = {
            .lower = 10,
            .upper = 20
```

```
        };

        clamp = Clamp::makeFromBounds(bounds);
    }

    void test_inputAboveUpper_clampsToUpper() {
        int16_t expected = bounds.upper;
        int16_t actual = clamp.clamp(25);
        TS_ASSERT_EQUALS(actual, expected);
    }

    void test_inputBelowLower_clampsToLower() {
        int16_t expected = bounds.lower;
        int16_t actual = clamp.clamp(5);
        TS_ASSERT_EQUALS(actual, expected);
    }

    void test_inputBetweenBounds_noClamp() {
        int16_t expected = 15;
        int16_t actual = clamp.clamp(15);
        TS_ASSERT_EQUALS(actual, expected);
    }
};
```

Each `test_` function in the test suite demonstrates the functionality of the `Clamp` class. It also includes the way we recommend instantiating the `Clamp` class. That is, by using the static factory method as opposed to the constructor.

```
clamp = Clamp::makeFromBounds({
        .lower = 10,
        .upper = 20 });
```

This is purely for readability, so the reader can understand that a `Clamp` object requires a `Bounds`.

Again, after instantiating the `Clamp` class, you can use it as,

```
int16_t clampedValue = clamp.clamp(5); // Returns 10 since the lower bound is 10.
```

5.4.4 PID

As discussed in section 5.1, a PID will provide reactive control to the servo from error feedback. In our system, the camera detects lanes. The further the car is from being centered between those lanes, the greater the error that is input into our steering PID. We've encapsulated the algorithm for a PID in a class.

```
#ifndef PID_HPP
#define PID_HPP

#include <stdint.h>

struct Pid {
    struct Component {
        int16_t proportional;
```

```
        int16_t integral;
        int16_t derivative;
    };

private:
    Component gain;
    Component error;
    int16_t scale;

public:
    int16_t updateError(int16_t error);

    constexpr static Pid makeFromGain(Component gain) {
        return Pid(gain);
    }

    constexpr explicit Pid(Component gain):
        gain{gain}, error{}, scale{1} {}


    constexpr static Pid makeFromScaledGain(int16_t scale, Component gain) {
        return Pid(scale, gain);
    }

    constexpr Pid(int16_t scale, Component gain):
        gain{gain}, error{}, scale{scale} {}

    Pid() {}
};

#endif
```

The PID takes in a structure of gains (proportional, integral, and derivative). Then, when `updateError` is called, it will return the control variable output of the PID.

Since we may have gains that are decimal values, and assuming we don't want to do floating point arithmetic on a microcontroller, we've included a scale variable which divides the final output of the PID. This means if we provide a gain value of `1` (for one of the control gains) and a scale of `10`, then we equivalently have a gain value of `1/10`. To get this behavior, we use the `makeFromScaleGain` to instantiate the `Pid` class, instead of `makeFromGain`.

The definition of the `updateError` function is

```
int16_t Pid::updateError(int16_t newError) {
    error.integral += newError;
    error.derivative = newError - error.proportional;
    error.proportional = newError;

    return (gain.proportional*error.proportional +
            gain.integral*error.integral +
            gain.derivative*error.derivative) / scale ;
}
```

Notice the result is divided by scale, as discussed above. Furthermore, we've provided a unit test that demonstrates how to use this class.

```
#include <cxxtest/TestSuite.h>
#include "Pid.hpp"

class TestPid: public CxxTest::TestSuite {
public:
    void test_errorInput_returnsCorrectedOutput() {
        Pid pid = Pid::makeFromGain({
                .proportional = 1,
                .integral = 2,
                .derivative = 3 });

        int16_t actual = pid.updateError(2);
        int16_t expected = 12;

        TS_ASSERT_EQUALS(actual, expected);
    }

    void test_scaledGain() {
        int16_t scale = 100;

        Pid pid = Pid::makeFromScaledGain(scale, {
                .proportional = 100,
                .integral = 200,
                .derivative = 300 });

        int16_t actual = pid.updateError(2);
        int16_t expected = 12;

        TS_ASSERT_EQUALS(actual, expected);
    }
};
```

Ideally, the user would call `updateError` iteratively though.

5.4.5 USART PID Results

The `main.cpp` code in the code appendix section 8.1.1 depcits the general flow of the program. Ultimately, the Arduino asks for an initial position for the servo (in microseconds), and then the servo will go to that position. After a short delay, the PID will begin updating its control variable output in a for loop. That control variable output is directly used to position the servo. For debugging purposes, the Arduino also sends the PID control variable output over USART, so we can see the values the PID is giving.

The figure below illustrates an example where I send the servo to $800\mu s$, and then the PID corrects the servo's position to $1500\mu s$.

```
>> 800
GOT: 800
1640
1332
1561
1455
1521
1487
1506
1497
1501
1500
1500
1500
1500
1500
1500
>>
```

Figure 22: Note that this PID is not tuned at all, and that the error we are using is not based on a real feedback error from a sensor. This demo is purely to show that the system is prepared for real feedback error from the OpenCV system.

6 Power Supply Design

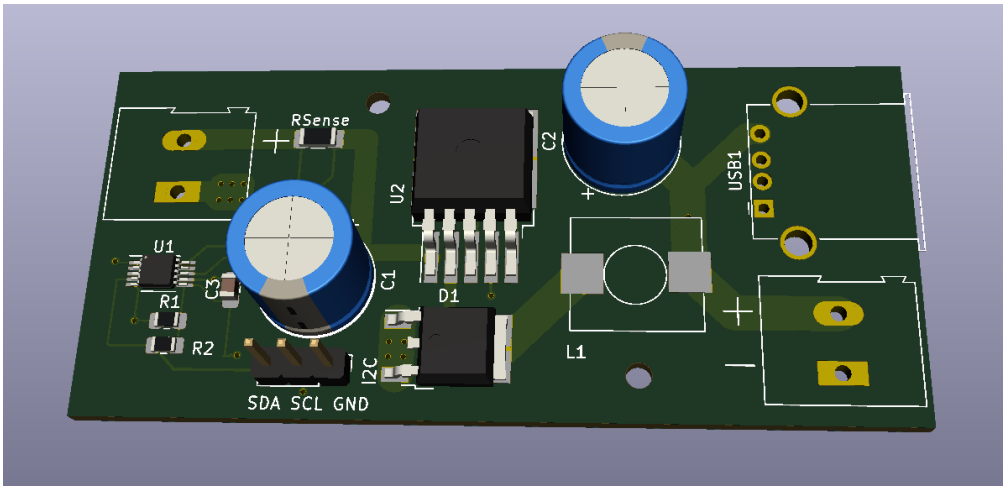


Figure 23: 3D Render of Power Supply

The function of the power supply is to step down the 12v (nominal) battery voltage to 5v for powering the vehicle's systems. The power supply is designed to be capable of outputting 3A with a 9v input (9v is the lowest safe charge for a 3-cell LiPo battery). The system should use much less than 3A on average, but a 3A output provides a large margin of error. The power supply is also capable of measuring the battery voltage and battery current draw and reporting these values over the I2C bus.

6.1 Switching Regulator

The power supply is implemented using a TI LM2596SM-5.0 buck switching regulator. The switching regulator section of the power supply consists of C1, C2, L1, D1 and U2 in the schematic diagram. A switching regulator is used instead of a linear regulator (ex. 7805) for efficiency reasons. A linear regulator has an efficiency of about 40% with a 12v input, whereas a buck converter will have an efficiency of about 80-90%. A linear regulator also produces a large amount of heat, which would require a large heatsink.

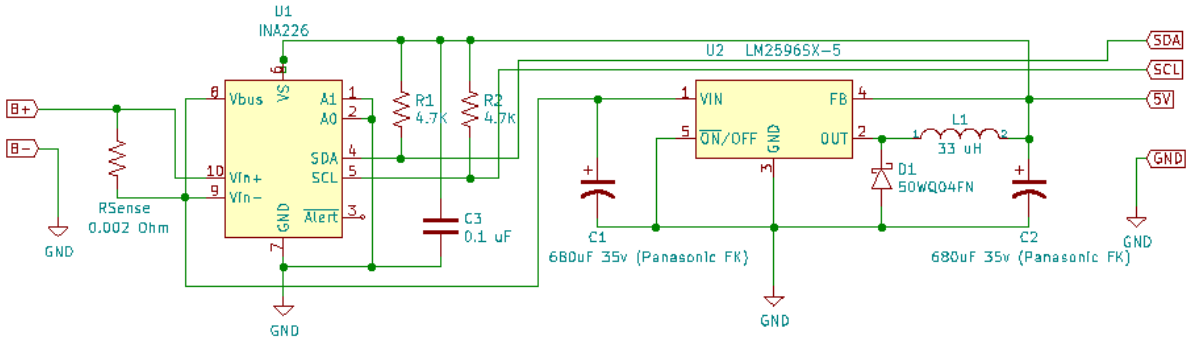


Figure 24: Power Supply Schematic

The regulator operates in a buck configuration, in continuous mode (the inductor current never falls to zero). A buck regulator operates in two cycles, shown in 25. The switching regulator IC (U2) acts as the switch. When the switch is closed, the inductor charges and produces an opposing voltage. This reduces the voltage across the load (following KVL). When the switch opens again, the inductor acts as a current source and discharges through the load. This switching creates voltage spikes, so the output capacitor is used to smooth them out. The output voltage is controlled by varying the ratio of switch on time vs off time.

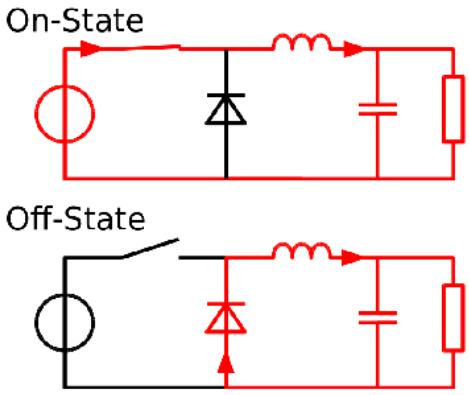


Figure 25: Buck Converter Cycles

The capacitors used for the switching section, C1 and C2 are both low ESR (equivalent series resistance) electrolytics. Low ESR capacitors are used so that they are capable of handling high current spikes without overheating. The inductor has a value of 330uH, which allows the regulator to operate in continuous mode to improve efficiency. The specific inductor (Bourns SRP1038C-330M) was chosen because it is capable of handling 4A of DC current and has a saturation current of 6A. A saturation current of over 3A is needed for the regulator to operate, there were issues with the prototype where the inductor was saturating at about 2A and causing the output voltage to go out of regulation. The diode D1 is a 50WQ04 Shottky diode, which was chosen because of its low forward voltage and fast recovery. Fast recovery is important because the regulator switches at 150kHz and a standard silicon diode would not switch fast enough.

6.2 Power Measurement

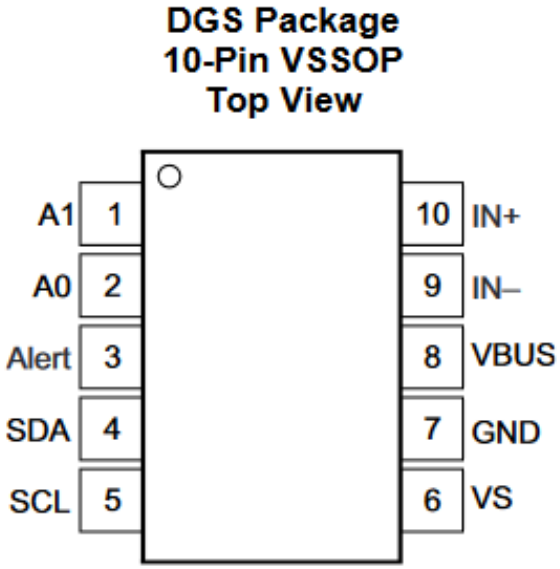


Figure 26: INA226 Pinout

A TI INA226 power monitor IC is used to measure the total power used by the system. This IC measures the total system power by measuring the input voltage and the voltage across a shunt resistor. The shunt resistor (shown as Rsense in the schematic) is connected in series with the input of the switching regulator to provide a more accurate reading. The resistor has a value of 0.002 ohm, and the current flowing through it can be calculated using Ohm's law.

The INA226 communicates using I2C and provides current, voltage and power measurements as well as the voltage across the shunt resistor. It also supports alerts over I2C and a dedicated digital output. The INA226 will be connected into the system wide wired-bus with an I2C address of 0x40. The power and voltage measurements will be used to calculate the remaining battery life, and the voltage measurement will be used to allow the system to put the car into a safe state before the battery is empty.

6.3 PCB Design

A PCB (printed circuit board) was designed for the power supply to provide more physical durability, as well as more stability for the buck converter. Since the buck converter operates at 150kHz traces connecting the inductor, output capacitor and diode need to be as thick and short as possible. The back side of the board uses copper fill as a ground plane to keep the impedance to ground as low as possible, and to act as a heatsink for the LM2596. Multiple vias are placed under the LM2596 to help with thermal and electrical conductivity. Surface mount components are used to keep leads short and make the finished product as compact as possible. Since the board will be hand soldered, larger pads are used for all components.

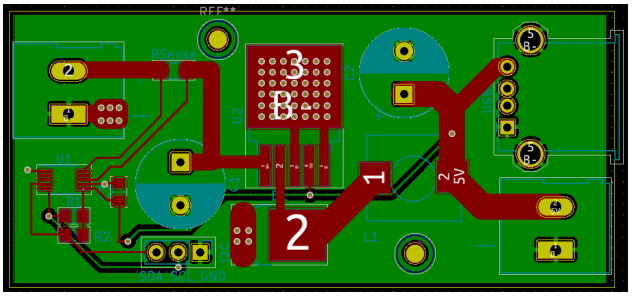


Figure 27: PCB Layout

The PCB and schematic were designed in KiCad, and while component footprints were available for most parts there was nothing matching the inductor so a custom footprint was made. The custom footprint was designed in the KiCad component editor. The component's datasheet was referenced for the size and spacing of the solder pads. The solder pads were widened and lengthened slightly to allow for easier hand soldering, although the spacing between them remains unchanged. Two 2.1 mm diameter mounting holes are included in the layout.

The power input and output connectors are 5mm pitch terminal blocks, in addition to a USB output port. The terminal blocks were picked because they are capable of accepting a 20 to 12 AWG wire which will allow more flexibility when connecting the power supply into the system. The battery will be directly connected to the input terminals, the Raspberry Pi will be connected to the USB port (with a small microUSB cable) and all other 5v devices will be connected to the output terminals. The I2C signals from the INA226 are routed to a pin header, with pull up resistors.

6.4 Testing

To test the power supply, the setup shown below will be used. The voltmeters and ammeters are used to measure the current and voltage at the input and output of the power supply. The potentiometer (or decade resistance box) will be used to place a load on the system. The bench power supply can be used to test the range of input voltages, and an oscilloscope can be placed across the power supply's (DUT) terminals to measure the ripple on the output.

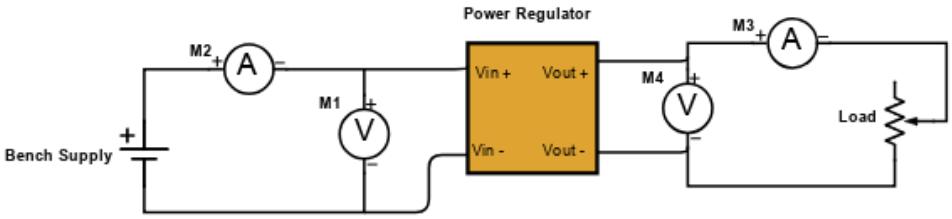


Figure 28: Test Setup

The switching regulator will be tested first by providing 12.6v to the input, which is the batteries maximum voltage. Then the load on the output will be increased from 0A to 3A in 0.5A steps, and the input and output voltages and currents will be recorded, along with the peak-to-peak ripple. This will then be repeated with an input voltage of 12.0v 11v 10v and 9v which will simulate the full range of battery voltages. These tests will ensure that the switching regulator is capable of providing the required output at all possible battery voltages.

Next, the power measurement section will be tested and calibrated. An Arduino will be connected to the I2C terminals on the power supply, and a simple sketch will be used to report the measured voltage, current and power over the serial console. The bench power supply will be adjusted until it is outputting exactly 10v, and then the difference between the measured and actual voltage will be found to get a calibration factor. Next, the power supply will be loaded until the input current is exactly 1.0A. Then the difference between the measured and actual current will be found to find a calibration factor. The calibration factors will then be written into the calibration registers on the INA226.

7 Vehicle Design

7.1 Parts List

The vehicle used in our design is a 1/10 scale rc car. The design was created in 2013 by Daniel Noree, who published the car as open source on thingaverse.com. We are fortunate to use this design, as it provides many features of a real car. However, we will not document any design of the car other than what we have added. The car is equipped with all-wheel drive, front and rear open differentials, and a suspension system. The chart below shows 3D printed parts only. Besides the part name and quantity, we have provided some of the print settings that we have chosen with respect to durability. The percent infill represents the ratio of material to air in each part. The layer height represents the z-axis increment of each layer. Higher infil and smaller layer height increase the strength substantially. It is important to use these settings on smaller parts such as gears and high stress components. Likewise, larger parts such as the frame do not require such settings. Note, all parts were printed in PLA material except gearing, which was printed in ABS. PLA gears would strip before ABS gears because of the lower melting point.

Part	Quantity	% Infil	Layer Height (mm)
Axle Shaft	4	100	0.15
Axle Shaft Cover	4	100	0.15
Battery Holder	1	40	0.3
Bevel Drive Gear	2	100	0.15
Bumper	1	40	0.3
Central Diff Case 3	1	60	0.3
Central Diff Case 5	1	60	0.3
Central Drive Shaft	2	100	0.15
Central Drive Shaft Joint	4	100	0.15
Chassie Plate Front	1	40	0.3
Chassie Plate Rear	1	40	0.3
C-Hub	2	80	0.2
Diff Case Shape	1	40	0.2
Diff Housing Lower (modified)	2	40	0.2
Diff Housing Upper (modified)	2	40	0.2
Drive Gear	1	100	0.15
Left Shield Assembly	1	40	0.3
Pivot Shaft	2	100	0.2
Rear C-Hub	2	60	0.2
Right Shield Assembly	1	40	0.3
Servo Arm	1	40	0.2
Servo Holder	1	40	0.3
Shock Tower	2	50	0.2
Simple Body Holder	1	40	0.3
Spur Gear	1	100	0.15
Steering Stabilizer	1	80	0.2
Steering Block	2	60	0.2
Steering Part 1	1	40	0.15
Steering Part 2	1	40	0.15
Steering Part 3	1	40	0.15
Steering Pin	2	100	0.15
Steering Plate	1	60	0.2
Top Deck Front	1	40	0.3
Top Deck Rear	1	40	0.3
Turnbuckle (modified)	4	60	0.15
Turnbucle Spacer	4	60	0.15
Wheel Hub Hex 12mm	4	100	0.15
Wishbone Front (modified)	2	50	0.2
Wishbone Rear (modified)	2	50	0.2
Camera Mount	1	40	0.2

Apart from the printed parts, we also purchased various prebuilt components for the vehicle. The table below shows the sourced parts.

Part	Quantity	Material	Source
Ball Joint	5	Zinc	Redcat Racing
5x10x4 mm Bearing	8	Steel	Fast Eddy
6x12x4 mm Bearing	4	Steel	Fast Eddy
10x15x4 mm Bearing	10	Steel	Fast Eddy
CS M3x5 mm Screw	5	Zinc	Bolt Depot
CS M3x8 mm Screw	54	Zinc	Bolt Depot
CS M3x10 mm Screw	6	Zinc	Bolt Depot
CS M3x14 mm Screw	4	Zinc	Bolt Depot
CS M3x16 mm Screw	6	Zinc	Bolt Depot
HSC M3x8 mm Bolt	3	Zinc	Bolt Depot
HSC M3x10 mm Bolt	14	Zinc	Bolt Depot
HSC M3x14 mm Bolt	8	Zinc	Bolt Depot
HSC M3x16 mm Bolt	14	Zinc	Bolt Depot
M3 Nut	26	Zinc	Bolt Depot
M4 Nut	4	Zinc	Bolt Depot
2x150 mm Rod	10	SS	Sutemribor
3x120 mm Rod	5	SS	Uxcell
188004 Shocks	4	N/A	Hobby-Park-US
188015 Drive Shaft	4	SS	Redcat Racing
ZD 7170 Open Differential	2	N/A	jsbay88
83mm Drive Axle Dogbone	2	SS	RC-Express

7.2 Vehicle Assembly

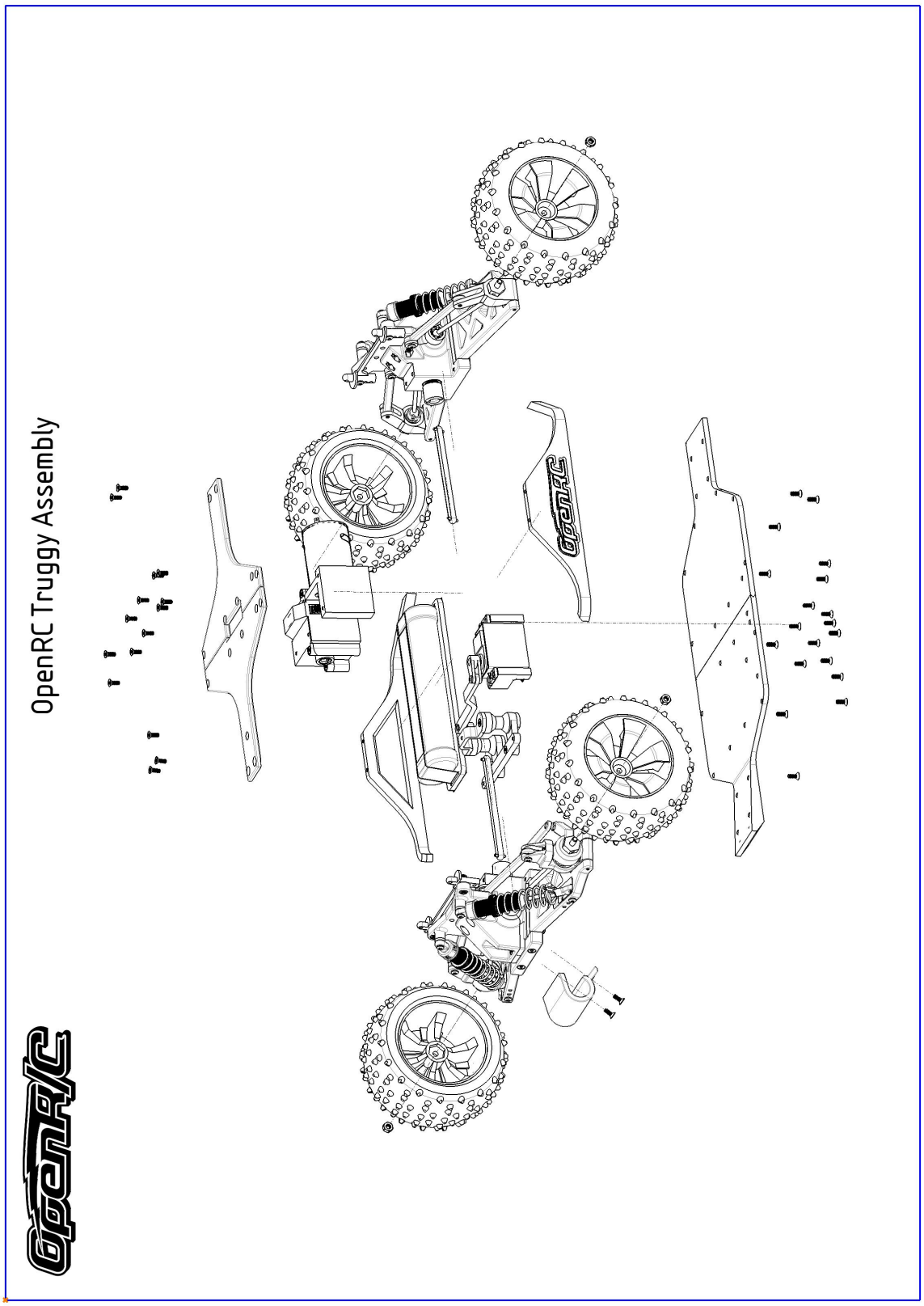


Figure 29: Overall Assembly

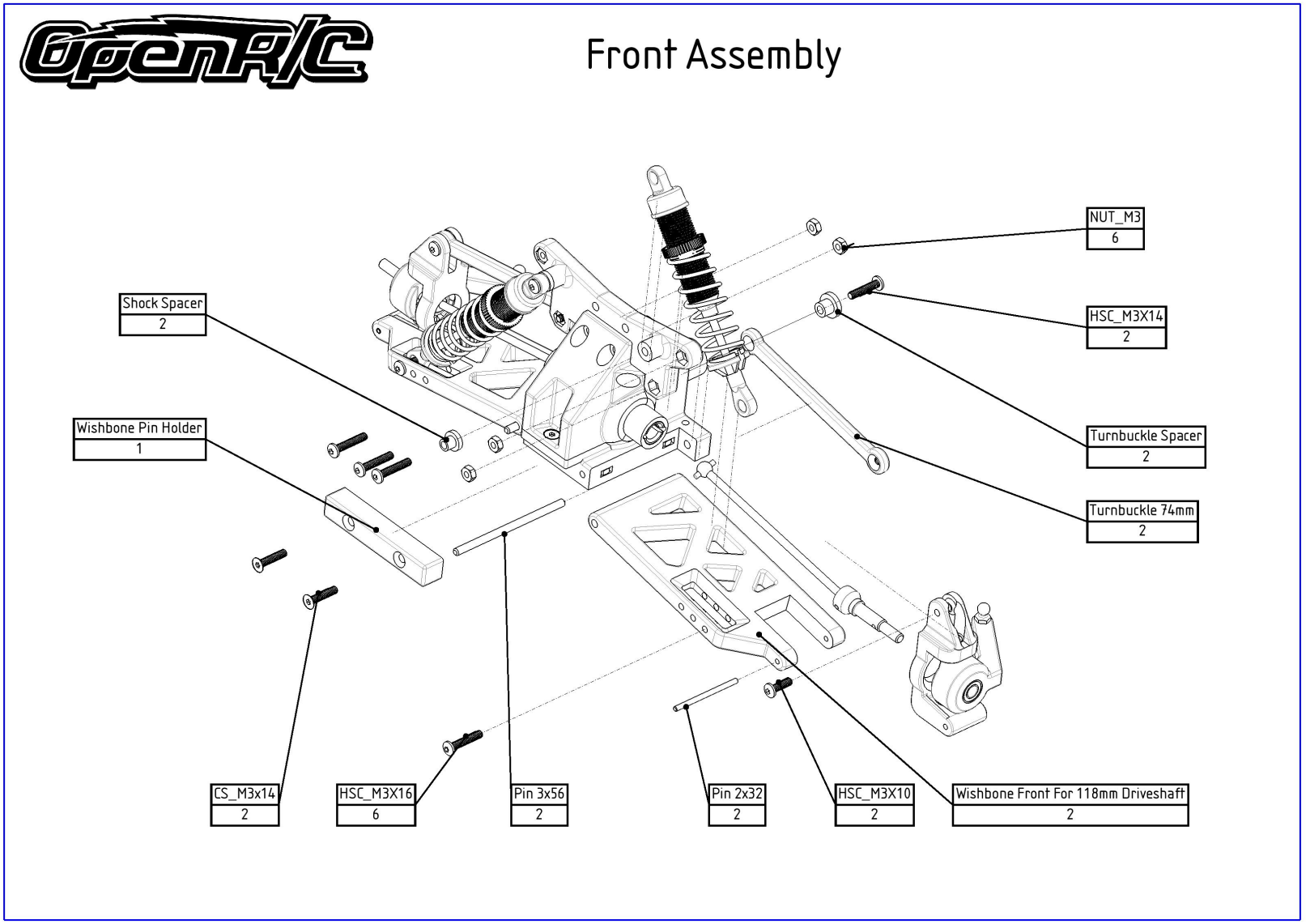


Figure 30: Front Assembly

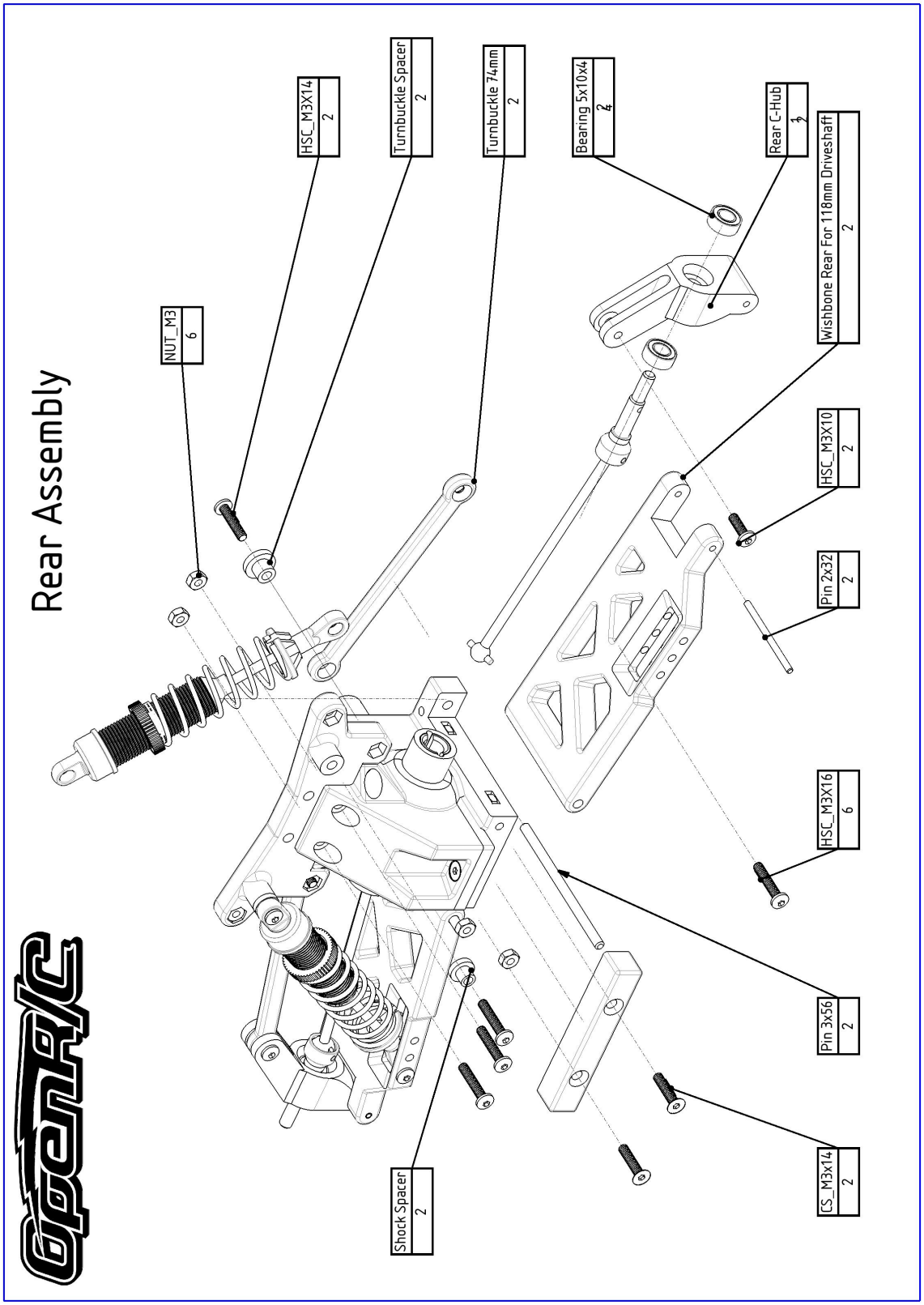


Figure 31: Rear Assembly

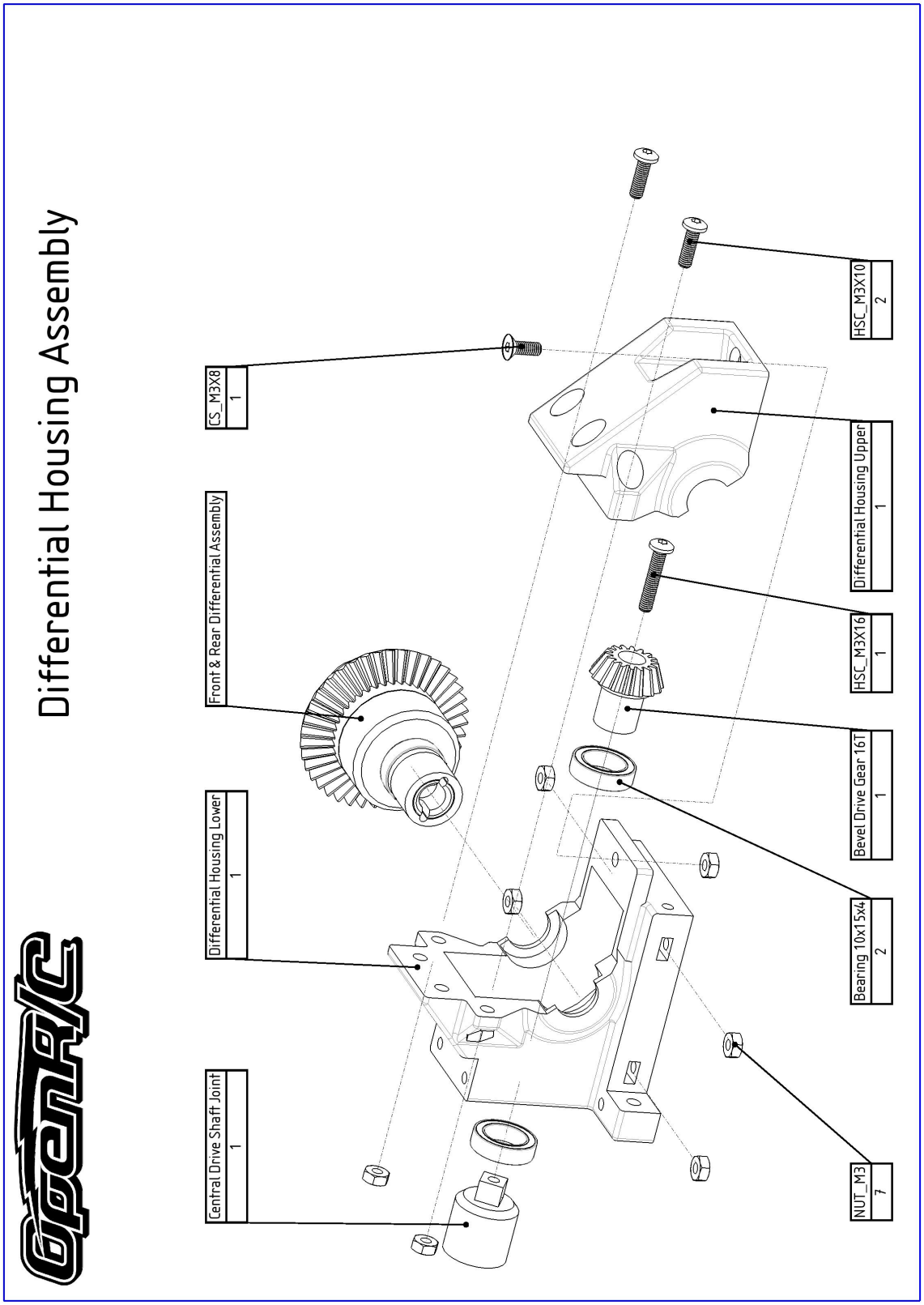


Figure 32: Differential Housing

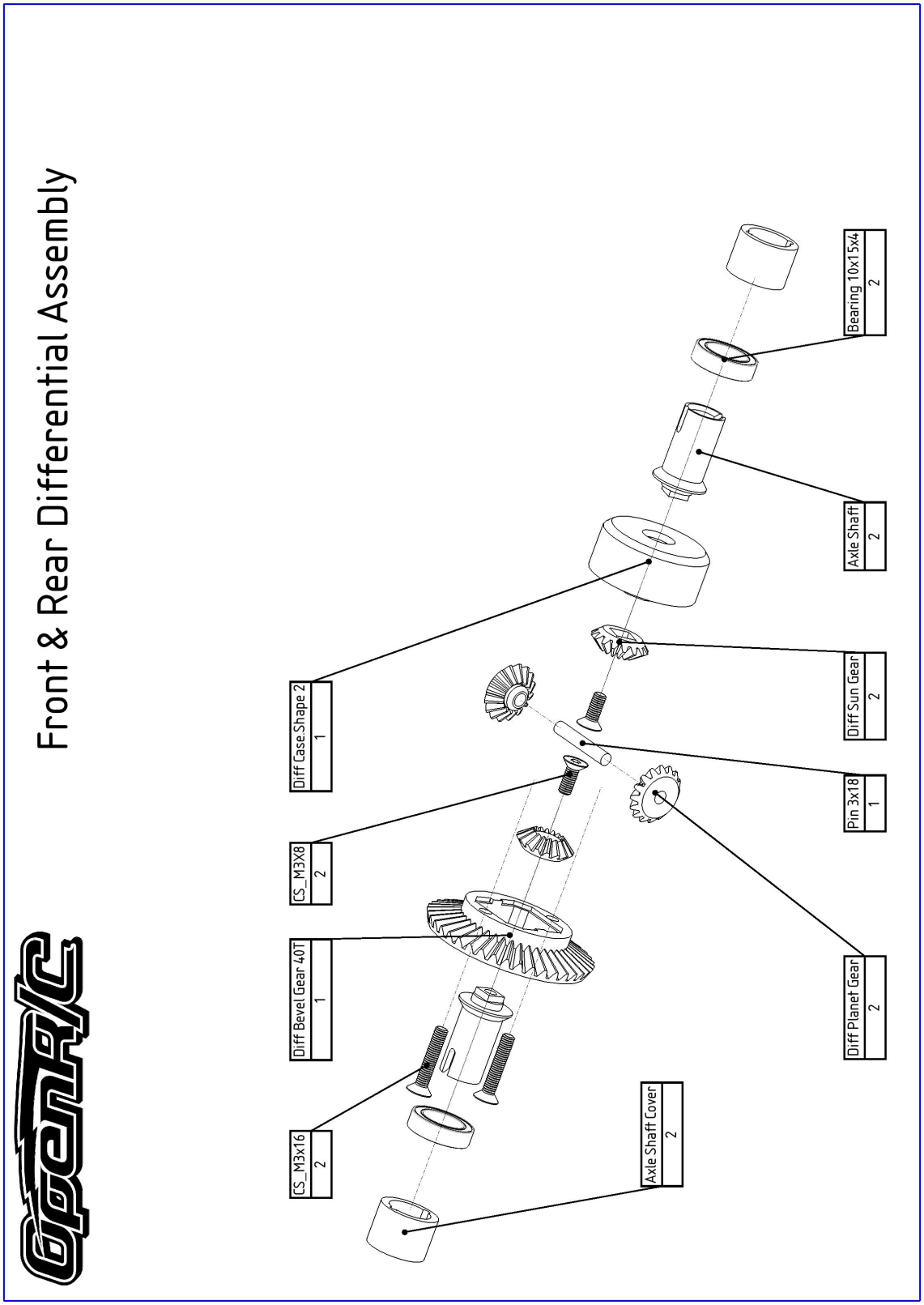


Figure 33: Differential Assembly

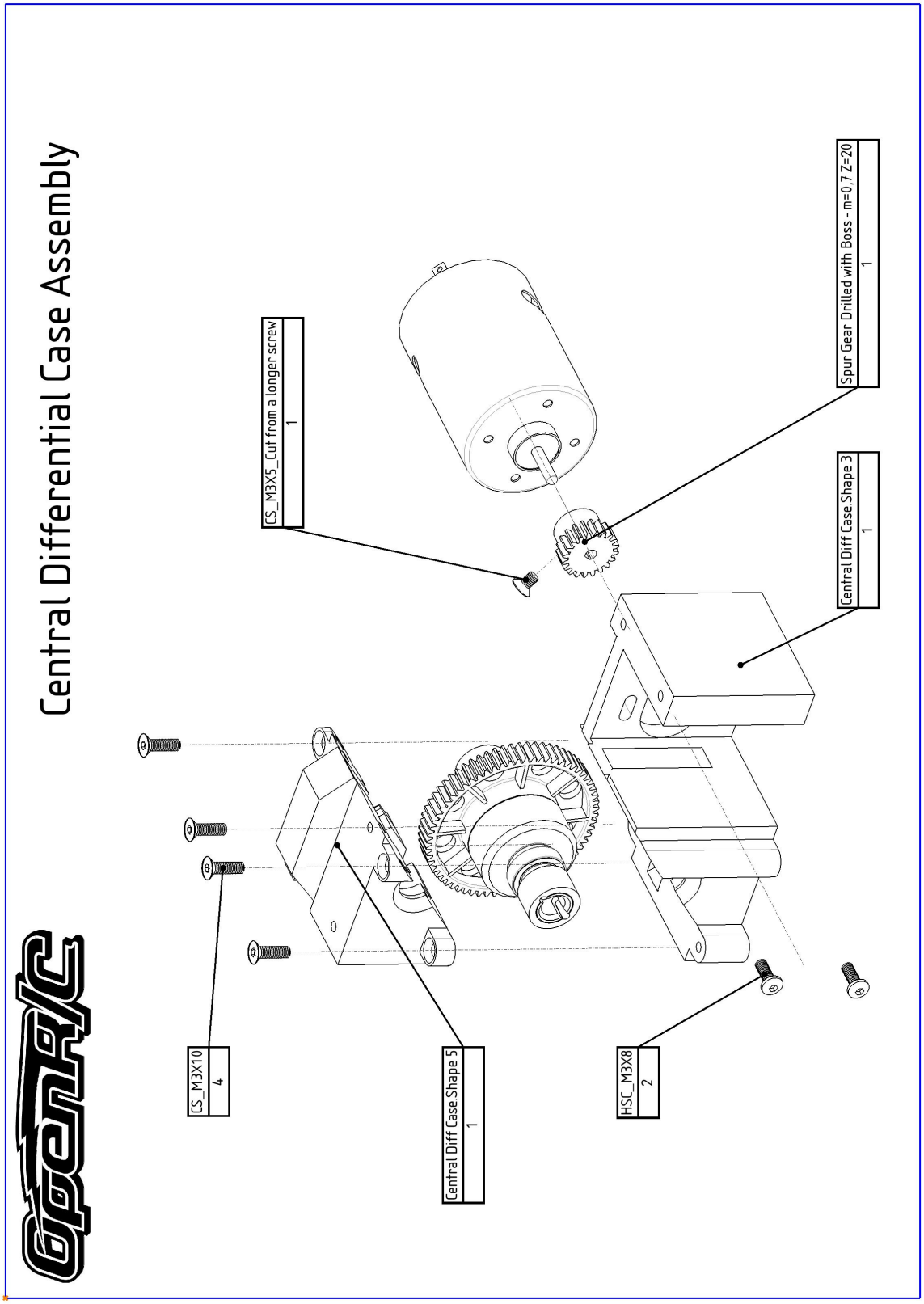


Figure 34: Central Differential Housing

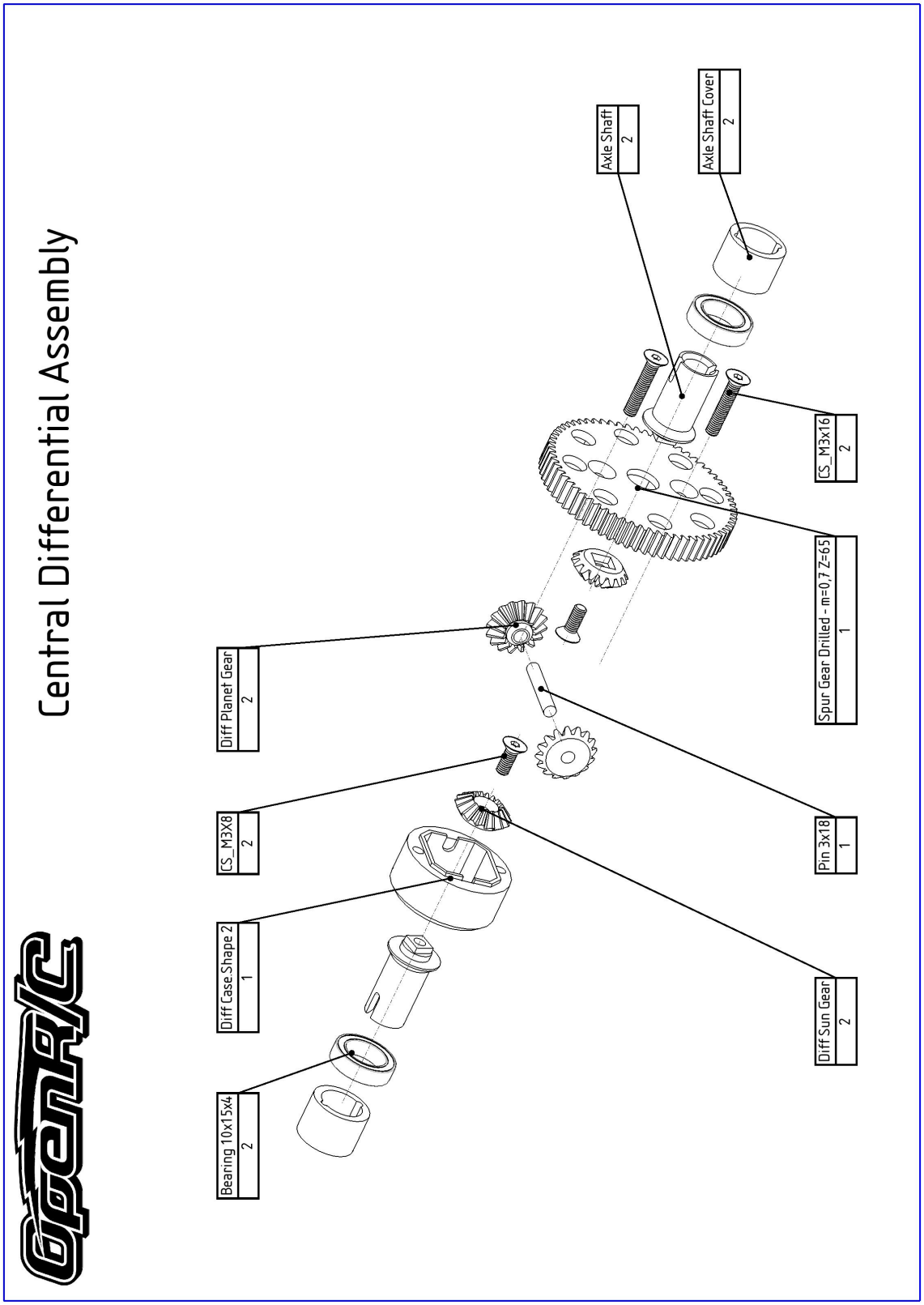


Figure 35: Central Gearbox

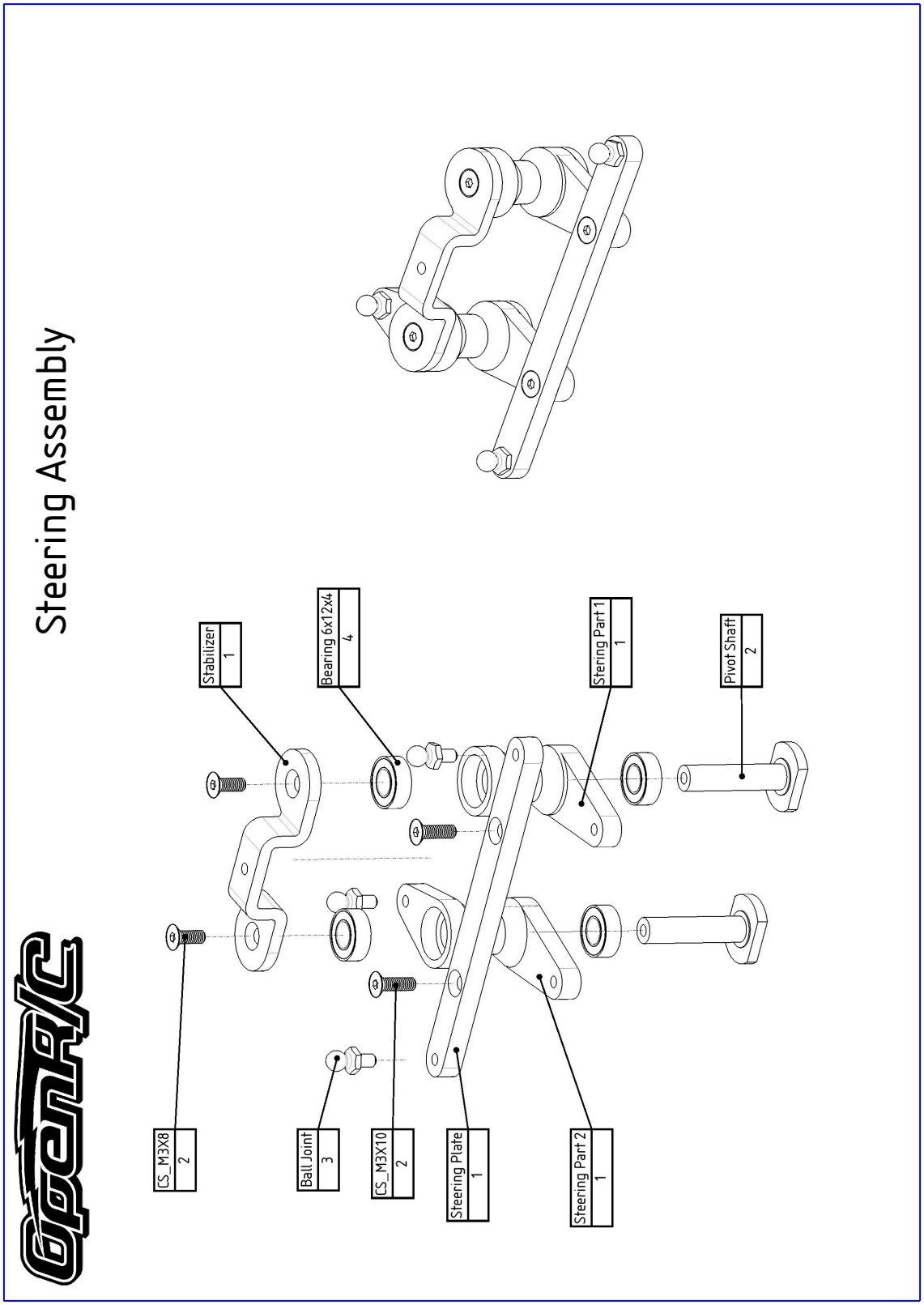


Figure 36: Steering Assembly

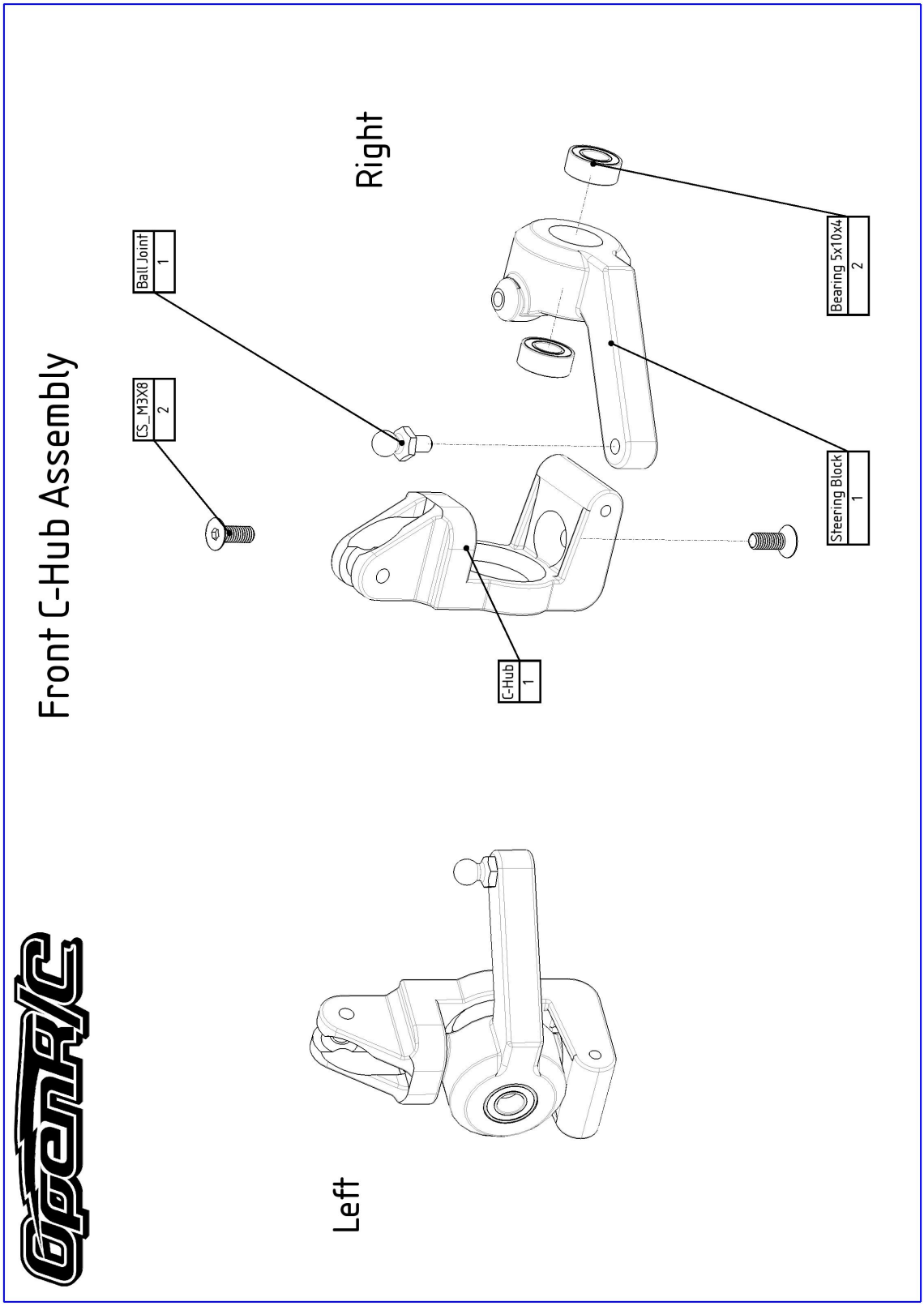


Figure 37: Front C-Hub Assembly

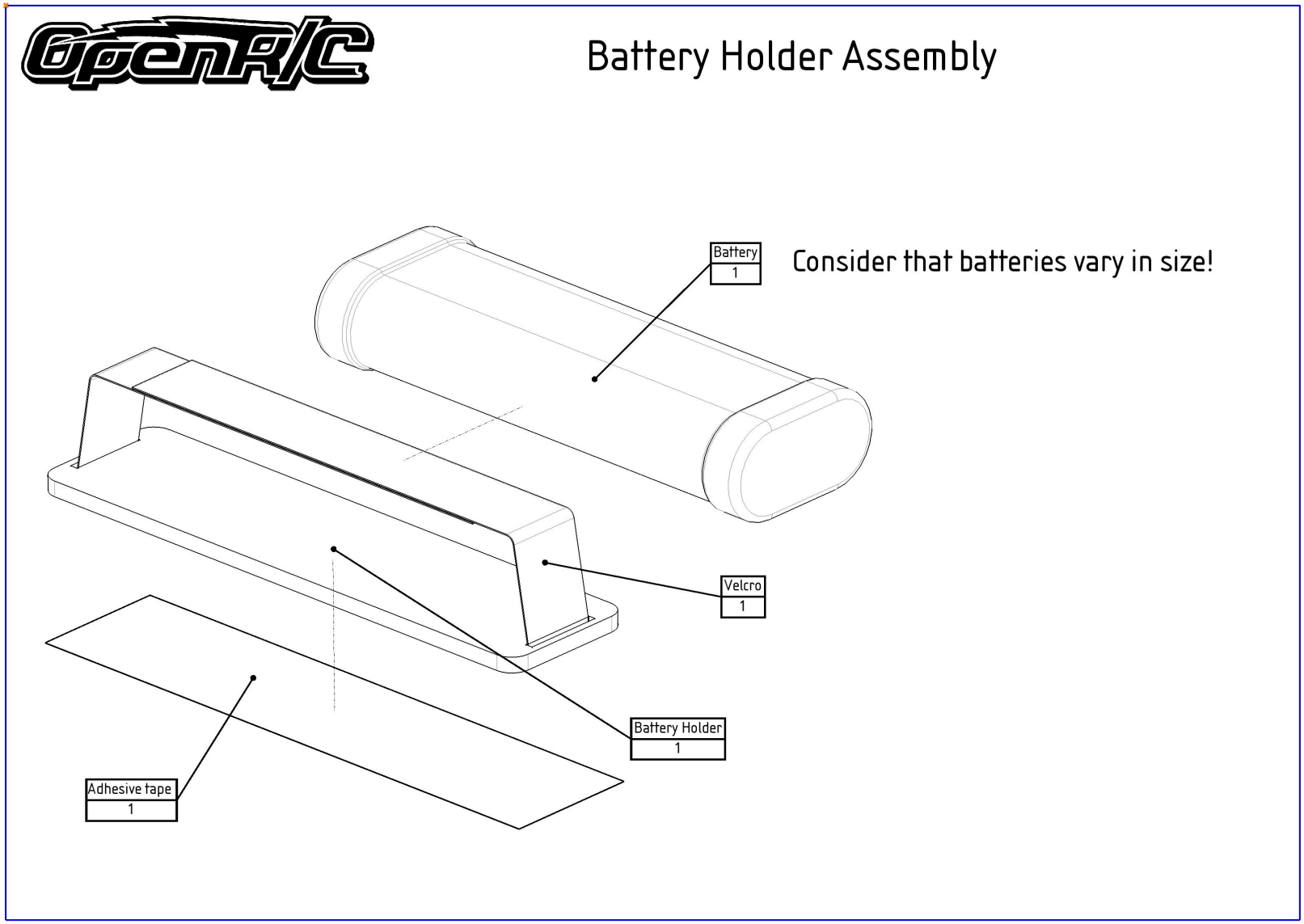


Figure 38: Battery Holder

8 Appendix: Code

8.1 Steering PID

8.1.1 main.cpp

```
#include <avr/io.h>
#include <util/delay.h>
#include <stdlib.h>
#include "Pid.hpp"
#include "Clamp.hpp"
#include "String.hpp"
#include "usart.hpp"

constexpr uint8_t prescaler = 8;
constexpr uint32_t clockFrequency = F_CPU;
constexpr uint8_t pwmFrequency = 50;
constexpr uint32_t baud = 9600;

static constexpr uint32_t microsToCycles(uint16_t micros) {
    constexpr uint32_t unitConversion = 1E6;
    return (clockFrequency/unitConversion/prescaler) * micros;
}

static constexpr uint32_t hertzToCycles(uint16_t hertz) {
    return clockFrequency/prescaler/hertz;
}

static void setupServoPwm() {
    DDRB |= 1 << PINB1; //Set pin 9 on arduino to output

    TCCR1A |=
        1 << WGM11 | //PWM Mode 14 (1/3)
        1 << COM1A0 | //Inverting Mode (1/2)
        1 << COM1A1; //Inverting Mode (2/2)

    TCCR1B |=
        1 << WGM12 | //PWM Mode 14 (2/3)
        1 << WGM13 | //PWM Mode 14 (3/3)
        1 << CS11; //Prescaler: 8

    //50Hz PWM to cycles for servo
    ICR1 = hertzToCycles(pwmFrequency)-1;
}

int main() {
    usart::setup(clockFrequency, baud);
    setupServoPwm();

    Clamp steeringClamp = Clamp::makeFromBounds({
            .lower = 800,
            .upper = 2200 });
    Pid steeringPid = Pid::makeFromScaledGain(10, {
            .proportional = 10,
            .integral = 0,
            .derivative = 2 });

    String<10> message;
```

```
    char currentChar;
    int16_t idealServoMicros = 1500;
    while(1) {
        currentChar = usart::getChar();

        if (currentChar != '\n') message.append(currentChar);
        else {
            usart::print("GOT: ");
            usart::print(message);
            usart::print('\n');

            int16_t initialServoMicros = atoi(message);
            int16_t servoMicros = steeringClamp.clamp(initialServoMicros);
            OCR1A = ICR1 - microsToCycles(initialServoMicros);
            _delay_ms(1000);

            int16_t error = idealServoMicros - servoMicros;
            for (uint32_t i = 0; i < 15; ++i) {
                servoMicros = steeringPid.updateError(error) + servoMicros;

                String<10> buffer;
                itoa(servoMicros, buffer, 10);
                usart::print(buffer); usart::print('\n');

                servoMicros = steeringClamp.clamp(servoMicros);
                OCR1A = ICR1 - microsToCycles(servoMicros);
                _delay_ms(100);

                //TODO replace with actual feedback error
                error = idealServoMicros - servoMicros;
            }

            message.clear();
            usart::print(">> ");
        }
    }
}
```

8.1.2 usart.hpp

```
#ifndef USART_HPP
#define USART_HPP

#include <avr/io.h>
#include <stdint.h>

namespace usart {
    void setup(uint32_t clockFrequency, uint16_t baud) {
        uint8_t ubrr = clockFrequency/16/baud - 1;
        UBRR0H = ubrr >> 8;
        UBRR0L = ubrr;

        //Enable Transmitter & Receiver
        UCSR0B = 1 << RXEN0 | 1 << TXEN0;
        //Frame Format: 8 data, 2 stop
        UCSR0C = 1 << USBS0 | 3 << UCSZ00;
```

```
    }

    void print(char c) {
        while (!(UCSR0A & 1<<UDRE0));
        UDR0 = c;
    }

    void print(char* string) {
        while (*string) print(*string++);
    }

    char getChar() {
        while (!(UCSR0A & 1<<RXC0));
        return UDR0;
    }
}

#endif
```

8.1.3 Pid.hpp

```
#ifndef PID_HPP
#define PID_HPP

#include <stdint.h>

struct Pid {
    struct Component {
        int16_t proportional;
        int16_t integral;
        int16_t derivative;
    };

private:
    Component gain;
    Component error;
    int16_t scale;

public:
    int16_t updateError(int16_t error);

    constexpr static Pid makeFromGain(Component gain) {
        return Pid(gain);
    }

    constexpr explicit Pid(Component gain):
        gain{gain}, error{}, scale{1} {}


    constexpr static Pid makeFromScaledGain(int16_t scale, Component gain) {
        return Pid(scale, gain);
    }

    constexpr Pid(int16_t scale, Component gain):
        gain{gain}, error{}, scale{scale} {}
```

```
    Pid() {}
};

#endif
```

8.1.4 Pid.cpp

```
#include "Pid.hpp"

int16_t Pid::updateError(int16_t newError) {
    error.integral += newError;
    error.derivative = newError - error.proportional;
    error.proportional = newError;

    return (gain.proportional*error.proportional +
            gain.integral*error.integral +
            gain.derivative*error.derivative) / scale ;
}
```

8.1.5 Clamp.hpp

```
#ifndef CLAMP_HPP
#define CLAMP_HPP

#include <stdint.h>

struct Bounds {
    int16_t lower;
    int16_t upper;
};

class Clamp {
    Bounds bounds;

public:
    constexpr static Clamp makeFromBounds(Bounds bounds) {
        return Clamp(bounds);
    }

    constexpr Clamp(Bounds bounds): bounds{bounds} {}

    int16_t clamp(int16_t value) {
        return (value < bounds.lower) ? bounds.lower :
            (value > bounds.upper) ? bounds.upper:
            value;
    }

    Clamp() {}
};

#endif
```

8.1.6 String.hpp

```
#ifndef STRING_HPP
#define STRING_HPP

#include <stdint.h>

template<int16_t capacity>
class String {
    int16_t size;
    char string[capacity];

public:
    void append(char c) {
        if (size < capacity) string[size++] = c;
    }

    String(): size{0}, string{0} {}

    void clear() {
        while (size != 0) string[--size] = 0;
    }

    operator char*() { return string; }
    operator const char*() { return string; }
};

#endif
```

8.1.7 Makefile

```
BIN=sweep

CPP = $(BIN).cpp $(wildcard *.cpp)
OBJ = $(CPP:%.cpp=%.o)
DEP = $(OBJ:%.o=%.d)

CROSS_COMPILE=avr-
CXX=$(CROSS_COMPILE)g++
OBJCOPY=$(CROSS_COMPILE)objcopy
CXXFLAGS=-Os -Wall -DF_CPU=$(F_CPU) -mmcu=atmega328p -std=gnu++17

F_CPU=16000000UL
PORT=/dev/ttyACM0
DEVICE=ATMEGA328P
PROGRAMMER=arduino
FLASH_BAUD=115200
COM_BAUD=9600

TESTS = TestPid.hpp TestClamp.hpp
TEST_SOURCES= Pid.cpp
TEST_RUNNER = runner



$(BIN).hex: $(BIN).elf
```

```
    ${OBJCOPY} -O ihex -R .eeprom $< $@

$(BIN).elf: $(OBJ)
    $(CXX) $(CXXFLAGS) -o $@ $^

-include $(DEP)

%.o: %.cpp
     $(CXX) $(CXXFLAGS) -MMD -c $< -o $@

flash: $(BIN).hex
    avrdude -c $(PROGRAMMER) -P $(PORT) -p $(DEVICE) -b $(FLASH_BAUD) -U flash:w:$<

test: $(TESTS)
    cxxtestgen --error-printer -o $(TEST_RUNNER).cpp $(TESTS)
    g++ -o $(TEST_RUNNER) -I$CXXTEST $(TEST_RUNNER).cpp $(TEST_SOURCES)
    ./$(TEST_RUNNER)

com:
    picocom -b $(COM_BAUD) $(PORT) -p 2


clean:
    rm -f ${BIN}.elf ${BIN}.hex $(OBJ) $(TEST_RUNNER)* $(DEP)

.PHONY: clean
```