Autonomous Vehicle Prototype
Group B7

1st Jad Al Jourdi
Systems Engineering and Prototyping
Electronic Engineering Department
Hochscule Hamm Lippstadt University
Lippstadt, Germany
jad.al-jourdi@stud.hshl.de

2nd Imad Chaar
Systems Engineering and Prototyping
Electronic Engineering Department
Hochscule Hamm Lippstadt University
Lippstadt, Germany
imad.chaar@stud.hshl.de

3rd Maxim Speczyk
Systems Engineering and Prototyping
Electronic Engineering Department
Hochscule Hamm Lippstadt University
Lippstadt, Germany
maxim-emile.speczyk@stud.hshl.de

4th Stephanie Chinenye Okosa
Systems Engineering and Prototyping
Electronic Engineering Department
Hochscule Hamm Lippstadt University
Lippstadt, Germany
stephanie-chinenye.okosa@stud.hshl.de

***Abstract* — Precision farming is a term used to describe the method of technology implementation in agriculture to increase productivity. Autonomous vehicles are one of the tools used to achieve this evolution. In this document, we develop an autonomous vehicle prototype which aims at collecting bales of straw from specified locations on test track coordinates and then transporting them to the desired location with the test track environment. The system entails all electronic components to be assembled within a body composed of 6cm thick plywood parts, connected together using screws and bolts.**

Keywords* — *autonomous, prototype, technology, components, communication, precision, farming.

I. Introduction

The advancements in agricultural practices such as farming, have been evolving over the years through the revolutionization of computers and electronic components [1]. Autonomous vehicle research has increased over the years because of the need to boost productivity, efficiency and enhance operational safety. An autonomous vehicle prototype, which follows the test tracks of the environment and collects bales of straw, is being implemented to evaluate our capabilities in precision farming. Precision farming leads to less manual labour, with a higher outcome in productivity, which can also have positive impact on the environment.

A. Purpose

The constant demand for crop yield production is rapidly growing along with the need to increase productivity, which in turn causes the need for more advanced tools to meet the supply. The aim of an autonomous vehicle prototype is to increase productivity in the farm by collecting bales of straw precisely from specified coordinates. The prototype must be able to drive through the test track, count each colour on the track, take turns at specific coordinates, as well as bring the bales of straw to a certain specified position.

This document is intended for any individual interested in automation in the agricultural sector of precise farming and would like to make positive contributions. Future models of this prototype could use GPS coordinates for higher precision at a larger scale.

B. Scope

The Autonomous vehicle functions with the aid of UART communication via the Arduino Uno and the colour coordinates are viewed on the serial monitor. The autonomous vehicle is required to follow the colour lines on the test track, count each coordinate through its path, and collect bales of straw from specified coordinates. The bales of straw can only be moved to a specified location but cannot be picked up. The aim of this project is to maximize productivity in farming and enhance technology in the agricultural sector of precise farming.

C. Definition

- Precise farming: An approach to farming management that focuses on observing, measuring, and responding to navigational instructions to attain higher crop productivity.
- Autonomous vehicle: A programmable vehicle that senses the environment and operates without human assistance.
- UART: Stands for Universal asynchronous reception transmission. A protocol allowing the Arduino Uno to communicate with serial devices.
- Prototype: A first version of the vehicle from which other forms are developed.

D. Overview

This document contains a detailed description of the design, various functions of the autonomous vehicle prototype as well as, system modelling diagrams such as the use case, state machine, and block diagrams. The organization begins with an overall description of the features, followed by the hardware characteristics and other related software interfaces as well as function of the prototype. Assumption and constraints are also stated. To conclude, a requirement diagram is shown, to enable designers design a system, testers to test that its satisfactory and attributes of the system are explained.

II. Overall Description

There are two aspects of this project:

1. Design:
    a. Creating a CAD model, with the various parts of the electronic components.
        i. The size of the vehicle prototype was limited to 20cm x 30cm x 20cm.
        ii. Selecting a sci-fi design style and creating a 3D model for the prototype.
    b. Assembling the electronic components on Tinker CAD.
    c. Cabling of the autonomous vehicle prototype.

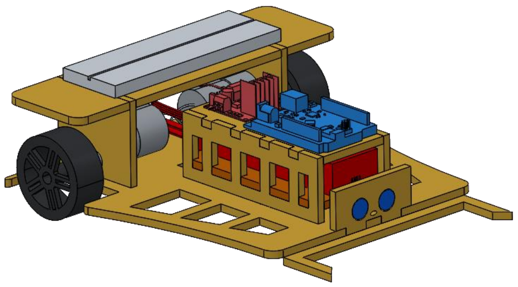

Figure 1: CAD Model

2. Programming:
    a. Programming a working code and simulating in the environment of Tinker CAD.
    b. Using the Arduino 1.8.19 version to upload the code to the Arduino through a COM5 Arduino Uno port.

A. *System Perspective*

The autonomous vehicle prototype consists of electronic components such as ultrasonic, infrared, color sensors, motor controller, microcontroller, battery, breadboard, switch, motor wheels, and front wheel which are connected by wires. Below is a block diagram showing their interconnection.

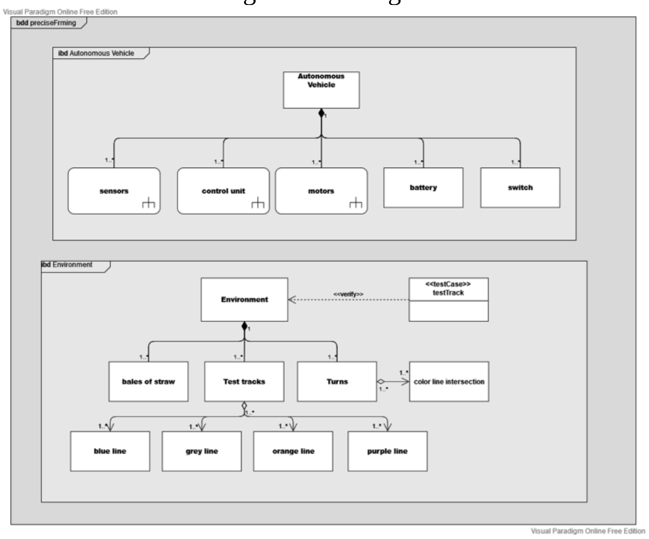

Figure 2: Block Diagram

B. *System function*

Figure 3 defines the major role of this autonomous vehicle prototype which is to collect bales of straw from a specific coordinate and transport them to a specified location. For this to happen, the vehicle must be able to drive through the instructed path, make turns at specified intersections, count the color of the lines along its path and come to a stop once the specified location is reached.

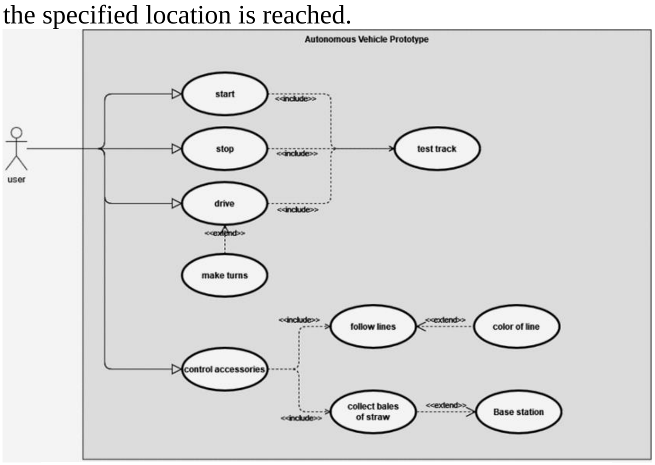

Figure 3: Use Case Diagram

Some electronic components have to be checked/replaced before the vehicle is switched on:

- Battery: To ensure that the motor controller generates enough voltage to power the motor wheels.

Also, the vehicle must be switched off while uploading the code on the microcontroller and when not in use.

C. *User Interface*

The user interface is the Arduino software IDE 1.8.19. This allows the user to write programs and upload to the Arduino Uno board. The user can save written codes on their desktops and to the cloud, which enables them to access the code from any device. The software automatically updates to the latest version without the user having to do so.

D. *Hardware Interface*

The autonomous vehicle prototype consists of various electronic components.

1) *Microconroller:*

The Arduino Uno consists of 14 digital input/output pins, a USB connection, a power jack, an ICSP header and a reset button. It can be connected to a computer with a USB or powered by a battery.

Figure 4: Arduino Uno Wi-Fi R2

2) Motor controller:

It is an extension board that controls up to two direct-current motors at once with a constant voltage from 5V to 35V.

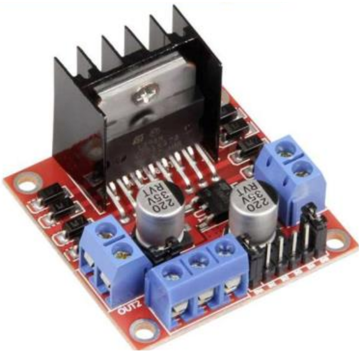

Figure 5: SBC-MotorDriver2 (L298N)

3) Infra red sensor:

Used to differentiate the white and black lines on the track. It can be calibrated by tuning the white knob seen in the diagram below:

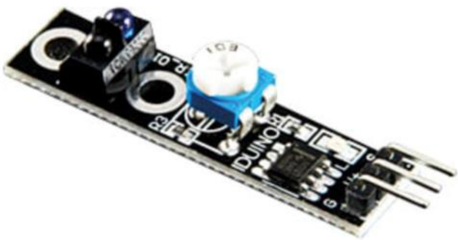

Figure 6: Irduino line tracking module (ST1140)

4) Motors:

Drives the motor wheels.

a) Constraints

To avoid internal gear motor damage, always check the screw size and length on external dimension drawing before assembling.

Figure 7: 2x Modelcraft RB 35 gearbox with motor (1:30)

5) Front and Rear Wheel:

Responsible for the vehicle’s motion.

Figure 8: Front wheel

Figure 9: Rear Wheel

6) Color sensor:

Detects the different color lines frequency on the test track.

Figure 10: ARD Color sensor (TCS3200)

7) Battery:

A 12V battery used to power the electronic prototype.

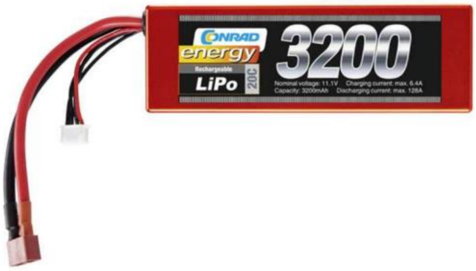

Figure 11: LiPo 11.1v 3200mAh

8) Breadboard:

Holds the electronic components together using wires.

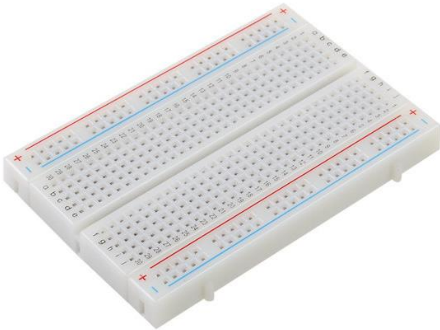

Figure 12: Breadboard

9) Ultrasonic sensor:

It senses and relays the distance of the object from the vehicle.

Figure 13: Ultrasonic sensor Iduino ST1099 (HC SR-04)

E. User Characteristics

The intended user of the system is anyone who has passion for precision farming and can operate an autonomous vehicle prototype. The individual should have knowledge in C++ programming, must understand how to operate, and maintain the prototype.

F. Constraints

There are various factors that could decrease the performance of the vehicle prototype such as:

1) The light intensity of the environment can affect the color frequency from the color sensor thus, causing wrong movement of the autonomous vehicle prototype on test tracks.

G. Assumptions and dependencies

1) It is assumed that all electronic components have been properly assembled.

2) The speed of the vehicle depends on the voltage of the battery.

III. SPECIFIC REQUIREMENTS

The functional requirement diagram shown in Figure 14, elaborates the system's requirements such as motion, navigation, and detection. The system should operate by following the line and making turns at specified coordinates, mapping the environment, and counting the colors throughout its movement on the path. Finally, the system should detect the color of lines, collect, and transfer the bales of straw throughout its movement on the path to the specified coordinates.

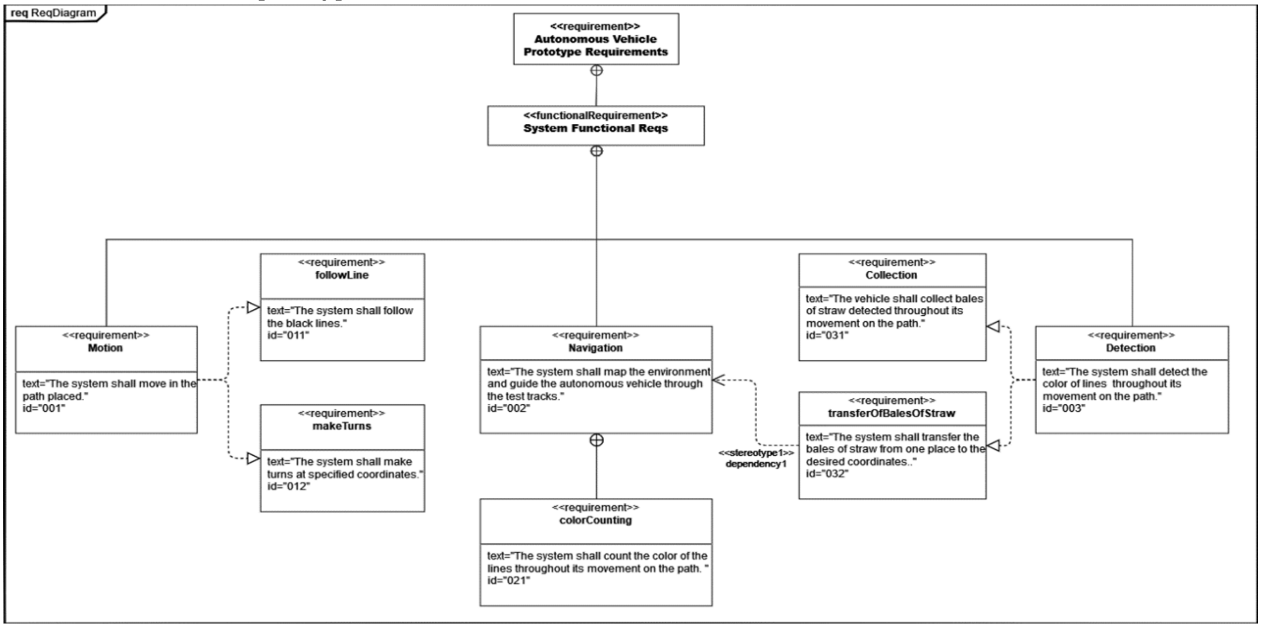

Figure 14: Functional Requirement Diagram

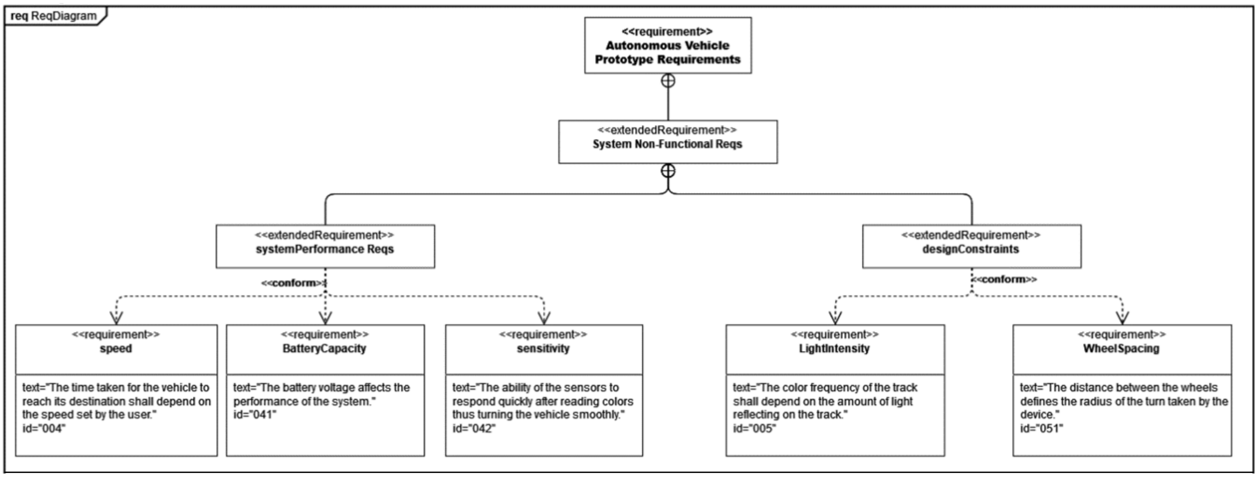

Figure 15: Non-Functional Requirement Diagram

Figure 15 above describes the non-functional requirement diagram, indicating the performance requirements and design constraints of the autonomous vehicle prototype.

A. *Design*

The software used to develop the design were Rhino 7, Solidworks, and Blender. There were several limiting factors that inhibited the design such as dimensions ranging from 20cm x 30cm x 20cm as well as the material, either being plywood or 3D printed PLA. The functional design was made entirely on Solidworks, which lead to a majority of the sci-fi style design parts to be designed on it again; however, the elevated base was designed on blender and the motor cover plate was designed on Rhino 7.

One of the requirements for uploading the designed components was to be able to upload the parts as step files, which both Blender and Solidworks were capable of. Designing on solid works proved to be of ease when designing in 2D with plywood in mind; however, Blender deemed itself superior in polygon manipulation and designing complex structures. Blender's capabilities allowed multiple faces to be inserted and extruded with ease on the elevated base. The elevated base and its lid were designed entirely on blender and exported as step files as well.

Rhino was used to apply a lattice hinge pattern on a sheet of plywood, giving it the capability to bend the flat sheet of plywood over ninety degrees. A Straight Lattice was first applied on a test sheet of plywood, but the test proved that the spacing was too small as the test sheet only bent around a 30-degree angle before showing signs of stress. That, in turn, caused the second test to cut the space between the lattice hinges by ¼. The second test showed great success as that sheet of plywood was able to touch ends without showing great signs of stress.

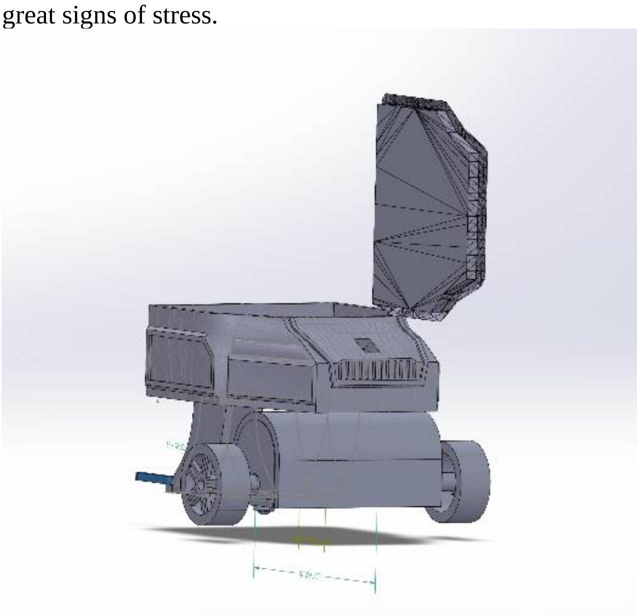

Figure 16: Sci-Fi Model

Given the choice between a retro or sci-fi design, the sci-fi design was seen as the obvious choice. The choice was inspired by the great sci-fi influences common within each member's childhood, such as Star Wars, as well as the great constant advancement seen in technology and used in space travel.

An elevated base with separate front and back plates that held common parts respectively was designed as an influence from the many sci-fi vehicles seen on the mood board. The front base held all sensors that were being utilized by the vehicle whereas the back plate held the motor driver and motors that controlled the movement of the vehicle. The elevated base housed the battery, breadboard, and Arduino Uno and acted as the main brain of the vehicle. Multiple cutouts could also be found throughout the design to allow for better wire management.

B. *System Attributes*

1) *Reliability*

For the autonomous vehicle prototype to be reliable, the system must have all electronic components checked and the right code uploaded into the Arduino at the time of delivery.

2) *Availabilty*

The prototype is available at time of use if all the aforementioned conditions for reliability are met.

3) *Safety*

When the cables are not connected properly, a malfunction in the autonomous vehicle would occur. It is advised to double check the cable connections before turning on the switch.

4) *Maintainability*

It is easy to replace the electronic components when damaged or to make adjustment in their positioning.

5) *Portability*

The code can be uploaded with the Arduino software which can be installed into the computer or used online. The autonomous vehicle prototype does not occupy space, has small weight, and can be easily stored in a portable space.

IV. CODE EXPLANATION

A. *State Machine*

Before writing the code, a state machine diagram is shown below which explicitly aids our understanding in order to implement an efficient program.

Figures 17 and 18 below show the loop state which begins with the system driving straight, facing north on the Y axis until the first given coordinate is reached. At the intersection on the Y axis, it spins with the stSpinNorth state to the right side on the X axis. Then, it switches to stStraightEast, where it keeps on driving until the first given coordinate on the X axis is reached. At the coordinate, it choses between spinning to the left or to the right, depending on the next Y coordinates value. After its spun, it goes into the stStraightNorth or stStraightSouth state depending on the specification of the user, where it drives to the second given Y coordinate. When it reaches the intersection, it turns to the left or to the right, depending on the second given X coordinate and drives straight until the given coordinate is reached. Upon arriving at the second specified coordinate, it makes a TurnThree which adds stSpinWest and stSpinWestLeft. All these steps get repeated until the third coordinate is reached, then the program switches to the state turnZero which directly returns the vehicle to the origin without detours.

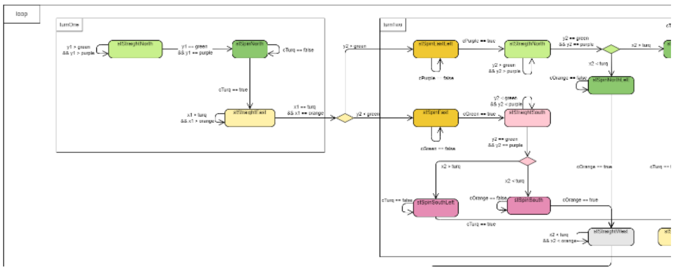

Figure 17: Loop State Machine Diagram (1)

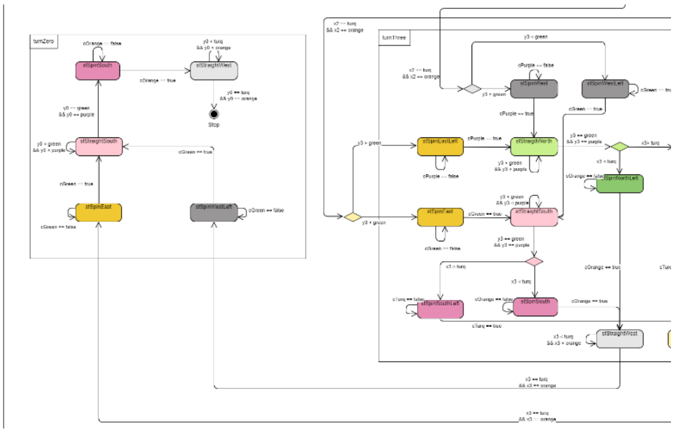

Figure 18: Loop State Machine Diagram (2)

In Figure 19, when the vehicle is facing north or south, the program switches between the states stGreen and stPurple with respect to the color lines of the track. Every state change depends on the specified direction of the vehicle and the color lines are counted to know which coordinates the vehicle is located at any given time. While moving north, it adds a count for every color line in its path; however, in the case of south, it subtracts a count. When the vehicle is in the west or east direction, the program switches between the state stOrange and stTurq respectively. Also, every time the vehicle detects a color line change in its path, it adds a count in the east direction and subtracts a count in the west direction.

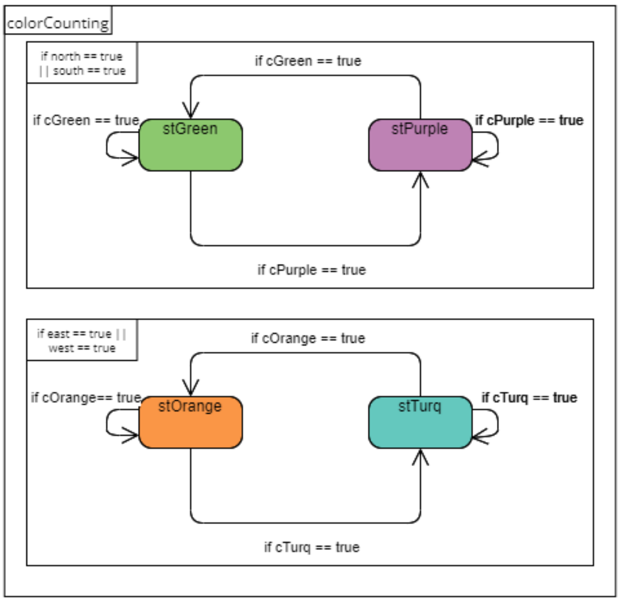

Figure 19: colorCounting State Machine Diagram

The followLine state is required to keep the vehicle on the line. In the straight state, we require both infrared sensors to be HIGH or LOW for the vehicle to move forward along the path.

The case for making turns at intersections of specified coordinates are considered where the black lines are shortly interrupted, but the vehicle should be moving straight along its path. The vehicle turns to the right direction, when the right sensor is on the black line, and the left sensor is on the white line. Whereas, the vehicle turns to the left, when the sensors behave in opposite manner.

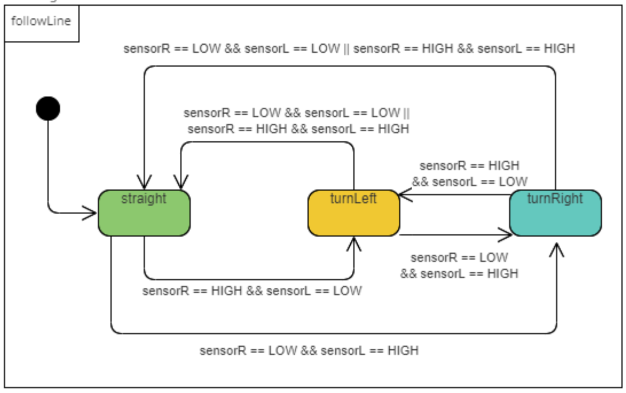

Figure 20: followLine State Machine Diagram

B. Code

The readColor function [Figure 21] obtains the frequencies from the RGB- sensor. These frequencies describe the color on the ground which the function classify to the accounting colors with if- functions for every single color.

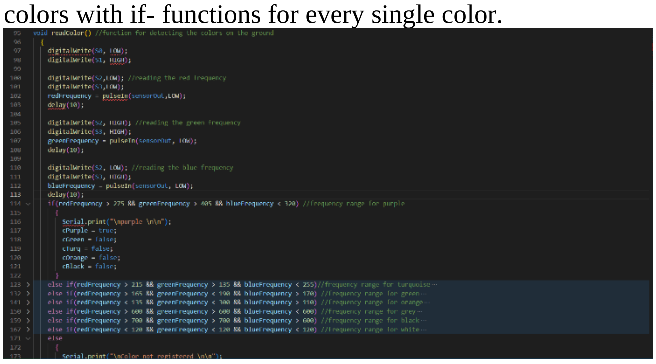

Figure 21: readColor Function

The measureDistance function [Figure 22] measures the distance by calculating the output of the ultrasonic sensor by 0.034/2. This equation defines the output of the ultrasonic sensor which senses and relays the distance of the object from the vehicle, giving an output for this distance in centimeter.

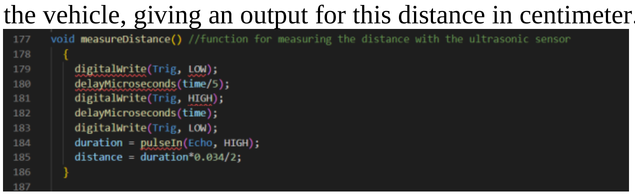

Figure 22: measureDistance Function

The followLine function [Figure 23] keeps the vehicle on the line.

The vehicle drives forward in 2 cases:

1. Both sensors are LOW: both sensors are on the black lines.

2. Both sensors are HIGH: sensors are either on white or different color of lines.

The second case is for intersections where the black lines are shortly interrupted, but the vehicle should still move straight along its path. The vehicle turns to the right, when the right sensor is on black, and the left sensor is on white. Whereas, the vehicle turns to the left, when the right sensor is on white, and the left sensor is on black.

```
void followLine() //function for following the line
  {
    speed = 110;
    speedL = 80;

      sensorR = digitalRead(2); //reading the line sensors
      sensorL = digitalRead(4);

      if(sensorR == HIGH && sensorL == HIGH) //if both sensors are on black -> straight
        {
          straight();
        }

      else if(sensorR == LOW && sensorL == LOW) //if both senors are on white/colors -> straight(intersections)
        {
          straight();
        }

      else if(sensorR == LOW && sensorL == HIGH) //if right sensor on white/colors -> turning right
        {
          turnRight();
        }

      else if(sensorR == HIGH && sensorL == LOW) //if left sensor on white/colors -> truning left
        {
          turnLeft();
        }
  }
```

Figure 23: followLine Function

The colorCounting function [Figure 24] is better described as the coordinate system. It counts every time when a color changes on the test track. The vehicle moves in the north, east, south, and west directions respectively. When moving towards the north [Figure 25], the coordinate system adds a count; however, it subtracts a count when moving towards the south [Figure 26]. This also applies to the east and west movement, therefore adding or subtracting a count respectively.

```
void colorCounting() //function for couting the colors (coordinate system)
  {
    readColor(); //calling the function for detecting the colors

    if(north == true) //when the car is facing north···

    if(south == true) //when car is facing south···

    if(east == true) //when car is facing east···

    if(west == true) //when car is facing west···
  }
```

Figure 24: colorCounting Function

```
    if(north == true) //when the car is facing north
    {
      switch(status)
      {
        case stGreen: //when car changes from green to purple, green plus 1 and status to purple

          if(cPurple == true)
          {
            green++;
            status = stPurple;
          }

          if(cGreen == true)
          {
            status = stGreen;
          }

        break;

        case stPurple: //when car changes from purple to green, purple plus 1 and status to green

          if(cGreen == true)
          {
            purple++;
            status = stGreen;
          }

          if(cPurple == true)
          {
            status = stPurple;
          }

        break;
      }
```

Figure 25: colorCountingNorth Function

```
    if(south == true) //when car is facing south
    {
      switch(status)
      {
        case stGreen: //when car changes from green to purple, green minus 1 and status to purple

          if(cPurple == true)
          {
            green = green - 1;
            status = stPurple;
          }

          if(cGreen == true)
          {
            status = stGreen;
          }

        break;

        case stPurple:  //when car changes from purple to green, purple minus 1 and status to green

          if(cGreen == true)
          {
            purple = purple - 1;
            status = stGreen;
          }

          if(cPurple == true)
          {
            status = stPurple;
          }

        break;
      }
    }
```

Figure 26: colorCountingSouth Function

The stop function [Figure 27] deactivates the motor, this is achieved when the right and left motors are set to LOW.

```
void stop() //function for stopping
  {
    digitalWrite(InputOneRight, LOW); //right wheel
    digitalWrite(InputTwoRight, LOW);
    analogWrite(EnableRight, speed);

    digitalWrite(InputOneLeft, LOW); //left wheel
    digitalWrite(InputTwoLeft, LOW);
    analogWrite(EnableLeft, speedL);
  }
```

Figure 27: stop Function

The straight function [Figure 28] indicates that for the vehicle to drive forward, InputTwoRight and InputTwoLeft on the motor controller should be set to HIGH.

```
void straight() //function for driving forward
  {
    analogWrite(EnableRight, speed); //right wheel
    digitalWrite(InputOneRight, LOW);
    digitalWrite(InputTwoRight, HIGH);

    analogWrite(EnableLeft, speedL); //left wheel
    digitalWrite(InputOneLeft, LOW);
    digitalWrite(InputTwoLeft, HIGH);
  }
```

Figure 28: straight Function

The turnRight function [Figure 29] determines that for the vehicle to turn right, only the InputTwoLeft of the motor controller should be set to HIGH.

```
void turnRight() //function for turning to the right
  {
    analogWrite(EnableRight, speed); //right wheel
    digitalWrite(InputOneRight, LOW);
    digitalWrite(InputTwoRight, LOW);

    analogWrite(EnableLeft, speedL); //left wheel
    digitalWrite(InputOneLeft, LOW);
    digitalWrite(InputTwoLeft, HIGH);
  }
```

Figure 29: turnRight Function

The turnLeft function [Figure 30] determines that for the vehicle to turn left, only the InputTwoRight of the motor controller should be set to HIGH.

```
void turnLeft() //function for turning to the left
  {
    analogWrite(EnableRight, speed-10); //right wheel
    digitalWrite(InputOneRight, LOW);
    digitalWrite(InputTwoRight, HIGH);

    analogWrite(EnableLeft, speedL); //left wheel
    digitalWrite(InputOneLeft, LOW);
    digitalWrite(InputTwoLeft, LOW);
  }
```

Figure 30: turnLeft Function

The directionLeft function [Figure 31] is enabled so that when called, the direction changes from the current one to the one on the left.

```
void directionLeft() //function for changing the direction of the car when turnig to the left
  {
    if(north == true)
      {
        north = false;
        west = true;
      }

    else if(east == true)
      {
        east = false;
        north = true;
      }

    else if(south == true)
      {
        south = false;
        east = true;
      }

    else if(west == true)
      {
        west = false;
        south = true;
      }
  }
```

Figure 31: directionLeft Function

The directionRight function [Figure 32] is enabled so that when called, the direction changes from the current one to the one on the right.

```
void directionRight() //function for changing the direction of the car when turnig to the right
  {
    if(north == true)
      {
        north = false;
        south = false;
        west = false;
        east = true;
      }

    if(east == true)
      {
        north = false;
        west = false;
        east = false;
        south = true;
      }

    else if(south == true)
      {
        south = false;
        north = false;
        west = true;
        east = false;
      }

    else if(west == true)
      {
        west = false;
        east = false;
        south = false;
        north = true;
      }
  }
```

Figure 32: directionRight Function

At the specified intersection of the coordinates, the spinLeft function [Figure 33] allows the vehicle to spin left when the InputTwoRight of the motor controller is set to HIGH. The speed is also reduced to prevent the vehicle from going off course.

```
void spinL() //function for spinning to the left
  {
    analogWrite(EnableRight, speed-8); //right wheel
    digitalWrite(InputOneRight, LOW);
    digitalWrite(InputTwoRight, HIGH);

    analogWrite(EnableLeft, speedL-10); //left wheel
    digitalWrite(InputOneLeft, HIGH);
    digitalWrite(InputTwoLeft, LOW);

  }
```

Figure 33: spinLeft Function

At the specified intersection of the coordinates, the spinRight function [Figure 34] allows the vehicle to spin right when the InputTwoLeft of the motor controller is set to HIGH. The speed is also reduced to prevent the vehicle from going off course.

```
void spinR() //function for spinning to the right
  {
    analogWrite(EnableRight, speed-15); //right wheel
    digitalWrite(InputOneRight, HIGH);
    digitalWrite(InputTwoRight, LOW);

    analogWrite(EnableLeft, speedL); //left wheel
    digitalWrite(InputOneLeft, LOW);
    digitalWrite(InputTwoLeft, HIGH);

  }
```

Figure 34: spinRight Function

The main loop [Figure 35] starts with the switch case that determines to which coordinate the vehicle is driving to. First, it drives from the origin to the first coordinate. When it reaches, it goes to turnTwo, where it drives directly to the second coordinate. Then, it drives directly to the third coordinate, which is its final destination and goes back without detour to the origin.

```
void loop()
  {
    switch(turn)
      {
      case turnOne: //driving to the first coordinate

        switch(run) ···

      break;

      case turnTwo: //driving to the second coordinate

          Serial.print("\n turn Two\n");
        switch(run) ···

      break;

      case turnThree: //driving to the third coordinate

          Serial.print("\n turn Three\n");
        switch(run) ···
      break;

      case turnZero: //driving to the beginning coordinate

        switch(run) ···

      break;
      }
```

Figure 35: loop Function

In the stStraightNorth case [Figure 36], the vehicle follows the line and calls the colorCounting function to keep track of its present coordinate location. When the given Y coordinate is reached, the system goes directly into the spinning case. The spinning case depends on if the next X coordinate is bigger or smaller than the current one. If bigger, it spins to the right. If smaller, it spins to the left. Whereas, for stStraightSouth case, it works in an opposite manner.

```
case stStraightNorth:
  colorCounting();
  followLine();

  if(y2 > green && y2 > purple)
    {
      run = stStraightNorth;
    }

  else if(y2 == green && y2 ==purple)
    {
      if(x2 > turq)
        {
          run = stSpinNorth;
        }

      if(x2 < turq)
        {
          run =stSpinNorthLeft;
        }
    }

break;
```

Figure 36: stStraightNorth case

The stSpinNorth/stSpinNorthLeft case [Figure 37] calls the spinR/spinL function and the readColor function to detect the color lines on the test track. The vehicle spins until it detects the colored line using the RGB sensor. When it detects the line, it calls the directionRight/directionLeft function to change the direction of the system as specified by the given coordinates, as well as switching the case again to going straight and to the color that the vehicle should be on. This implies to every spin case.

```
case stSpinNorth:

  spinR();
  readColor();

  if(cTurq == false)
    {
      run = stSpinNorth;
    }

  else if(cTurq == true)
    {
      directionRight();
      run = stStraightEast;
      status = stTurq;
    }

break;
```

Figure 37: stSpinNorth case

The stStraightEast [Figure 38] as well as the stStraightWest case drives the vehicle on the X axes. They call the followLine and colorCounting functions to stay on the line and keep track of where the car is at the moment. When the given X coordinate is reached, the system goes directly into the spinning case for the next coordinate. In case of stStraigthEast is bigger, it spins to the left. If smaller, it spins to the right. Whereas, for stStraightWest case, it works in an opposite manner.

```
      case stStraightEast:
        east = true;
        north = false;
        Serial.print("\neast\n");
        colorCounting();
        followLine();
         if(x1 > turq && x1 > orange)
          {
            run = stStraightEast;
          }

        else if(x1 == turq && x1 == orange)
          {
            stop();

            if(y2 > green)
              {
                run = stSpinEastLeft;
              }

            else if(y2 < green)
              {
                run = stSpinEast;
              }

            turn = turnTwo;

          }
      break;
    }
  break;
```

Figure 38: stStraightEast case

V. CONCLUSION

This electronic document summarizes the process of developing a working prototype of an autonomous vehicle for precision farming. Understanding the concept of the topic was crucial for knowing what hardware components are to be used. The design enabled us to have a visual representation of the prototype which accelerated the process of achieving a functioning autonomous vehicle prototype.

The prototype phase started with creating a CAD design, assembling, and cabling the hardware components on a physical level, to writing a functioning program that enabled the vehicle to attain its main goal.

AFFIDAVIT

I hereby confirm that I have written this paper independently and have not used any sources or aids other than those indicated. All statements taken from other sources in wording or sense are clearly marked. Furthermore, I assure that this paper has not been part of a course or examination in the same or a similar version.

Jad Al Jourdi	Lippstadt, 18.06.22	
Name, Vorname Last Name, First Name	Ort, Datum Location, Date	Unterschrift Signature

Imad Chaar	Lippstadt, 18.06.22	
Name, Vorname Last Name, First Name	Ort, Datum Location, Date	Unterschrift Signature

Maxim Speczyk	Paderborn, 18.06.22	
Name, Vorname Last Name, First Name	Ort, Datum Location, Date	Unterschrift Signature

Stephanie C. Okosa	Hamm, 18.06.22	steph.o
Name, Vorname	Ort, Datum	Unterschrift
Last Name, First Name	Location, Date	Signature

REFERENCES

[1] Andres, P. Design of an autonomous navigation system for a mobile robot [thesis]. Montreal: McGill University; 2005 [Accessed 17.06.2022]

[2] IEEE Recommended Practice for Software Requirements Specifications. 1998.