The University of Akron
IdeaExchange@UAkron

Honors Research Projects | The Dr. Gary B. and Pamela S. Williams Honors College

Spring 2015

Free-Range Pre-Programmed RC Car

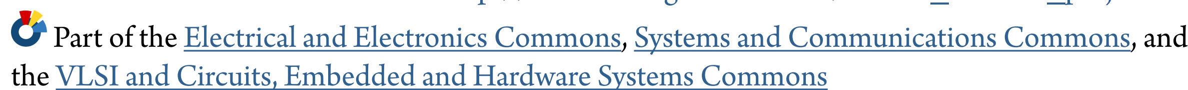
Alexander L. Aubihl
University of Akron Main Campus, ala54@zips.uakron.edu

Andrew S. Hopwood
University of Akron Main Campus, ash40@zips.uakron.edu

Benjamin J. Riggs
University of Akron Main Campus, bjr40@zips.uakron.edu

Tyler P. Vance
University of Akron Main Campus, tpv3@zips.uakron.edu

Please take a moment to share how this work helps you through this survey. Your feedback will be important as we plan further development of our repository.
Follow this and additional works at: http://ideaexchange.uakron.edu/honors_research_projects


Part of the Electrical and Electronics Commons, Systems and Communications Commons, and the VLSI and Circuits, Embedded and Hardware Systems Commons

Recommended Citation
Aubihl, Alexander L.; Hopwood, Andrew S.; Riggs, Benjamin J.; and Vance, Tyler P., "Free-Range Pre-Programmed RC Car" (2015). *Honors Research Projects*. 153.
http://ideaexchange.uakron.edu/honors_research_projects/153

This Honors Research Project is brought to you for free and open access by The Dr. Gary B. and Pamela S. Williams Honors College at IdeaExchange@UAkron, the institutional repository of The University of Akron in Akron, Ohio, USA. It has been accepted for inclusion in Honors Research Projects by an authorized administrator of IdeaExchange@UAkron. For more information, please contact mjon@uakron.edu, uapress@uakron.edu.

Senior Design Project Proposal

Free-range Pre-programmed RC Car

Alex Aubihl EE

Andrew Hopwood CpE

Ben Riggs CpE

Tyler Vance CpE

Table of Contents

Title Page 1
Table of Contents 2
Abstract 3
Problem Statement 4
Need Statement 4
Objective Statement 4
Research Survey 5
Marketing Requirements 7
Objective Tree 8
Design Requirements Specification 9
Engineering Requirements 9
Requirement Matrix 11
Accepted Technical Design 12
Overview 12
Qt Basics 14
Graphic User Interface 15
Track Parser 28
Data Transmission 33
RC Vehicle 38
Collision Detection/Avoidance 57
Parts List 60
Design Team Information 61
Conclusion 62
References 63
Appendices 64

Abstract BR, TV, AH, AA

There is a growing interest in the capabilities and utilization of autonomous vehicles. The objective of this project is to design a small scale illustration of an autonomous vehicle driven by user input. An application will be designed that will allow a user to create a track that an RC car will accurately follow. As the car follows the track, a microcontroller on the vehicle will send movement information back to the application. This information is used by the application to process where the vehicle is currently at and where it needs to go. While in motion, on-board sensors will actively detect obstacles in the path and adjust the vehicle's direction to avoid collision.

Problem Statement BR, TV, AH, AA

Need BR, TV, AH, AA

The free-range pre-programmed RC car has many possibilities, ranging from entertainment to a small-scale representation of autonomous vehicles. The technology could be employed in various environments depending on the customer's need. An example would be an autonomous vehicle on a factory floor with scheduled tasks throughout the building. The track would be determined by user input from the PC application's user interface. The input would then be converted into a readable form that directs the motion of an RC car. The vehicle would be able to detect obstacles to avoid collisions and also correct its course if necessary. After receiving the initial command, the vehicle would successfully complete its movement and then wait for a new order.

Objective BR, TV, AH, AA

The objective of the project is to have a remote controlled vehicle be able to autonomously maneuver along a desired path set by the user. An application will be created on a personal computer that allows the user to view the different performance characteristics of the vehicle and to design the path of the vehicle. The program will allow the user to select between several premade track pieces and connect them into the shape of the desired path. The user will be able to select track pieces that will then be appended one by one to the current location in the track. These tracks can also be saved and reloaded at will by the user. After executing the track, the information of the user-designed path will be wirelessly transmitted to the vehicle. The RC car will then follow this track at a given velocity while avoiding any potential collisions.

Research Survey BR, TV, AH, AA

One option for the application is building it using C++, Qt libraries, and Windows APIs. Qt has libraries that assist the development of both Wi-Fi and Bluetooth communication. The application will instantiate communication to the vehicle and continue doing so until the application is powered down. Algorithms will be constructed inside the code to calculate the data that needs to be sent to the vehicle to control the motor and sensors. Error handling will also have to be implemented to keep the vehicle on the determined path. If the vehicle strays off course, the error handling will guide the vehicle back onto the proper path. In addition to error handling, the car will also have to avoid colliding with obstacles in the path.

Using Bluetooth as a means of communication between the application and the vehicle provides both advantages and disadvantages. The main advantage of using Bluetooth is the cost and power efficiency of operation. When compared to Wi-Fi communication, the amount of power consumption for Bluetooth is significantly less. According to a comparative study performed by Oriana Riva from ETH Zürich and Jaakko Kangasharju from Helsinki University of Technology, the consumption of power in several Nokia cell phones was more than 100 times greater when using Wi-Fi as compared to using Bluetooth [4]. The reason for this significant difference in power consumption is directly related to the vast difference in data transfer rate. The theoretical data transfer rate for Bluetooth is nearly 11 times slower than Wi-Fi data transfer [4]. However, if the group wanted to incorporate video communication or derive complex sensor data, Wi-Fi communication would be better suited for higher data transfer rates. According to IEEE standard 802.15.1-2005, the effective range for Bluetooth communication is between 1 meter and 100 meters, depending on the class of Bluetooth device used [1]. The application will limit the maximum range the vehicle can be away from the user, thus helping to eliminate problems the vehicle will have with losing communication. If communication with the vehicle is lost, a mechanism will be hardwired into the vehicle to shut the motors off.

One of the main limitations of the design of the project is that the user can only choose between premade track pieces, which limit the user's ability to control the vehicle to go wherever the user pleases. The vehicle will only be capable of following the distances and curve angles of the premade track pieces, and not be able to make sharp or gradual turns. Despite this limitation, there should ideally be a wide enough selection of track pieces for the user to design whatever track he or she desires. While this is a limitation, it was a strategic decision by the designers to develop the interface as such. The program will be able to better track the position of the vehicle by only allowing it to travel at certain speeds and turn at certain angles. Without this limitation, the program would have to handle and compensate for the error of invalid maneuvers. The predetermined pieces have already been tested and are known to not cause error with the vehicle.

A limitation of the current technology is that similar programs for remote control vehicles are made almost exclusively for Apple and Android products. By being designed on these operating systems, this product can only be operated on either Apple or Android devices, meaning they can only be run using smart phones and tablets. With the designers choosing to use a Windows application, the program for the vehicle could be used on a desktop computer with Windows. By using Windows as the operating system, the vehicle and program will have more of a marketable value, considering that most companies use Windows in their offices.

Design teams at the University of California have developed robotic vehicles that follow the same general concept that will be used in making this project. By using off-the-shelf components, the UC design teams have been able to create several different autonomous vehicles incorporating Android operating systems, Arduino electronic boards, and R/C vehicles [3].

Through utilizing off-the-shelf components, they were able to cost-effectively create an autonomous system where sensors and other electronic components could be added without needing extra fabrication. To communicate between the Android operating system and the vehicle, a Bluetooth connection was established to transmit/receive data. A C++ application was written to allow communication between the Android operating system and a desktop computer over Wi-Fi. Through UDP packets, information such as video transmission, sensor signals, and GPS locations can be transmitted between the Android operating system and desktop application [3].

One patented technology that is very similar to our idea is a remote controlled toy device that uses a single chip main control unit. Its developer, Zhang Fengchuan of China, created a processor chip that collects video data and transmits a compressed signal over a Wi-Fi communication channel. By adding this chip onto a remote controlled toy, the user can control the toy using a smart phone, tablet, or personal computer. The chip receives the input controls from the user, relays the information to the motors, LED lights, and related electrical control components, and transmits measured data back to the user interface [2]. The design team would like to implement a system similar to this processor chip in our vehicle. Being able to wirelessly translate commands from an application-based system to the hardware of the vehicle is the ultimate goal of this project. Rather than designing a processing chip to do this, the group would prefer to use off-the-shelf components to implement the same process.

Another patent that relates to our project idea is a wireless system that was developed to control an R/C car via a handheld game player device. Inventor Patrick Tze Man Ho was able to achieve this by combining a game cartridge and a message transmitter. The user will be allowed to control the R/C car with the handheld gaming device, and have the display screen of the gaming device show functionalities to and for the remote control unit. The transmitter would send a radio frequency to a receiver on the vehicle. Using the gaming device, the user would be able to control the speed and direction of the R/C car [6]. This process of wireless communication is the ultimate goal of the group, but the means of making the communication between the vehicle and the user interface will be slightly different. Rather than using radio frequency, the group will be using Bluetooth or Wi-Fi communication.

Marketing Requirements BR, TV, AH, AA

- The RC car should follow the designed track accurately
- The program should be able to run on Windows PCs
- The RC car should have a reasonable battery length
- The RC car should avoid collisions
- If off-track, the RC car should auto-correct
- The program's GUI should be simple to use
- The application should take up as little memory/resources as possible
- The system should be affordable
- The parts should be interchangeable
- The system should be lightweight and portable

Objective Tree BR, TV, AH, AA

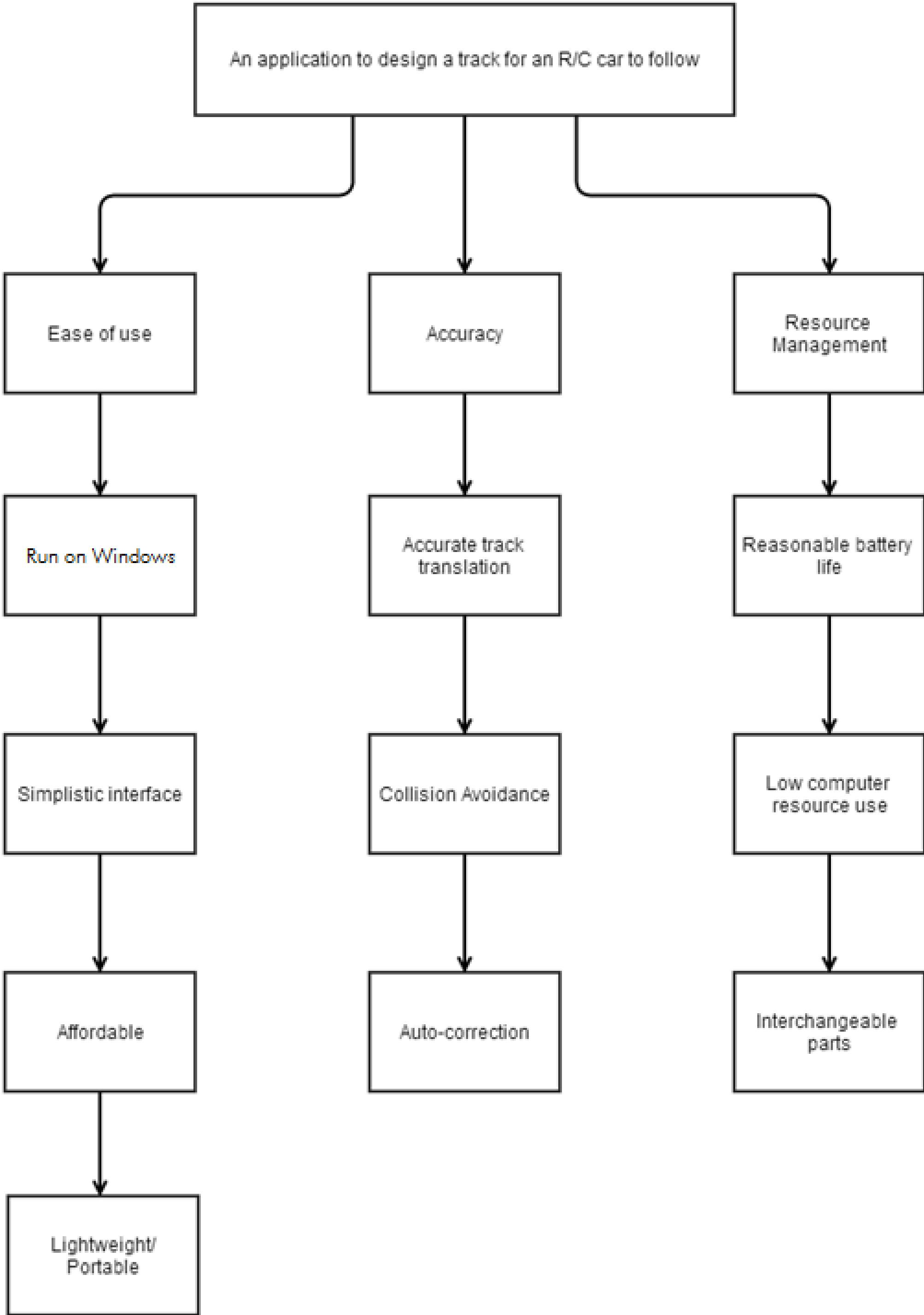


Figure 1: Objective Tree

Design Requirements Specification

Engineering Requirements BR, TV, AA

In order to allow for the marketing requirements to be realized, there needs to be a set of engineering requirements that need to be created that will allow for the vehicle and its system to fulfill the requirements desired by the customer. By analyzing the requirements requested by the customers, the group must determine how the best method of satisfying these requirements can be achieved. Without detailing how the goals will be achieved, the group must unambiguously state what goals and requirements the vehicle, the communication, and the PC must reach. These engineering requirements need to be made not only to give a verifiable, traceable specification for the vehicle to achieve, but also to set limitations on the maximum performance of the vehicle. Below, the group will explain the engineering requirements they have determined for the Free-Range Pre-Programmed RC Car system, and show how they relate back to and achieve the marketing requirements. The engineering requirements and marketing requirements are also compared in a tradeoff matrix, which demonstrates how increasing or decreasing one parameter affects the other. The requirements table and tradeoff matrix are found as Figures 2 and 3, respectively.

Marketing Requirements	Engineering Requirements	Justification
3, 9	1. The vehicle should run off a 9-VDC battery.	Using a 9-VDC battery should provide the vehicle with enough power to operate all the components on the vehicle.
1, 2, 4, 5, 7	2. The vehicle should be able to communicate with the PC program using Bluetooth communication at a maximum distance of 50m.	Bluetooth communication provides a good means for sending/receiving serial data from a transmitter to a receiver. This serial data can then be converted to the necessary voltage command data.
2, 6, 7	3. The maximum amount of memory taken up by the application and associated files should not exceed 250Mb.	By putting a limit on the amount of memory the system can use up, the program will need to be running as efficiently as possible upon completion.
1, 3, 6	4. The vehicle should travel at a constant speed that is determined by the user, with a maximum speed of 0.67 meters/second.	The slower the vehicle travels, the more accurately the vehicle can follow the path. By putting a maximum value on the speed of the vehicle, the vehicle will not be allowed to try and outperform its physical capability and allow the PC enough time to compute the necessary controls for the vehicle.
1, 4	5. The vehicle should be able to detect possible collisions at a minimum distance of 2 feet.	Setting the detectable distance of the sensor to 2 feet should allow the vehicle and avoidance algorithm plenty of time to correct the path of the vehicle to avoid the object.
1, 7	6. The user interface should allow the user the ability to choose pre-determined track pieces to form a path of any desired shape.	The user will have complete freedom to create a track of their choosing, but will be limited in the track pieces they can select from. Pre-determining the track pieces will allow for the vehicle to follow a smooth path, and not be restricted by difficult or impossible turns.
1, 2, 6	7. The interface should save each track as a file that can later be reloaded by the user.	The program will save each path created as a track file. This file is accessed by the track parser to break the track down into vehicle commands. The user can easily access previously created tracks since they are saved into the PC memory.
1, 5	8. Barring collision avoidance, the vehicle should have an error tolerance of 5%.	If the vehicle strays from its desired path by a magnitude of more than 1 foot, the path for the vehicle should be corrected to account for the error the vehicle is experiencing.
1, 4, 5	9. The vehicle should relay its position back to the PC using GPS coordinates.	The GPS position of the vehicle should be used as feedback from the vehicle to the program, relating its current position to the desired position from the created track.
8	10. The project should not exceed $400 USD in cost.	The design group is given a budget of $400 from the University of Akron, so the budget for the project should not exceed this budget.
3, 10	11. The vehicle should not exceed a size of 25cm x 15cm.	Limiting the size of the vehicle allows for a more user-friendly system and cuts down on the force required by the vehicle to drive forward.
3, 10	12. The vehicle should not exceed a weight of 1kg.	Limiting the weight of the vehicle allows for a more user friendly system and cuts down on the force required by the vehicle to drive forward.

Marketing Requirements
1. The RC car should follow the designed track accurately.
2. The program should be able to run on Windows tablets and PCs.
3. The RC car should have a reasonable battery length.
4. The RC car should avoid collisions.
5. If off-track, the RC car should auto-correct.
6. The program's GUI should be simple to use.
7. The application should take up as little memory/resources as possible.
8. The system should be affordable.
9. The parts should be interchangeable.
10. The system should be lightweight and portable.

Figure 2: Engineering Requirements Table

Requirement Matrix

Engineering-Marketing Tradeoff Matrix		Battery Size	Communication Distance	Maximum Memory	Maximum Speed	Collision Distance	GUI Control Ease	Track Files	Position Error	GPS Position	Cost	Size	Weight
Marketing Requirements		+	+	+	+	+	+	+	-	+	+	-	-
1. The RC car should follow the designed track accurately.	+		↑↑		↑↓	↑↑	↑↑	↑↑	↑↓	↑↑			
2. The program should be able to run on Windows tablets and PCs.	+		↑↑	↑↑				↑↑					
3. The RC car should have a reasonable battery length.	+	↑↑			↑↓							↑↓	↑↓
4. The RC car should avoid collisions.	+		↑↑			↑↑				↑↑			
5. If off-track, the RC car should auto-correct.	+		↑↑						↑↓	↑↑			
6. The program's GUI should be simple to use.	+			↑↑	↑↑			↑↑					
7. The application should take up as little memory/resources as possible.	+		↑↓	↑↓			↑↑	↑↓					
8. The system should be affordable.	-										↑↓		
9. The parts should be interchangeable.	+	↑↓											
10. The system should be lightweight and portable.	-											↓↓	↓↓

(The printed matrix also has a heading "Engineering Requirements" over the twelve engineering-requirement columns.)

Figure 3: Engineering/Marketing Requirement Tradeoff Matrix

Accepted Technical Design

Overview BR, TV, AH, AA

The general idea of the program is to take the user's desired track and replicate it through the RC car. In order to accomplish this, a combination of hardware and software will be needed. Figure 4 is a level 0 block diagram of the system, which illustrates the simplest view of the system. The functional requirement table for this block diagram is contained in Table 1.

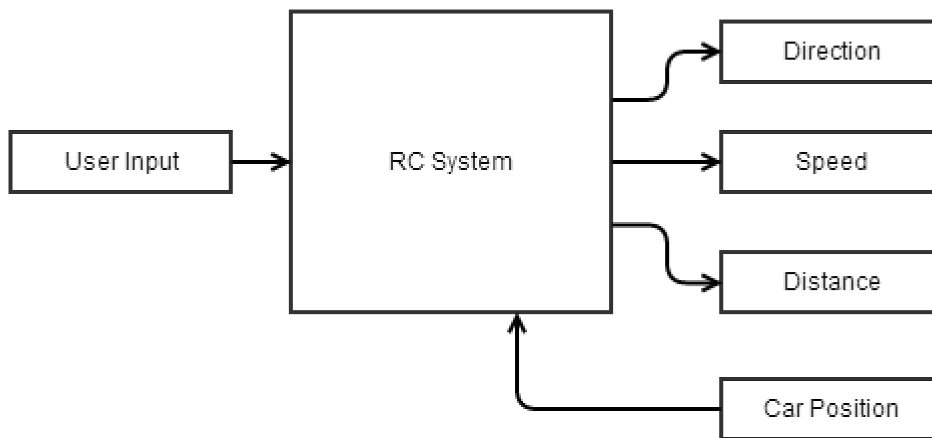


Figure 4: Level 0 Block Diagram

Module	RC System
Inputs	- User Track Input - Car Position
Outputs	- Direction - Speed - Distance
Functionality	User inputs a track design into the RC system that can compute and relay distance, speed, and direction to the RC car. The car position is relayed back to the RC system.

Table 1: Level 0 Functional Requirement Table

In order to ease the design process, the RC system was broken down into functional components. These components were divided and grouped according to functionality and each component has its own inputs and outputs. These components and their connections are represented in Figure 5. Each component's functionality is broken down in Table 2.

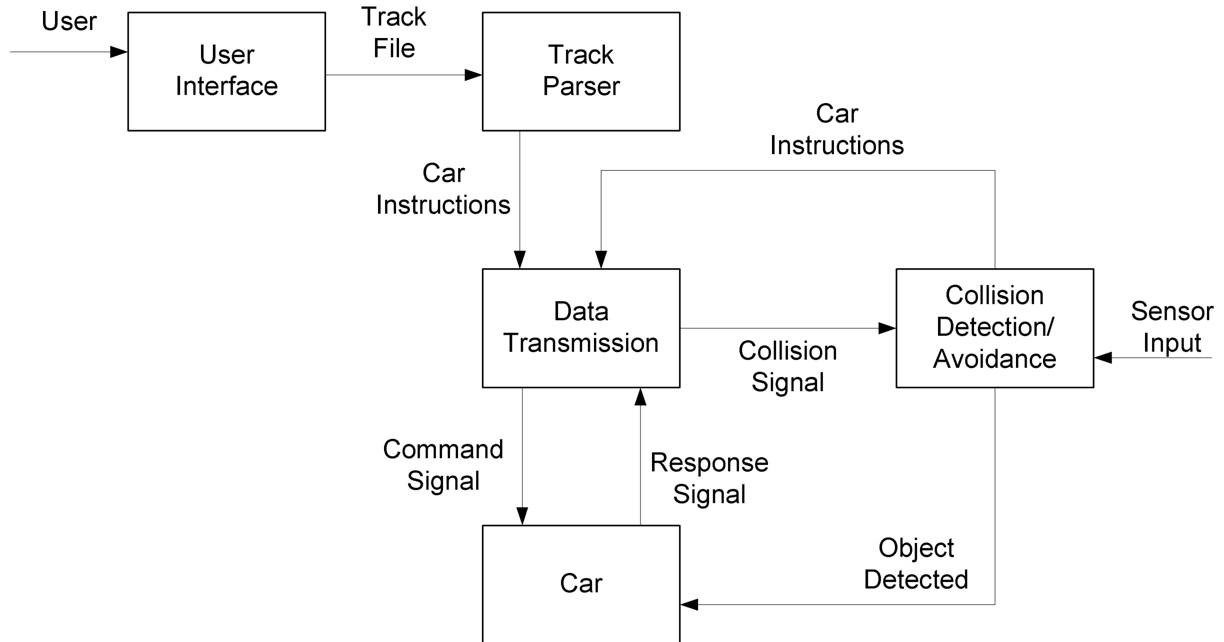


Figure 5: Level 1 Block Diagram

Module	Level 1 RC System Functionality
Inputs	- User - Sensor Input
Outputs	- Direction - Speed - Distance
Functionality	After the user selects a desired path for the vehicle to follow, the track file is sent through a parsing program, which breaks the track down into individual movement instructions for the car. These instructions are transmitted as a command signal to the car, which begins moving. As the vehicle moves, its position is monitored by the onboard IMU, which sends the vehicle's position signal back to the program. If the collision detection sensor is activated by the vehicle approaching an unanticipated object, the sensor input is used to alert the vehicle that it needs to adjust its path. Through the IMU and sensor feedback, the PC program corrects the path of the vehicle. This corrected path is then transmitted back to the vehicle, which adjusts its path according to the new car instructions.

Table 2: Level 1 Functional Requirement Table

Qt Basics BR

Qt is an open-source development tool that will be used in conjunction with C++ in order to accomplish useful tasks that would otherwise be extremely complex or even impossible. There are numerous libraries that Qt provides, each helping to ease the complexity of certain tasks. In addition, Qt also provides numerous objects that can either be used to override basic C++ objects, like a string or vector. Each of Qt's classes has a name that begins with 'Q', such as 'QString' or 'QMap'. This allows a programmer to easily differentiate whether a given object is from base C++ libraries or from Qt libraries.

Many of Qt's functionalities are abstract, requiring pointers to base classes instead of specific objects. This allows users to very easily inherit from base classes while still preserving the functionality that Qt provides. Throughout the application, numerous objects will inherit from some of the various Qt base classes, and many of the functions that will be used will be defined by these base classes.

Qt also provides an application to help with the creation of a dialog or widget. This program, called QtDesigner, allows the user to simply drag and drop other widgets into the dialog being created. The end result is a file with extension 'ui'. This file will then be compiled along with the other header and source files and turned into a header file of its own, detailing a 'UI' class that is equivalent to the dialog created in QtDesigner. This header file can be included in the header file of whatever dialog it was made for, and this dialog's class will store a 'UI' object in order to allow access to the various widgets created in QtDesigner.

The most important use of Qt in this application will be the signals and slots system. Signals and slots are new types of functions introduced by Qt which, when working together, act like an interrupt written directly into the C++ code. One of the major uses of this system is the allowance for a child class to directly talk to the parent class that holds it. When an object needs to send a signal to any object that receives it, it calls the signals function using the *emit* keyword. This will then emit the signal to anything that is listening. When an object listening for this signal receives it, it will immediately call the slot attached to the signal, no matter what was happening previously. Signals and slots can be connected to each other using the global *connect()* function defined by Qt's core libraries. In addition, in order for an object to use signals and/or slots, the 'Q_OBJECT' macro must be defined in the class's definition.

Qt will be directly integrated into every aspect of the application, and will even appear throughout the pseudo-code. Signals and slots will be primarily used in two areas: the user interface, in order to connect specific buttons to specific functionalities, and for exchanges between the communication and track parser components [7].

Graphic User Interface BR,TV

The user interface is responsible for obtaining and translating user input in order to create a usable track. Dialogs are to be created using Qt libraries which will process inputs and update the display accordingly. The end goal of this component is to create a track file which will be written in JSON format.

The application's main window should contain the usual menus in the top left. Below the menus, there will be toolbars with various shortcuts for convenience. Qt's main window class will assist in the creation of these menus and toolbars. The main window will also have a track view and track piece selector. Figure 6 shows the general design layout of the main window.

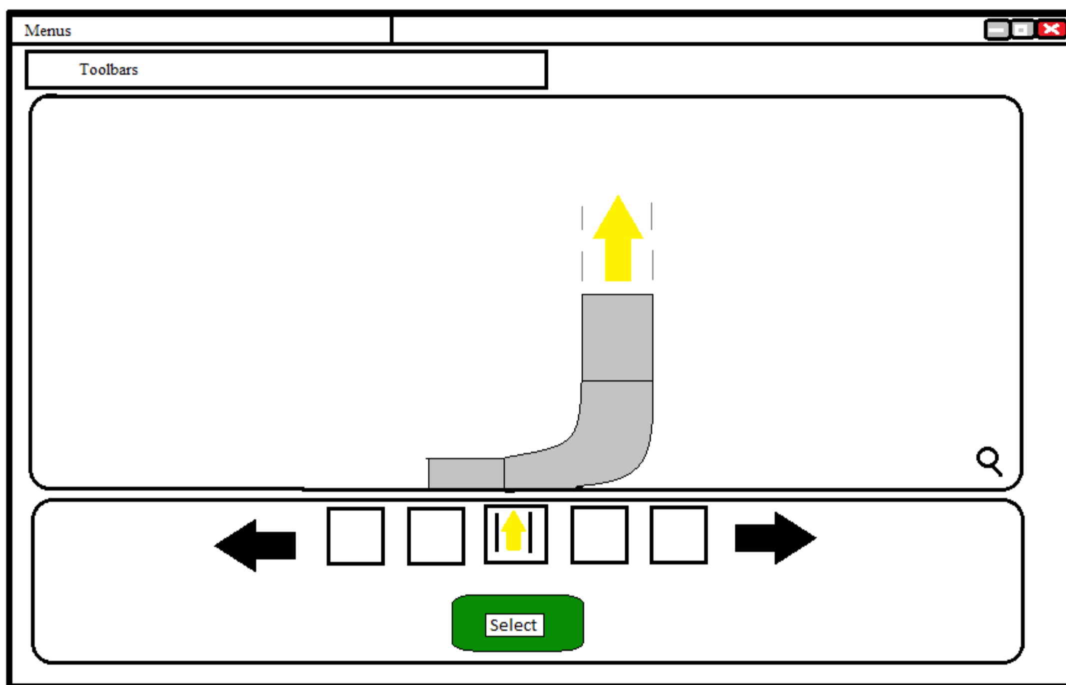


Figure 6: Main Window Design

The track view seen in the middle of Figure 6 displays the track pieces as they are placed. When a new document is created, a blank view is displayed. In order to add track parts, the user will click on various pieces in the part selector on the bottom of Figure 6. When hovering over a piece in the track selector, the track view should show a translucent view of what the piece would look like if added. Tracks can be saved and loaded, defaulting to the user's documents folder. When a previously created track is loaded, the track view will display the track in its entirety and automatically place the user at the end of the track.

The part selector should also have the capability to allow the user to traverse back though the track without deleting pieces by using the left and right arrows seen in Figure 6. The ability to traverse though the track would allow the user to make changes to the middle of the track. This traversal ability will be implemented in two ways. The arrows in the track selector will

navigate through the track one by one. The user should also have the ability to click on a track piece, automatically selecting it. When adding a track piece, it will append it to whatever the currently selected piece is. In addition, the track selector will also need a delete function that would remove the currently selected track.

The background of the track view will be a grid format in specific increments that will allow the user to better visualize and lay out the track. In addition, this will also help to get a better grasp of the distance the vehicle will travel. The track view will also need the ability to zoom in and out, allowing the user to either view more of the track at once or to look at a specific piece more closely. Additionally, by clicking and dragging on either end of a track piece, the user should be able to extend or retract that piece, allowing for more customization.

Lastly, this component will also need to be able to create a track file based on the track currently in the track view. This file will be in JSON format with each track piece being its own object and containing the track's specifications. The application should also be able to read JSON files and populate the track view based on what the file contains. Figure 7 shows a simplistic example of what a track file may look like.

```
{
    Name: "Track Name",
    NumPieces: 3,
    Pieces: [
    {
        Angle: 0,
        Length: 1,
        endX: 0,
        endY: 1
    },
    {
        Angle: 90,
        Direction: "RIGHT",
        Length: 2,
        endX: 2,
        endY: 3
    },
    {
        Angle: 90,
        Direction: "LEFT",
        Length: 1,
        endX: 3,
        endY: 4
    } ]
}
```

Figure 7: JSON Track File Example

Figure 8 shows a level 2 block diagram for this component. Table 3 shows the functionality requirements for this component.

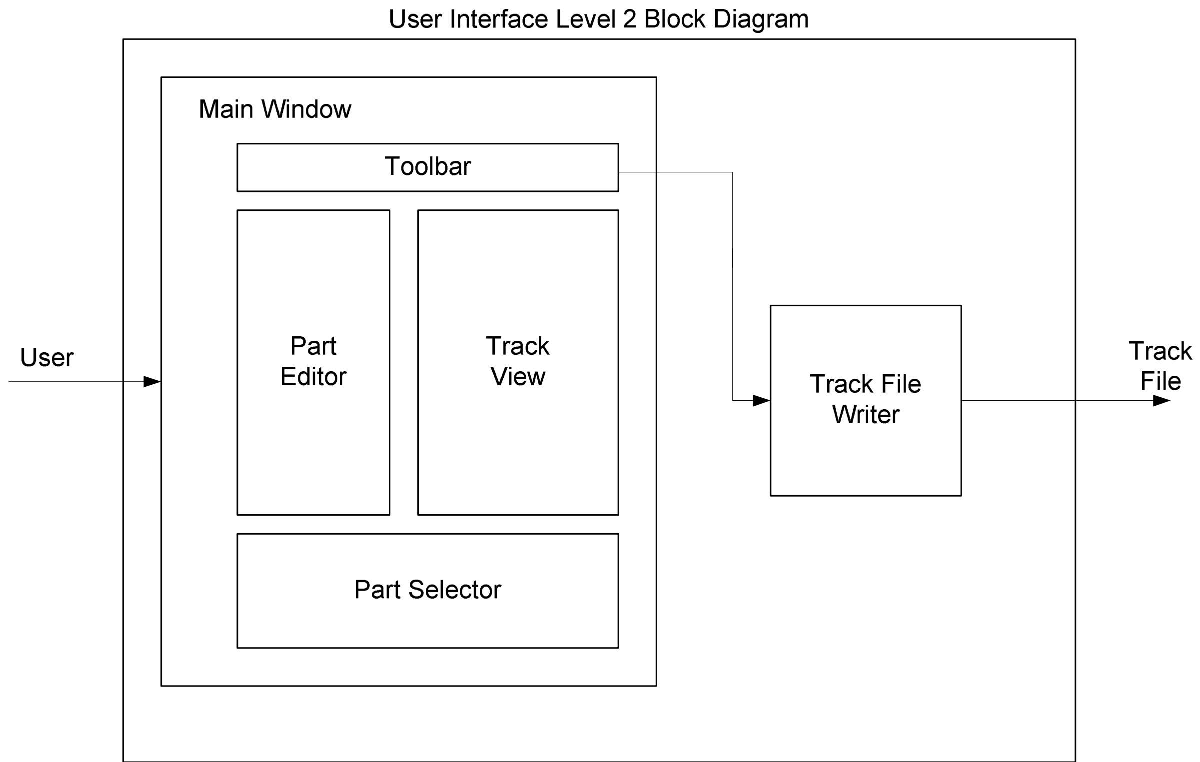


Figure 8: Level 2 User Interface Block Diagram

Module	User Interface
Inputs	- User
Outputs	- Track File
Functionality	At the user interface, the user will be allowed to create a customized track for the vehicle to follow. This track is created by the user selecting between vast arrays of track pieces found in the part selector. Once selected, the track pieces are placed in sequential order, creating the complete track. This track will be shown in the track view window. Pieces can be edited in the part editor window, where the user can adjust the length of the piece, as well as the speed of the vehicle. Once the user is satisfied with the track, the user can send the program to the vehicle to execute. To do this, the vehicle track is saved as a file, and the track file is sent to the track parser.

Table 3: User Interface Functional Requirement Table

All of the functionality required to implement the user interface can be implemented through three classes that inherit from various Qt classes. The classes and their interactions can be seen in Figure 9.

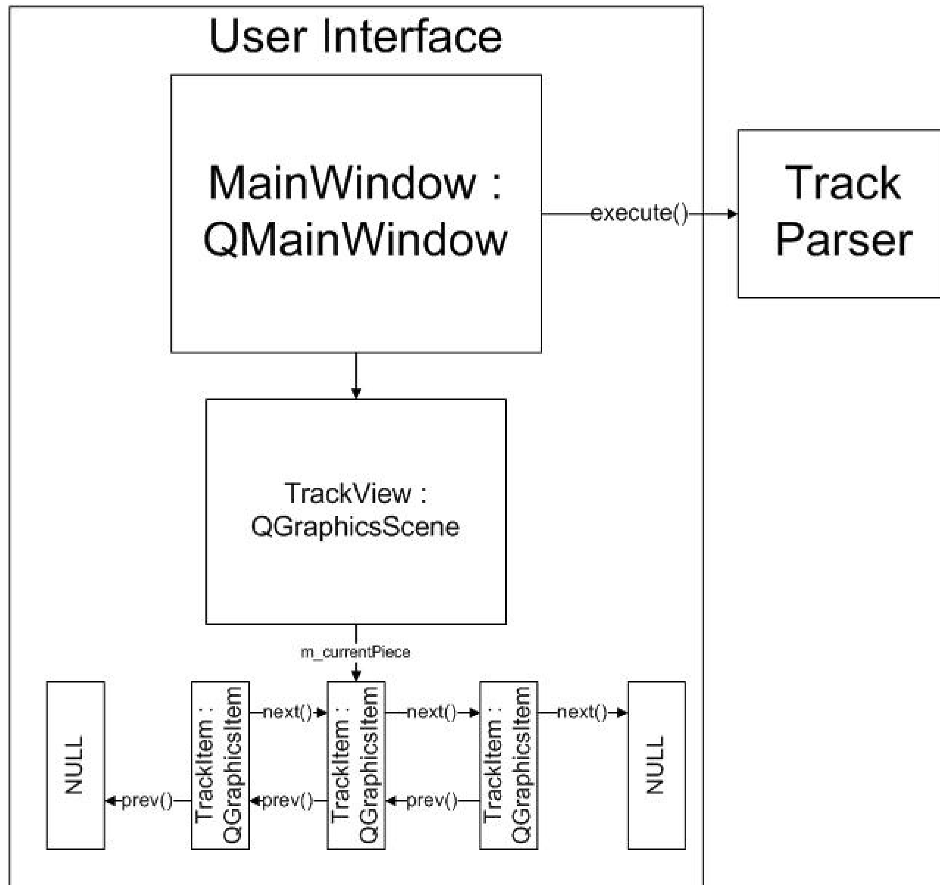


Figure 9: User Interface Class Diagram

At the bottom of the diagram is the TrackItem class. This class, which inherits from QGraphicsItem, represents a single track piece in the UI. QGraphicsItems are any object that can be placed in a QGraphicsScene, allowing for the visualization of movable objects to the user. Typically, these items are not kept in any order, but in the case of this project, the track pieces must be kept in a specific order; otherwise, the track pieces will be executed incorrectly. To maintain the order of the track pieces, each track piece will hold a pointer to both the previous track piece and the next track piece in the track. These pointers can be accessed through the prev() and next() functions shown in Figure 9. If the previous and/or next track piece in the track does not exist, signifying that the current track piece is either the beginning or end of the track, then these functions will return NULL. Pseudo-code for the TrackItem class can be found in Figures 10 and 11.

```cpp
// Header File
class TrackItem : public QGraphicsItem
{
    Q_OBJECT

public:
    // Constructors/Destructors
    TrackItem();

    ~TrackItem();

    // Getters/Setters (if any)

    // Access the next track piece in the track
    // If this is the last track piece, this should be NULL
    TrackItem* next();

    // Access the previous track piece in the track
    // If this is the first track piece, this should be NULL
    TrackItem* prev();

    // Overriding QGraphicsItem::paint()
    // This will be called extremely frequently, whenever the item
    // needs to be re-painted.
    void paint(QPainter*, const QStyleOptionGraphicsItem*, QWidget*);

signals:
    // Signal sent when the track is clicked on
    void clicked();

protected:
    // Overriding QGraphicsItem::mousePressEvent()
    // This will be called whenever the mouse is pressed on this item.
    // Pressing the mouse on a track piece should force the TrackView to
    // select that TrackPiece
    void mousePressEvent(QGraphicsSceneMouseEvent*);

private:
    // Use the member variables to set the path to be drawn
    void createTrackPath();


    // Member Variables
    TrackItem*  m_nextPiece;  // The next Track piece in the track
    TrackItem*  m_prevPiece;  // The previous Track piece in the track

    int     m_angle;        // The turn angle of the piece. + for right turns, - for left turns
    int     m_orientation;  // The starting orientation of the piece
    double  m_length;       // The length modifier of the track piece. Will be multiplied by the Unit length
    QPoint  m_startPoint;   // The starting point of the track piece
    QPoint  m_endPoint;     // The ending point of the track piece
    bool    m_isSelected;   // Holds whether the track piece is currently selected

    QPainterPath  m_path;   // Defines the path that will be drawn to represent this track piece
};
```

Figure 10: TrackItem.h Pseudo-code

```
// Source File
void paint(QPainter*, const QStyleOptionGraphicsItem*, QWidget*)
{
    if (m_isSelected)
    {
        // Set the fill color to a brighter form of the standard fill color
        // Set the border to a dotted line instead of a solid one
    }
    else
    {
        // Use the standard fill and border pens/painters
    }

    painter->fillPath(m_path, fillColor);
    painter->drawPath(m_path);
}

void Track::mousePressEvent(QGraphicsSceneMouseEvent*)
{
    // Check whether or not the mouse was clicked in the actual track piece
    // or just in the item's bounding rectangle.

    if (/* it was clicked in the actual track piece */)
    {
        emit clicked();
    }

    // The actual setting of m_isSelected to true will happen in the TrackView
}

void Track::createTrackPath()
{
    // Find the starting points of the inner and outer portions of the track
    // Rotate those points according to the orientation

    // For a straight line
    if (m_angle == 0)
    {
        // Find the end points of the inner and outer portions
        // Draw the path, from point to point to form a rectangle
    }
    // For a turn
    else
    {
        // Find the radius of the turn for the inner and outer portions
        // Create squares that encompass both of these arcs
        // Draw the lines of the path, including the inner and outer arc
    }

    // Set the end point
}
```

Figure 11: TrackItem.cpp Pseudo-code

In order to hold all of these TrackItems, as well as allow for simple addition, removal, and navigation of various track pieces, the TrackView class is necessary. This class inherits from QGraphicsView and holds a QGraphicsScene as a member variable. By overriding QGraphicsView, the graphics scene and view are much more customizable to the needs of our project.

In addition, this custom class is also able to manage the selected track piece. By storing a pointer to a TrackItem object, the TrackView has easy access to whatever track piece the user is currently working with, as well as the previous and next track pieces. If any track piece in the graphics scene is clicked on, it will send a signal to the TrackView. The TrackView will then shift its currently selected piece to this newly selected piece. This allows the TrackItem object to be blind to which track piece is currently selected, while TrackView does whatever track management is necessary.

The last advantage to this TrackView class is the ability to write ease of access functions. Instead of repeating the same code to add a track piece multiple times in the MainWindow, the TrackView can instead have simple functions for the addition, removal, or selection of a track piece. The more specialized each object is, the easier it is to navigate the code and find the cause of a potential issue. The pseudo-code for the TrackView class can be found in Figures 12 and 13.

```
// Header File
class TrackView : public QGraphicsView
{
    Q_OBJECT

public:
    // Constructors/Destructors
    TrackView(QWidget* parent = 0);

    ~TrackView();

    // Getters/Setters (if any)

    // Add a new track item after the currently selected piece
    // This function must also update the current piece to the newly added piece
    void addPiece(int angle);

    // Remove the currently selected piece
    void removePiece();

    void selectPiece(TrackItem*);

slots:
    void onPieceClicked(TrackItem*);

private:
    // Member Variables
    QGraphicsScene   m_scene;
    TrackItem*       m_currentPiece;
};
```

Figure 12: TrackView.h Pseudo-code

```
// Source File
void TrackView::addPiece(int angle)
{
    // Create the Track Piece and set its values
    TrackItem* track = new TrackItem();
    m_scene->addItem(track);

    // Set the current piece's next piece's previous piece to the new piece
    // Set the current piece's next piece to the new piece
    // Set the new piece's previous piece to the current piece
    // Set the new piece's next piece to the current piece's next piece
    selectPiece(track);
}

void TrackView::removePiece()
{
    // Set the current piece's previous piece's next piece to the current piece's next piece
    // Set the current piece's next piece's previous piece to the current piece's previous piece
    m_scene->removeItem(m_currentPiece);
    selectPiece(/* The current piece's previous piece */);
}

void TrackView::selectPiece(TrackItem* item)
{
    m_currentPiect->deselect();
    m_currentPiece = item;
    m_currentPiece->select();
    // Center the view on the selected piece
}

void TrackView::onPieceClicked(TrackItem* item)
{
    selectPiece(item);
}
```

Figure 13: TrackView.cpp Pseudo-code

The MainWindow class, which inherits from the QMainWindow class, will not only act as a controller between the user and the application, but will also act as a controller of the application as a whole. All of the various buttons and widgets that encompass the dialog will be placed into this class, and the signals and slots that allow them to interact will be connected through the MainWindow as well. This dialog will be active at all times, assisting the user with the creation and execution of a track file. In order to ease the creation of this class, QtDesigner will be used. Pseudo-code for the MainWindow class can be found in Figures 14 and 15.

```
// Header File
class MainWindow : public QMainWindow
{
    Q_OBJECT

public:
    // Constructors/Destructors
    MainWindow(QWidget* parent = 0);

    ~MainWindow();

    // Getters/Setters (if any)

slots:
    // Slots are called when this widget receives a signal
    // All of these slots are associated with various buttons that the user can click
    void onSaveClicked();

    void onSaveAsClicked();

    void onOpenClicked();

    void onExecuteClicked();

    void onTrackPieceClicked(int angle);

    void onTrashClicked();

private:
    // Save the current contents of the Track View into
    // the current file
    void save();

    // Read in a Track File and set the Track View's contents
    // to the contents of the file
    void readFile(QString);

    // Create a track parser that will execute the current file
    void execute();

    // Member Variables
    Ui    m_ui;  // This UI will be created by Qt Designer, and will
                 // contain all of the necessary widgets.

    QString  m_fileName;  // Full path to the opened track file
};
```

Figure 14: MainWindow.h Pseudo-code

```
// Source File
void MainWindow::onSaveClicked()
{
    if (m_fileName == "")
    {
        onSaveAsClicked();
    }
    else
    {
        save();
    }
}

void MainWindow::onSaveAsClicked()
{
    // Prompt the user for a file name/location
    m_fileName = // Obtained file path

    save();
}

void MainWindow::onOpenClicked()
{
    // Prompt the user for a file to read in
    readFile(/* Obtained file path */);
}

void MainWindow::onExecuteClicked()
{
    if (m_fileName == "")
    {
        onOpenClicked();
    }

    execute();
}

void MainWindow::onTrackPieceClicked(int angle)
{
    m_ui.trackView->addPiece(angle);
}

void MainWindow::onTrashClicked()
{
    m_ui.trackView->removePiece();
}
void MainWindow::save()
{
    // Write the file contents to m_fileName
    // TODO: Specify
}

void MainWindow::readFile(QString)
{
    if (/* file can be opened/read */)
    {
        m_fileName = str;
    }

    // Read the file contents into m_ui's TrackView
    // TODO: Specify
}

void MainWindow::execute()
{
    TrackParser parser;
    parser.parse(m_fileName);
}
```

Figure 15: MainWindow.cpp Pseudo-code

The visualization of an individual track piece in the TrackView requires complex trigonometric and geometric calculations. Each track piece must be two-dimensional instead of a simple straight line, meaning that the track piece must have a width, which will cause both an inner and outer track. In order to draw the curved lines that make up these inner and outer tracks, numerous potential methods could be used. It was decided to make use of the *arcTo()* function defined by the QGraphicsItem base class, which TrackItem inherits from.

This *arcTo()* function requires information that is not immediately available without specific calculations. Each track piece stores its own starting position in the TrackView grid, as well as its turn angle, length, and its starting orientation angle. Figure 16 shows the required metrics and the usage of the function.

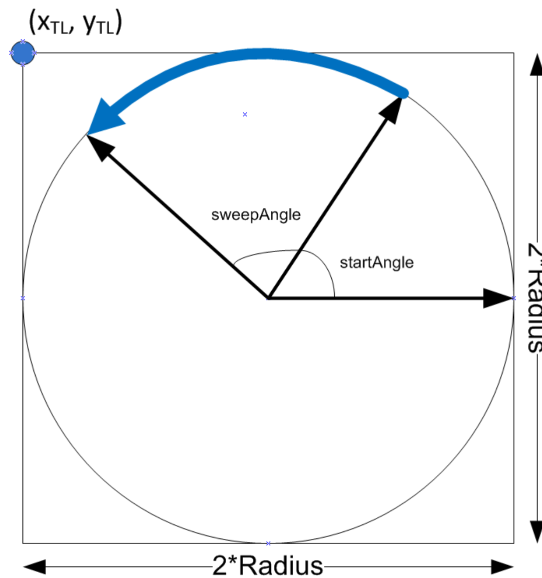


Figure 16: QGraphicsItem::arcTo() Function

The blue line shows what the resulting arc will look like with the given parameters. The sweep angle is equivalent to the inverse of the turn angle, since right turns are seen as positive in the case of a TrackItem. In addition, the start angle can be calculated based on the angle of orientation. In the case of a TrackItem, an orientation of 0° represents the left edge of the circle in Figure 16, or 180° in the *arcTo()* system.

Beyond the sweep angle and starting angle, the square that encompasses the circle being travelled must be given. In order to find this square, the top-left corner and the width of the square must be found. As Figure 16 suggests, the width of the square will simply be twice the radius of the circle being travelled, meaning that two parameters must be found: the top-left corner of the square, and the radius of the circle.

Figure 17 shows how to calculate the radius given the turn angle and length of the track piece. As the figure suggests, the length of a track piece is based on the maximum distance it travels above the y-axis, instead of the length of the arc itself.

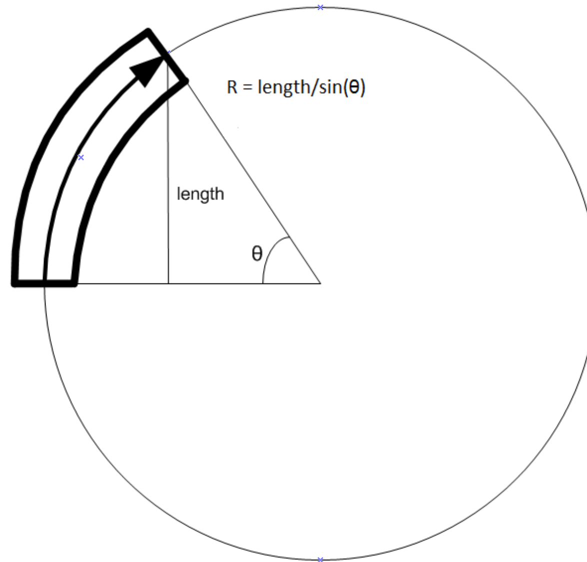


Figure 17: Radius Calculation

Finally, with the radius calculated, the top left- corner of the square encompassing the circle can be found.

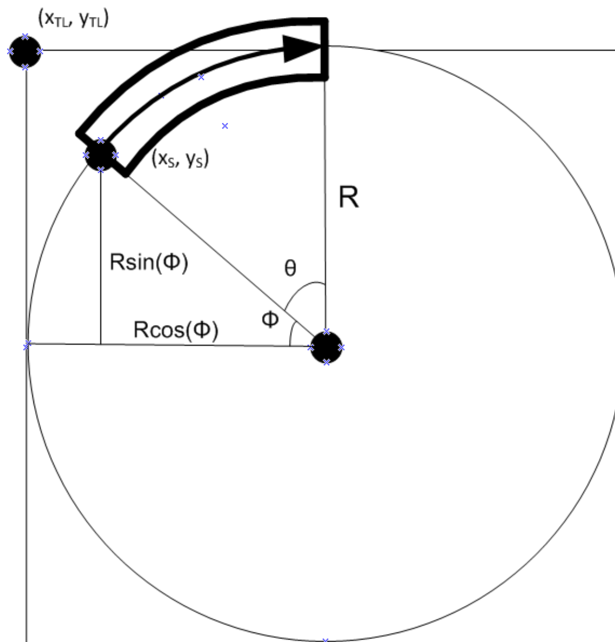


Figure 18: Top Left Calculation

Using Figure 18, equations can be formed to find the location of the top-left corner.

Finally, with all of the pieces in place, the track pieces can be placed into a QGraphicsScene. Simulations were run to ensure that all of the calculations came out correctly. The result is a prototype of what the TrackView may look like, shown in Figure 19.

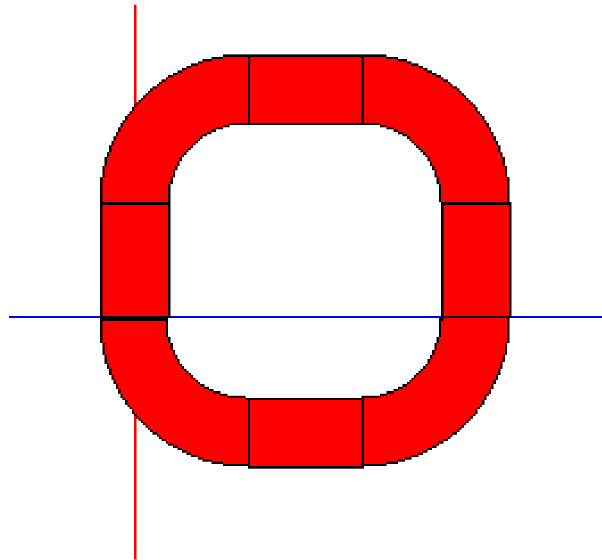


Figure 19: TrackView Simulation

In the figure, the blue line represents the x-axis while the red line is the y-axis. The first track piece placed is the straight line that starts where these two axes meet. This track piece goes straight up the y-axis, while the inner and outer lines are a specific distance away from the track's line, creating a visible rectangle.

Finally, when the track is saved by the user, each track piece is analyzed in order, and the necessary data for each piece is stored into the saved JSON format file. This file can then be used in two locations: reloading the file into the user interface in order to view or edit it, or executing the track with the Track Parser.

Track Parser BR, TV, AH, AA

The track parser's main goal is to take the track file and turn it into instructions that the microcontroller on the vehicle can interpret. This means that the component will need a method to parse the file and to turn each track piece into a series of instructions that the microcontroller on the vehicle can understand. These instructions are to be put into a vector and sent to the data transmission component.

In order to parse the track file, the component will need to be able to translate the JSON data into individual track pieces. These track pieces will need to be stored in a vector so that they remain in the order that they were given. Once the data transmission asks for the next track piece, the corresponding track piece will be converted to instructions.

The instructions that the converter creates will be specifically formatted and will contain a series of values that will correspond to voltages that the microcontroller will have to apply to different pins to run servos and motors, as well as the time that these voltages need to be applied. Figure 20 shows the level 2 diagram block diagram for the track parser and Table 4 shows the functional requirements for the component.

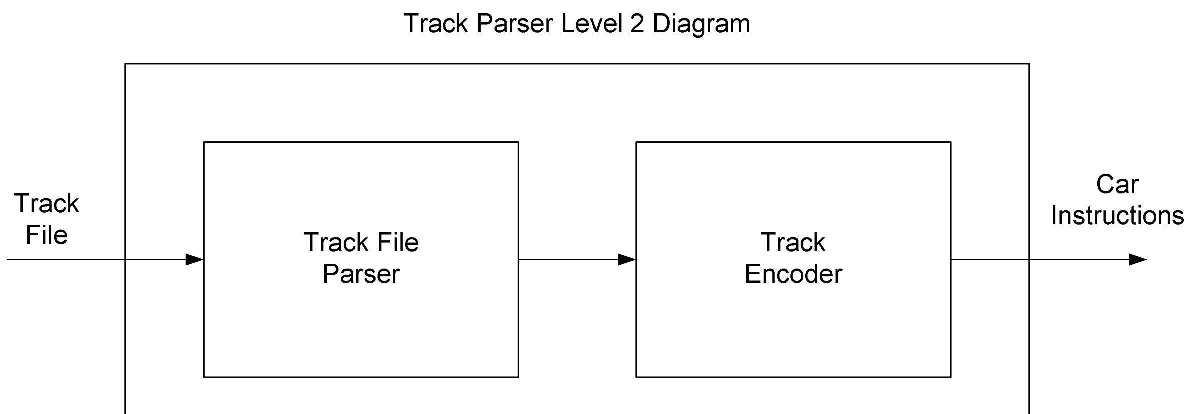


Figure 20: Level 2 Track Parser Block Diagram

Module	Track Parser
Inputs	- Track File
Outputs	- Car Instructions
Functionality	The track file created by the user on the interface is sent through a parsing file, which breaks the track down into several smaller commands. These commands are then encoded as binary messages and are transmitted to the micro-controller as the car instructions.

Table 4: Track Parser Functional Requirement Table

To create the instructions, the converter will use the information given for each track piece and run a series of equations. The known information about each track piece is the length, angle, and turn direction. First, the converter will calculate the instructions for the drive train motor. The radius of the turn angle can be found by using the following equation:

$$R_1 = \frac{L}{\sin(\theta)}$$

In this equation, L is the length of the track piece, and θ is the angle of the track piece. Once the radius is found, the circumference can be found. The circumference can be found using the following equation:

$$C = 2x\pi xR_1$$

The wheel circumference is also needed and can be calculated using the equation:

$$WC = 2x\pi xR_2$$

In this equation, R_2 is the radius of the car wheel, which is a measurable quantity. Then, the arc length of the turn can be calculated by using the equation:

$$A = \frac{\theta}{360}xC$$

The wheel circumference is also needed and can be calculated using the equation:

$$WC = 2x\pi xR_2$$

After A and WC are calculated, the number of revolutions that the wheel will need to rotate to achieve the correct arc length can be calculated using the equation:

$$N_{revs} = \frac{A}{WC}$$

The velocity that the vehicle is to travel will be predetermined in the software. The motor has to be tested at different voltages so that a graph can be created that relates the voltage to the revolutions per minute. To find the needed revolutions per minute needed, the following equation can be used:

$$Rev = \frac{V}{WC}$$

The desired revolutions per minute can then be related to the voltage to find the voltage value that needs to be applied to the motor. To determine the amount of time this voltage needs to be applied the following equations can be used:

$$T = \frac{N_{revs}}{Rev\ x\ \frac{1}{60}}$$

The time value will be in seconds. Also the angle that the turning wheels need to be to make the desired turn for the track piece needs to be converted to instructions. The turn direction will be used to tell the servo whether to turn right or left. To calculate the angle of the wheels, the turning radius needs to be calculated again. The length of the car will also need to be measured. Once the radius and the length of the car are found, the angle of the wheels can be found using the equation:

$$\theta = sin^{-1}(\frac{L}{R_1})$$

A graph will also need to be created that relates the servo movement to the angle of the turning wheels. The calculated angle will then be converted into a voltage that will move the servo to the correct position. After all of these instructions are created, they will be sent to the data transmission component.

To simulate the creation of track pieces and the way that the car will move with this information, a script was created in Matlab that allows for the creation of a track by connecting various points in a graph. An algorithm was created that takes this path as well as a given velocity, and turns it into the resulting path to be followed by the "RC car". The Matlab script and its various functions can be seen in Appendix Figure 1.

The predetermined path is created using two arrays, which hold the x and y points of the path. The traveling object will move from point to point in sequential order, simulating the RC car following track pieces. In this simulation, the traveling object must move one instance of the given velocity before it is given a new direction. After each of these movements, it will then check to see if it passed the current target point. If it did, it will head toward the next target point.

Figure 21 displays the program simulating a half-circle with the traveling object moving with a velocity of 1. It is important to note that it will sometimes pass a point entirely before realizing this and heading for the next. Figure 22 shows the same track when the velocity is reduced to ½. In this simulation, the traveling object moves along the path more accurately because it has more time to realize that it has passed a target point.

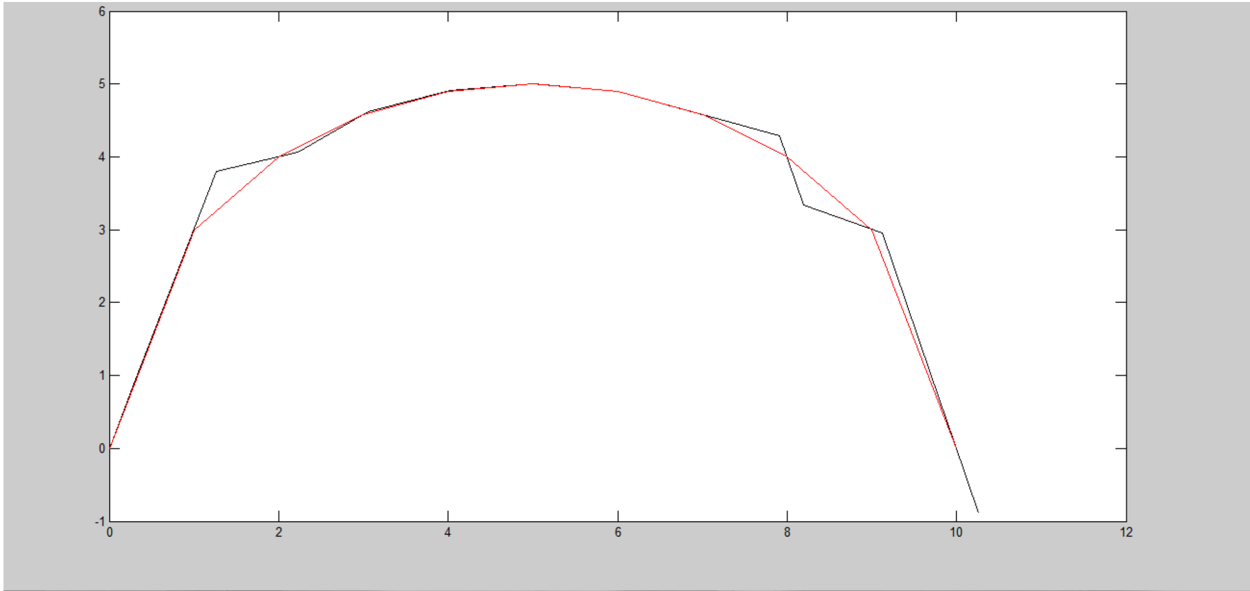


Figure 21: Half Circle with Traveling Velocity of 1

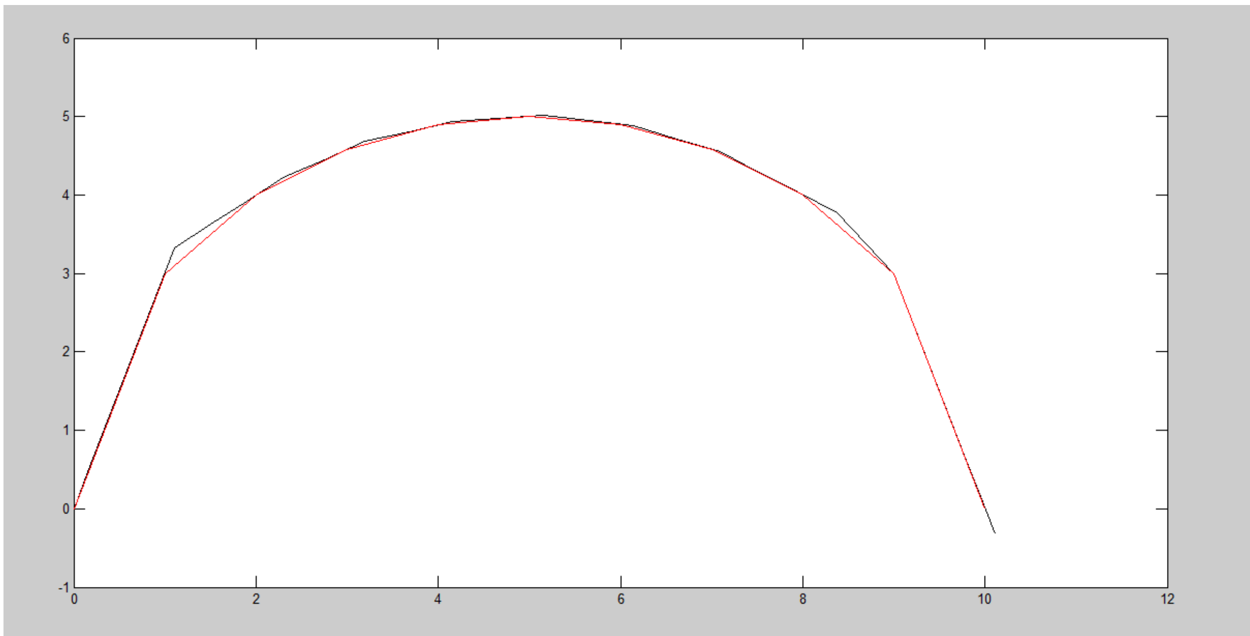


Figure 22: Half Circle with Traveling Velocity of 0.5

Figure 23 showcases the ability of the Matlab simulation to head in any direction, regardless of forward movement. Once again, when the velocity is lowered from 1 as it is in Figure 23 to ½ as it is in Figure 24, the traveling object moved along the desired path more accurately.

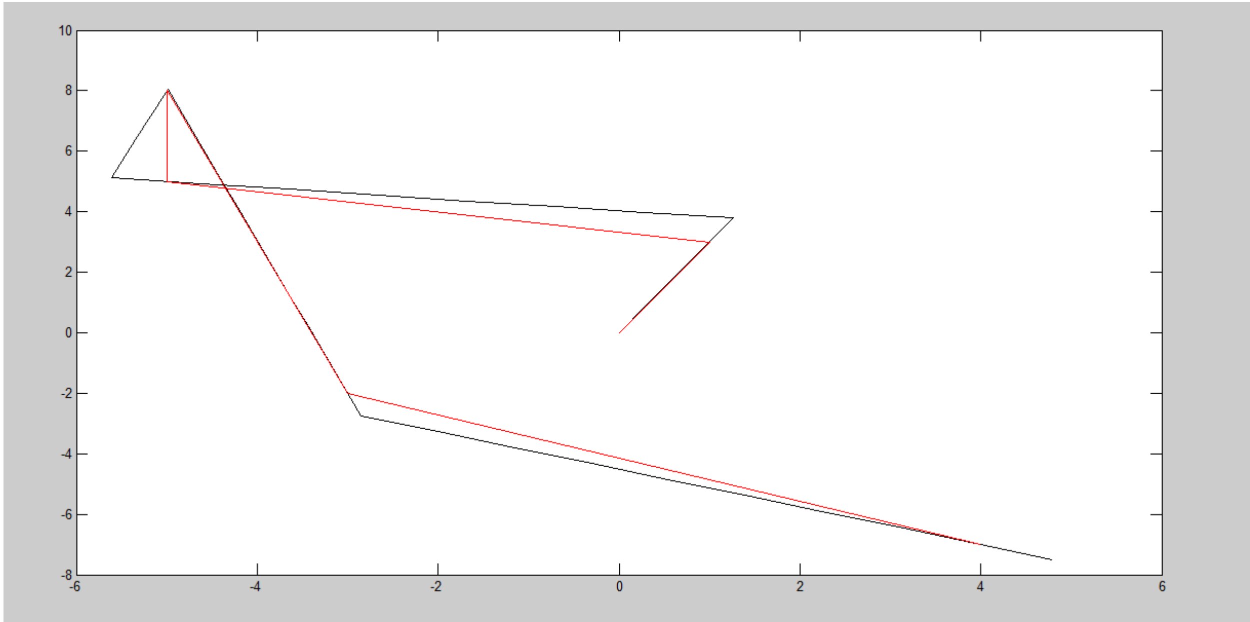


Figure 23: Jagged Path with Traveling Velocity of 1

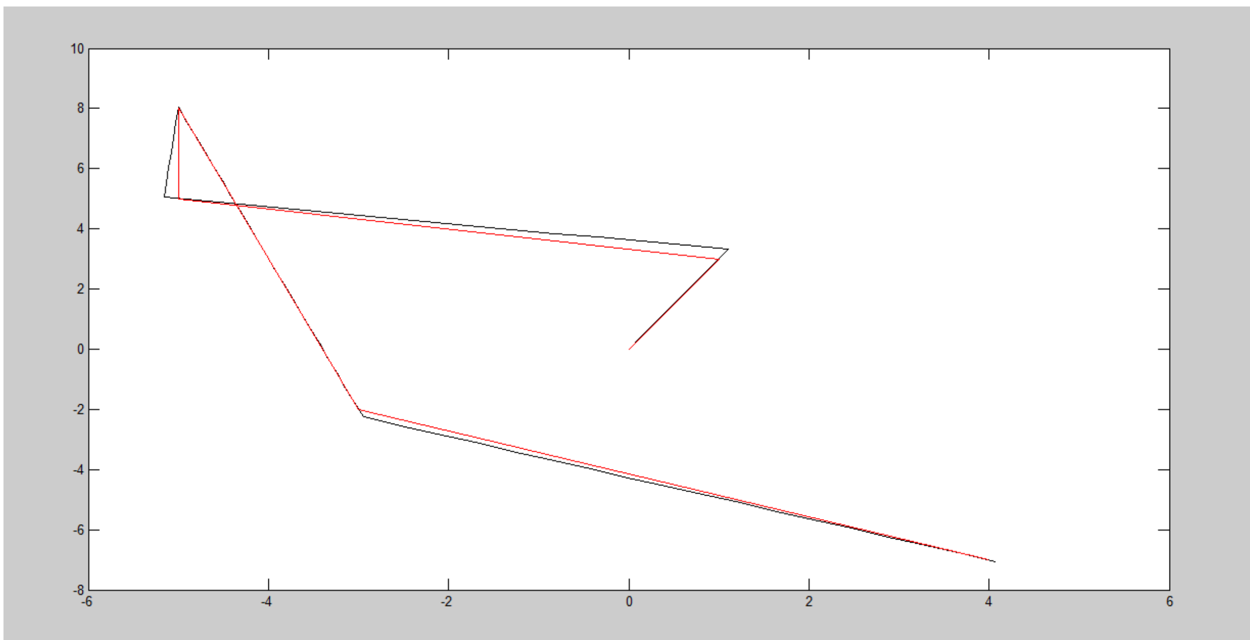


Figure 24: Jagged Path with Traveling Velocity of 0.5

Data Transmission **BR, AA, AH**

One of the key aspects of the entire system will be the ability to both transmit and receive data between both the application and the vehicle's microcontroller. This will be done using Qt networking libraries that open a serial stream that allows free communication between terminals. Several different commands will be encoded and sent. In order to both send and receive these packets, a transceiver will be connected to the PC using the application. In addition, this component must be able to decode any messages sent from the RC vehicle itself, and it must react accordingly.

Bluetooth was chosen due to a variety of factors. Bluetooth operates on the 2400-2483.5 MHz band, which is a regulated band used in a variety of fields. Bluetooth transmits data in packets of divided input data. A packet is sent on a clock-by-clock basis that allows for packet acknowledgment. This technique allows for reliable connections in open-air environments. Ease of use was another factor that led to Bluetooth being chosen. Bluetooth works with a master-slave relationship. In the case of this project, the computer terminal will act as the master while the vehicle will act as the slave. Finally, two technical parameters are important in Bluetooth functionality. Bluetooth works with relatively low power requirements while providing fast data transfer rates. There are three classes of Bluetooth. Class 2 was chosen for this project as it has versatile transmission range of 10 meters and moderate power consumption..

The transceiver will also be receiving signals from the vehicle during this time. Some of these signals will simply be update signals, or "UPD" signals, while some of them will be a collision detection signal. The network code will have to be able to interpret what kind of response signal was sent, and whether or not collision avoidance is needed. Figure 25 displays the Level 2 Block Diagram for this component, and Table 5 shows the corresponding Functional Requirement Table. Also, a diagram of communication flow is shown below in Figure 26.

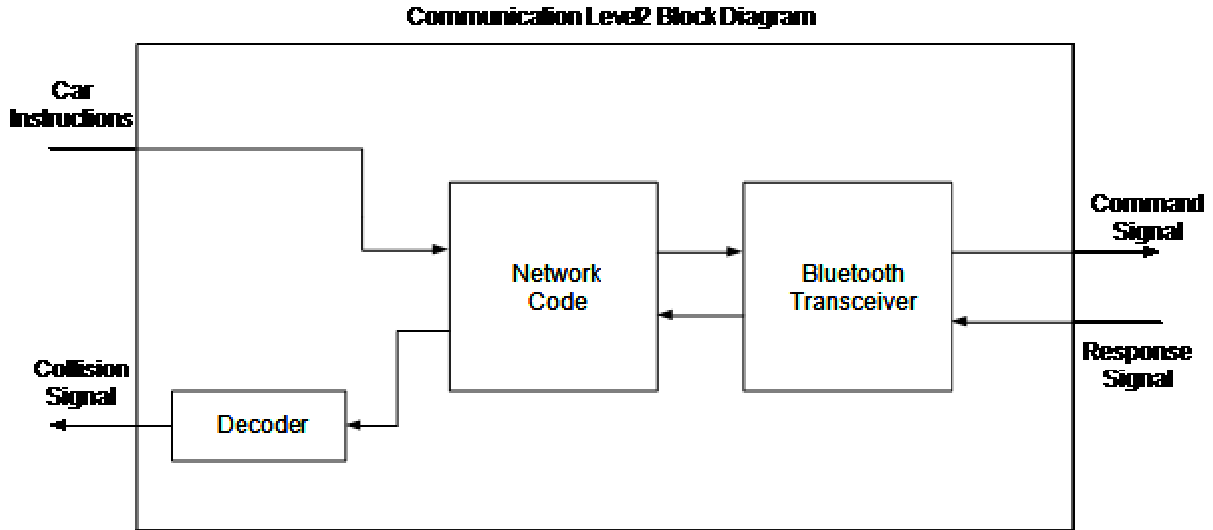


Figure 25: Level 2 Communication Block Diagram

Module	Communication
Inputs	- Car Instructions - Response Signal
Outputs	- Collision Signal - Command Signal
Functionality	Once the PC algorithm determines the instructions for the car, the signal is transmitted through the network code to the Bluetooth transceiver. The signal received by the Bluetooth transceiver is the vehicle command signal, which is relayed to the micro-controller. The vehicle's response signal, which is the IMU information and the collision signal, is transmitted through the Bluetooth transceiver to the network code. This signal is then broken down by the digital decoder, which gives the feedback collision signal to the PC algorithm.

Table 5: Communication Functional Requirement Table

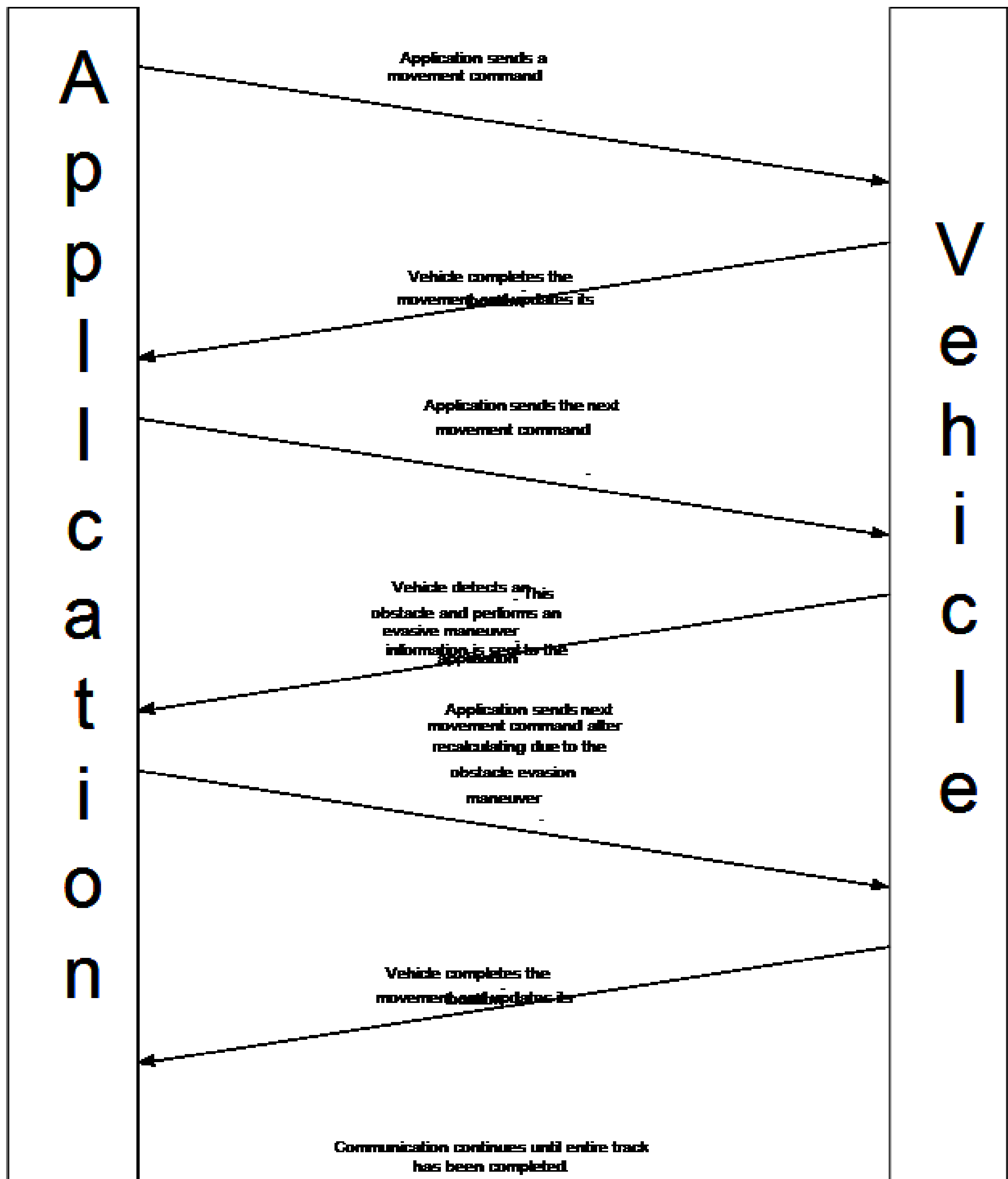


Figure 26: Data flow across communication system.

The vector of encoded track pieces created by the track parser must be sent to the vehicle's microcontroller one at a time. The vehicle will process each of these instructions for a set amount of time—enough to ensure that track piece instructions won't overlap. This, combined with request commands from the vehicle, will ensure that the vehicle receives a single instruction that is up-to-date with the current course correction parameters. To add another failsafe, the incoming commands will be stored locally on the vehicle. The application located on the terminal is being written within the QT framework using C++. Below in Figures 27 and 28, the header and source pseudo-code files are shown displaying the potential plan to develop the communications code. Qt's built-in libraries will be referenced to create code with the necessary functionality.

```
//Summary:  This class will set create a Bluetooth object
//      and open a window to connect to the vehicle.
//      Once the PC and terminal are paired, the Bluetooth
//      object will allow the application and the vehicle to
//      to communicate via a serial connection.

#ifndef BLUETOOTH_H
#define BLUETOOTH_H

//Header files.
#include <QBluetoothLocalDevice>
#include <QBluetoothDeviceInfo>


class Bluetooth : public QObject
{
    Q_OBJECT

public:

    //Constructor.
    Bluetooth();

    //Destructor.
    ~Bluetooth();

    //Setters/Getters if any.

    //Detects other area Bluetooth devices.
    void detectDevices();

    //Sets up connection between PC and vehicle.
    void setupConnection();

    //Sends message to vehicle.
    void sendMessage(Qstring message);

    //Retrieve message from storage.
    QString retrieveMessage();

private:
    //Creates a local device.
    QBluetoothLocalDevice localDevice;
    //Stores name of local device name
    QString localDevice;
    //Sets up server to listen for incoming messages from vehicle.
    QBluetoothServer connection;
    //Stores received messages for processing.
    QString messages[];
```

Figure 27: Pseudo-code header file for Qt Bluetooth communication

```
//Header files.
#include <QBluetoothLocalDevice>
#include <QBluetoothDeviceInfo>
Bluetooth::Bluetooth()
{
    //Check if Bluetooth is available.

    //Turn Bluetooth on.

    //Read and store local Bluetooth device name.

    //Make the Bluetooth device visible to the vehicle.

    //Store the connected devices.

}

Bluetooth::~Bluetooth()
{
    //Send closing message.

    //Close connections.

    //Delete local variables.
}

void detectDevices()
{
    //Find all devices that connected to it during initialization.

    //Store information about detected, connected devices.
}

void setupConnection()
{
    //Create a local server to handle local message IO.

    //Create SIGNAL/SLOT system to detect new incoming messages and...
    //execute the appropriate response.

}

void sendMessage(Qstring message)
{
    //Send message to vehicle.
}

QString retrieveMessage()
{

    //Returns next message from FIFO queue.
    return message;
}
```

Figure 28: Pseudo-code source file for Qt Bluetooth communication

RC Vehicle AA, AH

In order to accomplish the goal of following a user-created track, a vehicle needs to be created that can receive the parsed track data and convert that into the physical movement of the vehicle. For this application, an RC car was chosen as the best option of vehicle. An RC car is an inexpensive, low-power option for a vehicle that can be easily tested out in an indoor laboratory setting. The RC vehicle that was used in this experiment was a simple childhood toy of one of the members of the design team. After stripping the remote control components off the car, the only electronics that remained intact were the 3.3V DC motor that drove the rear wheels, and the rotational servo motor that turned the front wheels. The Level 2 Block diagram and functional requirement table for the vehicle are shown below in Figure 29 and Table 6.

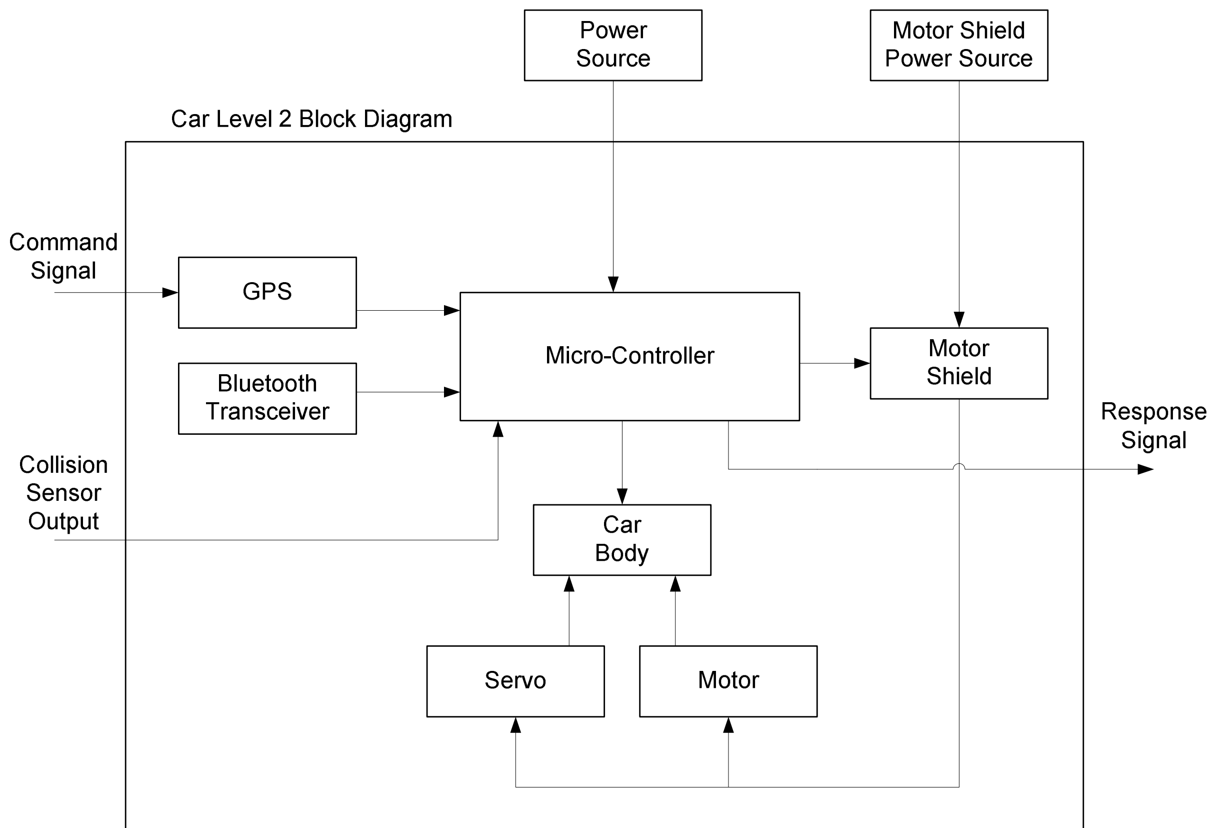


Figure 29: Car Level 2 Block Diagram

Module	RC Car
Inputs	- Command Signal - Object Detected - Power Source - Motor Shield Power Source
Outputs	- Response Signal
Functionality	The power source and the motor shield power source supply the power necessary to operate the micro-controller and the motor shield. The command signal input communicates a binary-coded message from the PC to the micro-controller, which distributes the necessary voltages to the servo and motor through the motor shield to control the speed and direction of the car. An object detection signal is relayed to the micro-controller from the collision sensor. This detection signal and the GPS location are sent back to the PC as the response signal. The response signal is broken down and analyzed by the PC algorithm, and is returned to the car as the command signal.

Table 6: Car Functional Requirement Table

Once the car was stripped of the old electronics, the new electronics were added onto the vehicle. The foundation of the new vehicle is the microcontroller. The microcontroller acts as an intersection between input and output signals and the required response. The vehicle will use an Arduino Mega 2560 microcontroller. This microcontroller was chosen for this application because it has 54 digital IO pins, 16 analog IO pins, 256k of memory, and runs off a 16 MHz clock. This board has more than enough IO pins to run all the onboard electronics, has sufficient memory for programming, and meets the low power requirements desired by the engineering requirements. The microcontroller layout is diagramed in Figure 30. Also, the wiring pinout required by the onboard embedded components can be found as Figure 31. These pin requirements will be explained later.

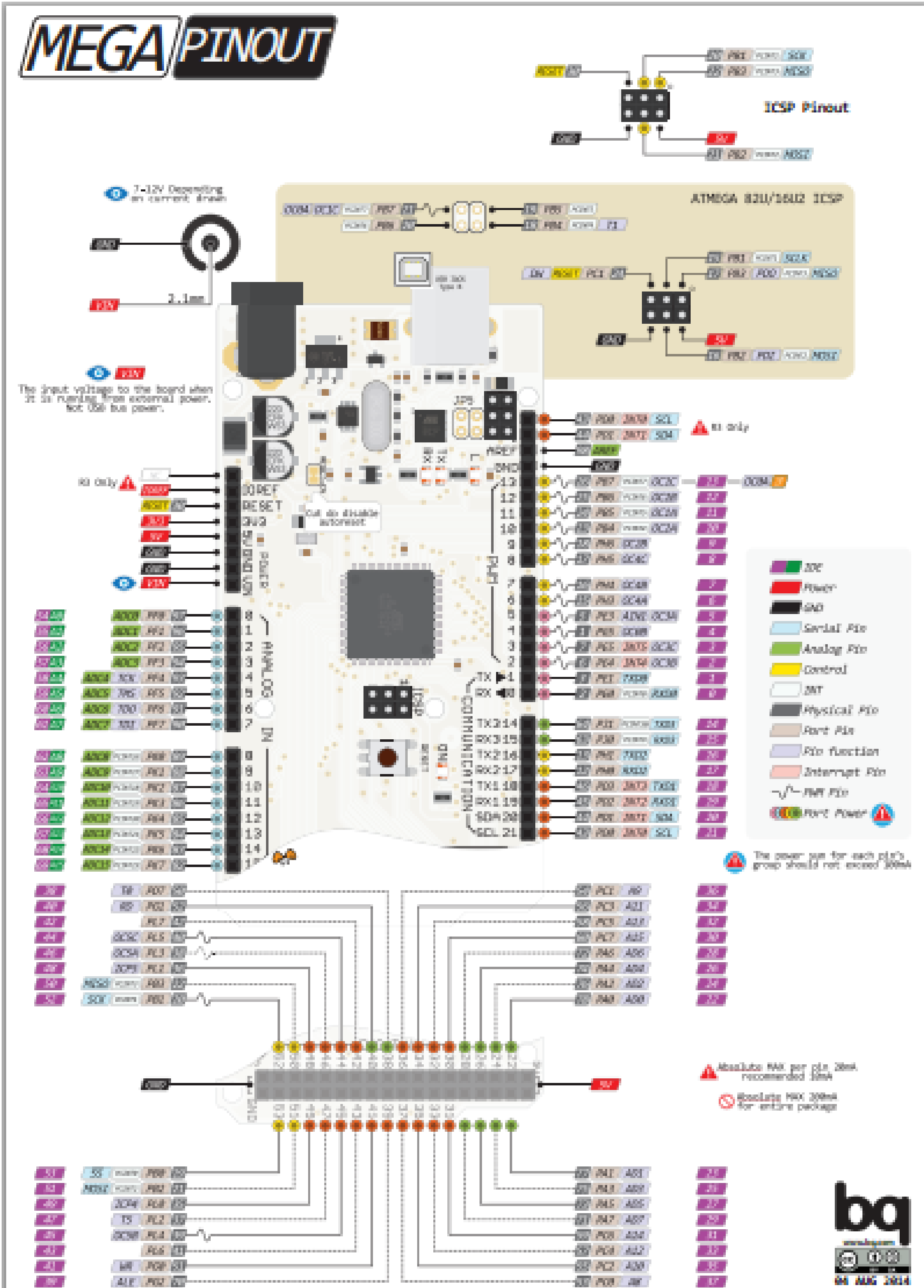


Figure 30: Microcontroller Layout

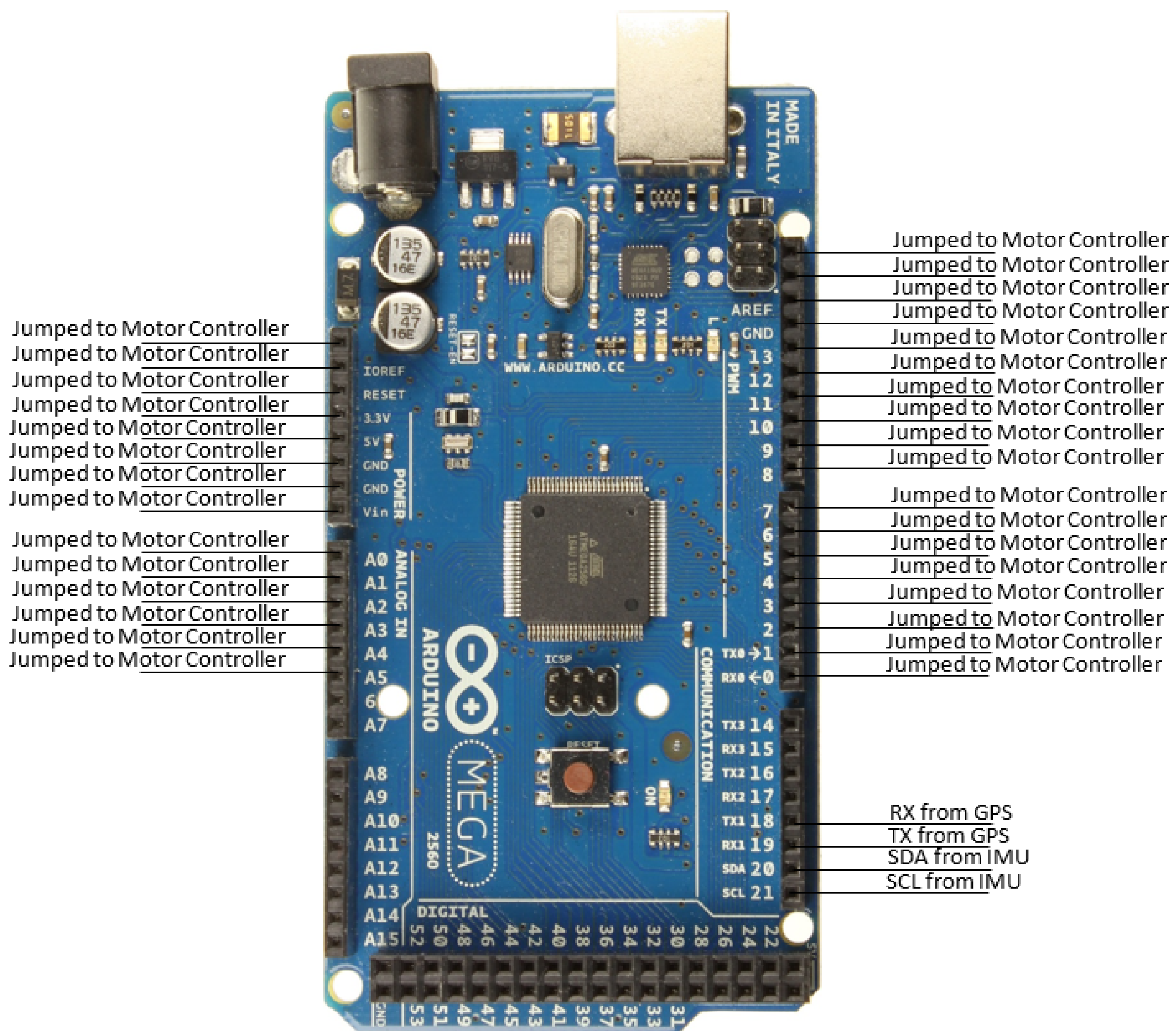


Figure 31: Arduino Mega 2560 Microcontroller Pinout

This microcontroller will be coupled with a motor shield, namely the Adafruit Motor Shield V2. The motor shield is a specialized full-bridge driver that steps up the input voltage and current to match the needs of the DC motors driving the vehicle. The motor shield takes input signals from the microcontroller and provides the correct gain to properly drive the motor. The signals to drive the motors would be determined by the PC and sent to the microcontroller. This information would include encoded instructions that the microcontroller would pass on as voltage signals to the motors. The pin requirements for the Motor Shield can be found on Figure 32. These pin requirements will also be explained later.

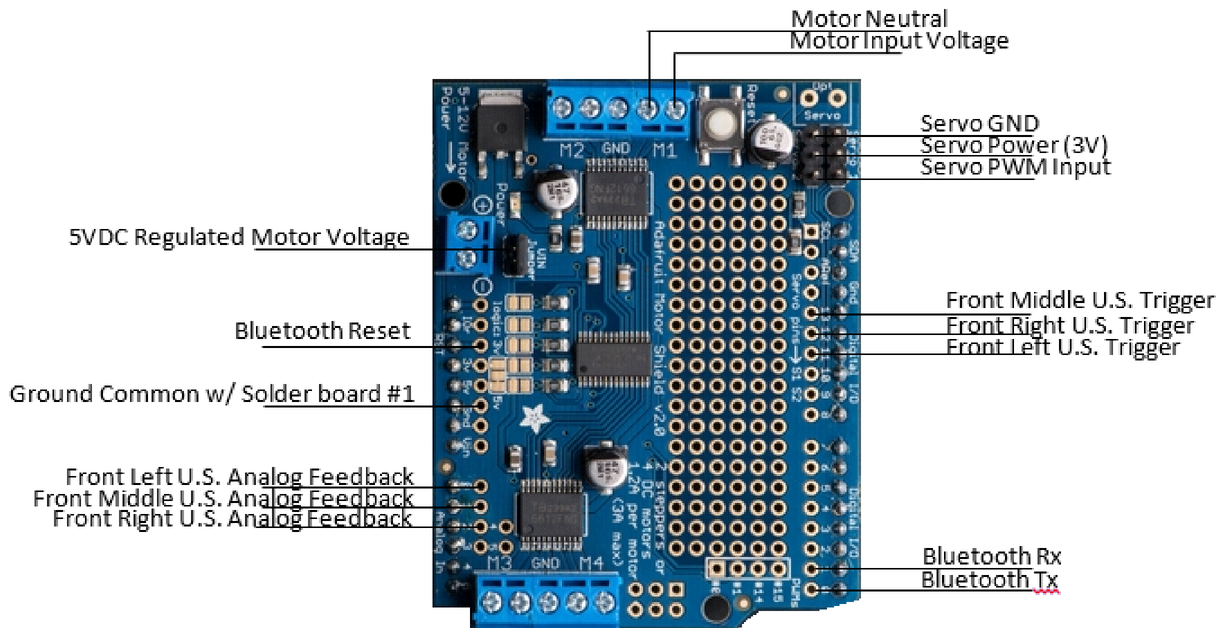


Figure 32: Adafruit Motor Controller Pinout

One of the most significant functions of the RC vehicle is wireless communication. Communication for between the vehicle microcontroller and the PC was determined to be over the Bluetooth frequency band. Using a Bluetooth transceiver, the RC vehicle would send motor and servo commands to the vehicle, and report sensor information to the PC with the microcontroller acting as a middleman. The pinout for the Bluetooth communication device can be found below as Figure 33.

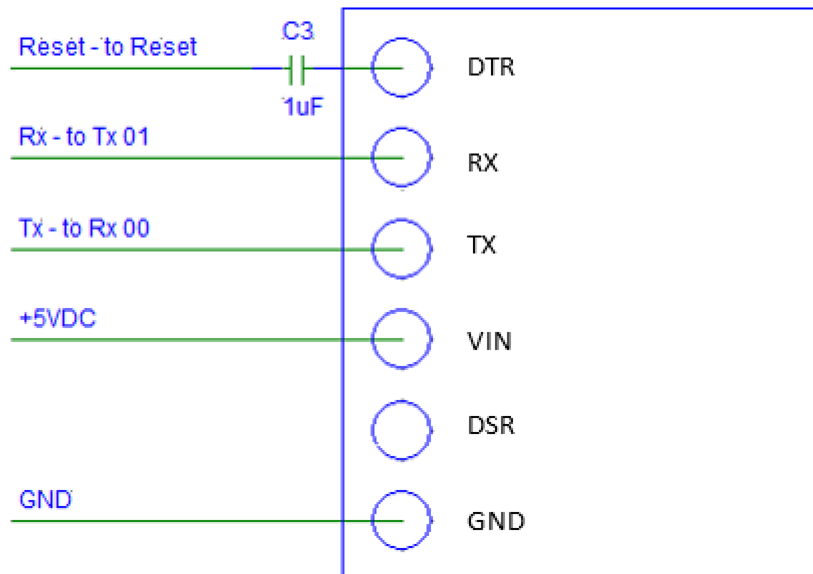


Figure 33 – Bluetooth Pinout Diagram

Along with the microcontroller, motor shield, and Bluetooth transceiver, there are several more pieces of embedded hardware that are used on the vehicle. An important portion of the embedded hardware is the inertial measurement unit (IMU). The inertial measurement unit chosen for this project has an onboard gyroscope, accelerometer, and compass. Using a filter, which will be explained in greater detail in the Collision Detection/Avoidance section, an accurate location can be determined in real-time to determine variation from the given track. This location is based on the acceleration and velocity determined by the accelerometer and the direction determined by the compass. If this occurs, algorithms would execute that compute a corrective movement for the vehicle. If the vehicle strays from the track, this information will be used to determine how to return to the desired path. The pinout for the IMU device can be found below as Figure 34.

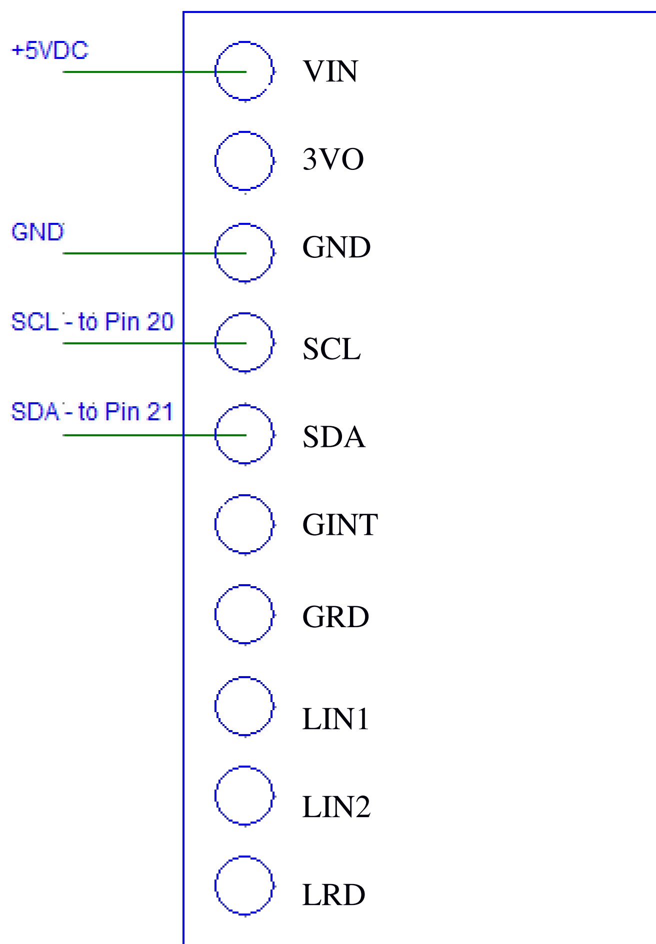


Figure 34 – Internal Measurement Unit Pinout Diagram

Another sensor that could be used to determine vehicle state would be a GPS module. The module would be used to track the current location of the vehicle. Initially, the GPS unit was intended to be used to track the actual position of the vehicle as it moved along the track. However, the unit was not able to provide precise enough positional feedback, so the IMU was used in its place for that purpose. The GPS was left connected to the vehicle circuit for future applications. The GPS device pinout can be found below as Figure 35.

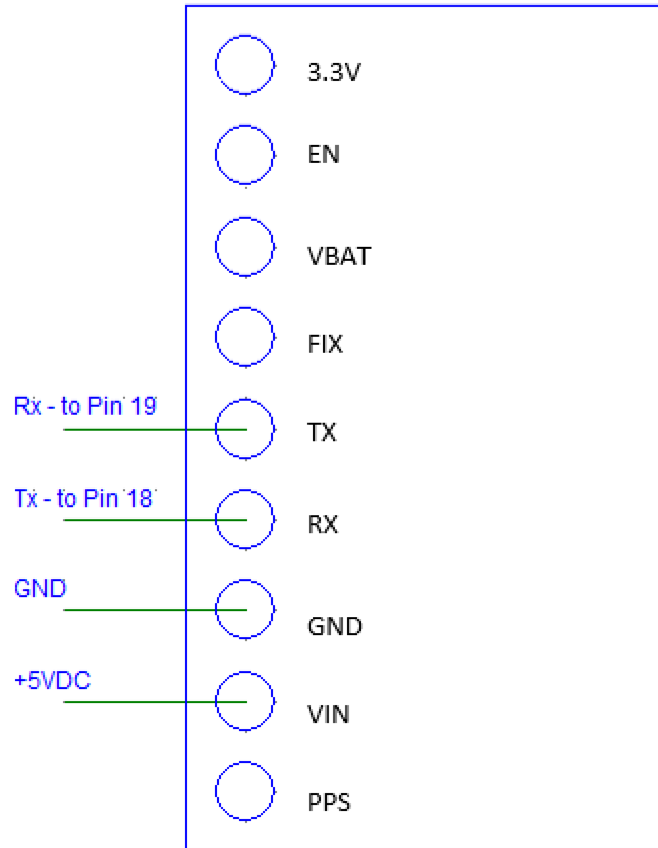


Figure 35 – GPS Pinout Diagram

In order to provide enough power to supply the motors, servos, and embedded hardware, great care was taken into the choice of power supply for this project. Initially, the vehicle was designed to run off of a 9 VDC batteries as the power supply. This power supply has been chosen because it is powerful enough to supply enough power to operate all of the onboard electronics as well as power the motors and motor drives. To calculate the maximum power required by the onboard electronics, the maximum current draws for each component were found by looking at the datasheets associated with the respective components. The power requirements for each component, as well as the overall amp draw and power draw for the onboard components are compiled in Table 7. Using a typical 9VDC alkaline battery, the minimum runtime of the vehicle from the battery can be determined from assuming the maximum amp draw is constantly used by the vehicle. With the average 9V alkaline battery being 600 mA·hrs, the vehicle would have a minimum runtime of 2.92 hours. This is also shown in Table 8.

Power Calculations for RC Car			
Component	**Max Power Draw (mW)**	**Max Amp Draw (mA)**	**Operating Voltage (V)**
Ultrasonic Range Detectors	30 (10mW each)	6 (2mA each)	5
GPS	220	44	5
Servo	250	50	5
Bluetooth	125	25	5
Relative Positioning	30	6	5
	625	**125**	

Table 7: Power Calculations Table

Battery Runtime (mA/h)	Max Amp Draw (mA)	Total Runtime (hrs)
600	125	4.8

Table 8: Vehicle Runtime Calculation Table for 9VDC Alkaline Battery

However, the one factor that is not included in these measurements is the current being drawn by the motor. This current draw varies not only with the velocity and acceleration required of the motor, but the current draw also increases as the input reference voltage for the motor shield rectifier varies. As the alkaline battery was used more over a period of time, the voltage provided by the battery would decrease due to the motor load. As less of a voltage was sourced, more current was required by the motor to move the vehicle. With this being factored in, the alkaline battery was only getting about 30 minutes of use out of each battery before there was not enough power available by the battery to drive the motor. With less power available to the motor, the vehicle was moving much slower than normal, and was unable to travel the desired lengths and angles that were desired. By having a more constant, regulated voltage as the motor shield reference, the motor would have a much more consistent voltage to reference for the motor output, meaning more consistent, repeatable results.

To solve this, one 12 VDC Nickle-Metal Hydride rechargeable battery and one 7.5 VDC Nickle-Metal Hydride battery were used in place of the 9 VDC alkaline batteries. The 7.5 VDC battery was connected only to the Arduino Mega microcontroller. By having this component on a separate power source, there is less noise that is seen in the other embedded components and the microcontroller can operate independently of these components. The 12 VDC NiMH battery is used as the source for the onboard embedded components, namely the ultrasonic range detectors, Bluetooth device, IMU device, and GPS device and motor shield. The NiMH has 2000 mAH of stored power amounts to 16 hours of use, which compared to the alkaline battery, proves that it is the better choice. These calculations are found in Table 9.

Battery Runtime (mA/h)	Max Amp Draw (mA)	Total Runtime (hrs)
2000	125	16

Table 9: Vehicle Runtime Calculation Table for 12 VDC NiMH Battery

All these components operate off a 5 VDC input. To step the 12 VDC down to 5 VDC, a voltage regulating circuit will be introduced. This regulation will be accomplished by introducing the voltage regulation circuit shown in Figure 36. This circuit utilizes the LM7805 voltage regulator that regulates input voltage from a range of 5-18 VDC to a nominal output voltage of 5 VDC. Two capacitors, one at the input (10 µF) and one at the output (1 µF), will be added in parallel to clean up any variance or ripple in the signal.

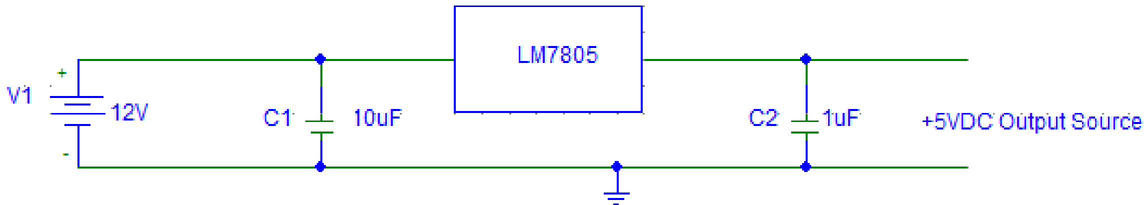


Figure 36: 12 VDC to 5 VDC Voltage Regulator Circuit

The 5 VDC output from this circuit acts as the input source for the IMU, GPS, and Bluetooth devices as shown below in Figure 37. Figure 37 also shows the input and output wires from each device that will be connected back to the microcontroller. The circuit connected in Figure 34 was constructed on a solder board and was mounted onto the vehicle.

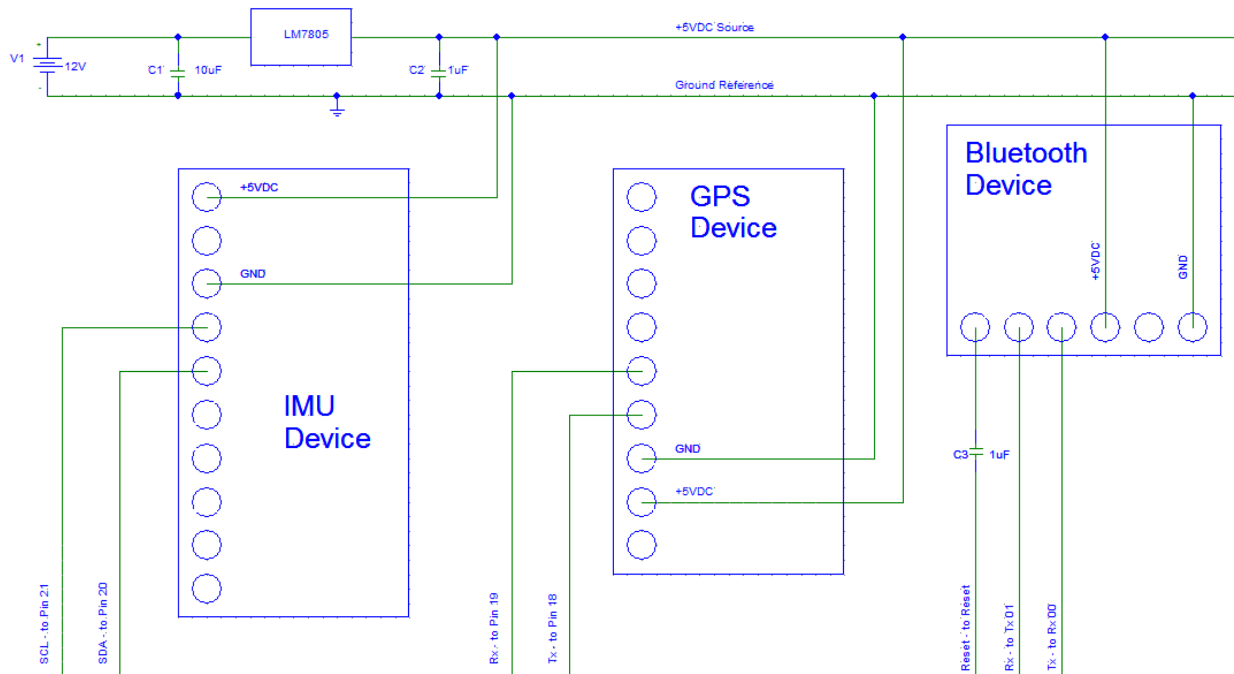


Figure 37: Solder Board #1 Circuit Design

As shown above in Figure 37, each device used only requires certain inputs and outputs to accomplish the tasks that are required for this application. The IMU uses only the SCL (serial clock) and SDA (serial data) pins to communicate accelerometer, gyroscope, and compass data back to the Arduino controller. The serial data is transmitted every time the serial clock pin is pulsed, which is a parameter that can be set by the user. The GPS device communicates the relative position of the vehicle through the Rx and Tx pins. Transmissions (Tx) from the GPS are received by the microcontroller (Rx). The opposite is true for transmissions from the microcontroller to the GPS. The communication pins used on the microcontroller for the GPS data transfer are Digital I/O pins 18 and 19, respectively. The same communication scheme of transmitting and receiving data is true for the Bluetooth device. This communication path is chosen between Digital I/O pins 0 and 1. Also, a reset function is required by the Bluetooth for testing purposes. This Reset pin is connected through a 1 μF shunt capacitor to the Reset pin found on the motor shield.

Below in Figure 38 a picture of the completed RC vehicle is shown. The electronics were mounted on a board made from fiberglass. Fiberglass is a non-conductive material, so it serves as a good medium with which to mount the devices. Brackets were also made to mount the ultrasonic range detectors onto the front of the car. The use for these devices will be explained later in the report. The vehicle, as designed, meets both the size and weight requirements that were outlined in the engineering requirements for the project.

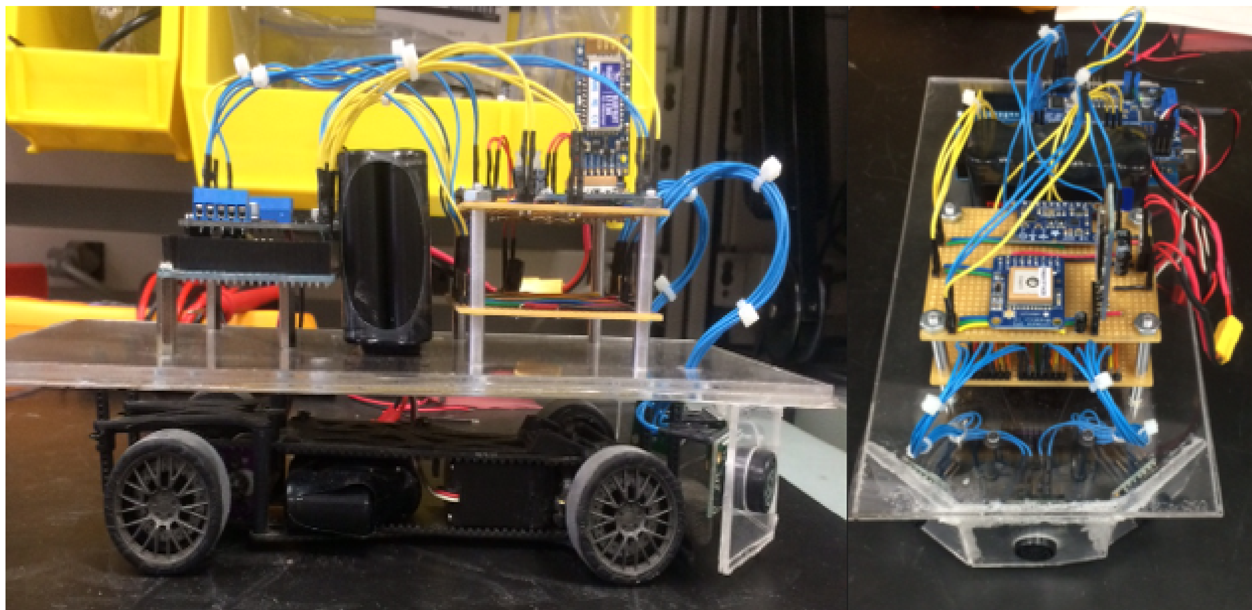


Figure 38: Photographs of vehicle used in project application (top and side view)

The microcontroller is controlled by embedded C code compiled in the open-source environment. This code will be programmed to gather sensor data, control the motors, and communicate with the terminal. A portion of the code will be functionality referenced from open-source libraries while the rest will be developed as a custom solution for this project. Many of the libraries were from Adafruit repositories. Ada fruit manufactured many of the components used for the project, and their libraries were used for simple interfacing between devices. The custom solution will function out of a header file that encapsulates all functions needed for the vehicle to operate.

Arduino code is broken into two, fundamental sections. Before these sections are entered, all functions to be used are defined in a header file. The specialized header file has functionality for course correction, obstacle detection, and code decoding among others. A sample of pseudo-code for the header file is shown below in Figure 39.

```
//Project.h file.
//Defines project related functions for use throughout the loop() block in Main.c.

//Required header files:
//Open-source libraries.
#include <TinyGPS.h>
#include <LSM303.h>
#include <Adafruit_MotorShield.h>
#include "utility/Adafruit_PWMServoDriver.h"
//Arduino libraries.
#include <Wire.h>
#include <SoftwareSerial.h>
#include <Servo.h>

bool checkCoordinates( double lat, double)
{
    //Checks the difference between current location and where the vehicle is supposed to be.
    //Returns true if coordinates are within a preset error threshold.
}

void correctCourse()
{
    //Runs course correction algorithm.
}

double filterCoordinate(double coord)
{
    //Filters noise from GPS coordinates.
}

double decodeCommand(string command)
{
    //Takes string from command and decodes the required speed, time, and angle.
}

double receiveLat()
{
    //Receives latitude coordinate from GPS module.
}
double receiveLon()
{
    //Receives longitude coordinate from GPS module.
}

int checkProximity()
{
    //Checks senser voltages to determine proximity.
    //Returns code if a vehicle is too close in one direction.
}
```

Figure 39: Pseudo-code header file of the embedded code

The first, fundamental section of Arduino code is *setup()*. Before *setup()* is entered, all global variables used in the code are defined. Once these declarations are complete, control moves to the *setup()* block. In *setup()*, previously defined declarations are initialized. As shown in the pseudo-code below, the Arduino is commanded to enable serial data transmission. Also, a local Bluetooth connection is set up using a third-party module so that the vehicle and application terminal can communicate. Following Bluetooth initialization, the ultrasonic sensors are configured to the proper pins. Lastly, the motor shield is initialized. The motor shield communicates with the microcontroller using I2C in a serial format. Below, a simplified flowchart shows the general process of the *setup() block* in Figure 40. Then, in Figure 41 and Figure 42, pseudo-code for everything up to and including the *setup()* block is shown.

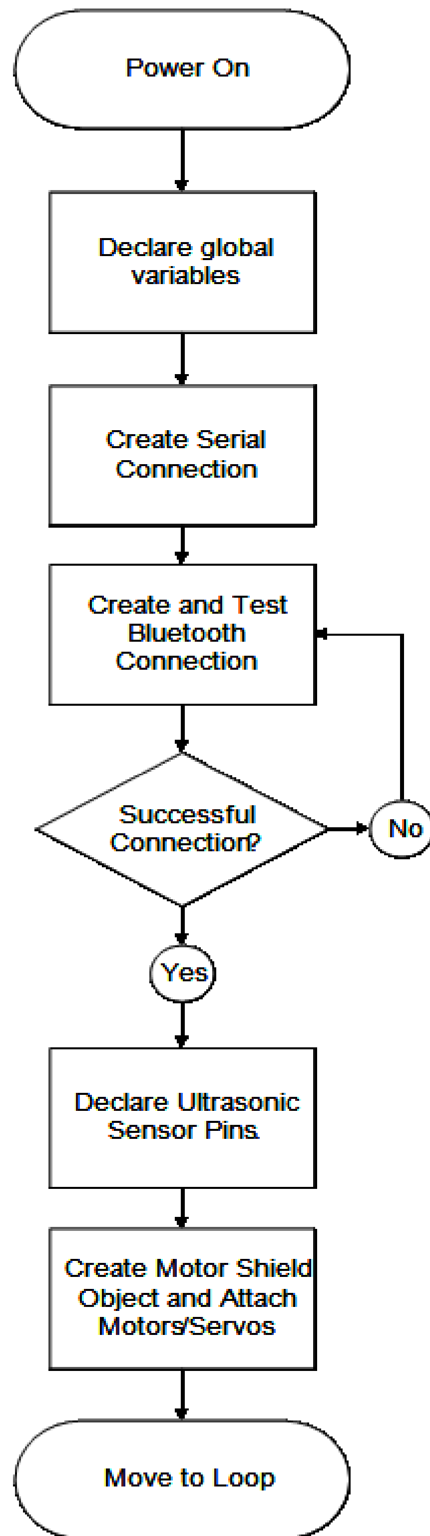


Figure 40: Flowchart of Arduino code setup

```
//Open-source headers.
#include <TinyGPS.h>
#include <LSM303.h>
#include <Adafruit_MotorShield.h>
#include "utility/Adafruit_PWMServoDriver.h"

//Arduino headers.
#include <Wire.h>
#include <SoftwareSerial.h>
#include <Servo.h>

//Custom header.
#include "Project.h"

//Bluetooth variables.
SoftwareSerial bluetooth(pin, pin); //Rx, Tx
char bluetoothData;
int baudRateBT;

//Serial variable.
int baudRate;

//Motor variables.
int defaultSpeed;
//Speed from user PC command.
int speed;
//Motor run time.
int time;
//Servo turn angle.
int angle;

//GPS object declaration.
//GPS bootup time <40s.
GPS gps;

//Motorshield declaration.
Adafruit_Motorshield motorShield = Adafruit_MotorShield();
//Motor declaration.
Adafruit_DCMotor *motor1 = motorShield.getMotor(1);
Adafruit_DCMotor *motor2 = motorShield.getMotor(2);
//Servo declaration.
Servo servo1;
Servo servo2;


void setup()
{
```

Figure 41: Variable Declaration Pseudo-code for the Arduino

```
void setup()
{
    //Serial library setup.
    Serial.begin(baudRate);

    //Bluetooth connection setup.
    bluetooth.begin(baudRateBT);
    bluetooth.printIn("Initialization message...");

    //Ultrasonic sensor declaration.
    //Repeat for each individual sensor.
    pinMode(trigPin, OUTPUT);
    pinMode(echoPin, INPUT);

    //Motorshield declaration.
    motorShield.begin();
    servo1.attach(pinA);
    servo2.attach(pinB);
    motor1->setSpeed(defaultSpeed);
    motor2->setSpeed(defaultSpeed);
}
```

Figure 42: *Setup()* block of the Arduino code

After the *setup()* block is completed, control moves to the *loop()* block of the code. The *loop()* block of Arduino code is where all action takes place. As can be surmised by the name, the *loop()* block is an infinite loop. When compiled and uploaded to the microcontroller, the code will run indefinitely unless a pre-programmed stimulus causes the code to break out of the loop. For this action, several actions take place every time the loop is completed. First, data is read from the Bluetooth serial stream. This data will include the newly requested command that will be run during the current loop cycle. This command will then be checked for errors prior to the car following its instructions. This prevents damage to the vehicle's mechanical componenets. Second, the next command to be followed will be run prior to execution. The command will be dissected to recover pertinent information for the microcontroller's servo and motors. This information includes, angle, time, and voltage. Before executing the next command, the third step in the *loop()* block will check the proximity readings of the ultrasonic sensors. If an obstacle is near, the motors will be sent an evasive maneuver command. An algorithm will then be completed to find the best route to return to the correct path while avoiding the obstacle. If no obstacle is detected, the fourth step is to run the command. While the movement is being completed, the IMU data stream will be check to determine the vehicle's location. The inertial measurement unit's sensor data will first be filtered to remove excess noise from the signal. Once the acceleration and angular velocity values are ready, the information will be sent to the

terminal for course correction. The information received by the application terminal from the inertial measurement unit is then used to determine the accuracy of the vehicle's position relative to the desired location. If a discrepancy is detected, the final action of the *loop()* block is to calculate a corrective path that will return the vehicle to the proper path. In the figures below, a flowchart outlines the actions taken in the *loop()* block, and pseudo-code outlines the potential code that will be developed.

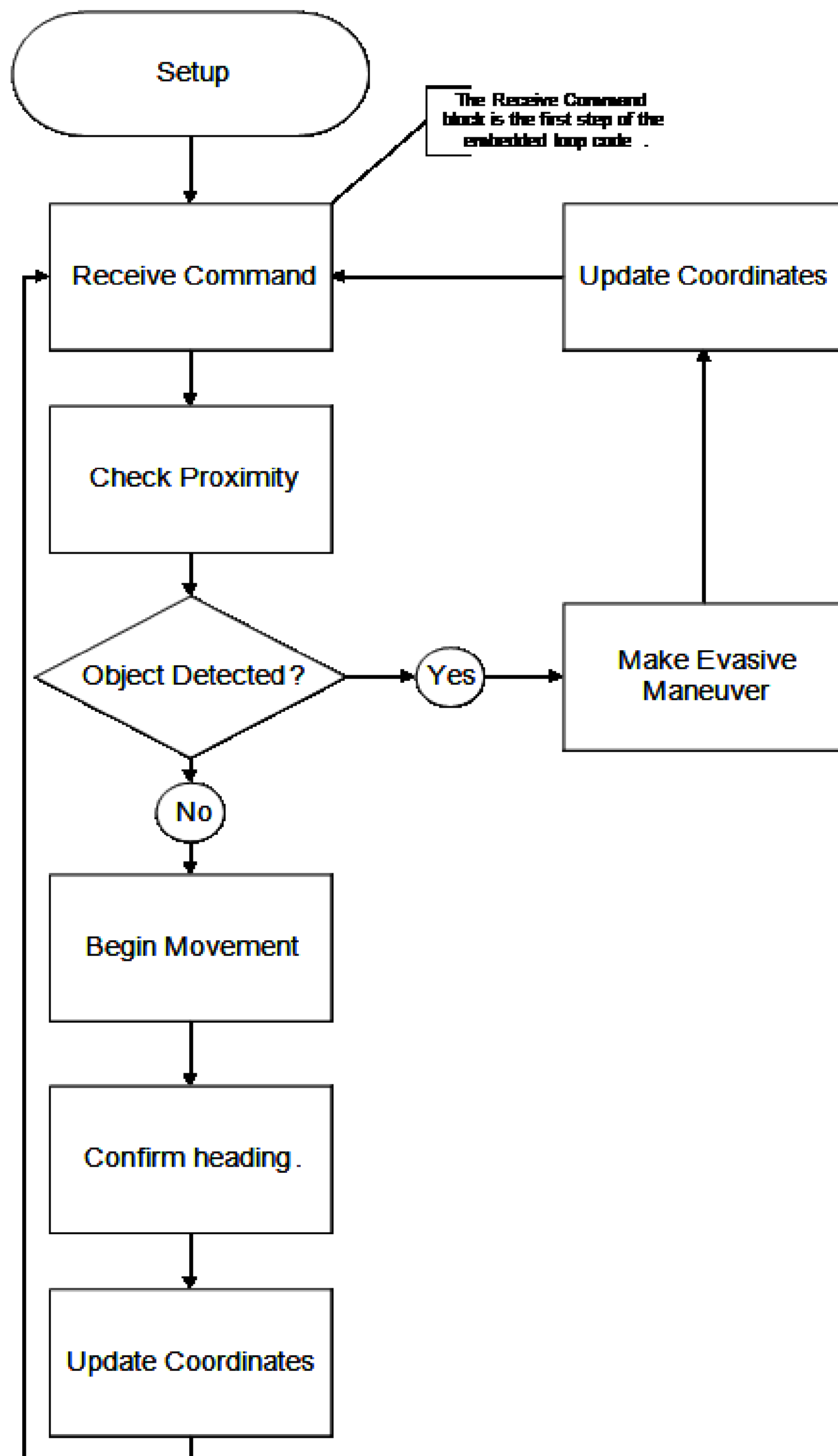


Figure 43: Flowchart of the loop

```
void loop()
{
    //Local variables:
    int i = 0;
    //Flag for new coordinates.
    bool gpsDataFlag  = false;
    bool location = false;
    //GPS storage variables.
    double lat;
    double lon;
    double newLat;
    double newLon;
    //Speed from user PC command.
    int speed;
    //Motor run time.
    int time;
    //Servo turn angle.
    int angle;
    //Sensor flag.
    int sensorFlag = 0;
```

Figure 44: Local *loop()* variables

```
    //Receive command via Bluetooth connection.
    if(bluetooth.available())
    {
        //Read and parse bluetoothData.
        bluetoothData=bluetooth.read();
        //Decode command.
        decodeCommand(blueToothData);
        sensorFlag = checkProximity();
        //
        if(sensorFlag)
        {
            //Perform evasive maneuver.
        }
        else if(sensorFlag)
        {
            //Perform evasive maneuver.
        }

        //Run new command.
        for(i=time; i!=0; time--)
        {
            //Set motor/servo values for proper movement.
            servo1.write(map(i, 0, 255, 0, angle));
            servo2.write(map(i, 0, 255, 0, angle));
            motor1->setSpeed(speed);
            motor1->setSpeed(speed);
        }

        //Result if new data arrived.
        if(gpsDataFlag)
        {
            receiveLat();
            receiveLon();
            //Store new coordinates.
            lat = filterCoordinate(lat);
            lon = filterCoordinates(lon);

            //Check location.
            location = checkCoordinates(lat, lon,  newLat, newLon);
            if(location != true)
            {
                //Moves vehicle to expected location.
                correctCourse();
            }
        }
    }
}
```

Figure 45: Embedded *loop()* code

Collision Detection / Avoidance BR, TV, AA

The collision detection and avoidance component is unique in that half of it takes place in the RC vehicle's hardware while the other half takes place in the application's software. The detection half requires that a proximity sensor on the vehicle is able to detect any objects obstructing its path. Meanwhile, the collision avoidance half of this component must use an algorithm to avoid that obstruction.

The proximity sensor must have a long enough range to allow the vehicle to both acknowledge the obstruction, send a collision PDU to the application, and receive a response generated by the collision avoidance algorithm. In order to properly sense any obstructions, the sensor must be mounted as far in the front of the car as possible. It may also need some proximity sensors on each side to help the algorithm determine which way to go to avoid the obstruction.

The collision algorithm must take in the information provided by these proximity sensors, as well as the vehicle's current and target locations in order to help the vehicle avoid the obstruction and get back onto the desired path. This means that it will have to create new instructions for the vehicle to follow and send them through data transmission. It is also possible that the obstruction is in the way of one or more of the vehicle's target locations. If this is the case, the algorithm must recognize this and skip these locations, preferring to move forward instead. Figure 46 contains this component's block diagram, and Table 10 contains its corresponding Functional Requirement Table.

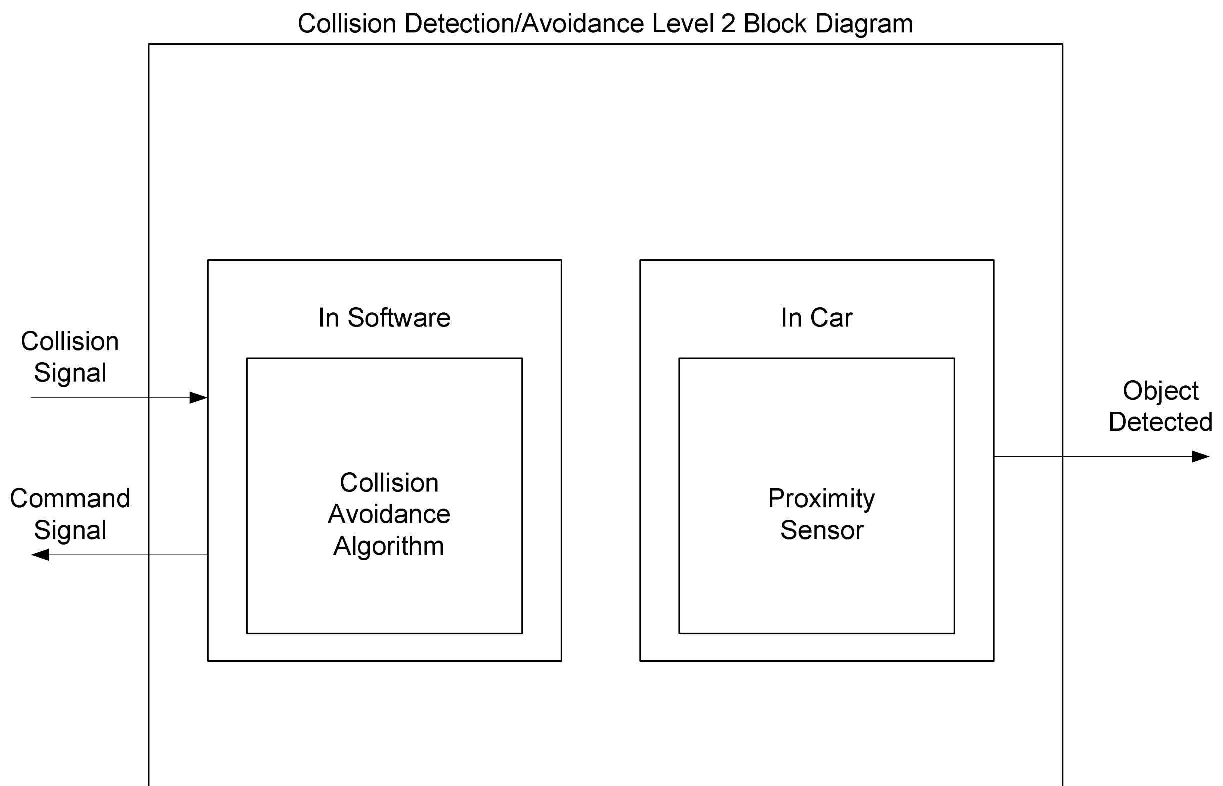


Figure 46: Level 2 Collision Detection / Avoidance Block Diagram

Module	Collision Detection/Avoidance
Inputs	- Collision Signal
Outputs	- Command Signal - Object Detected
Functionality	Once the proximity sensor on the car senses a nearby object, an object detection signal is sent from the car to the PC software. The incoming collision signal is then processed by the collision avoidance algorithm, which determines the path correction for the vehicle to avoid the object. The corrected path is then sent as the command signal from the software back to the micro-controller on the car.

Table 10: Collision Detection / Avoidance Functional Requirement Table

Another component of the microcontroller feedback to the system is the error tracking of the position of the vehicle. In order to verify that the vehicle has traveled along the desired path, real-time data needs to be fed back to the microcontroller in order to determine its actual position compared to its desired position. To do this, an onboard inertial measurement unit (IMU) is going to be used. This IMU has 6 degrees of freedom, meaning that the accelerometer component of the IMU has 3 degrees of freedom and the gyroscope component has 3 degrees of freedom. Using these components, the IMU will be able to sense what direction the vehicle is turning, and how fast the vehicle is turning. With this data fed back to the microcontroller, the position of the vehicle can be tracked through incorporating a complementary filter. A complementary filter, in its most basic sense, utilizes a combination of a high pass filter, a low pass filter, and numeric integration to determine more accurate angle and angular velocity measurements than can be found through direct measurements from the IMU. The sensor data collected by the IMU has a tendency to be noisy and can vary greatly from what the actual value is. The complementary filter eliminates this noise and will give a more accurate description of the actual position of the vehicle by focusing on calculating a better angle measurement for the direction of the path of the vehicle. The complementary filter uses the following equation to accomplish this:

$$\theta = 0.98(angle + gyroData\ x\ dt) + 0.02(accData)$$

The angle for the current time interval, θ, is not simply taken from the gyroscope data, rather it is the integration of the gyroscope data added to the angle of the previous time step. This is then added with the accelerometer data, which is used as an angle measurement using the arctangent function. These two quantities are multiplied by scalar constants, whose sum equals one. The past angle and gyroscope integrated sum is weighted much more than the accelerometer angle, as seen in the equation above. This weighting ensures that the measurement values will not drift in the long term, and the measurements will be very accurate in the short term.

This type of filter utilizes 3 main mathematical functions in order to filter out the unwanted noise: integration, a low-pass filter, and a high-pass filter. The integration that is used takes a summation of the angular rotation of the vehicle over one time interval through

integrating the angular velocity measurement from the accelerometer with respect to time. This gives the change in angular position since the last sampling instance. This is then added to the angular position of the last sampling instance, which results in an approximation of the angular position of the vehicle for the current sampling instance. This results in having incorporated the equivalent of a low-pass filter to the data. It is filtering out short-term fluctuations in the vehicle's angular position, and only allowing long term changes to be experienced. Any change in position is going to be incremental instead of instantaneous. The small scalar constant multiplied with the accelerometer angle also acts as a low-pass filter by only allowing that quantity to have a small impact on the current angle. The larger scalar constant multiplied with the gyroscope integration and previous angle acts as a high-pass filter. Acting in the opposite manner of the low-pass scalar constant, it allows for those quantities to have a larger impact on the current angle than the accelerometer angle and prevents the measurements of the vehicle's path to drift over time. The high-pass filter and the low-pass filter operate on the same time scaling, so they are being sampled at the exact same frequency.

Through comparing the desired vehicle position with the position measured by the complementary filter, the positional error of the vehicle can quickly be calculated. The error will be used to adjust the next movement command to bring the vehicle back to the desired position.

For the purpose of collision detection, three ultrasonic range detectors were attached to the front of the vehicle. By having one face straight ahead, one at a 45 degree angle to the right, and one at a 45 degree angle to the left, a full view of any object that may encounter the vehicle can be seen. Through chaining these sensors together and using analog voltage feedback, the vehicle is able to measure the distance an object is from the front of the vehicle, ranging between 3 and 72 inches from the front of the car. The circuit for the interconnection of the ultrasonic sensors can be found below as Figure 47.

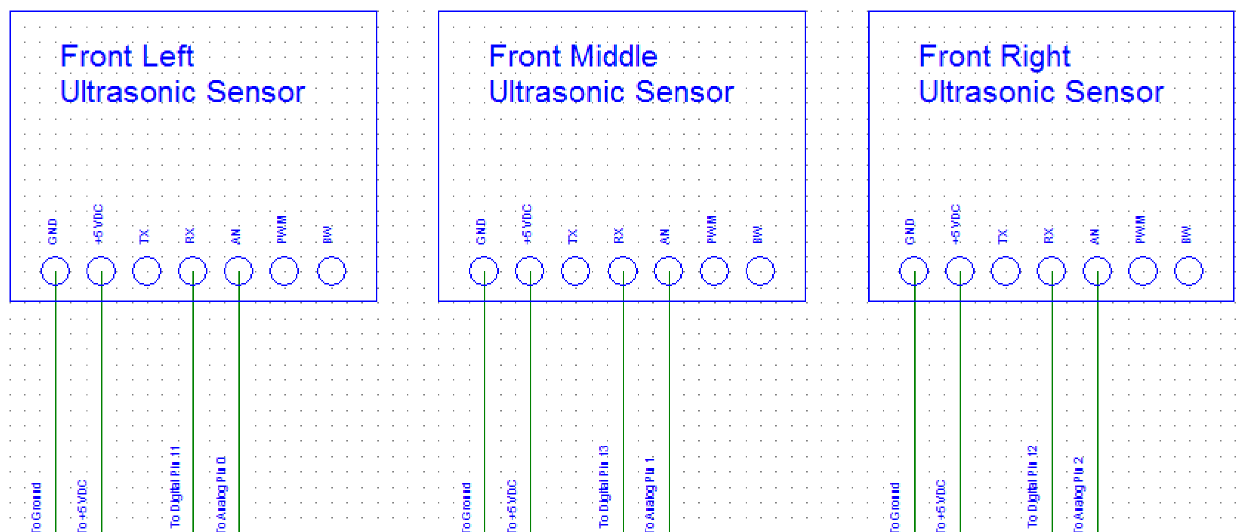


Figure 47: Proximity Sensor Circuit Diagram

Parts List AH

Use:	Part:	Part Number	Description:	Vendor	Link:	Price:	Quantity:
Microcontroller	Atmel 2560 (Arduino Mega 2560)	ATMEGA2560-16AU	The Arduino board with the AT Mega 2560 microcontroller requires an input voltages of 7-12V. The board has 54 digital IO pins and 16 analog IO pins. The microcontroller has 256k of memory and runs at 16Mhz clock speed.	SparkFun	https://www.sparkfun.com/products/11061	$45.95	1
Ultrasonic Range Detector	HC-SR04 Distance SensorUltrasonic Range Finder – LV-MaxSonar-EZ4	HC-SR04	The ultrasonic sensor needs a voltage of 5V to operate. The sensor can detect object within a range of 2cm to 4m with a measuring angle of 15 degrees.	SparkFun	https://www.sparkfun.com/products/8504	$27.95	3
Motor	Tamiya 70168 Double Gearbox L/R Independ 4-Speed (Mabuchi Motor)	FA-130RA	The dual gearbox setup contains two independent Mabuchi motors. They have an operating voltage of 1.5-3.0V and 0.2-2.20A. The motors have a maximum speed of ~9700rpm.	SparkFun	https://www.sparkfun.com/products/319	$10.50	1
Servo	Servo - Generic High Torque Full Rotation (Standard Size)			SparkFun	https://www.sparkfun.com/products/9347	$13.95	1
Voltage Regulator	+5V Fixed-Voltage Regulator 7805	7805	->5v	RadioShack	http://www.radioshack.com/-5v-fixed-voltage-regulator-7805/2761770.html	$1.99	1
Bluetooth	Bluefruit EZ-Link - Bluetooth Serial Link & Arduino Programmer - v1.3	1588	10m range.	Adafruit	https://www.adafruit.com/product/1588	$22.50	1
Relative Positioning	Adafruit 10-DOF IMU Breakout - L3GD20H + LSM303 + BMP180	1604	Gyro, accelerometer, compass	Adafruit	https://www.adafruit.com/products/1604	$29.95	1
Motor Controller	Adafruit Motor/Stepper/Servo Shield for Arduino v2 Kit - v2.3	1438	Can drive 2 motors and 2 servos.	Adafruit	http://www.adafruit.com/products/1438	$19.95	1
Rechargeable Battery	12V Tenergy 2000mAh NiMH Battery Pack with Bare Leads		Rechargeable battery.	Amazon	http://www.amazon.com/gp/product/B00408X4LU/ref=oh_aui_detailpage_o01_s00?ie=UTF8&psc=1	$23.92	1

Figure 48: Parts List

Design Team Information BR, TV, AH, AA

**Alex Aubihl - ** Electrical Engineer - Hardware Manager

**Andrew Hopwood - ** Computer Engineer - Project Leader

**Benjamin Riggs - ** Computer Engineer - Software Manager

**Tyler Vance - ** Computer Engineer - Archivist

Conclusion AH

The focus of the free-range pre-programmed car is to develop a system that can realize a user input in the form of motion. In order to accomplish this, the project will be broken into several components. First, input will be gathered and processed as a user inputs the desired track. Once the track has been processed, the information will be sent across a wireless connection to the radio-controlled vehicle. The vehicle will then execute the desired track. If an obstacle is detected, an interrupt will be sent through the vehicle's microcontroller to the computer for a new route to be determined.

After over a year of planning, designing, implementing, and testing our vehicle application, the project can be considered a success. The vehicle is able to use parsed data commands created by a user through the PC interface and autonomously manuever the vehicle about the desired track path. The vehicle is also able to measure the current position of the vehicle and compare that to the desired position from the algorithm. This difference is calculated as the positional error, and the vehicle corrects for this error in the next track piece sent by the user. The vehicle is able to incoroporate sensors on the front of the vehicle and avoid running into potential obstacles by braking the motors within a range of potential contact. All things being considered, this project was a success.

References TV, AA

[1] *IEEE standard 802.15.1-2005, Wireless medium access control (MAC) and physical layer (PHY) specifications for low-rate wireless personal area networks (WPANs).* 2005. Institute of Electrical and Electronics Engineers, 802.15.1-2005.

[2] Fengchuan, Zhang. 2013. *Remote control toy device.* China filed 2013.

[3] Krichmar, Jeffrey, and Oros, Nicholas. Android based robotics: Powerful, flexible and inexpensive robots for hobbyists, educators, students and researchers. 2013 [cited March 14 2014]. Available from http://www.socsci.uci.edu/~jkrichma/ABR/index.html.

[4] Oriana, Riva, and Jaakko Kangasharju. 2008. Challenges and lessons in developing middleware on smart phones. *IEEE Computer Society* (October): 23-31.

[5] Osthege, Michael. 2013. *Arduino as a MIDI/Bluetooth relay for Windows 8.1 apps* MSDN.

[6] Tze Man Ho, Patrick. 2002. *RC car device.* United States filed 2002.

[7] Qt Project Hosting. 2014. Documentation available from http://qt-project.org/

Appendices BR, TV, AH, AA

Referenced Datasheets:

WiFi Module: http://dlnmh9ip6v2uc.cloudfront.net/datasheets/Wireless/WiFi/WiFly-RN-XV-DS.pdf

Bluetooth Module: http://m2.img.dxcdn.com/CDDriver/sku.121326.pdf

Microcontroller: http://arduino.cc/en/Main/ArduinoBoardMega2560

Ultrasonic Range Detector: http://users.ece.utexas.edu/~valvano/Datasheets/HCSR04b.pdf

Infrared Range Detector:
http://www.acroname.com/products/Sharp_GP2D120_DATA_SHEET.pdf

Motor Controller: http://arduino.cc/en/Main/ArduinoMotorShieldR3

Motor:
http://www.robotgear.com.au/Cache/Files/Files/136_Mabuchi%20motor%20fa_130ra%20datasheet.pdf

GPS: http://cdn.sparkfun.com/datasheets/GPS/EM506_um.pdf

Servo: http://www.servocity.com/html/hs-5055mg_servo.html#.VDyKQ_nF_HU

Matlab Simulation Code:

```
% position of the car [x,y]
% velocity of the car [v_x,v_y]

% Target points obtained based on the predetermined path
[x_p,y_p]=Path_d(1,10,10);

T=100;
index = 1;
x=zeros(1,100);
y=zeros(1,100);
v_x=zeros(1,100);
v_y=zeros(1,100);
max_turn = (pi/6);

x(1)=0;
y(1)=0;
v_x(1)=1/sqrt(2);
v_y(1)=1/sqrt(2);

for t=1:T-1
    if t==1
        index = index + 1;
    else
        if ( x(t-1) < x_p(index) && x(t) >= x_p(index)) || ...
           ( x(t-1) > x_p(index) && x(t) <= x_p(index)) || ...
           ( y(t-1) < y_p(index) && y(t) >= y_p(index)) || ...
           ( y(t-1) > y_p(index) && y(t) <= y_p(index))
            index = index + 1;
        end
    end

    if index > length(x_p)
        break;
    end

    x_t = x_p(index);
    y_t = y_p(index);

    %destination velocity
    [vx_t,vy_t]=ajdust_v(x_t,y_t,x(t),y(t),v_x(t),v_y(t),max_turn);
    x(t+1)=x(t)+vx_t;
    y(t+1)=y(t)+vy_t;
    v_x(t+1)=vx_t;
    v_y(t+1)=vy_t;
end

x_final=zeros(1,t);
```

```
y_final=zeros(1,t);
for i=1:t
    x_final(i) = x(i);
    y_final(i) = y(i);

end
plot(x_final,y_final,'-k')

hold on
plot(x_p,y_p,'-r')

function[vx_t,vy_t]=ajdust_v(x_t,y_t,x,y,v_x,v_y,angle)

dx=x_t-x;
dy=y_t-y;
r=sqrt(dx^2+dy^2)/sqrt(v_x^2+v_y^2);
vx_t=dx/r;
vy_t=dy/r;
end


%predefined path
function[x_p,y_p]=Path_d(S,K,E)
% S is the predetermined shape of the path
% K is the steps we would like to focus on
% E is the horizontal length we predefine

%x_p = 0:(E/K):E;
%if S==1
%    y_p=sqrt(25-(x_p-5).^2);
%end

x_p = zeros(1,6);
y_p = zeros(1,6);

x_p(1) = 0;
y_p(1) = 0;

x_p(2) = 1;
y_p(2) = 3;

x_p(3) = -5;
y_p(3) = 5;

x_p(4) = -5;
y_p(4) = 8;

x_p(5) = -3;
y_p(5) = -2;

x_p(6) = 4;
y_p(6) = -7;

end
```

```
function[x_t,y_t,index]=Target_d(x_p,y_p,x,y,t,index)


if t==1
    index = index + 1;
    x_t = x_p(index);
    y_t = y_p(index);
else
    if (x(t-1)<x_p(index) && x(t) >= x_p(index)) ||...
            (x(t-1)>x_p(index) && x(t) <= x_p(index)) ||...
            (y(t-1)<y_p(index) && y(t) >= y_p(index)) ||...
            (y(t-1)>y_p(index) && y(t) <= y_p(index))
    index = index + 1;
    end
x_t = x_p(index);
y_t = y_p(index);
x_p(index)
end
```

Appendix Figure 1: Matlab Simulation Code