CMPE3815 - Final Project Report

Bluetooth+ Mobile Device Car Control

Report Author: Will Gambero

Group/Lab Partners: Aria Lindberg & Will Gambero

Background

The modern smartphone market has continued to evolve with global demand and technological improvements. In the early 2000s, portable phones were beginning to take shape–most of the first models were bulky to carry around. These slowly gave way to keypad mobile phones, as well as flip phones for making calls. While each concept of the mobile device did utilize some form of a graphical user interface, such as a call screen or logs of contacts, these were fundamentally limited by the current capabilities of the hardware [3]. Most models utilized a keypad to navigate menus, providing a slightly longer process to access a specific feature of flip phones. Nonetheless, the concept maintained popularity during the time period–it even has major popularity today to the surprise of many. This would remain a primary means of mobile communication up until 2007, the year when Steve Jobs unveiled the revolutionary concept of the iPhone. During his presentation, Steve pointed out these same fundamental flaws of the current mobile devices of the time, pinning it to one major drawback: the keyboard [4]. In most models, the keyboard took up more than half the space of the device and was deemed wasted space. In turn, he revealed that the new iPhone would completely remove the keyboard and create a smartphone with solely a giant screen. Therefore, instead of the physical keyboards, Steve proposed the concept of a dynamic graphical user interface, utilizing their new technology of touchscreens to allow users to navigate the iPhone using their fingers. Ever since this moment, touchscreen smartphones skyrocketed in popularity and is now the most popular mobile device type of the 21st century; this is all due to the revolutionary changes in graphical user interfaces.

In standard graphical user interfaces, these can be interpreted as the ways in which a user is able to view the options a complex device has, such as a settings screen on a computer; this bridges the gap between the intricate inner works of electronics and the person itself. With the introduction of the touchscreen, GUI concepts are able to transform from a read-only format to a direct way of controlling functions of the device using hand interactions. In fact, the group itself has been working for the full semester with the Arduino IDE interface, allowing them to write code to be programmed onto connected boards. After reading into the history of graphical user interfaces and seeing how truly revolutionary the invention of the iPhone was for technological and even human history, the group was inspired to pursue a project related to graphical user interfaces. If the team was able to construct and load a simple interactable graphical user interface onto a smartphone screen where a person could control a device or robot using their mobile device with its interactive touchscreen, they would be able to view for themselves just how innovative and incredibly versatile GUI usage is today in the modern world.

Project Overview & Goals

For the remaining weeks of the course, the focus and goal for the group's final project was to construct a graphical user interface using RemoteXY to allow them to remotely control the *LAFVIN* car kit constructed in Lab 8. They have chosen to split their projected work for the project into two categories depending on time constraints or software limitations. In order to deem their work successful, minimal viable product goals were drafted and approved by the instructor. First, the group would test fundamental Bluetooth control of the car. This would be done by interpreting single letter characters sent by a mobile device as commands, toggling specified test movement instructions to verify Bluetooth communication was possible with the *LAFVIN* car. Once successful, the group would introduce additional safety and visual aid features to the remote car to allow for a more intuitive level of control by the user. This involved adding a set of lights that would rest on the car, toggling on depending on a set of predetermined conditions. These would include detecting when the room is dark, the user toggles an emergency stop or is reversing, and if an obstacle has been detected in front of the car. Safety features were to also be built alongside the lights by internally sensing for obstructions directly ahead using the IR sensors of the car and autonomously stopping the wheels; as the sensitivity of these can be adjusted, the group was able to spend time fine-tuning just the right distance for the car to detect obstacles and stop to avoid a front collision.

Once these were proven to be successful, the group would work on a few of their stretch goals set out for this project–these were tasks that were written in the event that something may not work and cannot be completed, such as if the RemoteXY app does not function as intended for the team whether being the editor page of the library code. The major stretch goal the group had written is to implement a graphical user interface to replace the original serial Bluetooth letter command function from the minimal viable product goals. This would be accomplished using the RemoteXY editor, a free online website where users can visually construct a GUI for remote devices [5]. Once the GUI is ready, the website provides the user with the related code that needs to be pasted at the top of the Arduino project–this includes downloading the RemoteXY library as needed. Using the required library functions, the group is finally able to read parameters from each GUI component such as a slider or button using the RemoteXY dedicated syntax. These would then be connected to controlling the movement of the car, such as having a graphical joystick control the steering. While there was a chance the software may not work, the group was hopeful that this stretch goal could be accomplished. A second stretch goal written by the group was to allow for an "autopilot" mode where the car would move forward on its own. While this would have been also an interesting feature to incorporate, the group later deemed this stretch goal a bit too much for the current time; it would be more appropriate as a different final project as it relates to fully-autonomous robot movement compared to remote control commands with RemoteXY.

During the final tests of the car and their GUI on both partner's mobile devices, the group was incredibly successful with their project. **Not only was the group able to successfully complete all of their required minimal viable product goals–see Appendix E–but the RemoteXY GUI software was deemed a major success as it allowed the group to control the car using the interface as a remote**. This report will walk through all steps taken by the group to reach a majorly successful final project. Notes and programs written by the group can be found in the Appendices at the end of this report; a repository link is also provided in Appendix A where all group code written can be downloaded and tested directly.

Project Procedures & Results

Before starting any major goal for the project, the group wanted to spend some time fixing their car as it had some connection issues during Lab 8. This included rechecking wire connections and physical hardware screws as some were found to be loose during the original construction of the car.

Adjusting Car Hardware

The group first removed all major tape temporarily used to stabilize parts that had loose screws, most notably used for the DC motor wheel components. Since this was not a very secure way to keep parts connected due to non-ideal screws, the members brought a hot-glue gun for the lab session, substituting the old electrical tape connections with glue to create a much stronger bond between components. After letting the glue dry, each component was tested by attempting to slightly move them to see if it was fixed securely onto the car base itself. Overall, all components were found to be connected much stronger; there was an additional glue coat added to the battery bank on the back of the car as the test showed it was still slightly unsecured.

After sealing components to the car base itself, the group looked to organize the bundles of wires more effectively. In Lab 8, these were left as untied and resulted in a bit of a messier but working wire network. However, in order for the group to add a smaller breadboard as needed onto the car itself, they now connected bundles of wires together using zip ties as provided in the kit box. They tied wires of each side of the car into a small bundle, creating a space in the center of the car for the team to place a miniature breadboard; this was simply connected to the car using tape as glue would prevent wires underneath from being adjusted for debugging any unexpected results.

New Safety Lights

Using the miniature breadboard, the group constructed two simple circuits to implement a pair of safety lights. Each contained an input connection from the Arduino shield, a 220 Ω resistor, and two LEDs–one was red and the other was white. These were then connected to a common ground connection on the breadboard, finally wired to an available ground node on the Arduino shield. With this configuration, the group would be able to control **left and right lights** on both sides of the car individually or together, allowing for additional features beyond simple toggling of the LEDs. Due to this, they would need two extra pins from the Arduino shield to act as inputs into each pair of safety lights. As the Arduino itself had many of its pins already utilized for other components, the group spent time reviewing the empty pin connections that were available on the shield. Figure 1 shows a *LAFVIN* connection diagram of each used component and which pins they utilize [2]. This is provided on the website and is only shown here for quick reference:

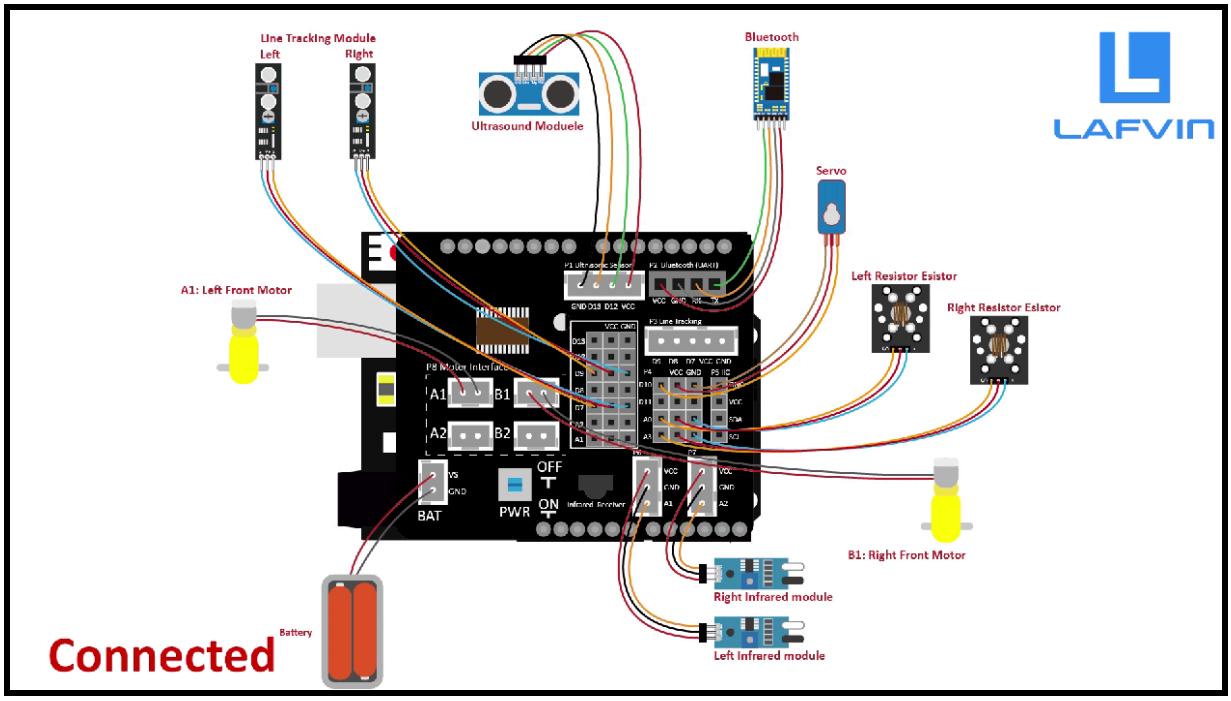


Figure 1 - *LAFVIN* car kit connection diagram [2]. This is available on the website and is only shown here for quick reference. Note that many of the unused connections on the shield are connected to already utilized pins through common nodes, such as “A2” or “B2”.

In the diagram in Figure 1 above, most of the pins are already used by much of the components needed by the car. However, the group recalled that some special pins, such as “SDA” and “SCL”, if not enabled for I2C communication can be used as normal inputs or outputs. Reading into the Arduino documentation, these I2C pins are **directly connected to analog pins A4 and A5 respectively** [1]. Therefore, the group could simply implement “pinMode()” for **pins A4 and A5 as outputs**, directly enabling the “SDA” and “SCL” pins for standard usage as an output for the pair of safety lights. In addition, these pins also have PWM support, giving an extra level of functionality if the group chooses to utilize pulse width modulation for the lights. Using this information, the group wired “SDA” and “SCL” as inputs to each side of the miniature breadboard to be later coded into the Arduino program for proper functionality. Figure 2 shows the simple breadboard connections made to successfully implement the pair of safety lights:

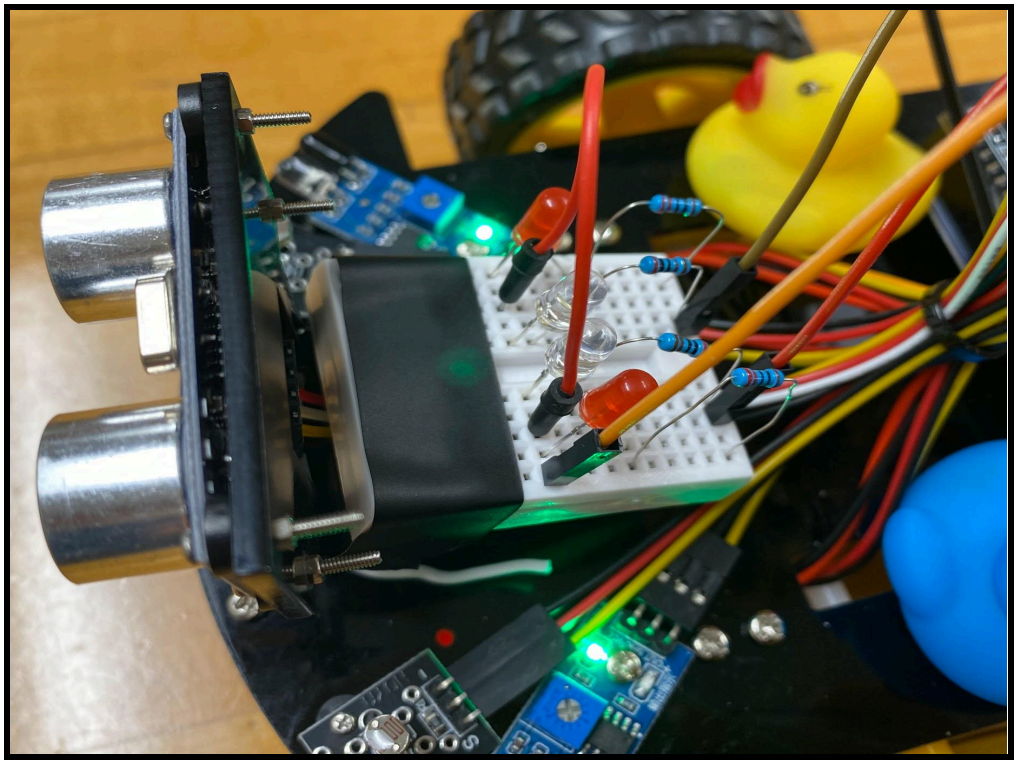


Figure 2 - Sample image of the safety lights implemented on a miniature breadboard. Both sides contain a red and white LED with a respective 220 Ω resistor in series for protection.

As seen in Figure 2 above, each side properly corresponds to one side of lights on the car, representing a way to toggle left or right LEDs depending on its current status. Each circuit was tested to ensure the corresponding LEDs would turn on using a wire connection to the 5V pin of the Arduino board, verifying that the 220 Ω resistors were also connected properly to allow for safe operation for the pair of lights. With the safety lights properly connected, as well as all major components of the car verified to be securely attached to the body, the group shifted to the software side of the project, beginning with adding functionality to each LED circuit shown above.

Safety & Additional Car Features

Now with the car ready to be programmed, the group turned to drafting code to implement safety and quality of life features to the car. As the car itself had many sensors for the group to choose from, they reviewed the functions each one provided to dedicate certain types to specific features to be implemented in their project. For front wide obstacle detection, the IR sensors were selected, while for nighttime detection, the photoresistors were utilized. As stated prior, the safety lights were to be implemented across a range of features, including toggling for object detection, nighttime sensing, reversing, and emergency stopping by the user. Note that before the group arrived at implementing the final code to be shown step by step next, they also constructed a small sample of **fundamental example programs** to independently test each important subset of functions. These include testing Bluetooth letter communication through a mobile serial monitor; interpreting changes in RemoteXY GUI features

by toggling lights; checking for obstacle detection using the IR sensor; and a demo counter to allow for flashing hazard lights when reversing. **As these were for only temporary use, the group has chosen to present all test code files in Appendix C; this report will only briefly mention the concepts behind the test code files as the final program instead reuses the same functions with new additions**, both covering the test programs and how it is implemented in the overarching Bluetooth-Plus Car Control code–see Appendix B. This section will walk through the major functions finalized from the four test programs, later explaining the functionality behind the RemoteXY block of code produced in the final parts of this report.

General Variables & Program Setup

Before starting any new programming, the group must summarize and implement variables to control the car once again. While all pins from Lab 8 are the same for this program, two new variables had to be set for **A4 and A5**–the respective "SDA" and "SCL" pins–to connect to each pair of headlights. Figure 3 below shows all pin values per component to enable full functionality of the *LAFVIN* car:

```
// ======= LIBRARIES =============== //
#include <Servo.h> // Official servo library
Servo servo_motor; // Create servo obj
// ================================= //

// ======= GLOBAL VARIABLES ========= //

// Head & trail lights
const int Rightlights_pin = A5;
const int Leftlights_pin = A4;

// Wheel control pins
const int EN1 = 2; // Enable pin 1 (Digital)
const int EN2 = 4; // Enable pin 2 (Digital)
const int MC1 = 5; // DC motor control 1 pin (PWM)
const int MC2 = 6; // DC motor control 2 pin (PWM)

// Ultrasonic/servo pins
const int Trigger_pin = 12; // Trigger connection
const int Echo_pin = 13; // Echo connection
const int Servo_pin = 10; // PWM

// Photoresistor, sense for light
const int PhotoResistorLeft_pin = A0;
const int PhotoResistorRight_pin = A3;

// Line trackers, sense for lines below car
const int LineTrackLeft_pin = 9;
const int LineTrackRight_pin = 7;

// IR sensors
const int IRSenseLeft_pin = A1;
const int IRSenseRight_pin = A2;
```

Figure 3 - Updated pin designations to track all components connections of the car to the shield. All pins are the same as prior with the new addition of A4 and A5 for "Rightlights_pin" and "Leftlights_pin"

respectively. The “Servo.h” library is also set up with a new object in the event it would be utilized; it however was not used anywhere else in the final code and can therefore be ignored.

Shown in Figure 3 above, all components on the miniature car have a specific connection that needs to be called when setting up each device in the “setup()” block soon to follow. All variables above have comments to help the group track each connection with the help of Figure 1 prior; the only change to the stored pins above is the addition of analog pins A4 and A5 to give a proper connection to the pair of headlights–both are connected to the shield using a simple jumper cable as “SDA” and “SCL” were both open for usage as needed. As this first set of code only stores pin values for the “setup()” block, the group recommends reviewing Figure 1 or the *LAFVIN* website for more information on how the shield is to be configured.

Moving on to important constants for remote car control implementation, Figure 4 below presents a mix of constant and changeable variables; all are utilized later in the program and will only be briefly shown here for reference:

```
// CONSTANTS ~~~~~~~~~~~~~~~~~~~~~~~~
// Constant values
const int LeftWheelTune = 0; // Offset value to approximate L-wheel power = R-wheel power, recommended value = 29
const int Nighttime_Threshold = 400; // Sets photosensor threshold for lights to turn on
const int Move_Threshold = 40; // Sets movement threshold before car is allowed to moved (solves "donut" behavior)
const int TurnMaxRatio = 40; // Sets max offset value to allow for turning
const int Blink_Threshold = 20; // Sets min value for X joystick value for turning side light(s) on

// Tracking values
int Speed_scalar = 0; // Tracks value of speed slider, 0 to 100

bool Nighttime = false; // Tracks if ambient area is dark
byte Move_direction = 1; // Tracks forward/backward movement; 1 forward VS 0 backward
bool Hazard_Stopped = false; // Tracks for hazard stopping; uses IR sensors
bool Emergency_Stopped = false; // Tracks emergency stopping; uses GUI button
int Flashing_Counter = 0; // Counts loop iterations for light flashing effect
int Flashing_Threshold = 2000; // Threshold to toggle lights to opposite state
```

Figure 4 - All important constants of the final program. The group has split them into fixed constants with the “const” term, and changeable values under the “// Tracking values” comment.

In the block of code above, all variables to be used in the program have been sorted into fixed and changeable terms. As all will later appear again in custom functions for car control, the group will only briefly point out a select few that were deemed important for their choice in values. For cars that may have a stronger wheel than the other, a user can change “LeftWheelTune” to adjust how much power the left DC motor is given to calibrate the net movement of the car to avoid swerving; for the group’s car, they have set to this a default of 0 as their fixes to the car structure have appeared to improve directional performance. Next, “Nighttime_Threshold”, “Move_Threshold”, and “Blink_Threshold” are constants chosen as a benchmark value before the car’s nightlight features toggles on, or for when the car is allowed to begin moving or turn on a pair of lights if turning–these final two features relate to the 2D position of the joystick graphic from the user interface and will be explained later. Lastly, “TurnMaxRatio” is a clamp value to prevent the car from turning too fast and beginning to spin instead of turn in a direction; 40 was experimentally found to be a solid value to mimic the

movement of a simple RC car. In contrast, all tracking values are instead **used to track states of the car** and perform related functions based upon these at a given time, such as “Move_direction”. As these are also utilized in the upcoming custom functions for car control, they will be explained in depth later and are only shown here for quick reference.

To now allow all components from Figure 1 to be toggled by the Arduino microcontroller itself, Figure 5 below sets the necessary modes of each part in the “setup()” block; the servo is also enabled with a special line of code but the component itself was not used in the final version of the program:

```
void setup() {
  // Set pin modes & connect servo
  // Wheels:
  pinMode(EN1, OUTPUT);
  pinMode(EN2, OUTPUT);
  pinMode(MC1, OUTPUT);
  pinMode(MC2, OUTPUT);

  // Ultrasonic:
  pinMode(Trigger_pin,OUTPUT); // Trigger is output to ultrasonic
  pinMode(Echo_pin,INPUT_PULLUP); // Echo is an input from ultrasonic

  // Photoresistor:
  pinMode(PhotoResistorLeft_pin,INPUT_PULLUP);
  pinMode(PhotoResistorRight_pin,INPUT_PULLUP);

  // Line trackers:
  pinMode(LineTrackLeft_pin,INPUT_PULLUP);
  pinMode(LineTrackRight_pin,INPUT_PULLUP);

  // IR sensors:
  pinMode(IRSenseLeft_pin,INPUT_PULLUP);
  pinMode(IRSenseRight_pin,INPUT_PULLUP);

  // Connect servo via library
  pinMode(Servo_pin,OUTPUT);
  servo_motor.attach(Servo_pin);

  // Head & tail lights
  pinMode(Rightlights_pin,OUTPUT);
  pinMode(Leftlights_pin,OUTPUT);

  // Setup serial monitor for debugging
  Serial.begin(9600);
  RemoteXY_Init (); // Begin RemoteXY library functions
}
```

Figure 5 - Mode designations for all components in the “setup()” block. The only outputs were to be the nodes of the **motor controller chip**–these are listed as “EN1”, “EN2”, “MC1”, and “MC2”–and each **pair of headlights**. All other components excluding the servo are set to “INPUT_PULLUP” to be read as sensor data. RemoteXY is enabled with the “RemoteXY_Init ()” function on the final line.

Shown in Figure 5 above, only the pins to the DC motor driver and pair of LED headlights are set to “OUTPUT” in mode, the rest all being designated as “INPUT_PULLUP” connections. As needed, all inputs will be interpreted as sensor information to be utilized in car behavior and functionality. The “servo_motor.attach()” line is again present but is ultimately unneeded in the final code as the group chose to select the **IR sensors over the ultrasonic for purposes of avoiding delay issues with RemoteXY**–see the Extra Project Discoveries section for more information. Now with all pins properly set up in the Arduino program, the group was ready to begin constructing their fundamental commands and custom functions to allow for full user control of the car through RemoteXY’s interface, communicating through Bluetooth. As they knew there would be specific overarching features the car should perform, the team drafted a set of overarching custom functions to be constantly run in the “loop()” block. The names of all are shown in Figure 6 below:

```
void loop()
{
  // Begin constant loop of checking RemoteXY values
  RemoteXY_Handler (); // Call RemoteXY handler
  DirectionConverter(); // Get joystick directions & write values to DC motor wheels
  FrontHazardCheck(); // Check for walls in front of car
  AmbientLightCheck(); // Checks brightness of room to toggle nighttime lights if needed
  BlinkerLightsCheck(); // Check for turning lights
```

Figure 6 - Beginning portion of the "loop()" function. Each custom function has a dedicated purpose in the final code as stated in the commands provided. The "RemoteXY_Handler ()" is required to be called in order for RemoteXY to be connected to the program–more on this later.

In the lines above in Figure 6, there are 4 important custom functions the group has settled on the car looping through to allow for full remote control; note that the "RemoteXY_Handler ()" is a required function to be called in order to enable a data request from the RemoteXY app and is instead part of a library. First, the group wants the car to interpret the **joystick** as directions to be written to the DC motor wheels, encapsulated in "DirectionConverter()". Afterwards, the group would want to quickly check for hazards in front of the car through "FrontHazardCheck()", forcing an emergency stop if the car determines there to be a wide obstacle too close to the front using the IR sensors. Depending on the brightness of the surrounding area, "AmbientLightCheck()" would quickly perform a light sensor check to be interpreted as being too bright–do not toggle the headlights–or too dark–turn on the headlights. Lastly, to determine if the car is in a turning state, "BlinkerLightsCheck()" would handle interpreting the x-directional input from the joystick, enabling the corresponding directional side of lights depending on if the turn is large enough from the GUI value. As the group would be also reusing fundamental code from Lab 8, such as for forwards or backwards movement, this will be shown in addition to the first function "DirectionConverter()" next.

Determining Direction & Revising Fundamental Movement

Beginning with the "DirectionConverter()" function, the group needed to first determine a way to account for **steering and interpreting movement** from the RemoteXY joystick. Since the program has enabled RemoteXY support with the final line from Figure 5 prior, the group could simply utilize the syntax provided on the website for creating the interface–this will be explained in depth later. In the group's interface, they have designated the joystick to have two variables called "Direct_joystick_X" and "Direct_joystick_Y", each of which contain the information related to the 2D position of the interface element. To read the value of any RemoteXY variable, the group can simply use **object notation with "RemoteXY.Direct_joystick_X" and "RemoteXY.Direcy_joystick_Y"**, returning an **integer** value between **-100 and 100**. Knowing this, the group first wanted to ensure the car was not already emergency stopped with the "Emergency_Stopped" boolean–this refers to the push button on the interface only–from the constants listed prior; this can be checked with an "if" statement, continuing into the block if this boolean is false when the function is called. One drawback that is present in the RemoteXY variable range is that the joystick position can be read as **negative**. To resolve this, the group would simply use a set of **"if" and "else if" statements on "RemoteXY.Direct_joystick_Y"** to determine if the car should be moving forwards, backwards, or neither–this is if the value is below a movement threshold

or the joystick is released, auto-resetting to the center. While a simple comparison against 0 would appear appropriate, there was an issue of **the minimum power each DC motor must have to turn on**, instead being around 35 as an analog value for required power. Therefore, the group set the constant "Move_Threshold" shown prior to **40**, rounding this value upwards. If "RemoteXY.Direct_joystick_Y" is **greater** than the threshold, the joystick is pointed up and should move forwards. If it is **less than the negative of the threshold** (to allow for an absolute value check with two negatives), the joystick is aiming down and **should be interpreted as moving backwards**. If neither is met, the car should be **fully stopped**; if "Emergency_Stopped" is also true, the car will also be forced to stop as a precaution. In addition, the car should not move forward if there is already a hazard detected in front, allowing the group to incorporate an "AND" statement with "Hazard_Stopped" being false. This other boolean variable tracks if the IR sensors have deemed there to be a wide obstruction ahead, requesting the car to stop immediately.

When the joystick is properly interpreted as looking to move the car forward or backwards, the group would first track the directional movement of the car by changing "Move_direction" to 0 for backwards or 1 for forwards, as well as send the **x-position and the absolute value of the y-position** to the corresponding "MoveForwards()" or "MoveBackwards()" fundamental functions. As a negative value cannot be sent to a DC motor through "analogWrite()", the decision to determine whether the car is going to forward or backward was a very useful solution, allowing the group to simply send the magnitude of the y-position of the joystick and only need to **call the correct movement function**. For the car to be stopped in the other cases, a function named "Stop()" is simply executed to turn off all DC motor power and completely stop the car in its place. This is all shown in its entirely in Figure 7 below:

```
// Checks GUI values and toggles movement parameters as needed
void DirectionConverter(){
  // Check for if emergency stop button is not held down
  if (Emergency_Stopped == false){
    // Forward or backward check (Y direction)
    if (RemoteXY.Direct_joystick_Y > Move_Threshold and Hazard_Stopped == false){ // Forwards & no hazard stopping
      Move_direction = 1;
      MoveForwards(RemoteXY.Direct_joystick_X,abs(RemoteXY.Direct_joystick_Y)); // Move backwards (send absolute val of Y position)
    }
    else if (RemoteXY.Direct_joystick_Y < -Move_Threshold){ // Backwards
      Move_direction = 0;
      MoveBackwards(RemoteXY.Direct_joystick_X,abs(RemoteXY.Direct_joystick_Y)); // Move forwards (send absolute val of Y position)
    }
    else { // Stopped, y = 0
      Stop();
    }
  }
  else{ // Emergency stop pressed, stop car immediately
    Stop();
  }
}
```

Figure 7 - Full "DirectionConverter()" code block. If the emergency stop button is not currently pushed with "Emergency_Stopped", the y-position of the joystick is checked against a threshold value in both the position and negative ranges. If above the magnitude of "Move_Threshold", the corresponding movement commands "MoveForwards()" or "MoveBackwards()" are run and sent the joystick values; "Move_direction" is also updated with the new resulting direction. If there is a hazard currently detected in front of the vehicle, "Hazard_Stopped" becomes true and prevents the program from running "MoveForwards()" using the "AND" statement.

As shown above, Figure 7 has the task of interpreting joystick positional values against current states of the car, first making sure the emergency button is not toggled–resulting in "Emergency_Stopped" turning true–and then determining which direction the joystick is facing to decipher which movement command to call. If the y-position is above the "Movement_Threshold", the "MoveForwards()" command is called; if it is below the negative of "Movement_Threshold", "MoveBackwards()" is run. If neither is met, or there is a hazard currently in front and the user tries to move the joystick above "Movement_Threshold", "Stop()" is immediately run to force all DC motors to untoggle.

Since all three fundamental movement commands from Figure 7–"MoveForwards()", "MoveBackwards()", and "Stop()"–were reutilized from a previous lab but now had new inputs, the team needed to make important changes to how each would write movement values to the corresponding DC motors. The team did not need to change the enable "digitalWrite()" code lines from the previous lab as these were only needed for determining movement forwards or backwards. For the first two functions–"Stop()" only required the addition of forcing "Movement_direction" back to 1 to avoid flashing light errors–each would be provided variables "TurnRatio" and "Speed" from the **x-position and y-position of the joystick respectively**. These two integers would need to be utilized in such a way that the car interprets **turning commands** versus straight forwards or backwards movement. The solution the group found involved **mapping the "TurnRatio"** (x-position) to a clamped range between **negative and positive "TurnMaxRatio"**. This was done since the group chose to **subtract** the turn ratio from one wheel's "analogWrite()" and add it to the other, avoiding the case where an **unclamped value** results in a subtraction that equals a **negative analog value**; this mapped range forces the value to be between -40 and 40 for addition and subtraction and is stored in a new variable "MapTurnRatio". As the group also wanted to account for the event that one wheel is stronger than the other given by **"LeftWheelTune" prior**, this is subtracted from the **left wheel** total in addition to adding the resulting "MapTurnRatio" to "Speed" –the y-position of the joystick which is from 0 to 100. In addition, the team would also be implementing a **maximum speed slide switch** to the GUI. Therefore, **the total summation would be mapped** from the original scale (given all possibilities of "MapTurnRatio") of **0 to 140** to a new scale from **0 to "RemoteXY.Speed_slider" times 2**. Since the graphical element can take on values from 0 to 100, this means that **double the speed slider is able to provide a maximum analog value of 200**, successfully giving the user a new interface item to directly control how fast they would like their car to travel. At low slider values near 0, the car will move very slowly, contrasting the maximum speed in which the value of "RemoteXY.Speed_slider" is set to 100–or the lever being pulled all the way up on the interface. One final benefit of this setup for speed and turning control is that because **the only difference required for moving forwards and backwards is changing the digital values of the two enabler pins**, the group is able to then have **"MoveForwards()" and "MoveBackwards()" use the same exact code** with the only change being the "digitalWrite()" lines at the very top. This is all shown fully in Figure 8 below:

```
// Wheel functions ~~~~~~~~~~~~~~~~~~~~~~~~~~
// Move forward
void MoveForwards(int TurnRatio, int Speed) {
  int MapTurnRatio = map(TurnRatio,-100,100,-TurnMaxRatio,TurnMaxRatio); // Scale turning value (-100 to 100) to allowed ratio
  digitalWrite(EN1, HIGH);
  digitalWrite(EN2, LOW);
  // Write corresponding x-joystick speeds to each wheel; scale by "Speed_slider" via mappingscale by 2*"Speed_slider" with map()
  analogWrite(MC1, map(Speed+MapTurnRatio-LeftWheelTune,0,140,0,2*RemoteXY.Speed_slider));
  analogWrite(MC2, map(Speed-MapTurnRatio,0,140,0,2*RemoteXY.Speed_slider));
}

// Move backwards
void MoveBackwards(int TurnRatio, int Speed) {
  int MapTurnRatio = map(TurnRatio,-100,100,-TurnMaxRatio,TurnMaxRatio); // Scale turning value (-100 to 100) to allowed ratio
  digitalWrite(EN1, LOW);
  digitalWrite(EN2, HIGH);
  // Write corresponding x-joystick speeds to each wheel;scale by "Speed_slider" via mappingscale by 2*"Speed_slider" with map()
  analogWrite(MC1, map(Speed+MapTurnRatio-LeftWheelTune,0,140,0,2*RemoteXY.Speed_slider));
  analogWrite(MC2, map(Speed-MapTurnRatio,0,140,0,2*RemoteXY.Speed_slider));
}

// Stop movement
void Stop() {
  Move_direction = 1; // Preset direction to forward to reset lights
  analogWrite(MC1, 0);
  analogWrite(MC2, 0);
}
// ================================== //
```

Figure 8 - Updated fundamental functions “MoveForwards()”, “MoveBackwards()”, and “Stop()”. The only change for “Stop()” is to force “Move_direction” to be set to 1 to prevent errors with the headlights–more on this next. For both “MoveForwards()” and “MoveBackwards()”, the inputted x-position and y-position from the joystick are stored as “TurnRatio” and “Speed” respectively; the x-position is next mapped to the range of negative to positive “TurnMaxRatio”. “Speed” is either added or subtracted by this new ratio value–with the left wheel also subtracting “LeftWheelTune” in the event of one wheel being more powerful–and is then mapped to a maximum range of 0 to 2*“RemoteXY.Speed_slider”. This is finally written to the respective analog pins to drive both DC motor wheels.

As seen above, these needed changes to the original fundamental functions have now enabled the car to successfully interpret all movement-based interactive elements from the GUI the group will show later. One extra choice the group could have made is to construct one function that would take a third input to determine if the car would be moving forwards or backwards to compress “MoveForwards()” and “MoveBackwards()” to a single function. The team decided that because these two functions were relatively simple, this would be unneeded as only two lines of four are the same from both. With the first major function from the “loop()” block complete, the group moved onto implementing obstacle detection and related emergency stopping.

Front-Obstacle Detection & Emergency Stopping

Moving to utilizing the IR sensors, the group did first construct test code to investigate the accuracy of the IR sensor of the car. They were most interested in what distance the car should be allowed to be from a wide obstacle such as a wall by physically tuning the screw of the components, writing then a simple test script to immediately run the original “Stop()” function to force all wheels to freeze. As this was only a test script that was revised for the final code, see Appendix C.3 for the

temporary test code written. To detect an obstacle, first note that the IR sensors **return a "HIGH" or "LOW"** value in contrast to an analog input. This meant that in order to check for any instance of a potential front hazard, the group would need to utilize **"digitalRead()" and check if either sensor returned "LOW"**, revealing that an obstacle is within the range of the sensor and has been sent to the program. Since there are two sensors, a simple **"OR" combination** of two "digitalRead()" lines checking for "LOW" can be incorporated. In the event either or both sensors return "LOW" and detect a hazard, the group wanted all headlights to toggle on and for the car to stop immediately. This is then done by utilizing the "Stop()" function explained earlier. In addition, the variable "Hazard_Stopped" from figures prior will be set to true. As seen in the previous section, if "Hazard_Stopped" is true, the car is forced to ignore joystick commands to move forward to prevent a collision, only allowing the user to reverse until the hazard is no longer detected. If neither sensor is "LOW", "Hazard_Stopped" should be reset to false as there is no longer an obstruction found ahead, as well as turning off both headlights. To toggle the states of the headlights, two "analogWrite()" lines can be used to either send 0 (off) or 255 (on) to both sets of LEDs. As an extra precaution, the car should only force an emergency hazard stop **only when "Move_direction" is 1, or moving forward**. This can be simply done by using an "AND" operator outside of parentheses housing the "OR" statement of the sensor readings. If this was not implemented, the user would never be able to reverse away from the obstacle as the original "OR" of both sensors would always return true, constantly running "Stop()" and never the other "else" code block. This is all shown fully in Figure 9 below:

```
// Checks for wide-front hazards with IR sensors
void FrontHazardCheck(){
  // Check if either IR sensor is LOW (detects obstacle) and car is moving forward; ignore if moving backwards
  if ((digitalRead(IRSenseLeft_pin) == LOW or digitalRead(IRSenseRight_pin) == 0) and Move_direction == 1) {
    Stop();
    Hazard_Stopped = true; // Set hazard stop to true, stop any forward movement
    analogWrite(Leftlights_pin,255);
    analogWrite(Rightlights_pin,255);
  }
  else{
    Hazard_Stopped = false; // Reset hazard stop to false, re-allow forward movement
    analogWrite(Leftlights_pin,0);
    analogWrite(Rightlights_pin,0);
  }
}
```

Figure 9 - Full "FrontHazardCheck()" function. Given that "Move_direction" is 1 (car is moving forward), if either sensor returns a "LOW" value and detects an object, the car is stopped immediately using "Stop()" and both headlights are toggled on with "analogWrite()" writing 255 for a PWM value; "Hazard_Stopped" is set to true to then prevent any future forward movement until it resets to false. If neither sensor detects an obstacle or the user is currently reversing ("Move_direction" is 0), the headlights will turn off and "Hazard_Stopped" is reset to false.

Simply shown, the block of code in Figure 9 is capable of simply determining if one or both sensors detect an obstacle using an "OR" operator to check for "LOW" readings. This is in the scenario that the car is moving forwards with "Move_direction" as 1 since trying to check for obstacles and subsequently emergency hazard stop when reversing will introduce a softlock for movement unless the

user picks up the car and places it elsewhere. If an obstacle is detected, both headlights are toggled on and “Stop()” is called to force all DC motors to stop immediately; “Hazard_Stopped” is set to true to prevent any forward movement. If there is no obstruction detected–both sensors are “LOW”–the headlights are turned off and “Hazard_Stopped” goes back to false, reenabling forward movement as there is no longer an obstacle in the front of the car.

In addition to the auto-hazard stopping feature, the group also incorporated an emergency stop button on the RemoteXY interface. When pressed down, the variable **“RemoteXY.Emergency_stop” is set to 1 internally**; once released, it returns to 0. Using this information, the group added an extra small block of code to the final part of the “loop()” block from Figure 6. When pressed, the group sets “Emergency_Stopped” to true to prevent any directional movement showcased in Figure 7–recall that if this variable is true, “Stop()” is run instead of “MoveForwards()” or “MoveBackwards()”. The headlights are also toggled on using the same “analogWrite()” code lines from Figure 9 sending 255 to both LED pins. If the button is released or not pressed, both lights are turned off by writing an analog value 0 to both headlights, as well as setting “Emergency_Stopped” back to false. Figure 10 shows the simple implementation of the emergency stop button fully; there is also one special line to be discussed next:

```
  // Check for emergency stopping (GUI button)
  if (RemoteXY.Emergency_stop == 1){
    Emergency_Stopped = true; // Button is held down
    // Turn on all lights
    analogWrite(Rightlights_pin,255);
    analogWrite(Leftlights_pin,255);
  }
  else{
    Emergency_Stopped = false; // Button not held down
    // Turn off all lights unless nighttime
    if (Nighttime == false){
      analogWrite(Rightlights_pin,0);
      analogWrite(Leftlights_pin,0);
    }
  }
}
```

Figure 10 - Bottom part of the “loop()” block of code. When the GUI emergency button element is held down, “Emergency_Stopped” becomes true and all lights are turned on. Due to this variable changing to true, the “DirectionConverter()” function will skip running “MoveForwards()” or “MoveBackwards()”, instead jumping to “Stop()” to force the car to stop immediately. When the button is released, “Emergency_Stopped” is reset to false and both headlights are turned off. Note that there is also an “if” statement checking for “Nighttime” being false–this will be explained next.

As seen above, the emergency button on the user’s mobile GUI will change the value of “RemoteXY.Emergency_stop” to determine which “if” block to run. When pressed down, the car stops immediately and the headlights turn on; once released, the car is free to move again and the lights are turned off. In addition to a simple boolean comparison, the group also wrote an extra “if” statement for checking if **“Nighttime” is false**. This variable keeps track of if the room is currently dark or not; if the room is then darkened for “Nighttime” to be set to true, **the LEDs will not be turned off**. This new feature is directly connected to the “AmbientLightCheck()” function and will be explained next.

Nighttime Detection

To implement nightlight-enabled headlights, the group would now need to focus on investigating the range of analog inputs the *LAFVIN* car photoresistors can take on. First, and as stated previously, the car direction is first checked for going forwards with "Move_direction" set to 1 to avoid an error with the headlights for the final function of reversal lights–this will be up next. If true, the **analog value of both photoresistive sensors are checked** for being above a threshold "Nighttime_Threshold". An important feature of both sensors is that **for darker environments, the input analog value becomes larger**. Therefore, by checking if **either** sensor–using an "OR" operator–is above this threshold, the program can determine if the surrounding environment is dark; this threshold was set to 400 from experimental tests with the preliminary code in Appendix C.3 that instead toggled the right set of LEDs to understand how these two sensors behaved. Once true, "Nighttime" is set to true and both headlights are toggled on with "analogWrite()" set to 255. If neither sensor is above this threshold, "Nighttime" is reset to false and the headlights are turned off. This is also shown in Figure 11 below:

```
// Checks for brightness and toggles headlights if dark
void AmbientLightCheck(){
  // Check for forward movement only
  if (Move_direction == 1){
    // Check for analog value above threshold (large value -> dark)
    if (analogRead(PhotoResistorLeft_pin)> Nighttime_Threshold or analogRead(PhotoResistorRight_pin) > Nighttime_Threshold){
      // Above threshold, turn ON lights
      Nighttime = true;
      analogWrite(Rightlights_pin,255);
      analogWrite(Leftlights_pin,255);
    }
    else{
      // Below threshold, turn OFF lights
      Nighttime = false;
      analogWrite(Rightlights_pin,0);
      analogWrite(Leftlights_pin,0);
    }
  }
}
```

Figure 11 - Full "AmbientLightCheck()" function. Given that the car is moving forwards with "Move_direction" as 1, if either photoresistor inputs an analog reading above "Nighttime_Threshold", the program deems the environment as dark and enables the headlights; "Nighttime" is also set to true to prevent emergency stopping from turning the lights off after release. If neither is above the threshold value, "Nighttime" resets to false and the headlights turn off.

As shown above, "AmbientLightCheck()" has the functionality to implement and quickly perform an auto-night light mode toggle of the car headlights by simplifying checking the values of the photoresistors against a constant analog threshold of "Nighttime_Threshold". With the addition of the "Nighttime" variable tracking if the ambient environment is considered dark or not, this explains the addition line of code from Figure 10 prior, preventing the headlights from toggling off if the user cancels the emergency stop as the car is still in a dark environment and must remain on. While this again utilizes the headlights, there are two remaining functions that add to this level of versatility to the LED pair from Figure 2 and will be explained next.

Reversal Light Pulse & Turn Auto-Signaling

Lastly, the group looked to implement two final functions to the car's headlights. To begin, the team chose to have a feature where if the car is reversing, the headlights would **flash at a slow rate** to indicate backing similar to a truck. To do this, "Move_direction" is checked for being equal to 0 (backwards) inside the "loop()" block, and if true, begins a **counter named "Flashing_Counter"** from Figure 4 prior. For every iteration of the "loop()" block, the counter increments by 1 and performs a check against **"Flashing_Threshold"**. If the counter is **less than the threshold**, both headlights are turned or kept on with "analogWrite()" sending analog value 255 to the LED pins. However, if the counter eventually **is greater than "Flashing_Threshold"**, the lights are turned off and remain off until the counter either goes above **two times "Flashing_Threshold"** or the car stops moving in reverse. If the counter goes above **2*"Flashing_Threshold"**, both headlights are re-toggled on and remain on; the counter is also **reset back to 0 to begin from the start** and repeat the flashing effect at the same frequency. In the event that the car begins to **move forward with "Move_direction" as 1, "Flashing_Counter" is immediately reset to 0 and stops** until the next time "Move_direction" is set to 0 again. Figure 12 shows this functionality fully and is placed in the center of the "loop()" block prior:

```
// If reversing, flash lights via counter (on-off sequence)
if (Move_direction == 0){
  Flashing_Counter ++; // Add 1 to counter
  if (Flashing_Counter > 2*Flashing_Threshold){ // If above 2*threshold, set lights to on and reset counter
    analogWrite(Rightlights_pin,255);
    analogWrite(Leftlights_pin,255);
    Flashing_Counter = 0;
  }
  else if(Flashing_Counter > Flashing_Threshold){ // If above normal threshold, set lights to off
    analogWrite(Rightlights_pin,0);
    analogWrite(Leftlights_pin,0);
  }
  else{ // If below threshold, set lights to on
    analogWrite(Rightlights_pin,255);
    analogWrite(Leftlights_pin,255);
  }

}
else{
  Flashing_Counter = 0; // Not in reverse direction, set counter to 0
}
```

Figure 12 - Middle portion of "loop()" for flashing reversal lights. When "Move_direction" becomes 0, a counter begins for "Flashing_Counter" and is checked against three conditions. If it is less than "Flashing_Threshold", the headlights turn or remain on; if it is greater, the LEDs now turn off and stay unpowered. Once the counter reaches two times "Flashing_Threshold", the lights turn on again and the counter resets to zero. At any time when "Move_direction" goes back to 1, the counter resets to 0 and the flash effect stops.

Shown above, a simple combination of three comparison blocks was required to implement a reversal flashing light feature to the car, successfully repeating the effect until "Move_direction" changed to 1. Note that in the code above**, the order of "if" statements is important as Arduino checks from top to bottom**. If the second "if" statement were instead checked first, the LEDs would first turn on and then turn off to never be toggled on again until "Move_direction" resets to 1. Therefore, checking the largest

comparison first ensures that the **counter will reset at two times "Flashing_Threshold" to allow for a looping flash effect** as desired by the group.

To close the final piece of software for general Arduino functions–the RemoteXY code will be shown afterwards with the interface constructed by the team–the final function in Figure 6 to be drafted is "BlinkerLightsCheck()". This function is meant to toggle the corresponding side LEDs depending on which way the car is currently turning, similar to a blinking vehicle signal but as a static light indicator. To accomplish this, **"RemoteXY.Direct_joystick_X"** can be compared against **"Blink_Threshold"** to check if the user is above the threshold for turning. Since the team wanted there to be a certain distance the joystick has to be moved on the x-axis to be considered a turn, adding the "Blink_Threshold" allows the pair of LED lights to only toggle on when the joystick is far enough left or right on the interface. For checking if the left lights should be toggled on, "RemoteXY.Direct_joystick_X" must be **less than negative "Blink_Threshold"**; for the right LEDs, "RemoteXY.Direct_joystick_X" must be above "Blink_Threshold". If it is instead between these two values, neither headlight pair is turned on. This is all shown in Figure 13 below:

```
// Check turning lights for toggling
void BlinkerLightsCheck(){
  // Compare joystick's X value against threshold turning val (must be greater than constant to toggle turning lights)
  if (RemoteXY.Direct_joystick_X<-Blink_Threshold){ // Turn left lights on
    analogWrite(Leftlights_pin,255);
    analogWrite(Rightlights_pin,0);
  }
  else if (RemoteXY.Direct_joystick_X > Blink_Threshold){ // Turn right lights on
    analogWrite(Leftlights_pin,0);
    analogWrite(Rightlights_pin,255);
  }
  else{ // Not enough X value, turn off lights
    analogWrite(Leftlights_pin,0);
    analogWrite(Rightlights_pin,0);
  }
}
```

Figure 13 - Final function of "BlinkerLightsCheck()" from the "loop()" block. In each iteration of the program, the x-position of the joystick is compared against "Blink_Threshold". If the position is less than negative "Blink_Threshold", the car is turning left and the left pair of LEDs should turn on. If the x-position is above "Blink_Threshold", the car is instead turning right and so the right pair of lights toggle on. If "RemoteXY.Direct_joystick_X" is between these two threshold points, both pairs of lights are forced off.

As shown above, depending on if "RemoteXY.Direct_joystick_X" is above the positive or below the negative of "Blink_Threshold", the corresponding headlight side depending on the car turning direction will be turned on. Note that only one pair can be turned on as seen above, the other pair of headlights being forced off to ensure **that when going forwards and turning,** only one set of headlights is toggled on. With this final function described, the team was now ready to begin experimenting with constructing their RemoteXY interface; this will also have code to be pasted into the program at the very top and will therefore be explained in this next section.

Full RemoteXY GUI Implementation

At this point in the project, the group has succeeded in fulfilling their minimum viable product goals with Bluetooth separately tested using a Bluetooth serial monitor app, also proving to be a success–see Appendix C.1 for the code that proved successful. Now the team moved to the final part of their project aiming to implement a functional graphical user interface to allow users to control the car remotely–the overarching stretch goal of the final project. This was chosen to be done through RemoteXY [5] as the website has a free online editor to construct a user interface for any smartphone or mobile device through their dedicated app.

Using the RemoteXY Editor

Using the RemoteXY editor, the group had the ability to drag and drop graphical elements onto a smartphone test screen template such as sliders, switches, buttons, or even dropdown menus. As stated in the previous code images shown, the team planned on **utilizing a slide lever, push button, and a joystick**. In addition to interactives, the website also allows users to add **text elements**. For free users, there is a maximum limit of 5 elements that are allowed to be placed. Knowing this, the team chose to label the slide lever as "Speed" and the push button as "EMERGENCY STOP" to help inform others the functions each interactive element provides; a joystick is relatively simple to interpret as a directional controller and so it was left without a label. In the editor, the mode of communication, type of IDE, board name, and Bluetooth chip type can be selected. For the group's *LAFVIN* car, it utilizes **an Arduino UNO R3**–and the related Arduino IDE–and an **HM-10 Bluetooth receiver**. Recall that in a previous lab, the JDY-16 receiver was instead used but is not supported in RemoteXY. Figure 14 shows the group's final user interface concept constructed with the RemoteXY editor:

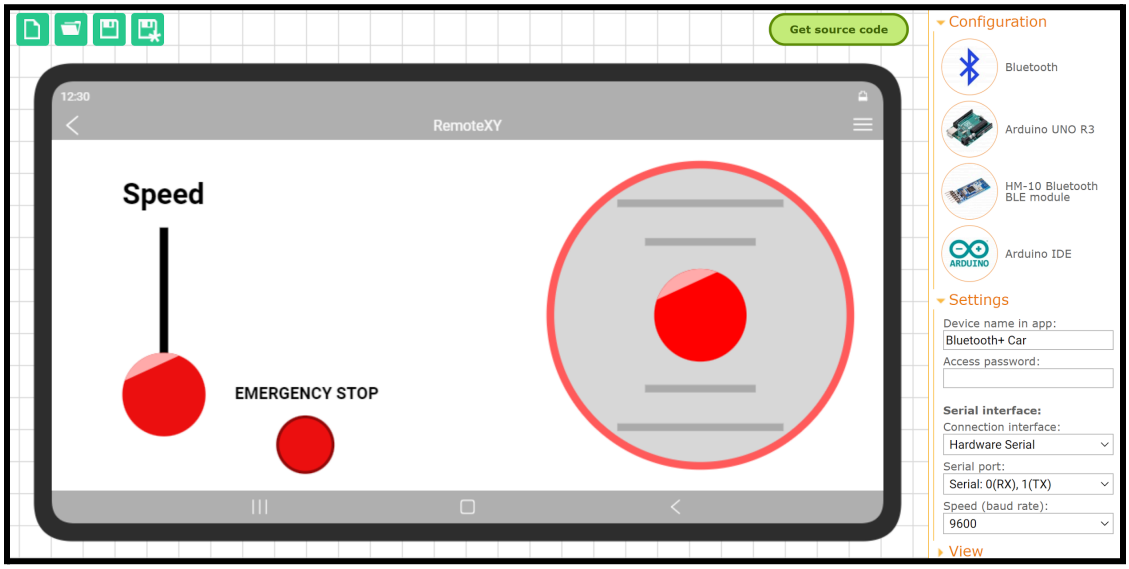


Figure 14 - Mobile car control GUI created in RemoteXY [5] by the group. The joystick, slider, and push button as stated previously are now visible and ready to be tested on the RemoteXY app.

Shown above, all three major interactive elements to be used in the functions explained prior are now present, each with customizable parameters for the group to choose from such as color and if the joystick will reset to the center if released–this was toggled on by the team. While the team did set the baud rate to 9600 to enable outputting information to the serial monitor in the Arduino IDE, they found it to be much more intuitive to instead toggle **the lights of the car to test communication of each component shown above through Bluetooth**. To accomplish this, the code must first be compiled by clicking the button called “Get Source Code”.

Library Syntax & Code Implementation

When pressing the editor’s source code button, the team was provided a giant page of code and was instructed to paste into the top of their final program. As the majority of the code is for connecting the variables to the RemoteXY library, the provided source code will only be briefly discussed here. Figure 15 below shows the source code compiled from the GUI from Figure 14:

```
// ====== REMOTE XY SETUP ============================== //
// RemoteXY setup (GNU Lesser General Public License)
// Code generated from editor page of RemoteXY website

// RemoteXY select connection mode and include library
#define REMOTEXY_MODE__HARDSERIAL

// RemoteXY connection settings
#define REMOTEXY_SERIAL Serial
#define REMOTEXY_SERIAL_SPEED 9600

#include <RemoteXY.h>

// RemoteXY GUI configuration
#pragma pack(push, 1)
uint8_t RemoteXY_CONF[] =   // 117 bytes
  { 255,4,0,0,0,110,0,19,0,0,0,66,108,117,101,116,111,111,116,104,
  43,32,67,97,114,0,31,2,106,200,200,84,1,1,5,0,5,21,80,60,
  60,119,5,74,74,32,36,30,31,4,48,52,7,86,17,11,20,60,0,1,
  24,1,23,81,57,57,54,66,14,14,0,1,31,0,129,1,124,71,29,17,
  10,20,7,64,24,83,112,101,101,100,0,129,7,112,71,29,44,59,35,4,
  64,24,69,77,69,82,71,69,78,67,89,32,83,84,79,80,0 };

// this structure defines all the variables and events of your control interface
struct {

    // input variables
  int8_t Direct_joystick_X; // from -100 to 100
  int8_t Direct_joystick_Y; // from -100 to 100
  int8_t Speed_slider; // from 0 to 100
  uint8_t Emergency_stop; // =1 if button pressed, else =0

    // other variable
  uint8_t connect_flag;  // =1 if wire connected, else =0

} RemoteXY;
#pragma pack(pop)
// ====================================================== //
```

Figure 15 - Downloaded source code of the team’s GUI created in the RemoteXY [5]. Much of the code performs library connections and is therefore not of major importance for explaining. However, the constants with RemoteXY syntax from before are now seen above.

As seen above, most of the code compiled on the RemoteXY editor for the team's interface is primarily focused on performing library functions and is not very insightful for interpretation. There are however four important variables introduced: **"Direct_joystick_X", "Direct_joystick_Y", "Speed_slider", and "Emergency_stop"**. Recall that these all were utilized previously but with a special syntax such as "RemoteXY.Direct_joystick_X"; this is due to the RemoteXY library **requiring the use of "RemoteXY" as an object to read values from each of the variables above**, each of them corresponding to information about a particular GUI element. A special case is that for the joystick, there are **two variables** that can be read to obtain the x-position and y-position when the interface is being interacted with. In addition to the source code compiled from the GUI constructed, the website also instructed the team to add **"RemoteXY_Init ()" to the "setup()" block, and "RemoteXY_Handler ()" to the "loop()" section**. Each function is required to enable connection to the library, the first one initializing RemoteXY functions and the second obtaining information about each of the interface elements upon each iteration of "loop()". As this code only needed to be pasted into the team's program, they were ready to begin direct tests with the RemoteXY app and their final car program. But first, the team did want to spend some time testing basic Bluetooth communication through RemoteXY.

Basic Bluetooth & RemoteXY GUI Tests

Before immediately running the code, the group constructed a test script in Appendix C.2 to **check a basic setup of implementing the emergency push button and speed slider**. Note that prior to reaching the level of RemoteXY testing, the group also replicated a serial Bluetooth test script to stop and start car movement using single letter commands–this was also one of the minimum viable product goals which was proven to be fulfilled. As the script is similar to the code written by the team in a previous lab, see Appendix C.1 for the full program. To summarize the serial monitor test, the group was successful in the car acknowledging a connection to an Arduino mobile app and interpreting "a", "l", and "s" as move forward, backward, or stop commands. This code was expanded to now instead utilize the two sets of headlights to **check the interpretation of RemoteXY elements**. The source code in Figure 15 was again copied to the top of the test file in Appendix C.2, allowing for initialization of RemoteXY and proper syntax for obtaining current values of the interface. As a test of RemoteXY through Bluetooth, the team would toggle one pair of lights to on or off depending on if the emergency button is pushed down (on) or released (off) using "digitalWrite()". To check the resolution of the slider, the other pair of LED headlights would instead use **"analogWrite()" to send a mapped value originally from 0 to "RemoteXY.Speed_slider", to the new range of 0 to 255**. Since the team was able to utilize the analog pins of A4 and A5, there was PWM support for both nodes and could provide additional debugging visualizers for the team. While the serial monitor could have been utilized to read the current value of the slider and push button, the team believed it would be much more engaging and intuitive to connect the readings to a visible brightness. Figure 16 below shows part of the sample code from Appendix C.2 to test this fundamental communication of the RemoteXY app through Bluetooth to the *LAFVIN* car:

```
// Basic test of UI functions; setup headlights (A4 & A5)
void setup()
{
  RemoteXY_Init ();
  pinMode(A4,OUTPUT);
  pinMode(A5,OUTPUT);
}

void loop()
{
  RemoteXY_Handler (); // Call RemoteXY handler
  digitalWrite(A4,RemoteXY.Emergency_stop); // Toggle A4 lights if EMERGENCY STOP button pressed
  analogWrite(A5,map(RemoteXY.Speed_slider,0,100,0,255)); // Adjust brightness of A5 lights via speed slider UI element
}
```

Figure 16 - Test script to interpret how RemoteXY variables can be read. Given that the push button can only take on a value of 0 or 1, “digitalWrite()” can simply write this input to the A4 headlight pair. As the speed slider can take on values between 0 and 100, “analogWrite()” can send a mapped value to A5 in the new range of 0 to 255 to make it easier for the team to observe a gradual change in brightness as they move the slider. See Appendix C.2 for the full test script.

As shown above, this is a very simple test script constructed by the team to test two of the GUI elements to ensure each were formatted correctly for reading the value of both variables “RemoteXY.Emergency_stop” and “RemoteXY.Speed_slider”. When running this code onto the car, the group was successful in observing the described behavior above through the RemoteXY app. The Bluetooth component connected successfully, and the full GUI appeared on one of the team member’s mobile devices. When pressing the push button, the right set of LEDs toggled on until the button was released, fulfilling the “digitalWrite()” test line above. Then for moving the slider from 0 to 100, the group successfully observed the left set of LEDs slowly brightening as expected. Therefore, this team code has proven to the group that this is the proper notation for obtaining RemoteXY variable values. Knowing this, the group implemented their finalized program on the car and were ready to begin testing with the true interface instead of the serial monitor.

Final Mobile GUI Car Control Testing

With all the code ready to test, the group began final quality tests of their program. Upon uploading their final program onto the car’s Arduino, one member would observe the behavior of the car as the other would connect to the HM-10 through the RemoteXY app. Once connected, the interface again successfully appeared on the team member’s phone and was ready for tests. First, they focused on testing the joystick. When moving it forwards, the car correctly began to move straight ahead, increasing slightly in speed as the joystick moved to the top of the screen until it was clamped by the speed slider set to a maximum value of 2*100, successfully outputting a PWM of 200 to both wheels. When turning heavily towards the left or right as the car moved forwards, the corresponding LED headlight-side pairs toggled on to indicate the car was turning in a direction; when moving left, the left-side lights toggled, and similarly for moving right, the right-side LEDs turned on successfully. Now when the team member moved the joystick downwards, the car correctly changed directions and started moving backwards **while correctly flashing the hazard lights** in a constant loop until they stopped movement or started

going forwards again. Turning was also successfully observed as it utilized the same lines of code for backwards movement, now without needing to toggle the headlights by side as the flashing lights were meant to only be shown when in reverse–this was done to lower the already high complexity of the program, correctly maintaining the hazard flash effect as desired. Additionally, the car would not move until the **joystick's y-position was a certain distance away from the center**, successfully proving that the car would be able to begin moving once the joystick was above a threshold value, avoiding the issue of donut-spinning–to be explained next–and negative PWM values.

Moving to the speed slider, when moving the car in any direction and decreasing the speed slider to zero, the group successfully observed the car begin to slow down to a stop once the slider reached the bottom. No matter where they moved the joystick, the slider successfully prevented any movement as it had a value of 0 when at the very bottom position. Moving the slider back to 100 resulted in the car correctly increasing in speed back to normal. In addition, pressing the emergency push button at any given time correctly stopped the car in place, toggling the hazard lights on until the button was released; once unpressed, the lights would untoggle and the car would resume its previous movement.

Lastly, the member would test the wide-front collision detection feature of the car by driving it slowly towards a wall or obstacle. When slowly approaching the obstruction, the car was observed successfully toggling all hazard lights on and stopping about **3 cm** away, preventing a collision; this was therefore a major success for this safety feature. Once stopped, any attempt to move the joystick upwards to go forward was ignored correctly, forcing the user to **move backwards** until the hazard was no longer detected. Once back far enough, the hazard lights would turn off and successfully enable forward movement once again. When moving the car into a dark area, the **headlights** would toggle on, proving the success of the night-time auto-light feature implemented. To conclude these successful tests, any attempt to emergency stop in a dark environment correctly **kept the headlights on** after releasing the button, fulfilling this final feature and resulting in a final project that was proven successful in all minimum viable product goals, as well as the first stretch goal, set out by the team. As sometimes the IR sensors can be a little too sensitive, the group had a mini screwdriver on hand to slightly adjust how close the car can be to a wide obstacle in front before stopping on its own. The last remaining photo for this report in Figure 17 below showcases the team's final project with all features listed prior implemented and verified in this section to be fully successful:

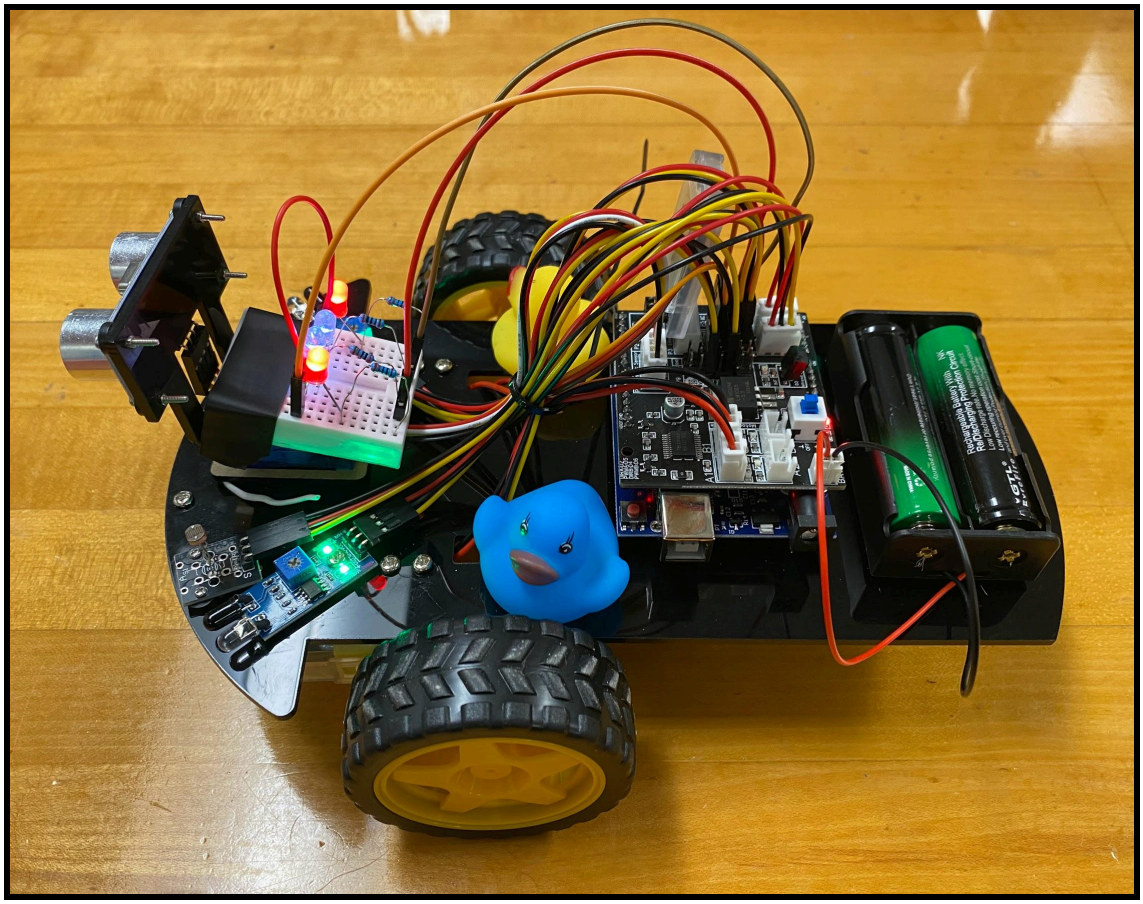


Figure 17 - Final car constructed and verified to be successful in fulfilling all goals set out by the group. All components are secured to the base of the car, as well as wires zip-tied together in bundles to allow for space to incorporate the miniature safety light circuits on a separate mini breadboard. Loose parts have been secured to the base using hot glue, achieving high stability in the wheels and screw connections of different components. The dual set of headlights can be seen on the miniature breadboard, toggled on as the left IR sensor has detected a hand in the image–this is the component with the two green LED lights, meaning that it has detected an obstacle.

Fully shown above, the team's final car has been a massive success for their final project as it has been tested and proven to fulfill and even exceed all minimum viable product goals and the RemoteXY stretch goal, providing users with not only a wide range of control features over the car from flashing hazard lights and emergency stopping, but an interactive and intuitive user interface that removes the original hassles associated with general Bluetooth serial monitors. In addition to working on one of the member's mobile devices, it also worked on the other teammate's phone; they each have an Apple or an Android individually, proving that the RemoteXY control over the car is functional across different mobile platforms and is not limited to just one. Across all lab sessions, the group is incredibly proud of how far they have progressed since the first day; they couldn't have been more proud seeing that they were highly successful in creating a working user interface for any classmate to try out with not just the team's car project, but potentially in future fun hobbies as an extension beyond the serial monitor of Arduino.

Extra Project Discoveries

During this final project, the group was very happy to discover great amounts of new information related to graphical user interfaces and Bluetooth communication. A few roadblocks were also encountered but successfully overcome by the group; they will be discussed in this section here. Written observations of these obstacles can be also found in Appendix D for further reference.

RemoteXY Limitations & Required Bluetooth Hardware Types

Upon utilizing the RemoteXY editor, the group was presented with a few limitations of the software. As it was free, there was a **total instance count** that could be used by the group; if they went above 5 total elements, the software would not allow the team to download the code unless they bought a PRO RemoteXY license. Therefore, the group had to carefully plan which functions they wanted to be directly controlled by the GUI, as well as which type of interactive object to use. In the end, they were able to settle on using **the joystick, slide switch, and push button** with two text labels; these were given values "EMERGENCY STOP" and "Speed" to place above the button and slider respectively, totaling to 5 GUI elements to remain in the free version limit. However, there was also an issue of the **allowed Bluetooth receiver types**. While the *LAFVIN* car kit came with the JDY-16 Bluetooth receiver, this was **not supported in the RemoteXY editor**. The group had to therefore request an assistant for an **HM-10** receiver which was an option in the editor window; they fortunately had a handful of these types available for use in projects, allowing the group to successfully use the RemoteXY editor software.

Searching for the Bluetooth Receiver Name

Similar to the Bluetooth communication lab session, the group would have to search for the Bluetooth receiver when using either the Arduino serial app or RemoteXY. However, as many laptops also have Bluetooth enabled, **there was a long list of devices the group had to search through to select** as hopefully their HM-10 receiver. Luckily, the RemoteXY editor also had a built-in feature where users can designate a custom device name to their receiver to be found in the app itself. Therefore, they set their HM-10 as "Bluetooth+ Car", allowing them to easily find the receiver in the RemoteXY app; this is also saved after selection, allowing the group to no longer need to search for the part again. Once done, they could later simply click on the stored "Bluetooth+ Car" icon to immediately reconnect to their receiver, allowing the group to overcome this smaller but important barrier for ease of use.

Important RemoteXY Syntax

One last drawback that is not as significant but deemed important is that in order for RemoteXY GUI elements to be read as Arduino variables, **the program must use modified object-oriented syntax**. For example, **"RemoteXY.Direct_joystick_X"** for x-values of the joystick–this goes from -100 to 100–compared to "Direct_joystick_X". Additionally, **"RemoteXY_delay()" must be used in place of "delay()" to not interfere with the internal Bluetooth library of RemoteXY**. With this in mind, the group made sure to follow this important syntax, allowing their GUI to be properly implemented in their code.

Reflections & Takeaways

This final project has given the group a great opportunity to explore their creative and innovative sides of the electrical engineering world, utilizing all their knowledge built up over the course of the semester to construct a car that goes beyond the basic functions of serial monitor control, instead expanding into the realm of user interfaces. Beginning with fixing up their car constructed in a previous set of labs, the team was able to directly investigate the wire connections and research even deeper into the *LAFVIN* shield for available pins and how different nodes were connected. This led them to the discovery of repurposing the "SDA" and "SCL" pins for general outputs as these were I2C lines that could instead be set up for normal usage. Shifting into constructing and drafting all the software for the project, the team constructed a series of test scripts to verify fundamental behavior of their car–all is presented in Appendix C–before consolidating all results into a final, polished program shown in Appendix B. This choice to have each teammate construct a small set of test programs allowed the group to be highly efficient overall as each was given an engaging task to experiment with and investigate. Once each was proven to be successful in their respective functions, the team moved to implementing all the small test code files into a final program, giving the group major insight into how effective their final car product worked with their small test code scripts now repolished and upgraded to interpret real sensor and interface data. Finally, the team being able to successfully construct their very own user interface with RemoteXY, as well as observe it successful connect to and control their car from a mobile device, provided the team with great insight into just how much additional functionality a GUI can provide for general projects and hobbies in contrast to only using the Arduino serial monitor, giving them a direct experience with a real-world application of graphical interfaces such as for car music players or interactive television screens. By lastly seeing their car succeed in all remote control and extra feature goals set out by the members, the team was given a chance to reflect on just how much progress they have made over the semester, incredibly proud of the major success their upgraded car has exemplified for their final project.

User interfaces were only slowly beginning to take shape in new, revolutionary forms in the early 2000s, previously limited by electronics and technology. As the invention of the brick phone slowly evolved along with the electronics world, companies began to shift focus onto the efficiency of user interactions with the technological side of products, continuously growing until the groundbreaking reveal of the iPhone by Steve Jobs in 2007 [4]. This moment in time changed the course of history by not only revolutionizing the ways in which society now communicates using daily, but introducing a massive improvement to user interfaces to the likes that no consumer has seen to that scale prior to the reveal. In the current day, consumers are continuing to see upgrades and innovations to this remarkable moment in history, such as the unveiling of the virtual reality headset, the XBox Kinect 3D, the Apple Pro vision, or even the Meta smart glasses where users can interact with 3D positions to navigate menus. From this final project with the addition of RemoteXY interfacing, the team has been given the greatest opportunity to explore the groundbreaking world of not only wireless communication, but just how incredibly versatile a graphical user interface can be in the real world. There is so much more to view in the world of microcontrollers–and this course has been a very insightful stairway for discovering the foundation of the modern world of technology and industries of today, uncovering breakthrough technological advancements to come in the near future.

References

[1] "Arduino Documentation." Arduino. https://docs.arduino.cc/ (accessed November 30, 2024).

This source was used to refresh on specific documentation for important Arduino functions and proper usage of UART communication for Bluetooth.

[2] "LAFVIN Smart Robot Car 2WD Chassis Kit." LAFVIN. https://lafvintech.com/products/lafvin-smart-robot-car-2wd-chassis-kit-upgraded-v2-0-for-arduino-robot-stem-graphical-programming-robot-car (accessed November 30, 2024).

This source was used to review connections for the 2WD v2 car kit. This was to ensure the group had extra pins to implement safety or additional car features.

[3] "Graphical user interface." Wikipedia. https://en.wikipedia.org/wiki/Graphical_user_interface (accessed November 30 2024).

This source was used to read more into the history and evolution of GUI concepts and functionality over time.

[4] "Steve Jobs debuts the iPhone | This day in History: January 9." History. https://www.history.com/this-day-in-history/steve-jobs-debuts-the-iphone (accessed November 30 2024).

This source was used to research the revolutionary reveal of the iPhone by Steve Jobs, the first major product to revolutionize communication and graphical user interfaces with a dynamic touchscreen concept.

[5] "RemoteXY Editor." RemoteXY. https://remotexy.com/en/editor/ (accessed November 30 2024).

This source was used to draft, test, and construct the graphical user interface to be displayed on mobile devices to control the car remotely through Bluetooth.

Appendices

[Appendix A - link(s) to the group repository]

Group repository link: https://github.com/HiWillCat/BluetoothPlus-Car-CMPE3815

★ *Examples and the final code are provided in the main branch folders of the GitHub!*

[Appendix B - final version of the "Bluetooth+ Mobile Device Car Control" code; this can downloaded from the repository using the GitHub link in Appendix A]

```
// === FINAL PROJECT - Bluetooth+ Mobile Device Car Control === //
// Students: Aria Lindberg, Will Gambero
// CMPE3815 - Microcontroller Systems
// ========================================================== //

// <<<<<<<<<<<<<<<< "Bluetooth+ Mobile Device Car Control" >>>>>>>>>>>>>>>> //

// Allow your mobile device to remotely control the LAFVIN car over Bluetooth!
// Use the RemoteXY app to enable mobile access!
// Comes with the following GUI features:
  // Joystick -> steering and movement
  // Slider -> speed control
  // Button -> emergency stop
// And built-in safety features:
  // Front-IR object detection -> stops if a wall is ahead
  // Night-time headlights -> toggles lights if the environment becomes dark
  // Turning lights -> toggles side lights depending on which direction the car swerves to
  // Flashing reverse hazardlights -> blinks lights when reversing

// <<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<< //

// ====== REMOTE XY SETUP ================================ //
// RemoteXY setup (GNU Lesser General Public License)
// Code generated from editor page of RemoteXY website

// RemoteXY select connection mode and include library
#define REMOTEXY_MODE__HARDSERIAL

// RemoteXY connection settings
#define REMOTEXY_SERIAL Serial
#define REMOTEXY_SERIAL_SPEED 9600

#include <RemoteXY.h>

// RemoteXY GUI configuration
#pragma pack(push, 1)
```

```
uint8_t RemoteXY_CONF[] =   // 117 bytes
  { 255,4,0,0,0,110,0,19,0,0,0,66,108,117,101,116,111,111,116,104,
  43,32,67,97,114,0,31,2,106,200,200,84,1,1,5,0,5,21,80,60,
  60,119,5,74,74,32,36,30,31,4,48,52,7,86,17,11,20,60,0,1,
  24,1,23,81,57,57,54,66,14,14,0,1,31,0,129,1,124,71,29,17,
  10,20,7,64,24,83,112,101,101,100,0,129,7,112,71,29,44,59,35,4,
  64,24,69,77,69,82,71,69,78,67,89,32,83,84,79,80,0 };

// this structure defines all the variables and events of your control interface
struct {

    // input variables
  int8_t Direct_joystick_X; // from -100 to 100
  int8_t Direct_joystick_Y; // from -100 to 100
  int8_t Speed_slider; // from 0 to 100
  uint8_t Emergency_stop; // =1 if button pressed, else =0

    // other variable
  uint8_t connect_flag;  // =1 if wire connected, else =0

} RemoteXY;
#pragma pack(pop)
// ============================================================ //

// ======= LIBRARIES =============== //
#include <Servo.h> // Official servo library
Servo servo_motor; // Create servo obj
// ================================== //

// ======= GLOBAL VARIABLES ========== //

// Head & trail lights
const int Rightlights_pin = A5;
const int Leftlights_pin = A4;

// Wheel control pins
const int EN1 = 2; // Enable pin 1 (Digital)
const int EN2 = 4; // Enable pin 2 (Digital)
const int MC1 = 5; // DC motor control 1 pin (PWM)
const int MC2 = 6; // DC motor control 2 pin (PWM)

// Ultrasonic/servo pins
const int Trigger_pin = 12; // Trigger connection
const int Echo_pin = 13; // Echo connection
const int Servo_pin = 10; // PWM

// Photoresistor, sense for light
const int PhotoResistorLeft_pin = A0;
```

```
const int PhotoResistorRight_pin = A3;

// Line trackers, sense for lines below car
const int LineTrackLeft_pin = 9;
const int LineTrackRight_pin = 7;

// IR sensors
const int IRSenseLeft_pin = A1;
const int IRSenseRight_pin = A2;

// CONSTANTS ~~~~~~~~~~~~~~~~~~~~~~~~
// Constant values
const int LeftWheelTune = 0; // Offset value to approximate L-wheel power = R-wheel power, recommended value = 29
const int Nighttime_Threshold = 400; // Sets photosensor threshold for lights to turn on
const int Move_Threshold = 40; // Sets movement threshold before car is allowed to moved (solves "donut" behavior)
const int TurnMaxRatio = 40; // Sets max offset value to allow for turning
const int Blink_Threshold = 20; // Sets min value for X joystick value for turning side light(s) on

// Tracking values
int Speed_scalar = 0; // Tracks value of speed slider, 0 to 100

bool Nighttime = false; // Tracks if ambient area is dark
byte Move_direction = 1; // Tracks forward/backward movement; 1 forward VS 0 backward
bool Hazard_Stopped = false; // Tracks for hazard stopping; uses IR sensors
bool Emergency_Stopped = false; // Tracks emergency stopping; uses GUI button
int Flashing_Counter = 0; // Counts loop iterations for light flashing effect
int Flashing_Threshold = 2000; // Threshold to toggle lights to opposite state

// ================================= //

// ======= BUILT IN FUNCTIONS ======= //
void setup() {
  // Set pin modes & connect servo
  // Wheels:
  pinMode(EN1, OUTPUT);
  pinMode(EN2, OUTPUT);
  pinMode(MC1, OUTPUT);
  pinMode(MC2, OUTPUT);

  // Ultrasonic:
  pinMode(Trigger_pin,OUTPUT); // Trigger is output to ultrasonic
  pinMode(Echo_pin,INPUT_PULLUP); // Echo is an input from ultrasonic

  // Photoresistor:
  pinMode(PhotoResistorLeft_pin,INPUT_PULLUP);
  pinMode(PhotoResistorRight_pin,INPUT_PULLUP);

  // Line trackers:
```

```
  pinMode(LineTrackLeft_pin,INPUT_PULLUP);
  pinMode(LineTrackRight_pin,INPUT_PULLUP);

  // IR sensors:
  pinMode(IRSenseLeft_pin,INPUT_PULLUP);
  pinMode(IRSenseRight_pin,INPUT_PULLUP);

  // Connect servo via library
  pinMode(Servo_pin,OUTPUT);
  servo_motor.attach(Servo_pin);

  // Head & tail lights
  pinMode(Rightlights_pin,OUTPUT);
  pinMode(Leftlights_pin,OUTPUT);

  // Setup serial monitor for debugging
  Serial.begin(9600);
  RemoteXY_Init (); // Begin RemoteXY library functions
}

void loop()
{
  // Begin constant loop of checking RemoteXY values
  RemoteXY_Handler (); // Call RemoteXY handler
  DirectionConverter(); // Get joystick directions & write values to DC motor wheels
  FrontHazardCheck(); // Check for walls in front of car
  AmbientLightCheck(); // Checks brightness of room to toggle nighttime lights if needed
  BlinkerLightsCheck(); // Check for turning lights

  // If reversing, flash lights via counter (on-off sequence)
  if (Move_direction == 0){
    Flashing_Counter ++; // Add 1 to counter
    if (Flashing_Counter > 2*Flashing_Threshold){ // If above 2*threshold, set lights to on and reset counter
      analogWrite(Rightlights_pin,255);
      analogWrite(Leftlights_pin,255);
      Flashing_Counter = 0;
    }
    else if(Flashing_Counter > Flashing_Threshold){ // If above normal threshold, set lights to off
      analogWrite(Rightlights_pin,0);
      analogWrite(Leftlights_pin,0);
    }
    else{ // If below threshold, set lights to on
      analogWrite(Rightlights_pin,255);
      analogWrite(Leftlights_pin,255);
    }

  }
  else{
```

```
    Flashing_Counter = 0; // Not in reverse direction, set counter to 0
  }

  // Check for emergency stopping (GUI button)
  if (RemoteXY.Emergency_stop == 1){
    Emergency_Stopped = true; // Button is held down
    // Turn on all lights
    analogWrite(Rightlights_pin,255);
    analogWrite(Leftlights_pin,255);
  }
  else{
    Emergency_Stopped = false; // Button not held down
    // Turn off all lights unless nighttime
    if (Nighttime == false){
      analogWrite(Rightlights_pin,0);
      analogWrite(Leftlights_pin,0);
    }
  }
}

// ==================================== //

// === CUSTOM FUNCTIONS ================ //

// Checks GUI values and toggles movement parameters as needed
void DirectionConverter(){
  // Check for if emergency stop button is not held down
  if (Emergency_Stopped == false){
    // Forward or backward check (Y direction)
    if (RemoteXY.Direct_joystick_Y > Move_Threshold and Hazard_Stopped == false){ // Forwards & no hazard stopping
      Move_direction = 1;
      MoveForwards(RemoteXY.Direct_joystick_X,abs(RemoteXY.Direct_joystick_Y)); // Move backwards (send absolute val of Y
position)
    }
    else if (RemoteXY.Direct_joystick_Y < -Move_Threshold){ // Backwards
      Move_direction = 0;
      MoveBackwards(RemoteXY.Direct_joystick_X,abs(RemoteXY.Direct_joystick_Y)); // Move forwards (send absolute val of Y
position)
    }
    else { // Stopped, y = 0
      Stop();
    }
  }
  else{ // Emergency stop pressed, stop car immediately
    Stop();
  }
}
```

```
// Checks for brightness and toggles headlights if dark
void AmbientLightCheck(){
  // Check for forward movement only
  if (Move_direction == 1){
    // Check for analog value above threshold (large value -> dark)
    if (analogRead(PhotoResistorLeft_pin)> Nighttime_Threshold or analogRead(PhotoResistorRight_pin) >
Nighttime_Threshold){
      // Above threshold, turn ON lights
      Nighttime = true;
      analogWrite(Rightlights_pin,255);
      analogWrite(Leftlights_pin,255);
    }
    else{
      // Below threshold, turn OFF lights
      Nighttime = false;
      analogWrite(Rightlights_pin,0);
      analogWrite(Leftlights_pin,0);
    }
  }
}

// Check turning lights for toggling
void BlinkerLightsCheck(){
  // Compare joystick's X value against threshold turning val (must be greater than constant to toggle turning lights)
  if (RemoteXY.Direct_joystick_X<-Blink_Threshold){ // Turn left lights on
    analogWrite(Leftlights_pin,255);
    analogWrite(Rightlights_pin,0);
  }
  else if (RemoteXY.Direct_joystick_X > Blink_Threshold){ // Turn right lights on
    analogWrite(Leftlights_pin,0);
    analogWrite(Rightlights_pin,255);
  }
  else{ // Not enough X value, turn off lights
    analogWrite(Leftlights_pin,0);
    analogWrite(Rightlights_pin,0);
  }
}

// Checks for wide-front hazards with IR sensors
void FrontHazardCheck(){
  // Check if either IR sensor is LOW (detects obstacle) and car is moving forward; ignore if moving backwards
  if ((digitalRead(IRSenseLeft_pin) == LOW or digitalRead(IRSenseRight_pin) == 0) and Move_direction == 1) {
    Stop();
    Hazard_Stopped = true; // Set hazard stop to true, stop any forward movement
    analogWrite(Leftlights_pin,255);
    analogWrite(Rightlights_pin,255);
  }
  else{
```

```
    Hazard_Stopped = false; // Reset hazard stop to false, re-allow forward movement
    analogWrite(Leftlights_pin,0);
    analogWrite(Rightlights_pin,0);
  }
}

// Wheel functions ~~~~~~~~~~~~~~~~~~~~~~~~~~
// Move forward
void MoveForwards(int TurnRatio, int Speed) {
  int MapTurnRatio = map(TurnRatio,-100,100,-TurnMaxRatio,TurnMaxRatio); // Scale turning value (-100 to 100) to allowed
ratio
  digitalWrite(EN1, HIGH);
  digitalWrite(EN2, LOW);
  // Write corresponding x-joystick speeds to each wheel; scale by "Speed_slider" via mapping scale by 2*"Speed_slider"
with map()
  analogWrite(MC1, map(Speed+MapTurnRatio-LeftWheelTune,0,140,0,2*RemoteXY.Speed_slider));
  analogWrite(MC2, map(Speed-MapTurnRatio,0,140,0,2*RemoteXY.Speed_slider));
}

// Move backwards
void MoveBackwards(int TurnRatio, int Speed) {
  int MapTurnRatio = map(TurnRatio,-100,100,-TurnMaxRatio,TurnMaxRatio); // Scale turning value (-100 to 100) to allowed
ratio
  digitalWrite(EN1, LOW);
  digitalWrite(EN2, HIGH);
  // Write corresponding x-joystick speeds to each wheel;scale by "Speed_slider" via mapping scale by 2*"Speed_slider"
with map()
  analogWrite(MC1, map(Speed+MapTurnRatio-LeftWheelTune,0,140,0,2*RemoteXY.Speed_slider));
  analogWrite(MC2, map(Speed-MapTurnRatio,0,140,0,2*RemoteXY.Speed_slider));
}

// Stop movement
void Stop() {
  Move_direction = 1; // Preset direction to forward to reset lights
  analogWrite(MC1, 0);
  analogWrite(MC2, 0);
}
// ================================== //
```

[**Appendix C -** fundamental test scripts created to verify minimal viable product goals; all are also provided in the "Examples" folder on GitHub]

Appendix C.1 – "SerialBluetoothCommands"

```
// === FINAL PROJECT - Bluetooth+ Car === //

// EXAMPLE 1 - Fundamental Bluetooth control
  // Ensures single letter commands can be sent & received over Bluetooth
  // Use the Arduino Bluetooth Controller app to send letter commands

// ============================================== //

// ======= LIBRARIES =============== //
// #include <IRremote.h>  // Official IR remote library (for early tests)
#include <Servo.h> // Official servo library
Servo servo_motor; // Create servo obj
// ================================== //

// ======= GLOBAL VARIABLES ========= //

// Head & trail lights
const int Headlights_pin = A4;
const int Taillights_pin = A5;

// Wheel control pins
const int EN1 = 2; // Enable pin 1 (Digital)
const int EN2 = 4; // Enable pin 2 (Digital)
const int MC1 = 5; // DC motor control 1 pin (PWM)
const int MC2 = 6; // DC motor control 2 pin (PWM)

// Ultrasonic/servo pins
const int Trigger_pin = 12; // Trigger connection
const int Echo_pin = 13; // Echo connection
const int Servo_pin = 10; // PWM

// Photoresistor, sense for light
const int PhotoResistorLeft_pin = A0;
const int PhotoResistorRight_pin = A3;

// Line trackers, sense for lines below car
const int LineTrackLeft_pin = 9;
const int LineTrackRight_pin = 7;

// IR sensors
const int IRSenseLeft_pin = A1;
const int IRSenseRight_pin = A2;
```

```
// CONSTANTS ~~~~~~~~~~~~~~~~~~~~~~~~
// Constant values
const int RestingSpeed = 30; // Normal move speed
const int SprintSpeed = 100; // Sprint speed (when going to point)

const int LeftWheelTune = 0;//29; // Offset value to approximate L-wheel power = R-wheel power

// ================================== //

// ======= BUILT IN FUNCTIONS ======= //
void setup() {
  // Set pin modes & connect servo
  // Wheels:
  pinMode(EN1, OUTPUT);
  pinMode(EN2, OUTPUT);
  pinMode(MC1, OUTPUT);
  pinMode(MC2, OUTPUT);

  // Ultrasonic:
  pinMode(Trigger_pin,OUTPUT); // Trigger is output to ultrasonic
  pinMode(Echo_pin,INPUT_PULLUP); // Echo is an input from ultrasonic

  // Photoresistor:
  pinMode(PhotoResistorLeft_pin,INPUT_PULLUP);
  pinMode(PhotoResistorRight_pin,INPUT_PULLUP);

  // Line trackers:
  pinMode(LineTrackLeft_pin,INPUT_PULLUP);
  pinMode(LineTrackRight_pin,INPUT_PULLUP);

  // IR sensors:
  pinMode(IRSenseLeft_pin,INPUT_PULLUP);
  pinMode(IRSenseRight_pin,INPUT_PULLUP);

  // Connect servo via library
  pinMode(Servo_pin,OUTPUT);
  servo_motor.attach(Servo_pin);

  // Head & tail lights
  pinMode(Headlights_pin,OUTPUT);
  pinMode(Taillights_pin,OUTPUT);

  // Setup serial monitor for debugging
  Serial.begin(9600);
  Serial.setTimeout(250); // Time between serial inputs in ms, default is 1000 ms or 1s
}
```

```
void loop() {
  // Start loop of waiting for message from Bluetooth device

  // Message received, only check 1st character
  if (Serial.available()){
    // Get 1st character
    String Message = Serial.readString(); // Full message
    String New_letter = Message.substring(0,1); // Get only 1st char
    Serial.println(New_letter); // Print the char

    // Check for forward/backward or stop commands:
    // Uppercase -> FAST; lowercase -> SLOW

    // Forward
    if (New_letter == "a" or New_letter == "A"){
      if (New_letter == "a"){
        MoveForwards(RestingSpeed);
      }
      else{
        MoveForwards(SprintSpeed);
      }
    }
    // Backward
    else if (New_letter == "l" or New_letter == "L"){
      if (New_letter == "l"){
        MoveBackwards(RestingSpeed);
      }
      else{
        MoveBackwards(SprintSpeed);
      }
    }
    // Stop (freeze any movement)
    else if (New_letter == "s" or New_letter == "S"){
      Stop();
    }
  }
}

// Wheel functions ~~~~~~~~~~~~~~~~~~~~~~~~~~
// Move forward
void MoveForwards(int NewSpeed) {
  analogWrite(Headlights_pin,255); // Test turn ON front lights
  analogWrite(Taillights_pin,0); // Test turn OFF rear lights
  digitalWrite(EN1, HIGH);
  digitalWrite(EN2, LOW);
  analogWrite(MC1, NewSpeed-LeftWheelTune); // Tuning left wheel
  analogWrite(MC2, NewSpeed);
}
```

```
// Move backwards
void MoveBackwards(int NewSpeed) {
  analogWrite(Headlights_pin,0); // Test turn OFF front lights
  analogWrite(Taillights_pin,255); // Test turn ON rear lights
  digitalWrite(EN1, LOW);
  digitalWrite(EN2, HIGH);
  analogWrite(MC1, NewSpeed-LeftWheelTune); // Tuning left wheel
  analogWrite(MC2, NewSpeed);
}

// Turn left
void TurnLeft() {
  digitalWrite(EN1, LOW);
  digitalWrite(EN2, LOW);
  analogWrite(MC1, RestingSpeed);
  analogWrite(MC2, RestingSpeed);
}

// Turn right
void TurnRight() {
  digitalWrite(EN1, HIGH);
  digitalWrite(EN2, HIGH);
  analogWrite(MC1, RestingSpeed);
  analogWrite(MC2, RestingSpeed);
}

// Stop movement
void Stop() {
  analogWrite(Headlights_pin,0); // Test turn OFF front lights
  analogWrite(Taillights_pin,0); // Test turn OFF rear lights
  analogWrite(MC1, 0);
  analogWrite(MC2, 0);
}
```

Appendix C.2 – “UITestControls”

```
// === FINAL PROJECT - Bluetooth+ Car === //

// EXAMPLE 2 - RemoteXY GUI Test
  // Test reading RemoteXY GUI functions using library and online editor

// ============================================ //

/*
   -- New project --

   This source code of graphical user interface
   has been generated automatically by RemoteXY editor.
   To compile this code using RemoteXY library 3.1.13 or later version
   download by link http://remotexy.com/en/library/
   To connect using RemoteXY mobile app by link http://remotexy.com/en/download/
     - for ANDROID 4.15.01 or later version;
     - for iOS 1.12.1 or later version;

   This source code is free software; you can redistribute it and/or
   modify it under the terms of the GNU Lesser General Public
   License as published by the Free Software Foundation; either
   version 2.1 of the License, or (at your option) any later version.
*/

//////////////////////////////////////////////
//        RemoteXY include library          //
//////////////////////////////////////////////

// you can enable debug logging to Serial at 115200
//#define REMOTEXY__DEBUGLOG

// RemoteXY select connection mode and include library
#define REMOTEXY_MODE__HARDSERIAL


// RemoteXY connection settings
#define REMOTEXY_SERIAL Serial
#define REMOTEXY_SERIAL_SPEED 9600


#include <RemoteXY.h>

// RemoteXY GUI configuration
#pragma pack(push, 1)
uint8_t RemoteXY_CONF[] =   // 117 bytes
  { 255,4,0,0,0,110,0,19,0,0,0,66,108,117,101,116,111,111,116,104,
  43,32,67,97,114,0,31,2,106,200,200,84,1,1,5,0,5,21,80,60,
```

```
  60,119,5,74,74,32,36,30,31,4,48,52,7,86,17,11,20,60,0,1,
  24,1,23,81,57,57,54,66,14,14,0,1,31,0,129,1,124,71,29,17,
  10,20,7,64,24,83,112,101,101,100,0,129,7,112,71,29,44,59,35,4,
  64,24,69,77,69,82,71,69,78,67,89,32,83,84,79,80,0 };

// this structure defines all the variables and events of your control interface
struct {

    // input variables
  int8_t Direct_joystick_x; // from -100 to 100
  int8_t Direct_joystick_y; // from -100 to 100
  int8_t Speed_slider; // from 0 to 100
  uint8_t Emergency_stop; // =1 if button pressed, else =0

    // other variable
  uint8_t connect_flag;  // =1 if wire connected, else =0

} RemoteXY;
#pragma pack(pop)

/////////////////////////////////////////////
//           END RemoteXY include          //
/////////////////////////////////////////////


// Basic test of UI functions; setup headlights (A4 & A5)
void setup()
{
  RemoteXY_Init ();
  pinMode(A4,OUTPUT);
  pinMode(A5,OUTPUT);
}

void loop()
{
  RemoteXY_Handler (); // Call RemoteXY handler
  digitalWrite(A4,RemoteXY.Emergency_stop); // Toggle A4 lights if EMERGENCY STOP button pressed
  analogWrite(A5,map(RemoteXY.Speed_slider,0,100,0,255)); // Adjust brightness of A5 lights via speed slider UI element
}
```

Appendix C.3 – “SafetyFeatures”

```
// === FINAL PROJECT - Bluetooth+ Car === //

// EXAMPLE 3 - Safety features
  // Run safety features (passive/active) with sensors
  // Reuse letters and Bluetooth app from EXAMPLE 1

// ============================================== //

// ======= LIBRARIES =============== //
// #include <IRremote.h>  // Official IR remote library (for early tests)
#include <Servo.h> // Official servo library
Servo servo_motor; // Create servo obj
// ================================== //

// ======= GLOBAL VARIABLES ========= //

// Head & trail lights
const int Rightlights_pin = A4;
const int Leftlights_pin = A5;

// Wheel control pins
const int EN1 = 2; // Enable pin 1 (Digital)
const int EN2 = 4; // Enable pin 2 (Digital)
const int MC1 = 5; // DC motor control 1 pin (PWM)
const int MC2 = 6; // DC motor control 2 pin (PWM)

// Ultrasonic/servo pins
const int Trigger_pin = 12; // Trigger connection
const int Echo_pin = 13; // Echo connection
const int Servo_pin = 10; // PWM

// Photoresistor, sense for light
const int PhotoResistorLeft_pin = A0;
const int PhotoResistorRight_pin = A3;

// Line trackers, sense for lines below car
const int LineTrackLeft_pin = 9;
const int LineTrackRight_pin = 7;

// IR sensors
const int IRSenseLeft_pin = A1;
const int IRSenseRight_pin = A2;

// CONSTANTS ~~~~~~~~~~~~~~~~~~~~~~~~
// Constant values
const int RestingSpeed = 30; // Normal move speed
const int SprintSpeed = 100; // Sprint speed (when going to point)
```

```
const int LeftWheelTune = 29; // Offset value to approximate L-wheel power = R-wheel power

// Tracking values
byte Move_direction = 1; // Tracks forward/backward movement; 1 forward VS 0 backward

// ================================= //

// ======= BUILT IN FUNCTIONS ======= //
void setup() {
  // Set pin modes & connect servo
  // Wheels:
  pinMode(EN1, OUTPUT);
  pinMode(EN2, OUTPUT);
  pinMode(MC1, OUTPUT);
  pinMode(MC2, OUTPUT);

  // Ultrasonic:
  pinMode(Trigger_pin,OUTPUT); // Trigger is output to ultrasonic
  pinMode(Echo_pin,INPUT_PULLUP); // Echo is an input from ultrasonic

  // Photoresistor:
  pinMode(PhotoResistorLeft_pin,INPUT_PULLUP);
  pinMode(PhotoResistorRight_pin,INPUT_PULLUP);

  // Line trackers:
  pinMode(LineTrackLeft_pin,INPUT_PULLUP);
  pinMode(LineTrackRight_pin,INPUT_PULLUP);

  // IR sensors:
  pinMode(IRSenseLeft_pin,INPUT_PULLUP);
  pinMode(IRSenseRight_pin,INPUT_PULLUP);

  // Connect servo via library
  pinMode(Servo_pin,OUTPUT);
  servo_motor.attach(Servo_pin);

  // Head & tail lights
  pinMode(Rightlights_pin,OUTPUT);
  pinMode(Leftlights_pin,OUTPUT);

  // Setup serial monitor for debugging
  Serial.begin(9600);
  Serial.setTimeout(250); // Time between serial inputs in ms, default is 1000 ms or 1s
}

void loop() {
  // Start loop safety feature checks
```

```
  AmbientLightCheck();
  FrontHazardCheck();

  // Also run Bluetooth receiver, only check 1st character
  if (Serial.available()){
    // Get 1st character
    String Message = Serial.readString(); // Full message
    String New_letter = Message.substring(0,1); // Get only 1st char
    Serial.println(New_letter); // Print the char

    // Check for forward/backward or stop commands:
    // Uppercase -> FAST; lowercase -> SLOW

    // Forward
    if (New_letter == "a" or New_letter == "A"){
      if (New_letter == "a"){
        MoveForwards(RestingSpeed);
      }
      else{
        MoveForwards(SprintSpeed);
      }
    }
    // Backward
    else if (New_letter == "l" or New_letter == "L"){
      if (New_letter == "l"){
        MoveBackwards(RestingSpeed);
      }
      else{
        MoveBackwards(SprintSpeed);
      }
    }
    // Stop (freeze any movement)
    else if (New_letter == "s" or New_letter == "S"){
      Stop();
    }
  }
}

// Checks for brightness and toggles headlights if dark
void AmbientLightCheck(){
  // Check for analog value above threshold (large value -> dark)
  int Night_threshold = 400;
  if (analogRead(A0)> Night_threshold or analogRead(A3) > Night_threshold){
    // Above threshold, turn ON head lights
    analogWrite(Rightlights_pin,255);
  }
  else{
    // Below threshold, turn OFF head lights
```

```
    analogWrite(Rightlights_pin,0);
  }
}

// Checks for front hazards with IR sensors
void FrontHazardCheck(){
  if ((digitalRead(IRSenseLeft_pin) == LOW or digitalRead(IRSenseRight_pin) == 0) and Move_direction == 1) {
    Stop();
    analogWrite(Leftlights_pin,255);
  }
  else{
    analogWrite(Leftlights_pin,0);
  }
}

// Wheel functions ~~~~~~~~~~~~~~~~~~~~~~~~~~
// Move forward
void MoveForwards(int NewSpeed) {
  Move_direction = 1; // Set direction to forward
  digitalWrite(EN1, HIGH);
  digitalWrite(EN2, LOW);
  analogWrite(MC1, NewSpeed-LeftWheelTune); // Tuning left wheel
  analogWrite(MC2, NewSpeed);
}

// Move backwards
void MoveBackwards(int NewSpeed) {
  Move_direction = 0; // Set direction to backward
  digitalWrite(EN1, LOW);
  digitalWrite(EN2, HIGH);
  analogWrite(MC1, NewSpeed-LeftWheelTune); // Tuning left wheel
  analogWrite(MC2, NewSpeed);
}

// Turn left
void TurnLeft() {
  digitalWrite(EN1, LOW);
  digitalWrite(EN2, LOW);
  analogWrite(MC1, RestingSpeed);
  analogWrite(MC2, RestingSpeed);
}

// Turn right
void TurnRight() {
  digitalWrite(EN1, HIGH);
  digitalWrite(EN2, HIGH);
  analogWrite(MC1, RestingSpeed);
  analogWrite(MC2, RestingSpeed);
```

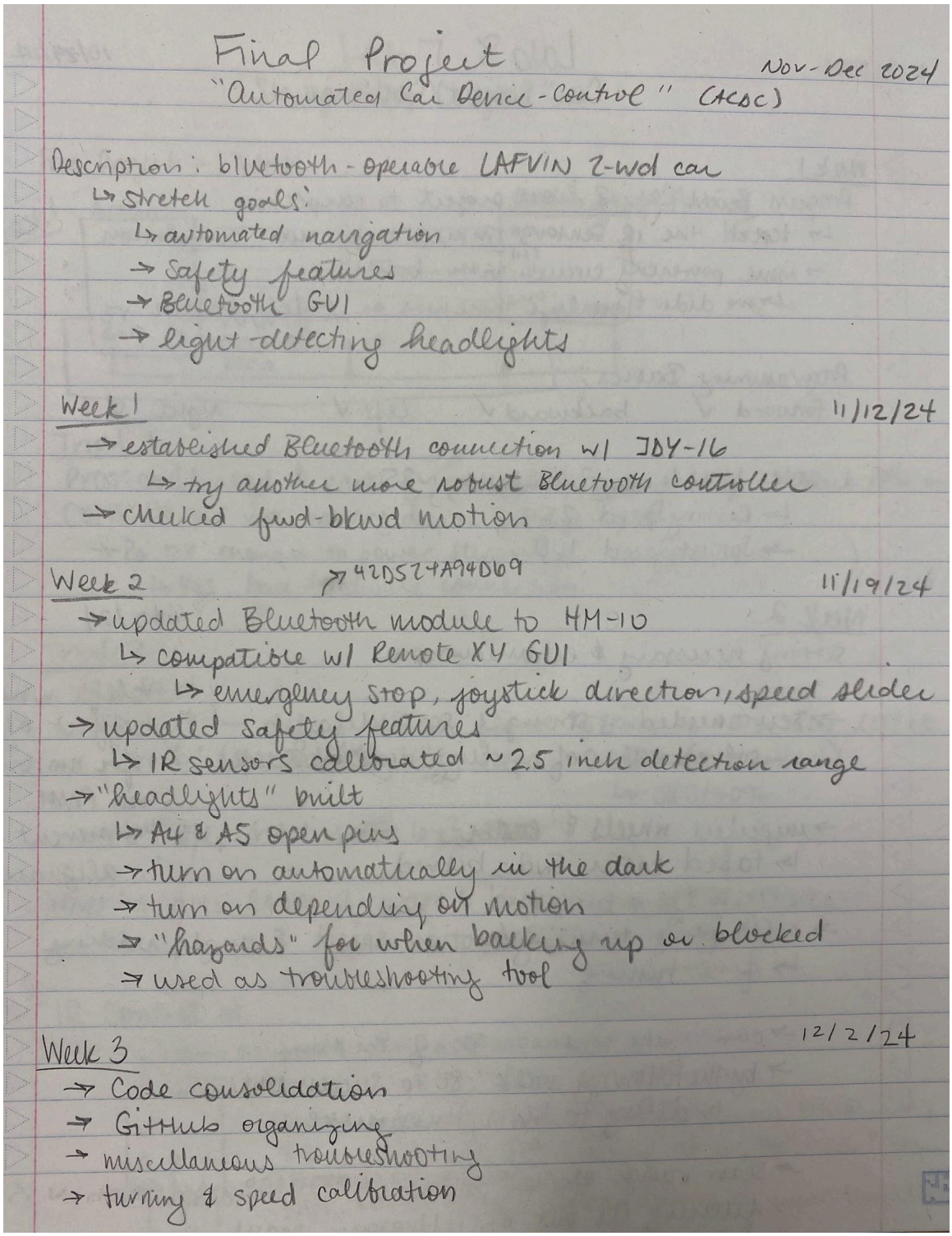
```
}

// Stop movement
void Stop() {
  analogWrite(MC1, 0);
  analogWrite(MC2, 0);
}
```

Appendix C.4 – “FlashingLights”

```
// === LAB #9 - Final Project === //
// Students: Aria Lindberg, Will Gambero
// CMPE3815 - Microcontroller Systems
// ========================================== //

// Code which runs the headlights of the car:

// Setting constants
const int right_headlights = A4;
const int left_headlights = A5;
bool HazardCondition = true;
int lightCounter = 0;

void setup() {
// set pin modes

pinMode(right_headlights, OUTPUT);
pinMode(left_headlights, OUTPUT);

}

void loop(){
  // LightsOn();
  // delay(500);
  // LightsOff();
  // delay(500);
  // LeftLights();
  // delay(500);
  // RightLights();
  // delay(500);
  // Hazards(true, 0);
  // delay(500);
}

  //function to activate headlights in the dark
  void LightsOn(){
    analogWrite(right_headlights, 255);
    analogWrite(left_headlights, 255);

  }

  //function to turn lights off
  void LightsOff(){
    analogWrite(right_headlights, 0);
    analogWrite(left_headlights, 0);

  }
```

```
//function for left lights
void LeftLights(){
  analogWrite(left_headlights, 255);

}

//function for right lights
void RightLights(){
  analogWrite(right_headlights, 255);

}

//Function to activate headlights when going backwards
void Hazards(bool HazardCondition, int lightCounter){
  LightsOff();

  while (HazardCondition == true && lightCounter <= 10000){
    LightsOn();
    lightCounter += 1;
  } //end while loop

  while (HazardCondition == true && lightCounter > 10000 && lightCounter <= 20000){
    LightsOff();
    lightCounter += 1;
  } //end while loop

  if (lightCounter == 20000){
    lightCounter = 0;
  }

}
```

[Appendix D - all written lab notes & diagrams by the team]



Final Project — Nov-Dec 2024

"Automated Car Device-Control" (ACDC)

Description: bluetooth-operable LAFVIN 2-wd car
- ↳ Stretch goals:
  - ↳ automated navigation
  - → Safety features
  - → Bluetooth GUI
  - → light-detecting headlights

Week 1 — 11/12/24
- → established Bluetooth connection w/ JDY-16
  - ↳ try another more robust Bluetooth controller
- → checked fwd-bkwd motion

Week 2 — 11/19/24 — ↗ 42D527A94D69
- → updated Bluetooth module to HM-10
  - ↳ compatible w/ Remote XY GUI
    - ↳ emergency stop, joystick direction, speed slider
- → updated safety features
  - ↳ IR sensors calibrated ~2.5 inch detection range
- → "headlights" built
  - ↳ A4 & A5 open pins
  - → turn on automatically in the dark
  - → turn on depending on motion
  - → "hazards" for when backing up or blocked
  - → used as troubleshooting tool

Week 3 — 12/2/24
- → Code consolidation
- → Github organizing
- → miscellaneous troubleshooting
- → turning & speed calibration

[**Appendix E** - minimal viable product goals for the group as a checklist]

MVP (minimum viable project) – the minimum requirements to deem the project successful:

- ☑ Fundamental Bluetooth control → **Success!**
  - Send, receive, and interpret single letters as movement commands. → **See Appendix C.1**
  - Implement Bluetooth receiver into car kit shield → Uses the HM-10
- ☑ Protection functions → **Success!**
  - Add internal functions to minimize the chance of the car hitting walls or edges. → Uses IR sensors
    - Utilize line trackers, ultrasonic, or IR sensors
- ☑ Extra car functionality; implement additional functionality to the car → **Success!** Team added:
  - **Night-time headlights**
  - **Reversal lights (flashing)**
  - **Turning lights by side**
  - **Front-wide obstacle detection & emergency stopping**