Introduction

Our project is a controllable car, which uses two Arduino Uno Boards that communicate with each other. It consists of a car, and a controller. The user can interact with the controller to manipulate the behaviour of the car in a variety of ways. The controller sends command signals to the car, which then reads these signals and reacts accordingly. Both the controller and car use an Arduino board to handle and process all inputs, outputs and communication. This project is a prototype for a final design, so the communication has been represented with a wired connection. If this were implemented as an actual product, this communication would be wireless.

Description

The controller consists of seven buttons and a potentiometer. The three leftmost buttons are used to define the state of the vehicle's motors: *forwards*, *neutral* and *backwards*, respectively. A green LED is situated next to the *forwards* button, and is lit up when the button is pressed. Similarly, a red LED is situated next to the *backwards* button. Only one of the two LEDs can be turned on at once. It should be noted that if the middle (*neutral*) button is pressed, both LEDs will be turned off. The speed at which the vehicle is moving can be adjusted via the potentiometer. To allow the car to turn *right* or *left*, two buttons are included on the far right side of the controller. If either of the buttons are pressed and held the car will start to turn accordingly. The car will stop turning only when the button is no longer being pressed. There is also a button to turn the LEDs of the car on or off. This is situated in the middle of the board. The blue LED next to the button is used to indicate the state of the cars' LEDs. Adjacent to this, is another button which is used to control the piezo speaker found on the car. By pressing and holding down this button, a sound, similar to that of an actual car's horn, will be played. An infrared sensor situated in the front of the vehicle is used to detect any obstacles in the vehicle's path. Next to this sensor is an arm, comprised of two servos and a body. If the sensor, senses any obstacles in the vehicle's path, the arm will be activated to clear the path. This process is automated and requires no user interference.
We have colour-coded the wires on our circuits to make them easier to interpret. Red wires connect to power, and black wires connect to ground. On the controller, white wires connect to the drive mode buttons, orange buttons connect to the potentiometer, brown wires connect to the LEDs, green wires connect to the *torch* and *horn* buttons, and yellow wires connect the *left* and *right* buttons. On the vehicle, orange wires connect to servos, green wires connect to the right motor, yellow wires connect to the left motor, blue wires connect to the LEDs and piezo speaker, and white wires connect to the infrared (IR) sensor. Finally, it should be mentioned that we are using an L293D motor driver, to control the vehicle's motors.

Implementation

Communication

We based our communication protocol on the Master/Slave model. More specifically, we utilized the *wire* library, which uses the I2C protocol. The controller acts as the master, and the vehicle acts as the slave. In every loop of the master program, it sends exactly 6 bytes within a transmission. Each byte represents one command, and they are always sent in the same order. This means when the slave receives a transmission, it knows exactly which byte corresponds to which command. To process the transmission, the slave goes through each byte in order of transmission, reads the value of the byte, and coordinates the appropriate response. Some commands are set based on the current input (e.g. if a button is being held down), whereas others are variables which are manipulated when an interrupt routine occurs.
The master also requests 1 byte from the slave in every loop. This byte represents the data from the IR sensor on the vehicle. The master processes this data and uses it to coordinate a command to control the arm, if necessary.
To physically implement this, we have used 3 wires. One wire connects the ground pins of the Arduinos, and the other two connect the SCL and SDA pins of the Arduinos to each other, as shown in the schematic.
Pros: Sending the same number of bytes each time makes the transmission easier to read for the slave. It also means it does not matter if different bytes have the same value, because they will be interpreted based on their position in the transmission.
Cons: This is not the most data-efficient solution, because commands to, for example, turn the torch on are sent in every loop, even if the torch is already on. Our design could be changed to only send commands when needed.

Movement

To control the drive mode of the vehicle, the user interacts with three buttons on the controller. The Arduino Uno, allows for just two pins to be used for interrupt service routines. To work around this, the three buttons are connected to a single interrupt pin, as well as their own individual analogue pins. This is achieved by connecting all three buttons to a single wire, which leads to the interrupt pin, with the help of diodes. Diodes are used to ensure the current will not flow back through any of the buttons. This implementation was chosen over using system time, given that it adds to the easier comprehension and following of the code. Whenever one of the buttons is pressed the *driveModeInterrupt* function is called. In this function, the button that was pressed is identified, by reading each of the buttons' states. The function then sets the value of *driveModeCommand* - a byte - to the appropriate value. 'F' is used to signify *forwards*, 'N' for *neutral*, and 'B' for *backwards*. This variable is sent to the vehicle every transmission. *driveModeCommand* is a volatile variable, because it is used in an interrupt routine.
Pros:Makes code comprehension easier. All three buttons can be used to access the interrupt routine, even though only one interrupt pin is used.
Cons: Requires additional hardware resources, namely; three diodes and an additional pin on the Arduino.
To turn the vehicle *left* or *right*, the user must make use of the *left* and *right* buttons situated on the right side of the controller. To achieve a turn in either direction, the corresponding *turn* button has to be pressed and held. The *getDriveDirectionCommand* function, will then get the

reading of the two buttons, identify which - if either - is pressed and return a character value to be transmitted over the I2C bus. The values this function returns are 'R' if the *right* button is being pressed, 'L' for the *left* one and 'S' - *straight* - if neither are pressed.
<u>Pros:</u> The implementation is hardware and software efficient - it does not require many hardware components and the software used for the reading and transmission of their values is easily comprehended. This implementation is easily extendable - newer versions of the system can be implemented with little change to this protocol.
<u>Cons:</u> Differ from conventional ways of controlling vehicles - making controlling the vehicle more complex
The speed at which the vehicle's motors are spinning is set via a potentiometer on the controller. The value of the potentiometer is read before each transmission of data to the vehicle. To do this the *getDriveSpeedCommand* function is called. In this function, the potentiometer reading, a value between 0 and 1023, is extracted through the *analogRead* function. This value is then mapped, between 0 and 255, to take a value suitable to be transmitted through the I2C bus. Finally, the suitable value is transmitted to the vehicle in the form of a byte.
When a transmission takes place, the slave is aware which bytes correspond to the *driveMode*, *driveSpeed*, and *driveDirection*. It uses these variables to set the *driveSpeed* the motors are operating at through the *editTurnMultiplier* function. In this function the *turnMultipliers* variables are updated. These variables are used to adjust the speed of the corresponding motor to achieve a turn. Finally, the appropriate commands are sent to the motors, through the *forward, neutral* and *backward* functions. A motor driver is used to allow for the motors to spin *forwards* and *backwards*.

Torch Control

The torch (LEDs) on the vehicle can be turned on and off by tapping the torch button on the controller. This button is connected to an interrupt pin, and uses an interrupt routine to detect when the button is pressed down. The interrupt routine calls the *torchInterrupt* function, which changes the torch command variable to represent the opposite of what it currently represents. This variable is volatile because it is accessed in an interrupt function. The command byte 'T' signals the car to turn the torch on, whereas the command 't' signals the car to turn the torch off. Every time the button is pressed, the variable switches between 'T' and 't', which turns the torch on and off.
Additionally, there is an LED on the controller that indicates the state of the torch. To control this, the master program reads the current state of the torch command, and turns this LED on/off.
When the slave reads this command, if it reads 'T' it activates both LEDs using digital pins, and if it reads 't' it turns both LEDs off. It controls these LEDs by storing their pin numbers in an array. When manipulating the LEDs, it loops through every pin in this array, and sets the output accordingly.
<u>Pros:</u> Storing the LED pin numbers in an array makes the code more maintainable and adaptable. It makes the addition of more LEDs easier.
<u>Cons:</u> Using an array to store LEDs means individual LEDs can not be controlled as easily. Although the pin can be accessed through the array, manipulating items in an array is less readable than having a variable for each.

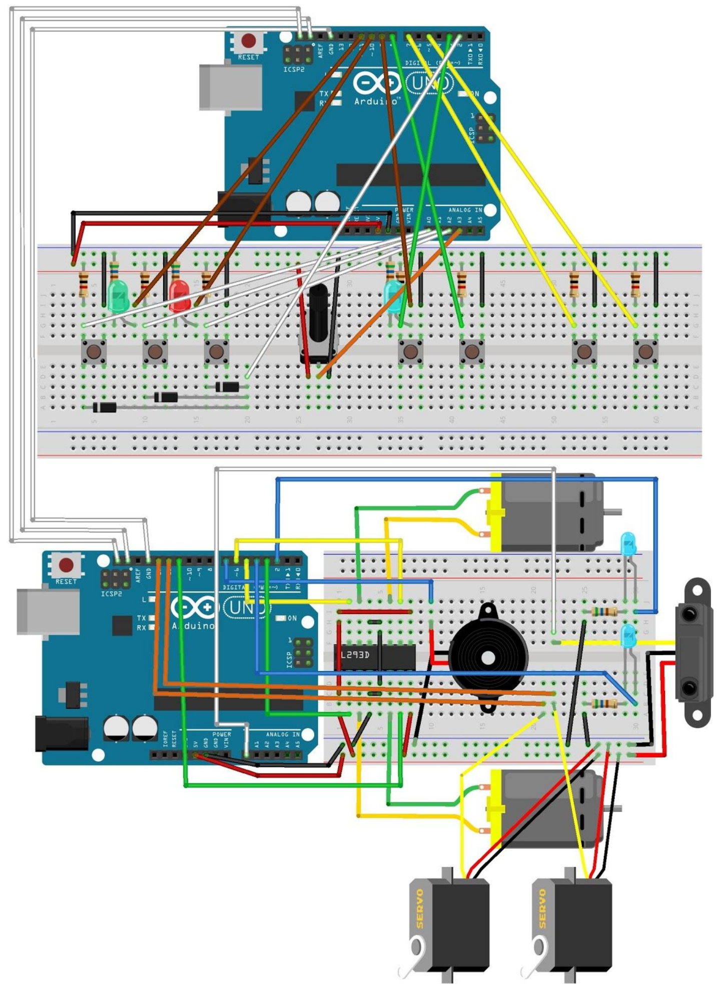
Horn Control

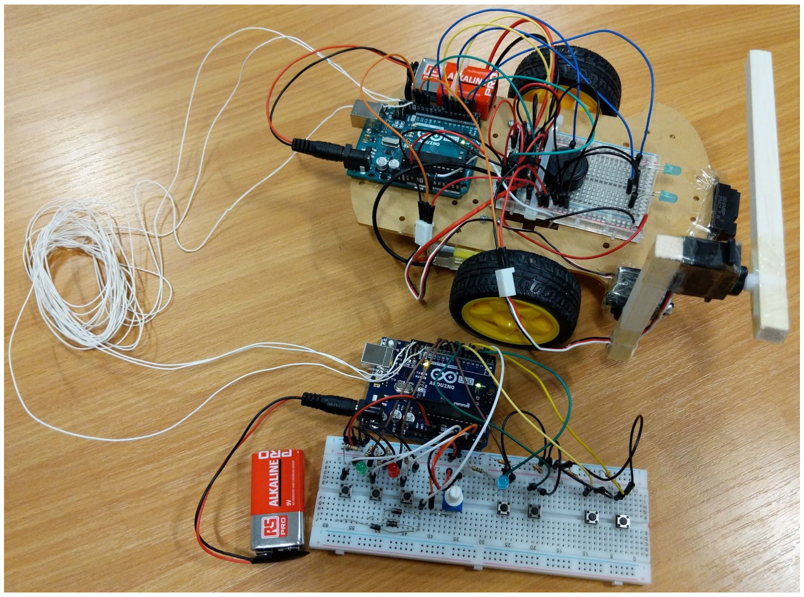
Similarly to the aforementioned *left* and *right* buttons, the *horn* button must be held down to activate the vehicle's piezo speaker. Before the *horn* command is sent, the master program reads the current state of the button, using the *digitalRead* function. If the button is being held down, the command byte 'H' is sent, while if it is not, the command 'h' is sent instead.
The vehicle controls the speaker via a digital pin. If the received *hornCommand* is 'H', it activates the piezo speaker using the *tone* function. While if the command reads 'h', any sound currently played by the piezo is stopped, using the *noTone* function.
<u>Pros:</u> The controlling of the horn, mimics the control of an actual car horn. (Pressing and holding down the button)
<u>Cons:</u> This could make controlling the car more complicated - for example holding the horn button whilst moving is difficult.

Sensor and Arm Control

An infrared sensor is situated on the front face of the vehicle. This sensor is used to calculate the distance between an object and the car itself. The value of the sensor is extracted through the *analogRead* function. This reading is a value ranging from 0 to 700. To make it suitable to be transmitted over the I2C bus, the reading is first mapped to a value from 0 to 255. Finally, this value is received by the controller through the use of the *Wire.requestFrom* function. The master then processes this information and sends the appropriate response to the slave board. If three consecutive readings of the distance sensor are above a given range, the *armCommand* - the command used to control the servos on the vehicle - is set to 'A' signifying the activation of the *arm*. Otherwise, an 'a' is sent, allowing the servos to return to their *passive* values. Once the *armCommand* is set, it is transmitted over the I2C bus, back to the vehicle, where it is processed accordingly. Three consecutive readings are needed to account for any errors in the sensor's readings. The *arm* consists of two servos attached to a wooden body. To activate the *arm*'s movement the servo controlling the vertical position of the *arm*, lowers the *arm* until it is parallel to the ground. Then the servo responsible for the horizontal position of the *arm* does a 'sweeping' motion to remove any obstacles in the vehicle's path. Once this is done, both servos return to their *passive* states. The servos' values are changed using the *write* function of the *Servo.h* library.
<u>Pros:</u> This implementation is robust, given it accounts for any errors in the infrared sensor's readings, does not require any user interaction with the system - everything is done automatically.
<u>Cons:</u> Having to get multiple readings before the *arm* is activated, leads to a slower reaction time - when compared to only having one reading.

Appendix





References

Arduino, 2019. *Arduino: Reference: Language: Analog IO: Analogwrite* [Online]. Available from: https://www.arduino.cc/reference/en/language/functions/analog-io/analogwrite/ [Accessed 30 November 2019].

Arduino, 2019. *Arduino: Reference: Language: Functions: Advance IO: Tone* [Online]. Available from: https://www.arduino.cc/reference/en/language/functions/advanced-io/tone/ [Accessed 1 December 2019].

Arduino, 2019. *Arduino: Reference: Language: Functions: External Interrupts: AttachInterrupt* [Online]. Available from:https://www.arduino.cc/reference/en/language/functions/external- interrupts/attachinterrupt/ [Accessed 29 November 2019].

Arduino, 2019. *Arduino: Reference: Servo library* [Online]. Available from: https://www.arduino.cc/en/Reference/Servo [Accessed 2 December 2019].

Arduino, 2019. *Arduino: Reference: Wire Library* [Online]. Available from: https://www.arduino.cc/en/reference/wire [Accessed 29 November 2019].