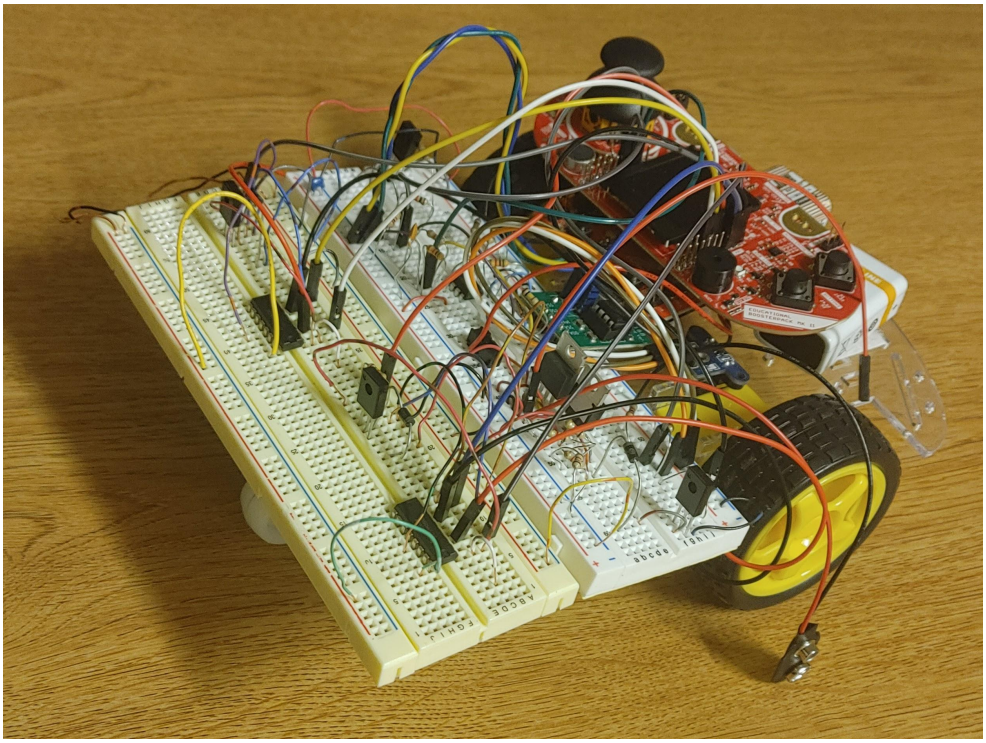
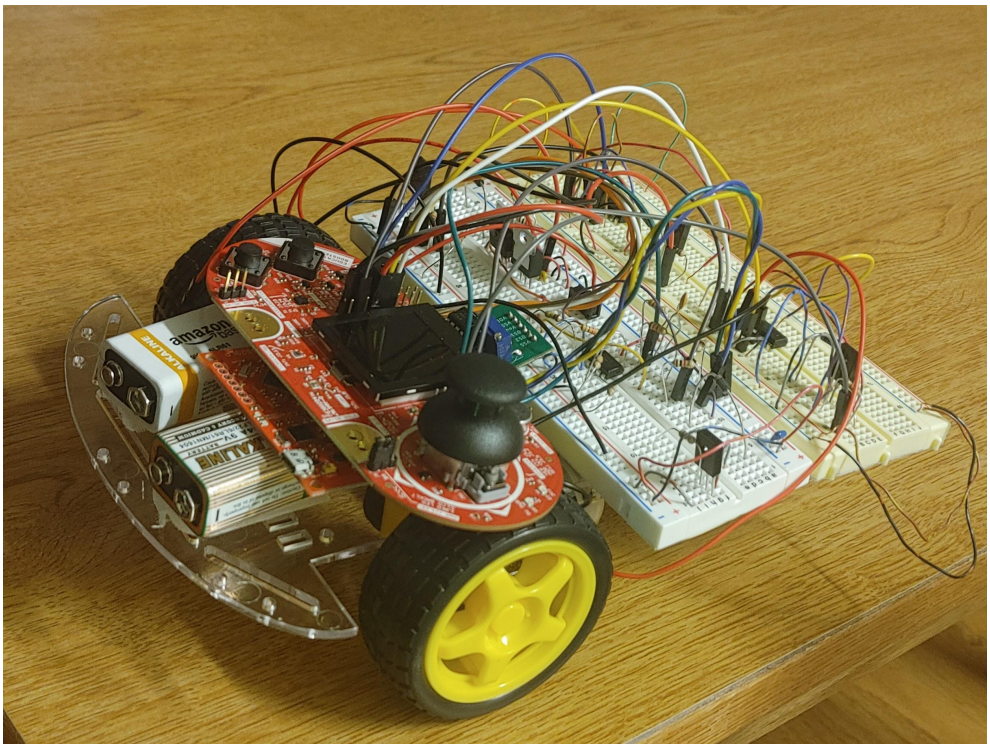
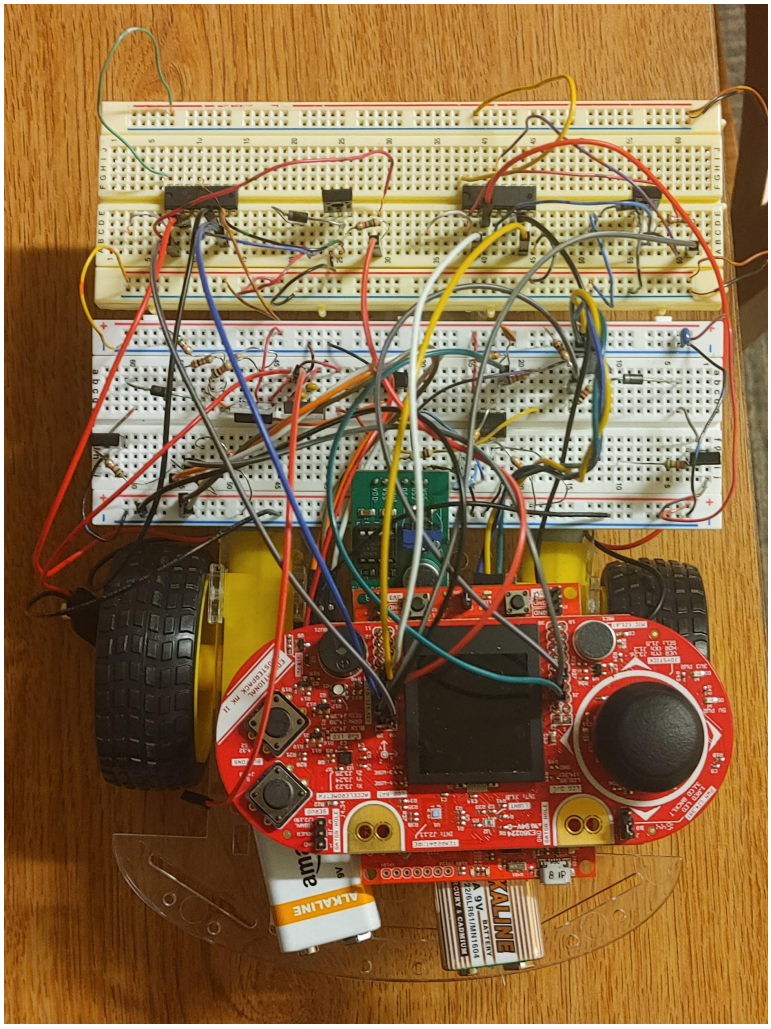
The !RC S1XT33N Car

Nikhil Jain, Ryan Ma, Andrew Huang







O' 16B car,
Reverse thy motor spin
And defy our will.

– Some 16B student, probably

Introduction

RC cars are cool, but how about not(!) RC-ing them? That's even cooler (if there's other stuff to make up for it, I guess). Given that the original car was controlled by voice commands and a thorough implementation of classification, we wanted to use the resources given to us to expand upon the original functionality of the car. While we are still working on perfecting the S1XT33N car for the perfect track, we found the screen, joystick, and the mechanical functions of the car particularly interesting for modification for the categorical track. Our car features two new functionalities. One of the functionalities is a new input type, being the ability to control the car with a joystick and displaying the car's direction on the screen. The other functionality is a

more experimental, proof-of-concept functionality, being the extra movement capability of the wheels/motors to be run in both directions depending on the input.

Perfect Car Tuning

Our first challenge was to consider how we could tune our car to be as perfect as possible. As our classification was pretty solid, we decided that we would prefer to tune the car's movement. While testing on the floor, there were a few shortcomings that we identified. For example, the car was drifting off to the right, especially during the short straight drive. Because we wanted to make both the short and long drives as straight as possible, even though only the long drive is being tested, we wanted to tweak both the δ_{ss} and `STRAIGHT_RADIUS` values so that both would go straighter. In addition, we decreased the length that the car drives straight for each of those commands in order to meet the desired distance requirement.

Then, we had one other challenge to tackle: the turning. We tested multiple values for the left and right turns, but we continuously came to the same problem: either it would miss the endpoint, overshoot too far in the original orientation of the car, or the car would have turned more than 90 degrees. Indeed, there were times where it might have looked more like an attempt to turn around. When we looked closer, we noticed that, before starting to turn, it would seem to stay straight for a certain amount of time, something that we guess might be due to jolting. So, when we moved the car a couple feet behind the starting point, it seemed to turn perfectly and hit the sweet spot with the correct orientation. However, to our knowledge, we are not allowed to give the car space to wind up and get started. So, we came up with a slightly different solution: we added a quadratic term in the timestep with coefficients `decay_left` and `decay_right`. These would allow the car to slow the amount that it changes its orientation as time increases. That way, we can finally have what we want: the car to reach the bullseye without ultimately changing its orientation much more than 90 degrees.

Joystick and Screen

The joystick and screen were meant to explore another means of utilizing physical and digital systems for interacting with the car. Using the joystick (that felt oddly solid, much like a game controller) and screen that came with the TI Educational Boosterpack MKII, we attached it to our TI MSP430 Launchpad and modified our integration code so that instead of accepting voice inputs (since the PCA vectors were too large to keep voice functionality), we used the joystick to take in an input angle that would tell the car which direction to go. For this part, we primarily used the code given to us in Integration (from the 16B staff) and occasionally referenced the examples provided by TI's Energia IDE (`LCD_screen_test.ino` and `LCD_joystick.ino`) to learn how to read inputs from the joystick and display images on the LCD screen.

We had to modify the code in several ways – first, we added a global variable that held a boolean to see if the joystick was pressed. A pressed joystick indicates that we begin the "recording" of the joystick movements. In the recording state, we added an array (much like for the voice recording) that contained the joystick position data sampled over an approximate one-second period. Since these values for the joystick are set as having the upper left corner as (0, 0), we wanted to recenter the output at the center of the screen, so we did (x - 64) and (64 - y) to find the relative position to the center. We used the maximum value of (x, y) in our one second sampling period to find atan2(y, x). Using the trigonometric hackery taught to us in 16B, we classified the angle of (x, y) relative to the origin. Anything that was from $-\pi/6$ to $\pi/6$ would be right, from $\pi/3$ to $2\pi/3$ would be far straight, from $5\pi/6$ to $7\pi/6$ would be left, and from $-2\pi/3$ to $-\pi/3$ would be drive close (later changed to reverse). Anything not in these ranges would not classify, much like our usage of Euclidean norms in Classification. We also calculated the distance of (x, y) from the origin, and we decided that if the distance was less than 28 pixels from the origin, we would not classify it either as it could just be accidental knocks on the joystick, much like the loudness threshold used in Classification. Based on these angles and distances, we used the same if-statement system as in Integration to set the drive mode and start driving. The rest of the feedback control and car functionality remain the same as nothing about the car's movement was modified.

We also decided to add some functionality to the screen (because RGB on a screen looks cool, and also totally not because the screen came attached with the TI Educational Boosterpack MKII). We wanted the screen to display relevant text for the user to interface with S1XT33N. When the car is not recording or moving, we display "Press joystick to start" on a gray screen to indicate the car is in a neutral state. We modified the code here by adding a `gText` call so that we display this text to the screen and call the function `clear(darkGrayColour)`. For the state where the car registers a joystick press, but is not ready to record yet (there is a delay), we display a blue screen and the text "Not recording." We display the text "recording" and change the screen to red when it is recording (much like how our car LED turns red from the original design). Lastly, we display the direction of the joystick classification and turn the screen green.

The main roadblocks we ran into while designing this portion of the car was unfamiliarity with the coding language and the codebase already provided to us. Unfamiliarity with C was not a huge issue for all of us – we had sufficient Java knowledge and could easily read and understand the gist of the code. However, understanding some of the more complex parts of `integration.ino` regarding the timers did cause us to slow down. At first, we tried implementing the whole thing without the use of the different clocks and our actual drive times were incorrect. We assume this is due to the extra time used in calculation and the slowness of the TI MSP430 Launchpad and without assigning the timers as the provided code did, our final run times for each movement were not accurate. We just reverted back to the given code from `integration.ino` after that, added our functionality as a replacement for the listen mode, and our implementation worked. We also were not familiar with programming the screen and using functions such as `delay()` or `clear()`, so trial and error and comparing to the

example code was the main way we were able to debug and fix our code to correctly display on the screen.

Dual-Direction Motors

O' 16B car,
Run off thy rocky roof
– Some 16B student, definitely

I defy your will.
– Some S1XT33N car, maybe

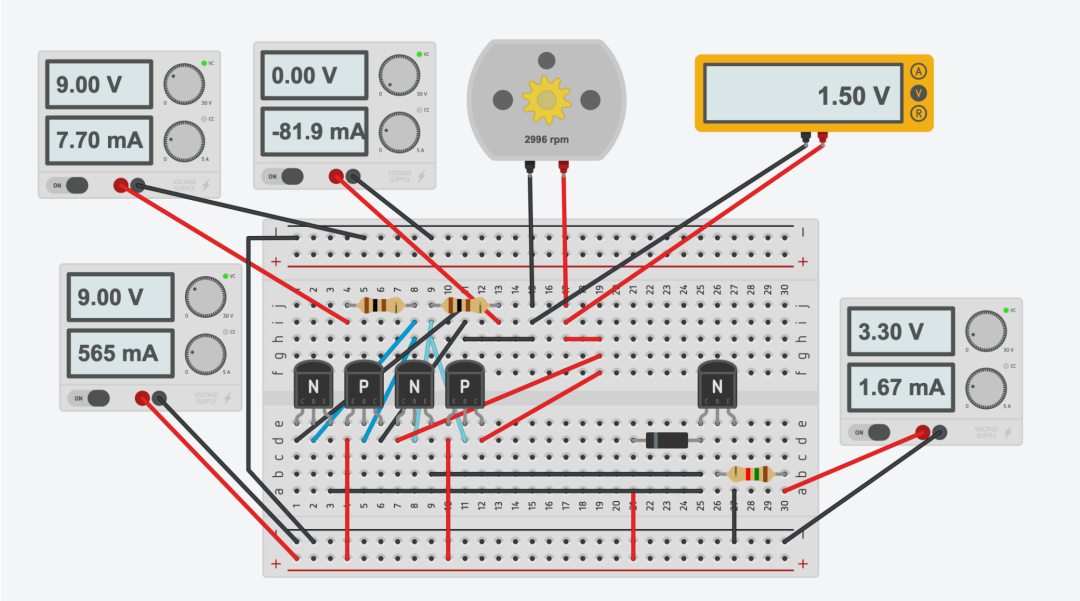
The original premise of the idea to allow our motors/wheels to run in both directions was that we noticed during one of the first labs that the motors were able to spin in opposite directions if we switched the motor's terminals in the motor circuit, showing us that the motor was capable of running with both positive and negative voltage differences across it. We realized that if our motors were independently able to run in both directions during operation, our car would be able to move both forward and backward as well as turn in place by spinning each wheel in opposite directions.

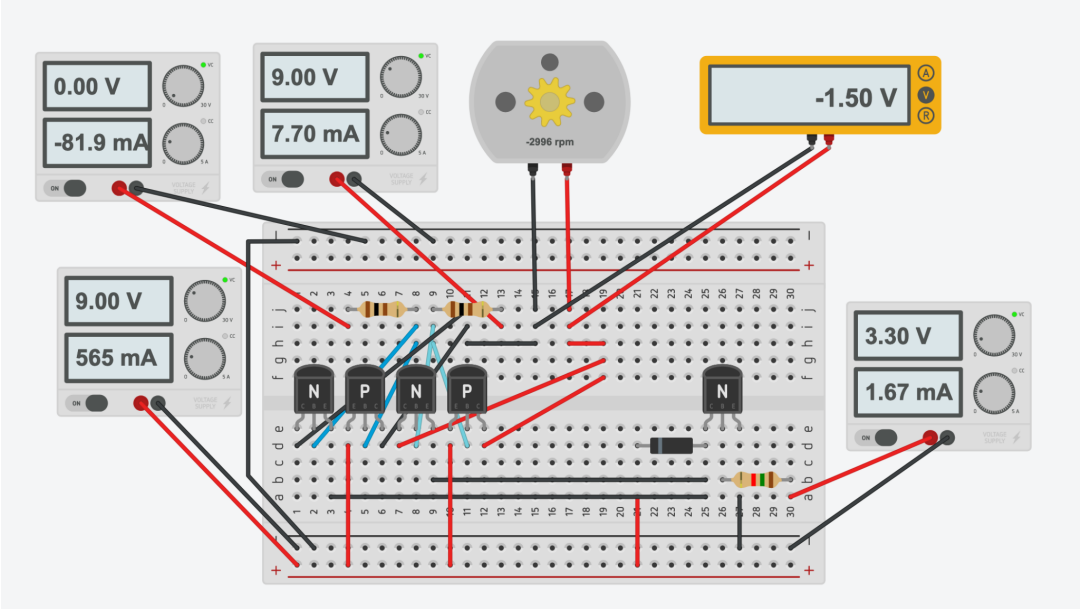
The experimental functionality of being able to move the wheels/motors in both directions involved the creation of a new circuit to reverse the inputs of the motor when desired, as determined by an input from the launchpad. The process of determining what this circuit would require was long and involved many attempts that did not succeed. We first considered if it was possible to simply input a negative voltage into the BJT base in the motor circuit, which we could do by using an inverting amplifier on the PWM input from the launchpad. However, this method does not work with the current motor circuit because the BJT, as a transistor, is only closed when a significant positive voltage is input into its base. Thus, a negative voltage input effectively acts similar to a zero voltage input and has no effect on the motor. Additionally, we tried to invert the voltages of the motor using op-amps, but because the op-amp terminals do not draw any current, the original motor circuit with the motor present as part of the circuit was not preserved and thus the circuit did not behave the same.

Thus, we looked at another potential way to *switch* the terminals of the motor, which led us to the idea of using transistors as voltage-controlled *switches*. Our first attempt with transistors involved using four small-signal NMOS transistors (since these were the only MOSFET transistors available to us, thanks Supernode!) to connect each of the terminals to both of the terminals in the motor circuit such that when an extra input voltage (different from the motor PWM signal) from the launchpad was a high voltage, the motor would have a positive voltage difference, and when the voltage from the launchpad was a low voltage, the motor would have a negative voltage difference. The advantage of this over the previous attempt was that the motor would effectively be within the circuit because the switches act close to wires when closed, which meant the behavior of the circuit should be the same as the original motor circuit,

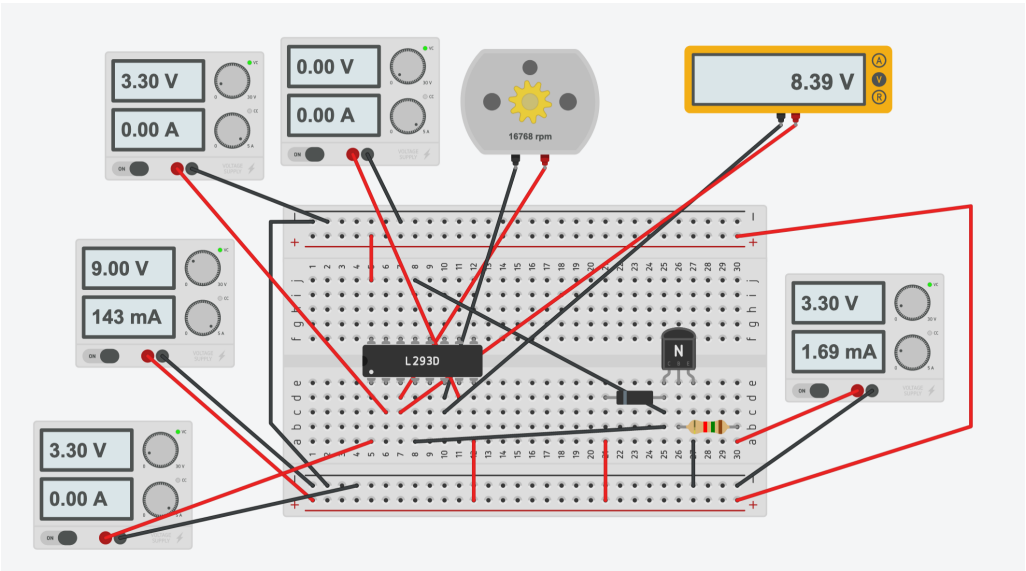
just with reversed directions. However, while an independent set of two NMOS transistor switches would work properly, the two connected systems would load each other due to the common motor in between and thus would not function properly in both directions.

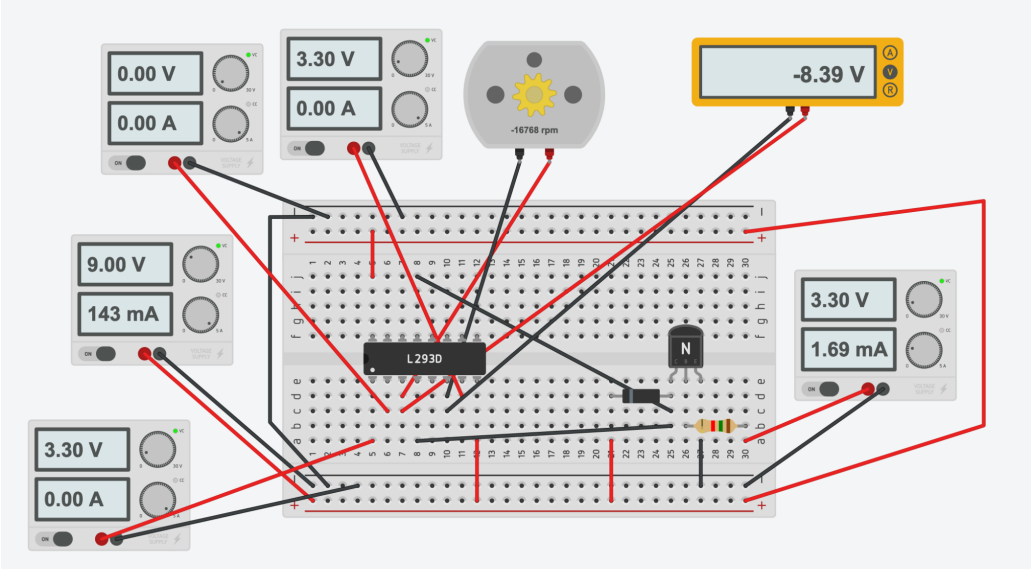
After some research, we discovered that a common circuit in robotics used for the purpose of allowing a motor to run in both directions is an H-bridge, a circuit that essentially consists of two CMOS switches (created with an NPN and PNP transistor each) operated between the 9V rail and the collector terminal of the BJT on both sides of the motor with opposite inputs at the gates of the CMOS circuits to allow for the terminals to be connected in opposite directions depending on the input settings. Our strategy was to try to manually create an H-bridge circuit and this strategy worked in that the motor would have opposite voltage differences depending on the voltage input at the transistor gates, but the NPN and PNP transistors we had access to were small signal transistors that could not handle the currents required by the motor to run properly. Thus, the strategy was ultimately not satisfactory with our current set of resources. Here are some simulation diagrams of this circuit:





However, from this experiment, we realized that an H-bridge circuit powerful enough for our motors would be a feasible solution for our problem. By using an H-bridge Integrated Circuit, we were able to effectively create the same circuit as the manual H-bridge that preserved the voltage output of our circuit with greater efficiency. Thus, we were finally able to get our motor to be able to turn in both directions with the same circuit configuration. We adjusted our code by defining delta as the difference between the absolute value of the encoder positions. Additionally, we created a new function `write_direction` that changes the launchpad input to low or high for each of the switch gates (4 gates in total, 2 for each motor because each motor has two corresponding switches) at the same time as the PWM value is input. Thus, we could allow our wheels to turn in opposite directions, which is an essential movement for turning in place. With all this, our car would be able to move forward, backwards, and turn in place without changing the circuit during operation. Here are some simulation diagrams of this circuit:





The issue is that the power for our motors through the H-bridge circuit was still inefficient to jolt on its own so a manual jolt is still necessary to overcome static friction. Through some experimentation, we determined that increasing the input voltage to the H-bridge IC increases the speed of the motor so increasing the power within the circuit could perhaps correct this inefficiency. However, to prevent from applying too much voltage to the circuit, we are operating it at voltages similar to the 9V battery voltage. Thus, this experiment shows a proof-of-concept that the S1XT33N car is capable of turning its wheels in both directions, given enough power and perhaps a more efficient circuit.

This added functionality involved several class concepts, particularly our knowledge and intuition about transistors as switches and circuit loading. Additionally, we learned about the issue of efficiency in circuits, particularly in relation to the difference between theoretical circuits and practical circuits. Learning about the H-bridge IC also involved reading and understanding the specifications of the circuit, an important skill for building complex circuits involving pre-existing IC modules. Ultimately, we were able to construct our desired result of moving the motors in both directions, even though at low speed, and learned a lot about circuit construction and design along the way.

Hardware and References

For our project, our added hardware included the TI Educational Booster Pack MKII, which was used for the joystick and screen functionality, and SN754410NE Quadruple Half-H Drivers (H-bridges), which were used to create the switch logic for the dual-direction motor.

Our code was primarily constructed on top of the code provided by the EECS 16B staff for integration. Overall, we essentially changed the listen mode to a neutral mode that, when activated by the press of the joystick, will detect the joystick direction and perform the corresponding output, displaying the current state of the program on the screen. To implement

the functions of the joystick and screen, we referenced the examples provided for the TI Educational Booster Pack MKII to understand how to measure and output the relevant values.