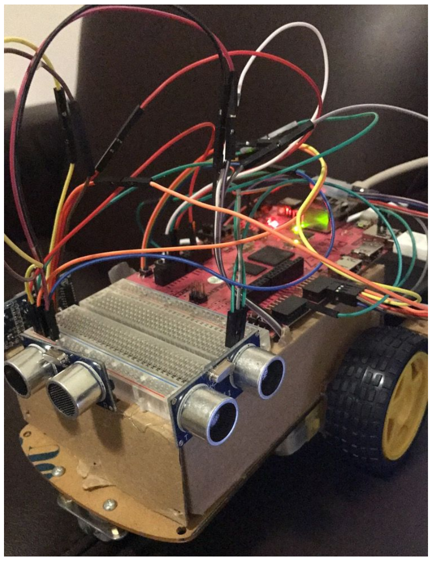
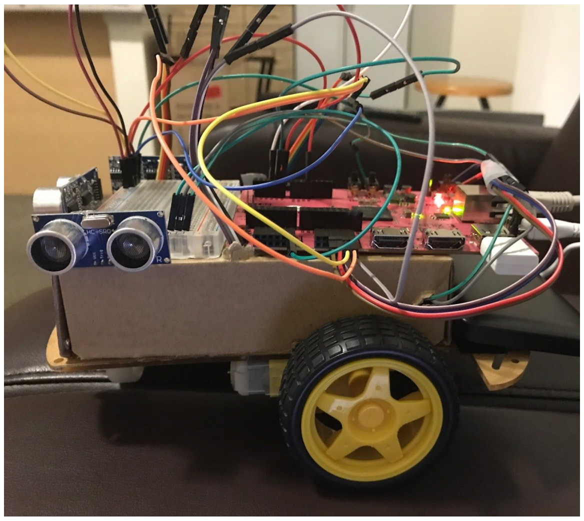
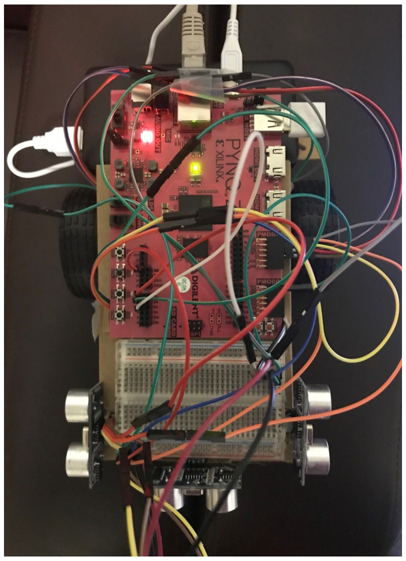
EECS 113 - Spring 2017
Prof. Pooria M. Yaghini
TA: Kasra Moazzemi

EECS 113 - Spring 2017
Processor HW/SW Interface

Final Report

Obstacle Avoiding Robot

Team 25: Boyo

Tomohiro Ohkubo - 27466929
William Floyd - 24489458

I. Goal

The goal of this project is to build a car that can successfully navigate a maze while using ultrasonic sensors for collision avoidance. This was all done using the PYNQ-Z1 board, ultrasonic sensors (for wall detection), and a car kit with motors and Dual H-Bridge Motor Driver.

II. Project Description and Features

Using the PYNQ-Z1 board, our goal is to create an autonomous maze solving vehicle that uses a recursive algorithm to keep track of its progress. Our wall detection is done with an 3 ultrasonic sensor placed facing in front, to the left, and to the right of the vehicle. The HC-SR04 sensor did not work natively with the PYNQ-Z1 board so we had to overwrite a MicroBlaze file to get it working without a 5 second delay. With our custom driver we were able to achieve performance with each sensing operation being completed in less than 0.5 seconds showing a 10 fold improvement in performance. With this improvement we were able to implement our algorithm for navigating a maze with steps of roughly 5 cm (depending on battery charge). This car as the ability to: Move forward in increments, check its sides while moving forward and adjust to make sure it will not bump into a wall, turn right if there is an object in front of it and the right is clear, turn left if there are objects in front of and to the right of the car, and correct a right or left turn if the car did not turn a full 90 degrees (due to low battery). If the car sees something close to it but not close enough to bump into it, it moves forward in smaller increments so it doesn't hit the wall. Custom logic like the aforementioned operations helps the car navigate any type of maze with eaze.

III. Procedure

Hardware:

The first thing we planned was a robot that navigated around an area and mapped out objects as it went. This goal might have been too ambitious a given our small group so we decided to just start with getting our ultrasonic Sensors to work with the PYNQ board. To start off we tried using the native PMOD_ IO and Arduino_IO functions and quickly realized that the functions far too much latency to give us accurate distance readings. The operations of the HC-SR04 are as follows. The Sensor receives a 10 microsecond pulse as an activation pulse, then the sensor emits 8, 40kHz pulses to get distance to the object in front of it (if any). After the sensor has calculated distance, it generates an echo signal back to the PYNQ board with pulse width between 150us and 25ms, corresponding to the distance from the object. As you can see in figure 1, the sensor requires precise timing as it has to measure the Echo Back pulse width with microsecond precision.

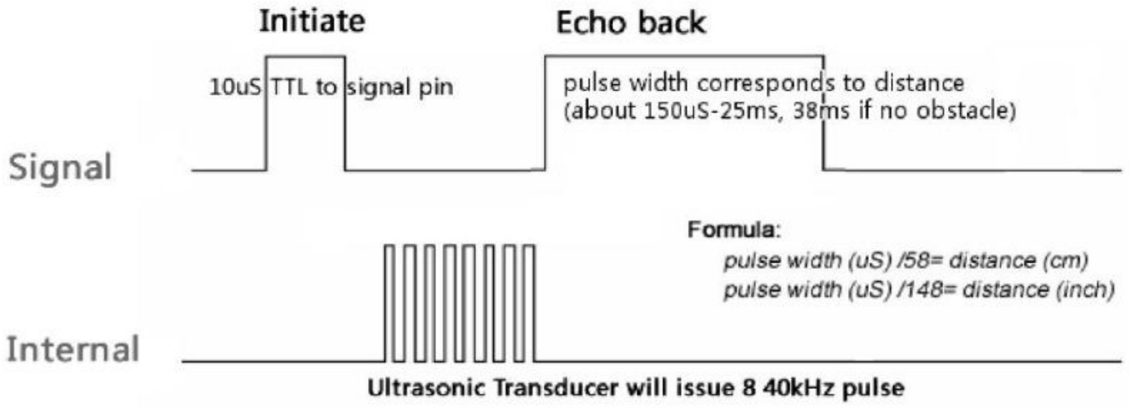

Initiate
Echo back
10uS TTL to signal pin
pulse width corresponds to distance
(about 150uS-25ms, 38ms if no obstacle)
Signal
Formula:
pulse width (uS) /58= distance (cm)
pulse width (uS) /148= distance (inch)
Internal
Ultrasonic Transducer will issue 8 40kHz pulse

Figure 1. Timing diagram of pulses in HC-SR04

The PMOD_ IO and Arduino_IO read-voltage functions were simply not fast enough to give us an accurate pulse width, so we had to try something else. Next we tried to use the PYNQ-Z1's built in trace-buffer functions to monitor a pin to get the pulse width. The trace buffer functions take voltage samples through a pin at a high frequency, then stores all of the data into a CSV file. We then made python functions to parse the CSV file to extract the pulse width out of the CSV file. This method gave us extremely accurate results but with a major drawback. The total processing time for finding the distance was somewhere between 5-6 seconds. A 5-6 second delay would not be practical for our application because this car needs to move at a reasonable speed. After this trial we realized that we had to write a Microblaze file to handle capturing the pulse width. To start the process, we used the Xilinx SDK to obtain the C file for the arduino_analog Microblaze function. After obtaining that, we edited one of the functions that wrote to the mailbox then we used the make function to generate a new arduino_analog.bin. We decided to edit and replace the read_voltage function from Arduino_analog with a custom function that measured the pulse width given to the pin. We had the custom program wait in a while loop until the signal became high, then while the signal was high, then program incremented a counter variable "cycles" , then returned "cycles" after the signal went low again. We edited an existing C file arduino_analog to avoid having to write a custom python function to match our potential new C file and bin file. This way, pulse width from our ultrasonic sensor was as simple as calling the `arduino_analog.read()` function. The modified arduino_analog's `read()` function was able to accurately measure distances within a std deviation of 0.928 and a percent error of 1.98%. This was improved upon by instead polling ten times, then taking the median of the ten polls. Our results improved greatly with standard deviation of 0.475 and a percent error 1.089%. We were able to use the PMOD_IO to set the trigger pin for the device and did not need to write a driver for that part of the HC-SR04 operation.

Next we put the Arduino car kit together using the instructions given in the kit. The only modification we had to make was with the way that the Dual H-Bridge received its pulse width modulation signal. If you are connecting the Dual H-Bridge motor drive you can declare 2 separate pulse width modulation pins and work each of the wheels completely independent of each other. The H-Bridge controls the 2 motors with a total of 6 pins. Two of the pins control the velocity at which the wheels turn, one for each wheel and then the other 4 pins are for controlling the direction of the motor's rotation(forwards or backwards), 2 for each motor. Unfortunately we found that we could only instantiate one PMOD pulse width modulation object at a time. We quickly realized that every motor operation required the wheels to move at the same time and at the same speed so we sent the same pulse width modulation signal in parallel. The hardware was now complete and we started the coding part of the project.

Software:

Mazes, by nature, are meant to be solved through the use of recursion due to its inherent ability to backtrack whenever it reaches a dead end (impasse) or the end of the maze (goal). While virtual mazes may otherwise indicate that returning a `bool` for every recursive call proves to be efficacious, our maze utilizes a map in which our obstacle-avoidance robot can free-roam while still detecting and dodging walls that it detects and returning a `None` for every recursive call on return. Due to time constraints and overly ambitious goals, especially with only two members on the team, we were not able to satisfy our original proposal of mapping out an area and marking object locations, in a grid-like fashion. We did however create an autonomous robot that avoids obstacles and detects impasses.

In order to keep a certain distance away from a wall, we implemented a `setUp()` function that gets called immediately before calling the very first navigation function. This was done in order to better align the robot parallel to a wall as to not veer off to the left or to the right. The following is the pseudo-code for our `setUp()` initializing function.

```
void setUp() {
    right = getRightDistance();
    left = getLeftDistance();
    if left < right
        hug = 'L';
    else
```

```
        hug = 'R';
    align = fixedDistance;
}
```

This aforementioned `setUp()` function allows the robot, immediately after initialization and before calling the `search()` function, to initially check its lateral position. The two UltraSonic Sensors on the sides poll the distance to the wall to the left and to the right, respectively, to determine which wall hug: if the robot is closer to the left, it will use the left wall to hug (assign the `hug` variable to 'L' for left wall (relative to itself)); otherwise, if the robot is closer to the right, it will use the right wall to hug (assign the `hug` variable to 'R' for right wall (relative to itself)). This allows the robot to keep track of its relative location by maintaining a `fixedDistance` from the wall (almost like using a handrail). We set this align variable to `38` (our own unit of distance based on pulse width in the Microblaze C code) for the entirety of the code in order to keep a safe distance that is not too close but not too far from the closest wall that it is hugging.

The pseudocode for our recursive `search()` function is as follows:

```
void search() {
    get all three sides' distances using the respective sensors
    name them front, right, left
    // forward analysis
    if front >= threshold {
        if tilted to either direction (not parallel with wall)
            leftAdjust if tilted right; rightAdjust otherwise
        if there's enough space to go forward
            take a normal step forward
        else
            take a mini step forward
        search()
    }

    // right analysis
    if right >= threshold {
        rotate 90 degrees to the right
        hug the left wall
        search()
    }
    // left analysis
    if left >= threshold {
        rotate 90 degrees to the left
        hug the right wall
        search()
    }

    If front, right, and left are all below threshold
        Make a u-turn
}
```

To explain, this algorithm utilizes a prioritized list of directions in which it polls values using the respective sensors and takes action in the following order: front, right, left.

This robot, using the front UltraSonic Sensor, checks to see if there's enough distance in the front to move forward; however, if it sees that there's only little bit of space to move, it will take a small step forward. If the robot sees that there's an object (or a wall) right in front of it, it will use the distance that was previously polled from the right sensor to analyze its surroundings on its right and make a decision accordingly.

Similarly, the right sensor will checks to see if there's enough distance to its right to rotate right and keep moving; however, if it sees that the distance to its right is less than the certain threshold, it will detect an object on the right. Although, if that same right distance is greater than the threshold, it will see it as an opening through which to pass. Then it can turn 90 degrees to the right.

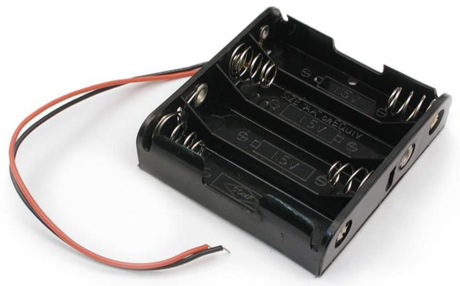
After it recognizes that the right sensor detects an object within the same threshold, it will then check to see if there's enough distance to its right; however, if it sees that the distance to its left is less than the same threshold, it will detect an object there on its left. Although, if that same left distance is greater than the threshold, it will see it as an opening through which to pass. Then it can turn 90 degrees to the left.

Then, after it checks the left and sees that there's an object there, we now know that it has come to an impasse. We would make a u-turn by making two 90-degree turns. The direction of these turns are determined by whichever side has more room; in other words, at the impasse, if there's more room to its left than to its right, it will turn left twice, but if there's more room to its right than to its left, it will turn right twice. From there, it continues its recursive `search()` function to keep avoiding obstacles.

IV. Parts List

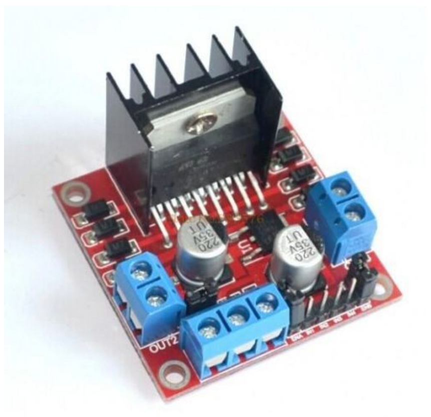
HC- SR04 Ultrasonic sensor x3

Arduino Car Kit including:
2 DC motors, Car frame

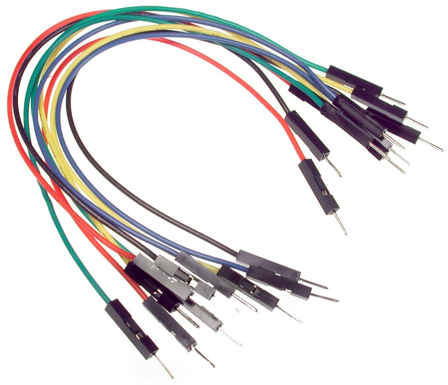


Battery pack

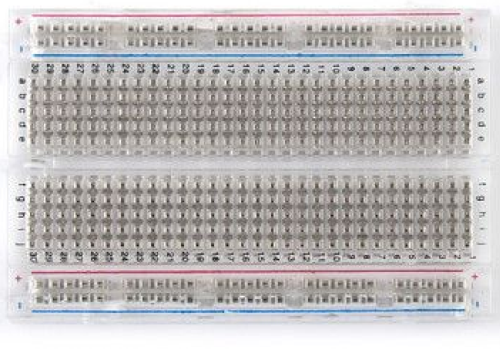
4 AA Batteries



Qunqi L298N Motor Drive Dual H-Bridge



Wires

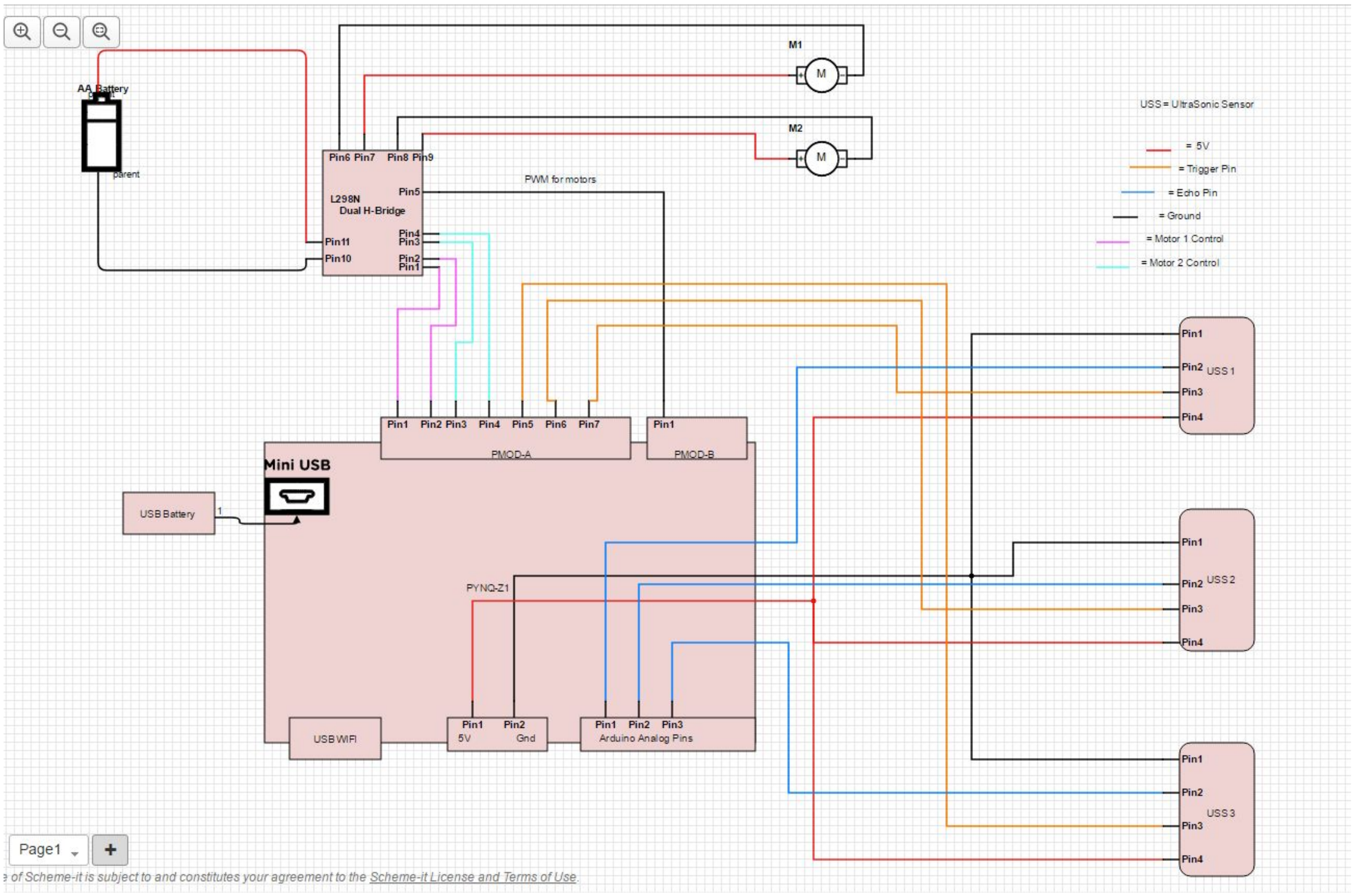


Breadboard

USB Power bank

Adafruit USB-Wifi Dongle

V. Schematic



https://drive.google.com/open?id=0B5KSCakkg68NSzY5dzhGRGdkekE (for better view)

VI. Task Delegation

Tomohiro Ohkubo (CSE) - Software (Algorithm) Specialist / Testing

- Wrote recursive functions in python to traverse and keep track of the robot's path taken within the maze.
- Tested algorithms and refined movements of robot including but not limited to: adjusting while going straight, turning right, left, and u-turn, optimized wall hugging algorithms for both right and left wall hugging

William Floyd (EE) - Sensor Implementation for HC-SR04 and hardware specialist

- Wrote Microblaze C driver for HC-Sr04 Sensor
- Assembled arduino kit and integrated all parts to work with the PYNQ including Dual- H-bridge
- Wrote a python script for Motor control for code reusability
- Designed Schematic for connecting all peripherals and power sources to vehicle
- Assisted in debugging vehicle movement

VII. Conclusion

All together this was a very interesting project where we learned a lot about the technology as well as working as a team. If we had more time with the project, one of the facets of accuracy we could have improved upon would have been to more accurately parse the data polled from the sensors. Specifically speaking, when the robot sees that the wall (or object) is at a slanted angle (of, for example, 45° with respect to the angle of the front Ultrasonic Sensor) the range of values would be very inconsistent. We also could have started to implement the maze solving functions to efficiently solve a maze or maybe optimized turning. Perhaps we could have also utilized IR sensors for its speed instead of UltraSonic Sensors. All this being said, we are content with the work that we have done, and with our completion of an obstacle avoiding car we have built something that can be expanded to make something truly extraordinary.