

Mechanical Engineering 135

Team Klesis

Christopher Le, William Tang, Jeffrey Chan, Shahil Zhangada
May 12, 2017

Introduction

1. Background
2. Motivation
3. Expected Technical Challenges
4. Experienced Technical Challenges
5. Code
6. Time Machine
7. Extra Credit



The Background

Our team is formed completely of computer science majors with no mechanical engineering experience. How did we get here?

In the fall of our senior year, the four of us knew that we still needed to meet our electrical engineering class requirement to graduate. But the class that is typically used to meet that requirements is geared toward freshmen and the CS department encouraged us to take other technical classes instead, giving us a list of other courses that would also fulfill this requirement. Being seniors, we were looking for a challenge as well as excited to learn one more new thing before we graduated and stepped out into this real world.

ME 135, Design of Microprocessor-based Mechanical Systems, was a course that stood out on this list that the CS department sent us. None of us had ever worked with hardware before beyond building apps that could run on phones, but we all really wanted to get some experience in this area and this class seemed like it would give us the opportunity to do that. At the same time, it seemed like some programming would need to be done and we knew that we had the skills to do just that. We just needed to understand the hardware portion. Little did we really understand how difficult that would be.

Working on this project, we ran into the many little quirks of hardware from wires needing to be “pushed more in” for things to work to sending too much power into a motor and frying it. While ultimately these quirks exist for logical and rational reasons, it ultimately took us a lot longer for us to solve hardware problems in our projects (and there were many) as we had never experienced these problems before in any of our previous work.

But along the way, we learned a lot, from the necessity of completing certain tasks in a set period of time and the benefits of having a system that does that to how to use a multimeter to do a basic sanity check on whether or not our wires were plugged into our sensors correctly. And ultimately, we were able to build a object avoidance vehicle robot that we’re quite proud of. As we improved our algorithms along the way and watched that robot get better and better at avoiding obstacles, we felt we were growing in the same way, getting better and better at finding hardware bugs and avoiding pitfalls both in our hardware as well as our code for that hardware.

To be completely honest, this class ended up being a lot more challenging than any of us expected or were prepared for, but we’re graduating thankful that for taking it. It taught us a lot about working with hardware and microprocessors and reminded us once again the joy of learning and making baby steps toward success.

The Motivation

For our final project, our group built an object avoidance vehicle robot.


When we originally conceived of this project, we wanted to build something that would be useful on the battlefield that could eventually replace humans. Given the gruesomeness of warfare and the advanced tactics of bloodshed that exist today, we'd figured if we could not stop the people in office from fighting, we could at least make it easier for the men and women who bear our nation's flag into battle.

Starting at Hello Universe

For the project, our original idea was a battlefield bot that could drive itself around a busy battlefield with the ability to target enemies and shoot projectiles at them while also being able to retrieve items out in the field. This robot would be able to take down enemy forces reducing the need for soldiers to actually be out there in the line of fire while also be able to perform important battlefield tasks such as retrieving a wounded soldier and bringing her back to home base. We envisioned using machine learning to be able to teach this bot to drive around a busy area with lots of obstacles while being able to target specific items and locations with its vision. It would all be controlled via voice commands that a soldier back at home base could give the robot. We had a grand vision for this robot and we set about bringing it to reality.

The Path to Hello World

As we began the work of making our battlefield bot into a reality, we ran into a number of issues. One such problem was the difficulty in being able to achieve real-time performance for our robot. We had such a high vision of being able to do image processing via machine learning to get our robot to see potential enemies to target or important objects to retrieve. But as we began working, we realized that this would be impossible to do on the PSOC. The machine learning software that we were using was too complex to be run on the PSOC. Keeping it on the PSOC would mean that the algorithm would take many seconds to run as the robot lost valuable time with no idea where to go. Adjusting on the fly to avoid obstacles would be difficult since the image processing took so long. In reality, the robot would just end up crashing into whatever was ahead of it, just going straight until it crashed without an ability to adjust fast enough. During this time, one idea we considered was sending photos taken on a camera connected to the PSOC to a laptop where the machine learning algorithms could be run with the processed data being sent back to the PSOC. Sending a photo from the PSOC to the laptop ended up taking too long as well as the photo was too large. Reducing the size of the photo meant compromising our ability to locate objects in the photo due to a major drop in photo quality so that route was out of




question. Lastly, we figured out a way to get our project to work by taking photos on a camera already connected to a laptop and then sending just the processed data about the target location to the PSOC. But ultimately, because of the complexity of an operating system such as Windows, we could not guarantee that our cycles of updates to the robot were done in real time. There was no way to guarantee that we would meet our set time to deliver results.

This major issue forced us to take a few steps back and reevaluate the complexity of our project. It did not help that beyond not being able to guarantee real-time, we were running into a host of other issues. Getting the voice commands to work was difficult because we kept running into issues with outside noise making it difficult to translate our speech to text. The machine learning libraries that we pulled in were — while powerful — rather complex and heavy and using them to seek arbitrary objects in our images proved more difficult than we thought. All of this pushed us to stop and reevaluate where we were headed and what we could really achieve.

It was at this moment that we chose to reduce the complexity and aspirations of our battlefield bot. Realizing the complexity of handling and processing vision, we decided to remove that altogether. We settled instead of keeping our sensors more simple, so we moved to using an ultrasonic sonar sensor instead that could tell us the distance between us and another object.

Using an ultrasonic sonar sensor instead of a camera whose image needed to be processed, we were able to do all of our processing on the PSOC instead of having to move data between the laptop and PSOC and doing processing on both. By doing all the processing on the PSOC itself, we could guarantee that our program was running in real-time. In regards to retrieval and projectile firing of targets, we realized that removing vision made it extremely difficult to still achieve those original parameters. Though we were sad at the moment realizing that we would no longer be able to achieve the vision we originally had (literally, we were giving up on the vision!), by reducing the objectives of what we trying to achieve, we were able to hone in on the sole objective that we had left — to avoid the obstacles in our robot's path. With an accelerometer and the ultrasonic sonar sensor as our inputs along with a renewed sense of motivation to make a robot that would still be helpful on the battlefield, we were able to — on just the PSOC alone — build a program that could make all the calculations we needed to avoid the objects in our way and plot a new path forward without any collisions.

While our final robot is not as helpful on the battlefield as we had originally envisioned, we still believe that it can help the men and women who serve. One vision we have for our object avoidance robot is that it can be the foundation to building a self-driving tank. This would allow soldiers within the vehicle to focus on the mission objective, such as shooting the enemy base, instead of also having to think about steering around on a crowded battlefield. We hope that our



research and implementation can be the beginning of automating work done on the battlefield and lead to more lives being saved.

Technical Challenges Prior to Proposal


While we were still avidly pursuing our “Hello Universe” aspirations, there were a number of technical challenges we anticipated, most of which revolved around software. At the time, we envisioned primary work and processing being done via software while our microcontroller would parse and execute in real-time based upon our software output. These challenges are summarized here and elaborated upon down below:

1. Training a machine learning model to accurately parse visual input.
2. Creating a system to translate audio input into usable strings for our program.
3. Programming our robot to physically move and adjust its movement pattern in real-time based on sensor input.
4. Executing each of these tasks within a multitasking context.

To begin, one of the challenges that we determined to amongst the most difficult and yet the most exciting was the training of a machine learning model that could take in visual input via camera and quickly identify the target object within it. At the time, we were considering either a person’s face or a large, noticeable object such as a basketball. From a military standpoint, this would be crucial in the process of recognizing a dangerous weapon, an approaching enemy, or a wounded ally. Although we knew that machine learning algorithms already existed, among the larger challenges we anticipated was finding a training dataset to build the model, synchronizing our software with a physical camera that would provide a live visual feed of the robot’s surroundings, and then running this entire sequence at a fast enough rate in tandem as our robot moved. Consequently, handling visual stimuli was certainly an issue we anticipated.

Additionally, we envisioned our robot as having the capability to take in vocal input provided by an ally. While a military context might have hundreds of different actions, we wanted our robot to be able to respond to three primary commands: stop, go, and fire. As one might expect, though, the primary difficulty here was finding a means of translating spoken words into data or strings that the robot or microcontroller could use. In particular, one of the biggest challenges we anticipated as a result of this was handling external noise. Both warfare and even an occupied lab contains enormous amounts of background noise that could interfere with our commands—an issue that we would have to handle.

Because we came into this course without any prior mechanical engineering experience at Cal, we knew that building an interface between our physical robot and the software we’d built would be no easy task, especially when programming a microcontroller that we’d never used before. Given that our machine-learning and voice translation software worked successfully, there was still the matter of feeding that to our robot and having it respond appropriately within a real-time



context. In this case, it was both the technological challenge and the lack of prior experience that most worried us.

Finally, given all of these moving parts, we anticipated that synchronization issues might arise as we tried to build each of these features into a multitasking system. Although we'd had some prior experience with multithreaded systems before, this was still no small task. Our robot would not only need to constantly take in external visual input, but also constantly be listening for any immediate vocal commands that its human controller might provide. Especially in the context of warfare, being able to appropriately handle a command such as "stop" while moving or analyzing its surroundings could mean the difference between success and failure.

Actual Technical Challenges

After pivoting from our original “Hello Universe” project to our somewhat more realistic “Hello World” vision, we were faced with an entirely new plethora of technical challenges. Some we anticipated in advance and were able to handle accordingly, while others arose during the implementation process.

Rewiring the RC Car to Take Commands from the PSOC


Upon first opening our RC car, the immediate challenge that presented itself was simply allowing the car a means of communicating with and taking commands from the PSOC. Aside from incorporating any sensor logic, our first order of business was rewiring the hardware so that we could supply voltage directly to the vehicle motors from the PSOC rather than utilizing the remote control. Although there wasn’t any C code involved in this process, our primary solution came from mounting an H-Bridge on the car that would connect to the car motors while serving as a medium through which we could pump power from the PSOC. The PSOC, in turn, received power via micro-USB from our laptops. To our delight, we were then able to manually probe different pins on the H-Bridge to stimulate left, right, forward, and reverse movement from the car.

Programming the PSOC

Rather than having to manually probe various pins in order to trigger movement, we instead faced a new challenge of programming the PSOC to control movement of the car instead. Although programming forward, reverse, left, and right movement may seem rather trivial, none of us had prior exposure to system design using microcontrollers. We began by manually implementing sequences of movements using a virtual hardware clock, before eventually moving on to software pins that would operate via commands from our C code on PSOC Creator. Through this, we were able to program a basic path for the vehicle to travel without receiving any sensor input.

Printing PSOC Output to Terminal

In order to facilitate communication between the PSoC and the computer, we had to find a way to create the UART interface so we could transmit and receive data through the PSoC. We eventually utilized a “tinyprintf” component that dealt with the printing. In order to implement such an interface, we had to wire the PSoC4 TX to the PSoC5 RX and the PSoC4 RX to the PSoC5 TX. This allowed us to transmit data through the mini-usb cable and into the COM port. Essentially, the arrays of characters (which are really just bytes) could now be sent through the



rewiring. By utilizing our “tinyprintf” module, we could now call printf and see those print statements on the terminal.

Supplying Higher Voltage to the Vehicle

Unfortunately, a separate hardware challenge we faced came in a simple form: lack of power. We realized that the current voltage we were supplying to the vehicle’s motors weren’t strong enough. Instead of making full turns, our vehicle wasn’t able to rotate its wheels and consequently crashed into any obstacles in its path. What had to be done instead was funnel power from a battery instead that could supply a higher voltage to our motors. This process required some basic soldering and rewiring.

Communicating Between the PSOC and Sonar


After wiring our sonar into our PSOC, the main challenge we faced was understanding how to program the PSOC to: 1) use the sonar to detect objects and 2) redirect the car along a new path to avoid the objects. To handle the first aspect of this challenge, we programmed the sonar to first emit an echo or signal. Our C code then begins a count that counts how much time elapses until the sonar signal returns to the trigger pin on the sonar. This reading is then used to determine if an object is in the sonar’s vicinity. The code for this process is shown under the “Sonar” heading in the “Code” section. As for the second aspect of the challenge, we implemented new logic that would direct the vehicle to turn right or left for an X amount of time before straightening. Although X was a manually set parameter, whether we turned left or right depended upon which of our two sonars was triggered (the left or right sonar). If both sonars are triggered, we then set a default turn direction of right.

Controlling Wheel Speed

The issue with adding more power to incorporate both sensors, accelerometers, and vehicle movement was that now the car was moving too quickly. The high signal that was emitted caused the motor to crank out more revolutions than we needed, making the car operate too quickly. To combat this issue, we decided to stabilize the speed of the vehicle with a pwm, where we set a period of 4000 and a compare value of 18000, which helped spread the rising clock edges out, this modulating the high signals sent to the motor. By doing this, we were able to cut the speed to a more manageable speed, where a sudden change in direction would not cause the car to flip. However, fine-tuning the period value, clock frequency, and compare value took some time before we were able to figure out that perfect balance.

Incorporating the Accelerometer

In addition to our sonar sensor, we incorporated an accelerometer to assist with speed stabilization of our vehicle. Although we were already controlling for our vehicle’s speed using



the PWM mentioned above, we desired a more rigorous means of measuring and controlling this metric. Consequently, we introduced an accelerometer to measure our X and Y acceleration. Based upon these readings, we then fine-tuned the speed at which we traveled. The code to handle this is shown under the “Accelerometer” heading in the “Code” section.

Attempting to Implement Real-Time

Although in hindsight our attempts to implement real-time within our project weren’t successful, our thought process at the time was as follows. Rather than utilizing hardware interrupts to set a hard operational deadline, we attempted to utilize delays in order to enforce this deadline. For example, if our operational deadline was to have finished calculations in increments of every 500 ms, and one such operation took 250 ms, then our delay would forcibly extend the runtime of the code by 250 ms to reach this deadline. However, looking back, we realized that because our code logically follows a variety of paths via different “if”, “else”, and logic statements, we weren’t able to properly enforce this operational deadline. Looking at our code, though, we can properly see that an ideal means of implementing real-time would have been to utilize these hardware interrupts mentioned earlier in synchronization with a timer that would count elapsed time.

Ensuring Multitasking

The two main tasks that we had to ensure operated in a multitasking context were handling sonar readings and calculating a new path of movement. Both tasks had to be executed simultaneously: lack of sonar readings at any point might result in crashing into an object while failing to adjust a path of movement would result in the car veering inconsistently. The PSOC would believe that the car was in a certain position while the physical hardware wouldn’t be responding appropriately. In addition to this, we also polled the accelerometer in the midst of these other operations as a side calculation. Whilst nontrivial, ensuring multitasking wasn’t the most challenging technical issue.

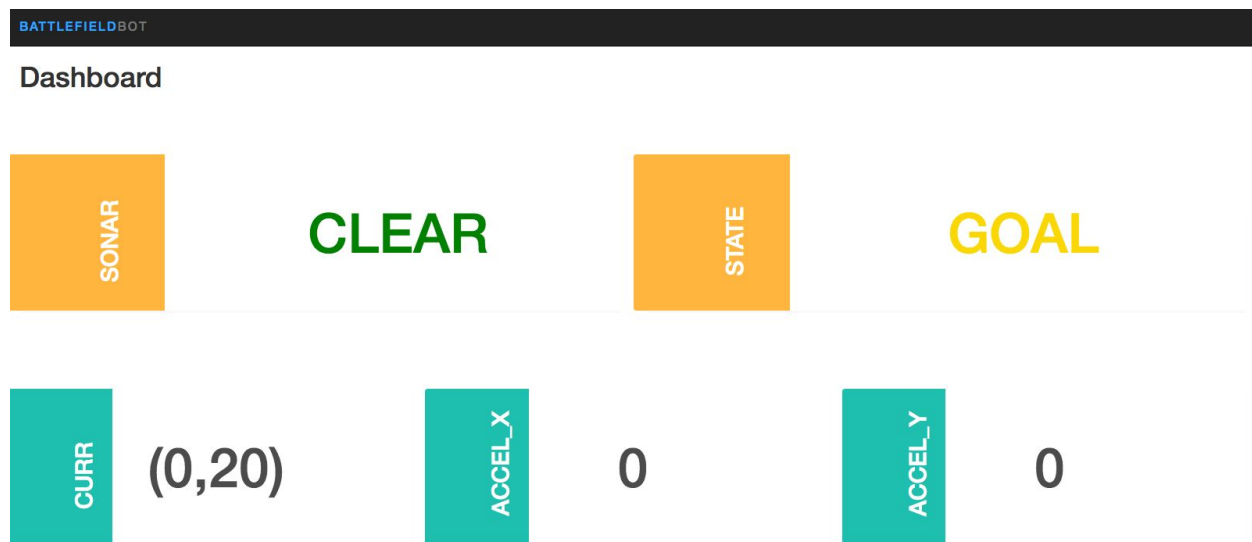
Connecting the GUI to the PSOC

The final challenge we had to deal with involved funneling the data, readings, and movement of our vehicle into a GUI. We implemented a basic web application using HTML/CSS and JavaScript. However, in order to feed this with live data from our vehicle as it moved, we had to develop a web socket framework that would transmit this data regularly. This data was then displayed via JavaScript charts and graphs in the web application. The code for this is shown under “The GUI” heading in the “Code” section.

The Code

The GUI

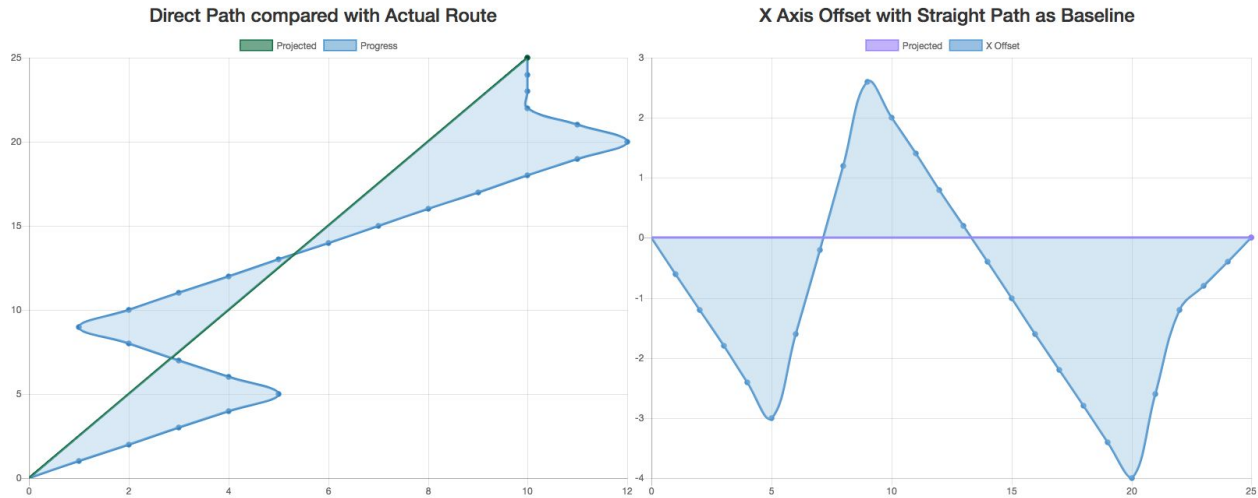
We decided to create our GUI utilizing a web server, HTML, CSS, and javascript because we wanted the ability to create awesome charts and display it in an easily digestible and aesthetic manner.



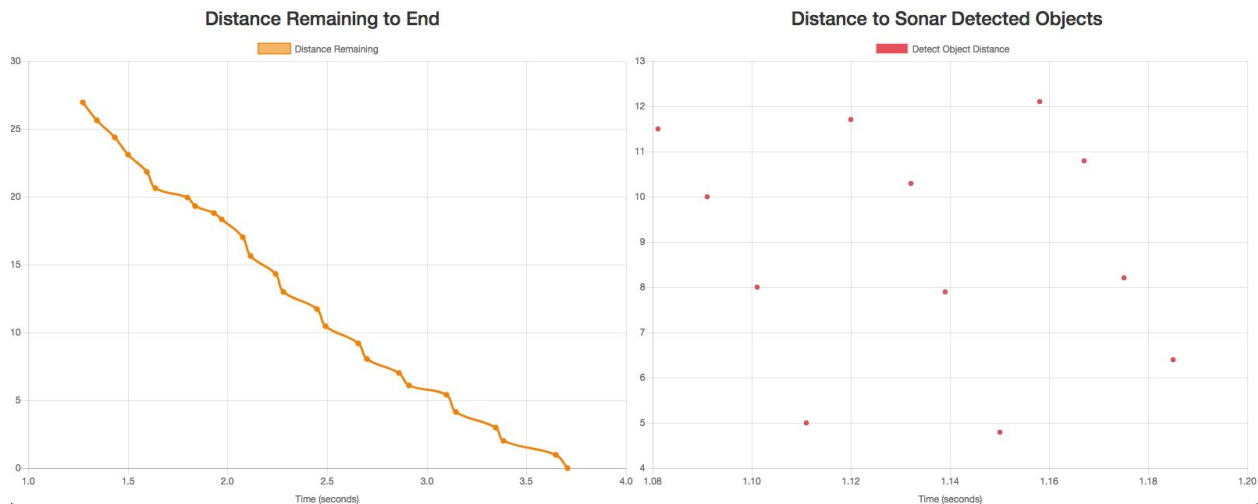
The Indicators

The top half of our GUI displays indicators for us to make sure that everything is working correctly. It displays whether or not something has been picked up by the sonars. It also shows what the current state of the robot is in. The four states are: INIT, NORMAL, AVOIDANCE, and GOAL. In the second reading, it display the output based on an internal struct of the robot's current location as well as the current readings from the accelerometer.

The Graphs



The first set of graphs show the route that the robot is taking as well as the adjustments that it is making. The graph on the left graphs the path of the robot with respect to an “ideal” straight shot path to the end destination. The graph on the right shows the adjustments that the robot needs to make in order to avoid objects. Dips under the line show that the car is currently going left while peaks above show that the car is going to the right.



The second set of graphs display distance the car from the end goal (left) and how far obstacles are away (right). The graph on the left helps us know that the state machine is working and properly allow the car to get closer to the end goal. The graph on the right allows us to see that the sonar is working properly and also helped us to fine tune the optimal range of the sensors before the car would initiate the turn (in centimeters).

How It Works

It took a long time to build but the framework is built upon several web sockets that are constantly listening for updates in order to adapt the indicators live or render the new points on the graphs. A look at the web app is shown below:

```
@app.route("/")
def chart():
    return render_template('index.html', async_mode=socketio.async_mode)

@socketio.on('connect', namespace='/gui')
def gui_connect():
    print 'connected gui'

#####

@socketio.on('connect', namespace='/client')
def client_connect():
    print 'connected client'

#####

@socketio.on('sonar_dist', namespace='/client')
def sonar_dist(msg):
    print 'sonar command received by server'
    send_sonar_dist(msg)

def send_sonar_dist(msg):
    print msg
    socketio.emit('sonar_dist', msg, namespace='/gui')

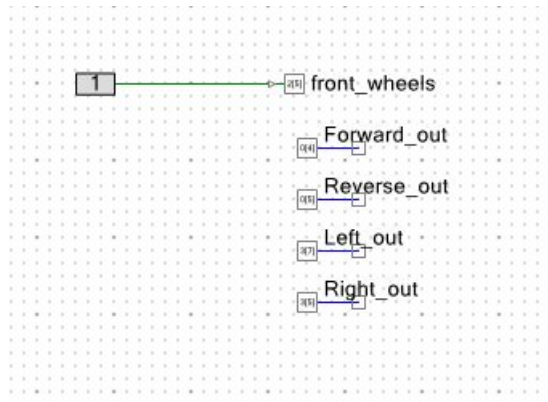
#####
```

Upon opening the localhost, a default index.html will be created. Then as data is emitted from the PSOC, this app will then transfer the data over to the websocket to the front end. The front end will also be listening:

```
socket.on('state', function(msg) {
    console.log(msg.data);
    new_state = msg.data.toUpperCase();
    $('#state').text(new_state);
    if (new_state != "AVOIDANCE") {
        $('#sonar_dist').text("CLEAR").css("color", "green");
    }
    if (new_state == "INIT") {
        $('#state').css("color", "blue");
    } else if (new_state == "NORMAL") {
        $('#state').css("color", "green");
    } else if (new_state == "GOAL") {
        $('#state').css("color", "#FFD700");
    } else {
        $('#state').css("color", "red");
    }
});
```

It will listen and update according to the key that is given. In the example above, it will update the “State” of the GUI and also change the color of the font depending on the state.

The Top Design

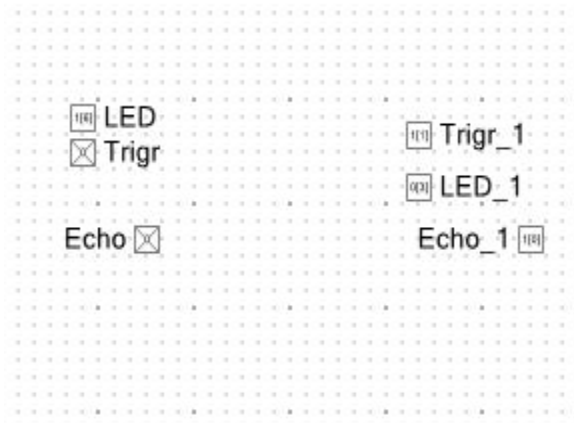
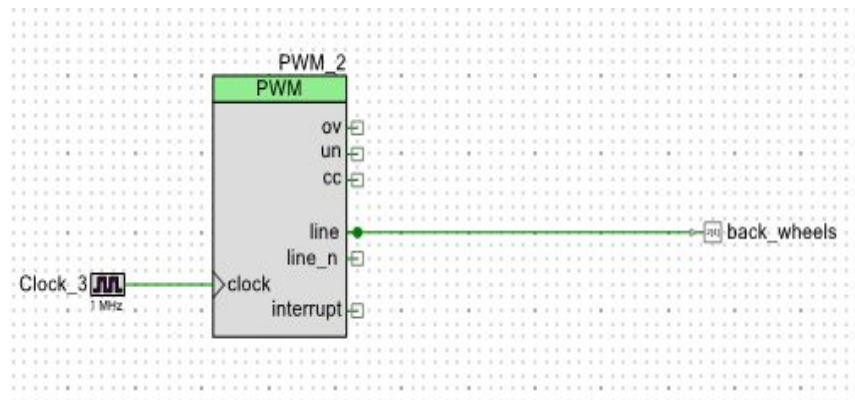


The Wheels

Although there is a PWM controlling the power output to the back wheel, which allows us to control the speed of the car, we decided to create a high hardware connection to the front wheels to enable maximum turning radius. Forward_out, Reverse_out, Left_out, and Right_out are software enable and allow us to control the movements of the car.

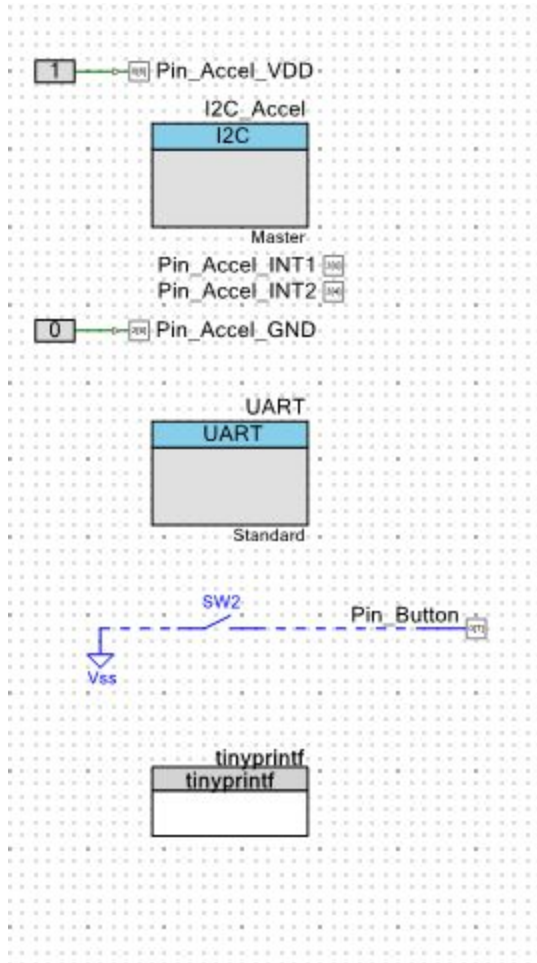
Back Wheels

The back wheels power the speed of the car. This PWM allows us to control the power given to the back wheels and therefore control the speed via software.



The Sonars

These components are the schematics needed for the sonar. The Trigr components start the sonars while the Echo components detect the distance of just how far an object is away. All of these are software controlled and the LED components allow us to see whether it is the left sonar (BLUE) or the right sonar (RED) that detected the obstacle. The components with "1" represent the right sonar.



The Accelerometer, UART, and Print modules

The accelerometer used I2C protocol to communicate to the PSOC and allowed us to get readings. The UART allowed us to print from the PSOC Creator to a terminal which allowed us to both debug and to transfer data to our GUI via a livestream. The tinyprintf is a submodule that is an imported component and handles the printing aspects of our project.

Sensor Logic

The Sonars

```
int range = 2200;

int detectObstacle()
{
    int count = 0;
    Trigr_Write(1);
    CyDelayUs(10);
    Trigr_Write(0);
    while(Echo_Read() == 0)
    {
    }
    while(Echo_Read() == 1)
    {
        count++;
        if (count > range)
        {
            LED_Write(1);
            return 0;
        }
    }
    printf("sonar_dist : %d\n", count);
    if(count <= range)
    {
        LED_Write(0);
        return 1;
    }
    return 0;
}
```

The majority of the logic for the 2 sonar sensors came from the documentation from the manufacturer. The sensors kick off by receiving a signal from Trigr for 10 nanoseconds. Then, it will subsequently begin to read in echo values and begin a count. The more echos it reads, the closer the object is. The documentation listed the count as being approximately 65 per centimeter but we found that it was closer to 73. Therefore, we fine-tuned a range of $2200 / 73 = 11.86$ cm or about 1 foot of range to be optimal given our constrained turn radius.

The Accelerometer

```
void getAccel(Robot_State_Struct *robot, int I2C_Status)
{
    uint8 accelStatus;
    uint8 accelX_MSB, accelX_LSB, accelY_MSB, accelY_LSB, accelZ_MSB, accelZ_LSB;
    int16 accelX, accelY, accelZ;
    uint16 accelXAbs, accelYAbs, accelZAbs;
    uint16 accelXAbsIIR, accelYAbsIIR, accelZAbsIIR;
    Trigr_Write(0);

    /* Start the I2C register reads by writing the target register */
    I2C_Status = I2C_Accel_I2CMasterSendStart(ACCEL_I2C_ADDR, I2C_WRITE);
    I2C_Status = I2C_Accel_I2CMasterWriteByte(ACCEL_STATUS_REG);
    /* Start the register read */
    I2C_Status = I2C_Accel_I2CMasterSendRestart(ACCEL_I2C_ADDR, I2C_READ);
    /* Read out 7 bytes of status and X and Y acceleration data */
    accelStatus = I2C_Accel_I2CMasterReadByte(I2C_ACK);
    accelX_MSB = I2C_Accel_I2CMasterReadByte(I2C_ACK);
    accelX_LSB = I2C_Accel_I2CMasterReadByte(I2C_ACK);
    accelY_MSB = I2C_Accel_I2CMasterReadByte(I2C_ACK);
    accelY_LSB = I2C_Accel_I2CMasterReadByte(I2C_ACK);
    /* Send a stop condition to end the I2C transaction */
    I2C_Status = I2C_Accel_I2CMasterSendStop();

    /* Convert the acceleration MSBs and LSBs to signed ints */
    accelX = ((int16)((accelX_MSB<<8) + accelX_LSB))>>4;
    accelY = ((int16)((accelY_MSB<<8) + accelY_LSB))>>4;

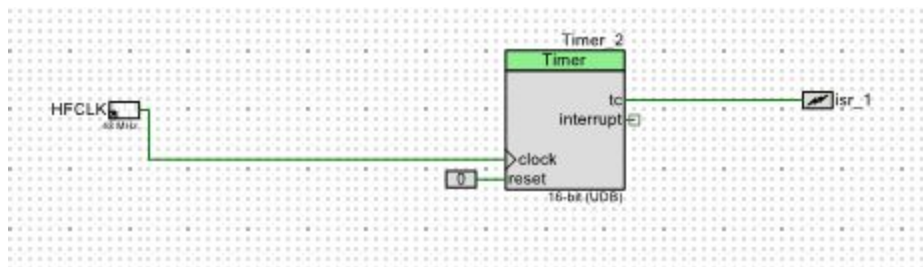
    robot->accelX = accelX;
    robot->accelY = accelY;
}
```

The motivation behind the accelerometer is that we wanted to slow the car down given the distance to a certain object and the current speed of the car. We found several tutorials online about how to do this given the X and Y readings from our accelerometer. If we took the square root of the squared sum of the X and Y over a given time, we could determine the change in distance of the car and therefore determine a rough estimate of its velocity. If an obstacle is detected, we take the estimated velocity and cut it down by controlling the back wheels via the PWM.

Real-Time Improvements and Multitasking

Current

Our current real time implementation depended on the state machine and how long we knew that longest branch would take. Then we could guarantee when is the latest that the front wheels need to turn. This helps us in avoiding obstacles and not crashing. However, Professor Anwar pointed out during our demo that our timing aspect will change depending on which state the car is in and which one it will transfer in. He suggested an interrupt that will always occur at a specified time no matter which state the robot is in.



Using this timer, which operates via a regular interrupt from a clock cycle, we were able to determine how long each of our processes took. Therefore, we were able to determine that we could do about 4 accelerometer readings (takes about 10 ms each) while the echos are being read from the sonars (40ms). After which, the car will either receive an instruction to either continue going straight, turn left, or turn right. These events are happening simultaneously given the readings on the clock. However, they do not necessarily guarantee the shortest time since the car could take different branches based on different situations and therefore mess up the real time component of it.

Revised

If we had more time, we would have implemented a Professor Anwar's suggestion of a global hardware interrupt to wrap our state machine logic and interrupt on a regular given clock cycle. In this case, it would not depend on which branch the robot took and would guarantee a response exactly at a given time.

Data-Transfer

```
import serial
import sys

from socketIO_client import SocketIO, BaseNamespace
import time

class ClientNamespace(BaseNamespace):

    def on_connect(self):
        print('Terminal connected')

    def on_disconnect(self):
        print('Terminal disconnected')

print("made it here")

socketIO = SocketIO('127.0.0.1', 5000)
client = socketIO.define(ClientNamespace, '/client')
client.on('connect', client.on_connect)
client.on('disconnect', client.on_disconnect)

client.emit('audio_command', {'data': 'Stop'})
time.sleep(3)
client.emit('audio_executed')
print("made it here")


com = sys.argv[1]
# Initializes the port on which we will be reading...check device manager for com number.
ser=serial.Serial(port=com, baudrate=9600,
    parity=serial.PARITY_NONE, stopbits=serial.STOPBITS_ONE,
    bytesize=serial.EIGHTBITS)

def process_line(line):
    # Assuming that data from PSOC comes in this format:
    # command : data
    print "LINE: " + line
    sys.stdout.flush()
    line_data = [x.strip() for x in line.split(":")]
    if len(line_data) == 2:
        client.emit(line_data[0], {'data': line_data[1]})

# client.emit('render', {})

line = ""
while 1:
    c = ser.read()
    # Print based off of new lines (for now), can modify to include other endings.
    # Currently checks for newline, carriage return (0x0D) and line feed (0x0A).
    if c == "\n" or c == 0x0D or c == 0x0A:
        # print line

        # Flush buffer
        # sys.stdout.flush()
        process_line(line)
        line = ""
    else:
        line += c
```



In order to simplify debugging procedures, we decided we wanted to use print statements to help identify bugs and improper sections of code. This led us to do research and figure out how to incorporate printf into our code. However, how would the transfer of data from PSoC to laptop or PC be made?

To do this, we wrote a Python script, that would read from a COM port--essentially reading the data given through UART. The main function of the script is to function like a socket waiting forever--it runs the infinite loop to wait for any data entering the COM port--when there is data present, we read the characters byte by byte and store everything into a string called "line". On the occurrence of a newline ('\n'), carriage return, or line feed characters, we return the line through a call to "print" and sys.stdout.flush() to display the data onto a terminal screen. The reason we would wait for these signal characters is because we wanted the data printed to have meaning and be organized--we wanted the data we saw to be the same as what we had intended it to be. So for example, if I wanted to print "hello world" from the PSoC, I would want to see "hello world" on the terminal screen rather than a bunch of random bytes, or all the bytes one character at a time.

With this in mind, we had a way to transfer data to the GUI. In order to do this, we incorporated flask.io and socket.io to create listening sockets that would update our webpage, which contained the gui. The code above the "process_line()" function sets up the declaration for the client class. Then when the script attempts to make a connection to the listening socket to update the GUI, we call the process_line() function on the line of data we want to send, which then does some error checking and emits the data as a tuple to the gui in json form, making it easy to extrapolate the data on the gui side.

To connect the two together, we would run the gui code first by running "python app.py," which initializes the socket to start listening for any data transfers. Then, in a different terminal, we run "python reading_com_socket.py COM5", where the "COM5" argument refers to which COM port we are trying to read from. In order to this correctly, we used the Device Manager to find which COM port number we were reading UART from and passed that into our function.

A Longing Look at the Time Machine


Looking back, there are plenty of things that we wish we could have done if we just had more time.

We definitely wish that we had not lost so many RC cars in the process of building our robot. Due to mistakes on our part (for example, soldering on the power connection wires while the power was on leading to smoke erupting from our car's onboard chip was one of our less bright ideas...), we lost a number of cars and a decent amount of money in the process. We did learn a whole lot though, but not making those mistakes would have been a top priority.

On a more serious note, one thing that we wished we had implemented was guaranteeing real-time using hardware interrupts. Stepping out of the time machine into the past, we would have implemented Professor Anwar's suggestion of using a global hardware interrupt to wrap our state machine logic and interrupting on a regular given clock cycle. In this case, it would not depend on which branch the robot took and would guarantee a response exactly at a given time.

Another thing that we wish we knew about was I2C_Stop that would reset the buffers for data being sent from the PSOC to our laptops for our GUI. While building out the connection between the PSOC and the GUI, we had a major issue with the PSOC endlessly sending the same piece of data to our laptop when it should have stopped a long while ago. As we were preparing to write this report and took a look back over our code, we found out about I2C_Stop which most likely would have solved our problem. By sending that signal when the last piece of data has been sent, we would have ended our endless data problem immediately.

One thing that we would have worked on had we had more time – or started our project earlier – was implementing priority scheduling into the multitasking element of our code. Currently, we are able to make use of time in between the sonar listening for return echos as well as time in between when the car is physically executing the new direction to run simultaneous tasks (e.g. reading in the accelerometer data at the same time the sonar data is coming in). But one way we can make this multitasking element more structured would be to bring in priority scheduling of tasks. If we implemented this, we would most likely give sending the car directions the highest priority with sonar coming in second and the accelerometer coming in third. While this would ultimately result in the same scheduling we have now where sonar tasks would run while the car executes directions and accelerometer tasks would run while the sonar is listening for the echos to return, this would bring a lot more structure in our code.



Another element that we would have finished implementing if we had more time was our simulator of the robot as a second element of our GUI. We had most of this working on the day of our demo, but ultimately our issues with the PSOC endlessly sending repeated data over to the laptop led us to table this part of the project. This simulator would have shown the robot moving on a grid on our laptops and turning left and right as our physical car doing just that, giving the viewer an overhead view of our robot's movements.

One last thing we would have definitely looked into if we could step into a time machine was adding more sensors to our car and building out more functionality to our robot. That would definitely allow us to go back to the original vision we had for this robot being incredibly powerful and useful to soldiers on the battlefield. In fact, given more time, we might have been able to figure out a way to literally bring back vision onto our robot opening the door to endless possibilities.

Extra Credit



SubVIs

```
void Truck_Stop()
{
    Forward_out_Write(0);
    Reverse_out_Write(0);
    Left_out_Write(0);
    Right_out_Write(0);
}

void Truck_Forward(Robot_State_Struct *robot)
{
    robot->currY += 1;
}

void Truck_Reverse()
{
    Forward_out_Write(0);
    Reverse_out_Write(1);
}

void Truck_Right(Robot_State_Struct *robot)
{
    robot->currX += 1;

    Left_out_Write(0);
    Right_out_Write(1);
}
```

Truck Components

We decided to modularize our code into different sections and components to improve the clarity and cleanliness of the code. By creating these helper functions, we were able to call a simple and intuitive function like “truck_forward()” and expect the truck to go forward. By creating these different layers, we had an easier time debugging, as the layers of complexity were built up from these functions.

```
void Truck_Left(Robot_State_Struct *robot)
{
    robot->currX -= 1;

    Right_out_Write(0);
    Left_out_Write(1);
}

void Truck_Straight()
{
    Right_out_Write(0);
    Left_out_Write(0);
}
```

Each function deals with correctly writing the high/low signals to the inputs and outputs of the car’s motor, causing the car to move in a certain direction or stop. Each function also updates the robot’s current location by setting the Robot_State_Struct’s currX and currY fields to maintain updated positions to the robot.


```

int avoidObs(Robot_State_Struct *robot)
{
    Truck_Forward(robot);
    Truck_Straight();

    int slow_down = sqrt(square(robot->accelX + robot->accelY));
    PWM_2_WriteCompare(PWM_2_ReadCompare() - slow_down);

    // Obstacle on right
    if (detectObstacle()) {
        Truck_Left(robot);
        return 0;
    }
    //Obstacle on left
    else if (detectObstacle2()) {
        Truck_Right(robot);
        return 0;
    }

    // No obstacle anymore
    return 1;
}

```

Similarly, we decided to break the logic of the object avoidance algorithm by creating a function `avoidObs()` to deal with the logic and implementation of object avoidance. This function basically calculates an integer value to determine how much to slow the car down by factoring in accelerometer readings, then check if there are obstacles to the left and right and if so, dodge the other way by turning in the opposite direction (i.e. if there was an object on the right, the car would turn left).

By modularizing the code in such a way, we were able to facilitate easier transitions when handing the code off to other members of our group, as well as spot bugs easier because components were so organized. By keeping the modularity of our program, we were able to isolate bugs quicker and not deal with the complexities at lower levels of abstraction.

Struct (Cluster)

```
typedef struct
{
    char state; // i for init, n for normsl, a for avoidance, g for goal
    int currX; // X of current pos
    int currY; // Y of current pos
    int destY; // Y of dest, how far to travel
    int32 accelX; // X from accelerometer
    int32 accelY; // Y from accelerometer
    Boolean goal_reached; // Achieved the goal
} Robot_State_Struct;
```

Robot_State_Struct: This struct is the centralized object of all our relevant information. The struct is used to update all important traits we are currently measuring such as the robot's distance to the destination, whether or not the robot's reached the goal, or what state the robot is currently in, etc. We incorporated different data types--strings (char), integers, and booleans to clarify type definitions within our project.

State: This field is a string that allows us to indicate which state we are currently in--the possible values are 'i' for initialization, 'n' for normal (where the robot is currently moving toward the goal), 'a' for avoidance (where the robot undergoes the object avoidance algorithm), and 'g' for goal (where the robot has reached the destination and quits the program).

CurrX: This field keeps track of the current robot's location with respect to the x-axis.

CurrY: This field keeps track of the current robot's location with respect to the y-axis.

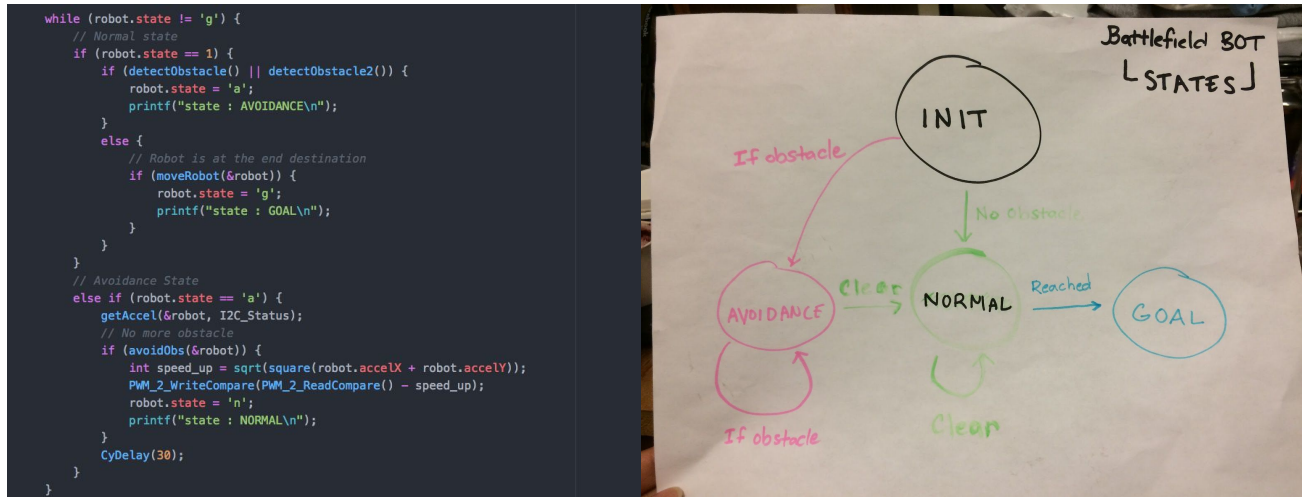
DestY: This field keeps track of the relative distance to the destination, keeping track of only displacement in the y-direction (which in this case would be distance ahead to the goal).

AccelX: This keeps track of the accelerometer reading in the x-axis.

AccelY: This keeps track of the accelerometer reading in the y-axis.

Goal_reached: This boolean helps keep track of whether or not we have reached the destination or not.

State Machine



Our state machine has four states: initialization, normal, avoidance and goal. Each of the states are outlined below:

Initialization: Power the back wheels, set the initialization values for the Robot struct, and check for an obstacle ahead

Normal: Continue to move forward, check accelerometer readings, and check for obstacles

Avoidance: Given a detected object, make a decision to go left or right depending on which sonar detected the object

Goal: Robot has reached its goal

Movements between the states can be seen as below:

Initialization starts off the movement of the robot and can send the robot to *Normal* or *Avoidance*, depending on whether there is an obstacle right in front or not. From *Normal*, the robot can change to *Avoidance* (if there is an object), stay in *Normal*, or change to *Goal* (if the goal has been reached). In *Avoidance*, the robot could stay in *Avoidance* if there is still an object detected or changed to *Normal* if there is no more obstacles. Finally, the *Goal* state can be transitioned into but once the robot reaches this state, it will stop and operation will cease.