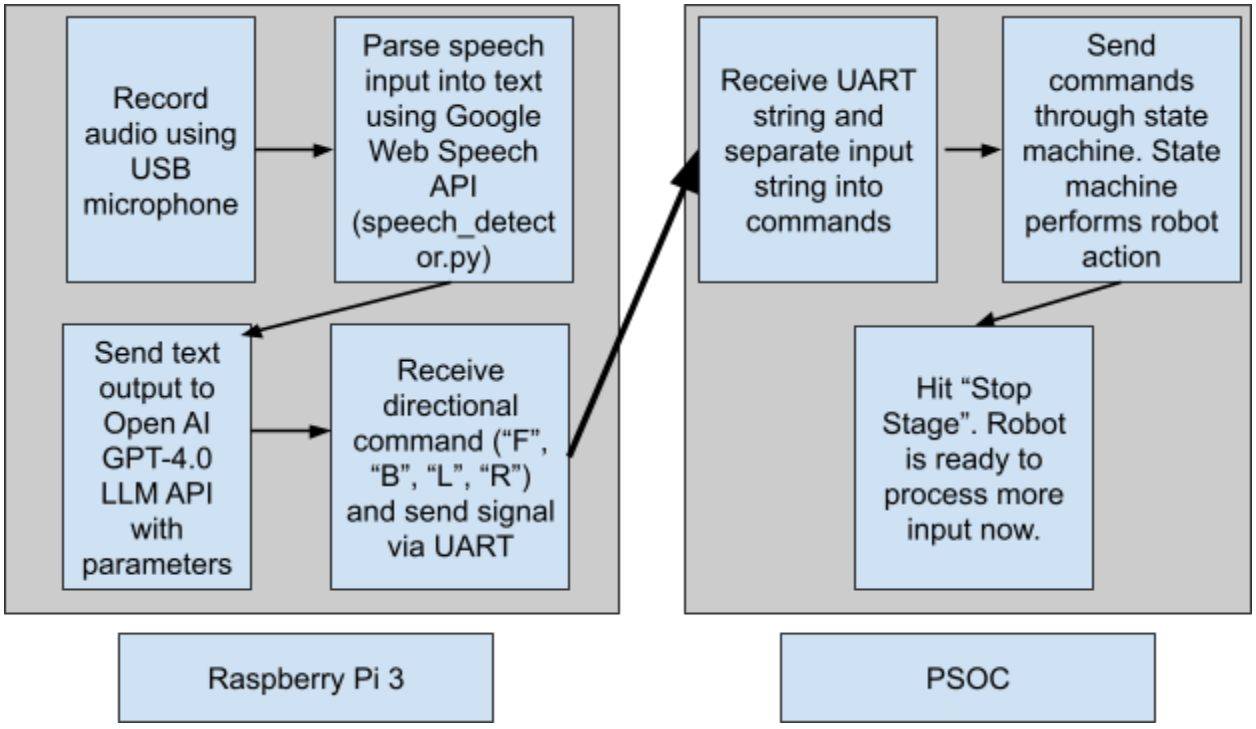
ECE 302: Final Project Report

Due: December 15th, 2023

Abhi Vellore & Ashwindev Pazhetam

Objective: To use Large Language Models (LLM) to control the mini car's motion. Incorporate a speech-to-text module to feed into the Open AI GPT-4.0 API, performed on a Raspberry Pi 3. The LLM will output a series of basic directional commands, that are then sent via UART communication to the PSOC. The PSOC then interprets the commands, and by using a state machine, determines which action to perform.

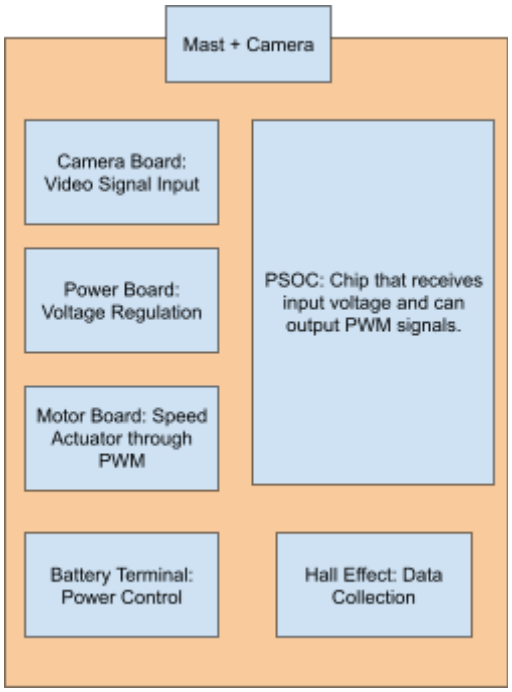
Final Project Layout:



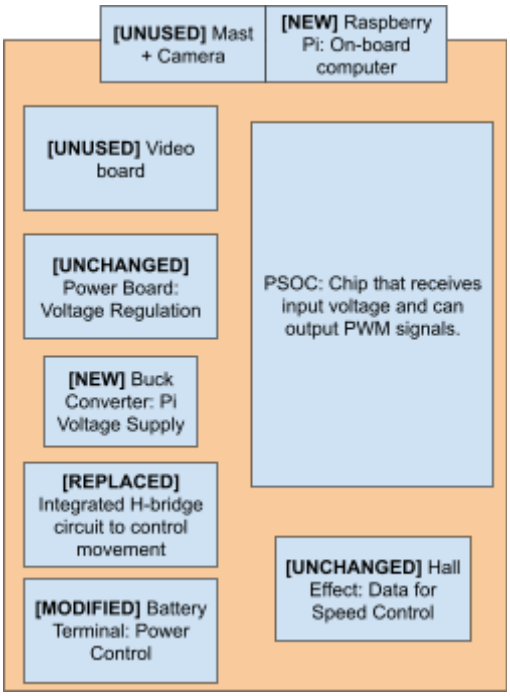
This report will first analyze the hardware required in this project before diving into software decisions.

Key Subsystems and Components (with schematics):

Original Car Layout:



Final Car Layout:



[New] Raspberry Pi 3 attached to the mast
[New] Buck Converter to power Raspberry Pi 3

1. **Components and Configuration:** When planning our robot, we quickly realized that our PSOC was not powerful enough to run API calls. We pivoted to an Arduino but quickly realized that even our Arduino could not send and receive large packets of data to API calls. Hence, we chose to use a Raspberry Pi 3, and coded on it in Python. We connected the Pi to the ECE346 wifi present in the lab to work around regulations that Eduroam/ServiceNet has for devices.

   The Pi required 5 V and at least 2.5 amps. Initially, we used a mobile power bank device to power the Pi. However, for it to function on the robot, we needed to send 5Vs from one of our batteries. Because of the higher current requirement, we could not use the existing power board's 5V regulator, as that only produced 1 A of current. So, we switched to using a buck converter, where we could take 9.6 V input and output 5 V and a current around 2.5 A. Since the Pi always needed to be on (as we were SSH'ing into it to run commands), we used a separate battery that went directly to the Pi, as using our existing 9.6 V would go through the switch in the battery terminal and constantly cause the Pi to shut on and off.

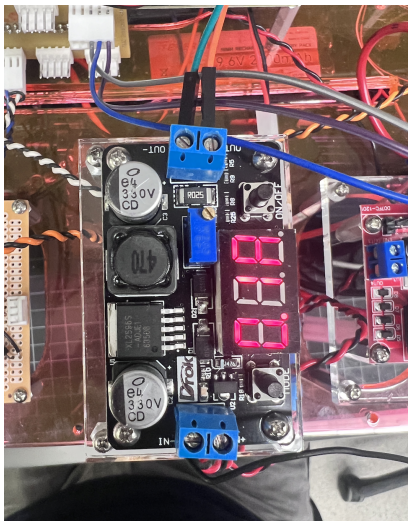
As mentioned, we eventually SSH'd (remote connected) to the Pi. Initially, however, we connected a mouse and a keyboard to the Pi via USB and then used a monitor to project the Pi onto a screen via HTMI. Once it was fully configured, we used the remote connection to make for a more seamless coding experience from our personal devices.

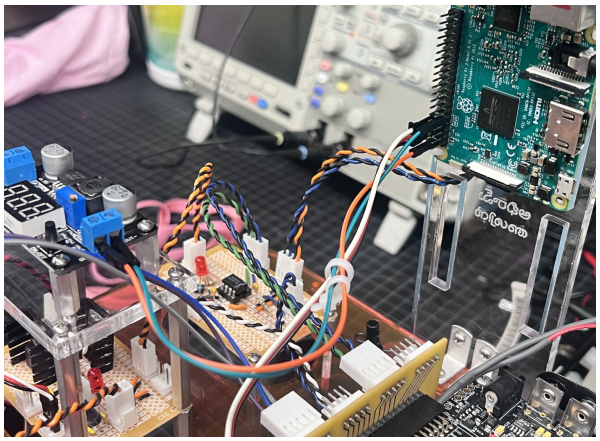
In terms of wiring, we needed to connect the Pi to three ports: 1) Power from buck converter (5 Volts); 2) Ground back to buck converter 3) TX transmitter port. The TX transmitter wire goes to the PSOC RX input terminal. As part of that, we must also ground the UART signal — otherwise, we noticed high amounts of noise that made the signal unreadable.

All of these connections were made using male-to-female or male-to-male jumper wires. Also, it's important to note that our entire new battery circuit for the Pi needs to be grounded to the ground of the rest of the robot, to ensure that there is no voltage spikes and for noise protection. Hence, we run a wire from the buck converter to our battery terminal. Finally, the eventual placement of the Pi on the mast was chosen simply for space and cleanliness constraints, as it produced very little noise.

2. **Design Choices and Challenges:** We initially purchased an Arduino Uno R2 Wifi, because based on our research, it could use the internet fully. However, we misinterpreted its power to transmit API calls, and we ran into multiple issues setting up the Arduino. Hence, we switched to a Raspberry Pi, which had a lot more power and was able to conduct both the speech-to-text and Open AI API calls with ease.

   It was also difficult to fully understand what needed to be grounded and what could cause ground loops, and we made mistakes that caused the PSOC to short out. Eventually, after discussing with professors, we were able to understand what needed to be grounded and what could cause ground loops.
3. **Improvements:** Outside of wire cleanup, we were very careful in our placement and our integration of the Pi into our overall car.

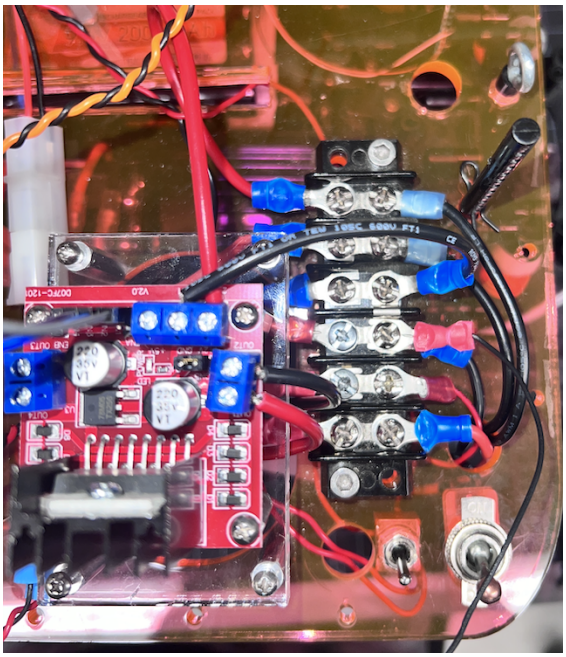




[Modified] Motorboard replaced with integrated H-bridge
[Modified] Battery Terminal altered to integrate new H-bridge

1. **Components and Configuration:** Our existing motor board only enables our robot to move in one direction, forward. However, to allow more movements in our robot, especially for it to go run continuously, we required it to also go backward. Hence, we needed an H-Bridge, an electronic circuit that allows a voltage to be applied across a load in either direction. Due to time constraints and our specific project need, we used an off-the-shelf L298 H-Bridge. The L298 contains two H-bridges, allowing it to control two DC motors independently, but we only connect to one. The bridge consists of four transistors arranged in an H-shape, which can switch the polarity of the voltage applied to the motor, thus controlling its direction. As our motors only require 7.2 Volts, we were able to use our existing 7.2 battery to power the motor — and despite limitations on the current allowed through the L298, our motor still worked fine. A “1” “0” signal would make it go forward while a “0” “1” signal would make it go backwards. It also takes in a PWM signal from the PSOC, similar to how our original motor board functioned.

   The battery terminal was rewired to incorporate the new H-bridge and ensure that the motor connections were clean.



[Unused] Video board and camera no longer used.

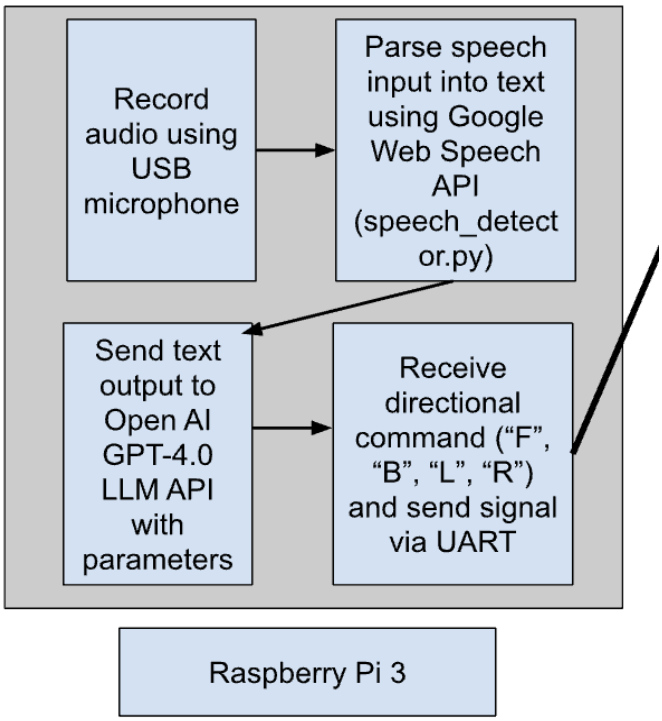
1. We decided to bucket our movements into the five categories of “forward”, “backward”, “right”, “left”, and “stop”. As these are fixed motions at certain PWM values, we did not need to use the video board or the camera anymore, as there was no “line following” feature. However, in future improvements of our robot, being able to interpret a command like “follow the line” and having the robot use the camera sensor would be a great addition. Code from navigation is commented in all appropriate parts and labeled below.

SOFTWARE

All code is uploaded to the Github repository found here:
https://github.com/apazhetam/carlab/tree/main

PSOC files are uploaded and are also part of GitHub.

Raspberry Pi



The core functionality that needs to be completed on the Raspberry Pi is to take in audio input via a USB microphone and end up with an output of directional commands. In order to ensure rigorous testing and maintain modularity in our code, we have separated our goals as follows.

1. **main.py** Central class that sets up the context of our LLM model, calls other classes and sends via UART commands to the PSOC
2. **speech_detector.py** Interpret audio signal via USB mic and convert it into a textual output using the Google Web Speech API.
3. **chat_request.py** Send text to Open AI ChatGPT-4.0 LLM, where we have a customized model, and receive back a series of directional commands.

We’ll walk through some key implementations in each of these categories.

Main.py

```
ser = serial.Serial("/dev/ttyAMA0", 115200)
```

1. To send via UART, we must configure the Pi to send a serial signal to a specific port, which is the “ttyAMA0” port. We send signals at a baud rate of 115200, which is the standard used in UART, and hence, our PSOC will eventually be set to receive it at the same rate.

```
context = "Categorize the following commands into one of these five buckets. \
1: Go forward (F). \
2: Go backwards (B). \
3: Turn left (L). \
4: Turn right (R). \
5: Stop, no movement (S). \
If unsure, default to Stop (S). \
Use either one or two commands in order to accomplish any task. \
Return a series of letters, for example 'FB'. Don't give any additional text."
```

2. Context. Open AI offers a “chat” API request, found here. The request can be fed a variety of parameters, including the specific type of LLM to be used, its randomness, and a “context.” The others will be explained later, but the context is meant to help train the LLM to provide specific types of output. Our primary goal in our project is to use the LLM to bucket commands into directional command buckets: F, B, L, R, and S. This final context was determined after a variety of tests on Open AI’s “playground,” an online feature that allows you to test the output of contexts without using any tokens (which costs money).

```
while True:
    # Detect a person's speech
    speech = ""

    try:
        speech = detect_speech()
    except KeyboardInterrupt:
        print("Program interrupted by user, exiting...")
        break
    except:
        print("Speech detection failed!")

    # Send the detected speech to ChatGPT
    if (speech != ""):
        try:
            response = chat_session.send_prompt(speech)
            print(response)  # Process the response as needed
            print()

            actions = response['choices'][0]['message']['content']
            print(f'{actions}\n')

            actions += "X"

            # ser.write(bytes(words,'utf-8'))
            for i in range(len(actions)):
                ser.write(bytes(actions[i],'utf-8'))
                time.sleep(0.1)

        except Exception as e:
            print("An error occurred:", e)

    # time.sleep(5)  # Sleep while actions are being executed
    time.sleep(0.5)
    input("Press Enter to continue...")
```

3. As we want to be able to take in commands one after another, we run a while loop that runs until the program is broken.
4. We use try/except statements to ensure we have working output and that our program does not stop working randomly. We call the detect_speech function in our speech_detector class, and only if text is saved, do we send that speech to ChatGPT. Similarly, we ensure we get an output in this try statement. We save the output of the LLM in “actions”. We also append an “X” to the end of the sequence of commands to eventually tell the PSOC when to stop searching for more commands.
5. Within the try statement, we then write the bytes of the action one at a time using the serial UART TX pin to the PSOC. There is a slight delay between each to ensure that the PSOC has time to save and interpret each command.
6. In the end, we have another short sleep time to ensure that the car has time to move before a user starts spamming new instructions. The “enter to continue” similarly ensures that the user remains in control the whole time.

Speech_detector.py

```python
# speech_detector.py
import speech_recognition as sr

def detect_speech():
    r = sr.Recognizer()

    # Select desired mic (find list of mics on conifgure_mic.py)
    mic_index = 1
    mic = sr.Microphone(device_index=mic_index)

    # Print the name of the selected microphone
    mic_list = sr.Microphone.list_microphone_names()
    selected_mic_name = mic_list[mic_index]
    print(f"Selected microphone: {selected_mic_name}")
```

1. We're using the speech_recognition module from Google, so we import the appropriate libraries. We also ensure we're selecting the correct mic, which is indexed at 1, corresponding to our USB microphone.

```python
    try:
        with mic as source:
            print("\nListening...")
            r.adjust_for_ambient_noise(source)  # Adjust for ambient noise
            audio = r.listen(source, timeout=5, phrase_time_limit=10)
            print("Done listening!")

        # Recognize speech using Google Web Speech API
        words = r.recognize_google(audio)
        print(f"You said: {words}")
        return words

    except sr.UnknownValueError:
        print("Google Speech Recognition could not understand audio")
    except sr.RequestError as e:
        print(f"Could not request results from Google Speech Recognition service; {e}")
    except KeyboardInterrupt:
        print("Program interrupted by user")
        raise
```

2. We use various methods of the Google Speech Recognition API, and write to it, listening specifically for speech. Documentation can be found here.

chat_request.py

```python
# chat_request.py
import requests
from credentials import openai_api_key

class ChatGPTSession:
    def __init__(self):
        self.session_prompt = ""  # Stores the initial context or model information

    def set_context(self, context):
        self.session_prompt = context
```

1. To access the OpenAI api, we need to create an account and import our own credentials, as is done here. OpenAI uses a token system to determine costs for using the API, with longer requests and longer outputs costing more. To minimize our costs, we optimized our API call to have a lengthy context with details, that is run only once, and shorter actual prompts/commands and responses.
2. Hence, we set the context only once per session, to minimize the number of tokens we are using. That's done using the set_context function (as seen in main.py)

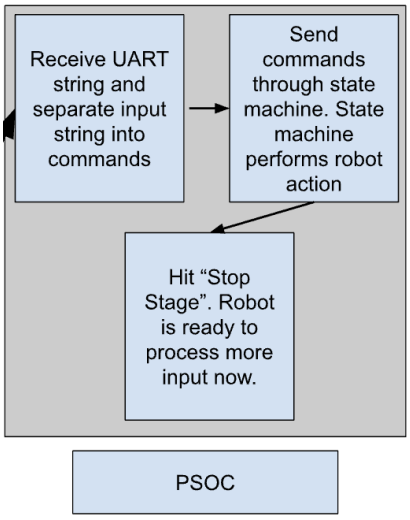
```python
    def send_prompt(self, prompt):
        combined_prompt = self.session_prompt + "\n\n" + prompt
        headers = {
            'Authorization': f'Bearer {openai_api_key}'
        }
        data = {
            'model': 'gpt-4',    # You can change the model as needed
            'messages': [{"role": "user", "content" : combined_prompt}],
            'max_tokens': 10,
            'temperature': 0.2,  # 0.17
            'top_p': 0.1,
            'frequency_penalty': 0,
            'presence_penalty': 0,
        }
        response = requests.post('https://api.openai.com/v1/chat/completions', headers=headers, json=data)
        return response.json()

# Create a global session object
chat_session = ChatGPTSession()
```

3. The bulk of the actual API is done using the send_prompt command. Here, we declare which model that we want to use, which we chose to be gpt-4.0. This was chosen after much deliberation, as we found that various GPT-3.5 models all failed to interpret spatial directions well. 4.0, in comparison, had significantly greater capabilities and was able to more accurately determine the right motion of the car. The research is linked here. The complexity of our context and our prompts is similarly chosen based on the various results we were getting using the different models.
4. We limit the number of tokens per result, as the ideal output should be no longer than a few letters, for example, "R F L F". Anything more than 10 tokens (which translates to far more than 10 letters) most likely means our model is not working as intended.

5. Temperature and Top_P are parameters used to determine how deterministic and consistent our model functions. A higher value of both would lead to a more “creative” model, one that would provide different and innovative answers even if two questions were identical. In comparison, the low values of both of these ensures that we have a more consistent model, that consistently returns the same values for the same command, something critical for our robot’s functionality.
6. We are returned a JSON that contains a variety of information, including the number of tokens spent, but more importantly, the output of the LLM, which can then be passed via UART to the PSOC.

As illustrated, our Raspberry Pi functionality covers the first half of the spectrum, going from microphone input to directional commands.

PSOC



Our PSOC code can be broken down as follows:

1. Interpret UART signals. Requires a delay between each input to ensure each are read correctly.
2. Once all directional commands (letters) have been interpreted, run them through the state machine. Terminate when finished.

We will not analyze variable presets here as they are all commented in the code. Instead, the focus will be on our UART and state machine implementations

UART SIGNALS

```
192  /* Read UART Information from Raspberry */
193  void RaspberryFunction() {
194      char strbuf[60]; // Buffer to hold commands
195
196      UART_PutString("Rx:    ");
197      char receivedByte = (char) UART_ReadRxData(); // Receieves one byte
198      UART_PutChar(receivedByte);
199      UART_PutString("\r\n");
200
201      // Halt byte. If receieved, then stop command collection
202      if (receivedByte == 'X') {
203          stringPointer = 0; // Reset pointer
204
205          sprintf(strbuf, "receivedString: %s\r\n", receivedString);
206          UART_PutString(strbuf);
207
208          UART_RXBITCTR_CONTROL_REG &= (uint8) ~UART_CNTR_ENABLE;    // Disable UART RX
209          raspberryEnabled = false;
210          continueExecute = true;
211      }
212      else {
213          receivedString[stringPointer] = receivedByte; // Store
214          receivedString[stringPointer + 1] = ',';
215          receivedString[stringPointer + 2] = '\0';
216          stringPointer += 2; // Increment to next byte
217      }
218  }
```

1. We create a String buffer to hold the incoming UART.
2. Each time the RaspberryFunction is called (done in a for loop in our main method), we read one more byte. We store that in receievedString[stringPointer], and stringPointer is incremented each time by two to store the next byte. We also store a null byte in case the current byte is the last one.
3. We implement a “halt” byte through the letter X. As mentioned above as part of our TX signal from the Pi, if the received byte is an ‘X,’ then we know that we should stop reading from the UART and disable it, and should instead switch to executing the commands, done through the “continueExecute” line. Since that variable is now true, we can then begin executing through our state machine.

STATE MACHINE

The actual commands only need to be slightly modified from our first two projects. The speed control portion of the car remains the same (moving at a fixed speed), with the only modification being that when we’re going backward, we also need to switch the signals going to the H-bridge.

In terms of the servos, however, we no longer need to set the servo's PWM based on where the line is. Instead, we chose fixed values for "left" and "right", and simply had the car move in those directions for some time.

In order to cleanly set all of these different functions, we built a variety of functions that set specific variables:

```
254  /* Commands for specific motions. */
255
256  void StartMotor()
257  {
258      PIDController_Init(&pidMotor, PID_KP_MOTOR, PID_KI_MOTOR, PID_KD_MOTOR);
259      PIDController_Init(&pidServo, PID_KP_SERVO, PID_KI_SERVO, PID_KD_SERVO);
260      HallEffect_Timer_Start();
261      PWM_Motor_Start();
262      hallEnabled = true;
263      HallEffect_Interrupt_SetPending();
264  }
265
266  void StopMotor()
267  {
268      hallEnabled = false;
269      HallEffect_Interrupt_ClearPending();
270      HallEffect_Timer_Stop();
271      PWM_Motor_Stop();
272  }
273
274  void StartServo()
275  {
276      PWM_Servo_Start();
277  }
278
279  void StopServo()
280  {
281      PWM_Servo_Stop();
282  }
283
```

By separating each functionality, we were then able to build more a more robust state machine that only calls certain commands, helping modularize our code.

```
/*---------------------------------------------------------------------------*/
/* State Machine Implementation*/

/* Implement the FORWARD state. */
void handleForwardState(enum Statetype prevState)
{
    StartMotor();
    StartServo();
    HBridge_Reg_Write(FORWARD_VAL);
    UART_PutString("Start Forward\r\n");
    PWM_Servo_WriteCompare(SERVO_PWM_FORWARD);
    servoPWM = SERVO_PWM_FORWARD;
}

/* Implement the BACKWARD state. */
void handleBackwardState(enum Statetype prevState)
{
    StartMotor();
    StartServo();
    HBridge_Reg_Write(BACKWARD_VAL);
    UART_PutString("Start Backward\r\n");
    PWM_Servo_WriteCompare(SERVO_PWM_FORWARD);
    servoPWM = SERVO_PWM_FORWARD;
}

/* Implement the LEFT state. */
void handleLeftState(enum Statetype prevState)
{
    StartMotor();
    StartServo();
    HBridge_Reg_Write(FORWARD_VAL);
    UART_PutString("Start Left\r\n");
    PWM_Servo_WriteCompare(SERVO_PWM_LEFT);
    servoPWM = SERVO_PWM_LEFT;
}
```

```
/* Implement the RIGHT state. */
void handleRightState(enum Statetype prevState)
{
    StartMotor();
    StartServo();
    HBridge_Reg_Write(FORWARD_VAL);
    UART_PutString("Start Right\r\n");
    PWM_Servo_WriteCompare(SERVO_PWM_RIGHT);
    servoPWM = SERVO_PWM_RIGHT;

}

/* Implement the STOP state. */
void handleStopState()
{
    HBridge_Reg_Write(FORWARD_VAL); // To reset HBRidge to default
    PWM_Servo_WriteCompare(SERVO_PWM_FORWARD);
    UART_PutString("Stopping....\r\n");
    StopMotor();
    StopServo();
}
```

As can be seen, each separate state uses the separate functionality of servos/motors to move in that direction properly.

Finally, we use an “execute” function to parse through the string of directional commands and call the states.

```
void execute()
{
    enum Statetype state = STOP;

    char * nextAction;
    char strbuf[32];
    UART_PutString("test.\r\n");

    sprintf(strbuf, "test receivedString: %s\r\n", receivedString);
    UART_PutString(strbuf); // Save UART string into string buffer

    // Marker for first char in sequence
    if (firstInstruction) {
        UART_PutString("first char.\r\n");
        nextAction = strtok(receivedString,",");
    }
    else {

    // STATE MACHINE IMPLEMENTATION
    if (nextAction != NULL) {
        sprintf(strbuf, "%c\r\n", *nextAction); // Take next action
        UART_PutString(strbuf);

        switch (*nextAction) {
            case 'F':
                handleForwardState(state);
                wentForward = true;
                turning = false;
                state = FORWARD;
                break;
            case 'B':
```

As can be seen here, we separate each action and execute it from the string buffer. The execute does one action at a time, hence, is called in the for loop in our main method.

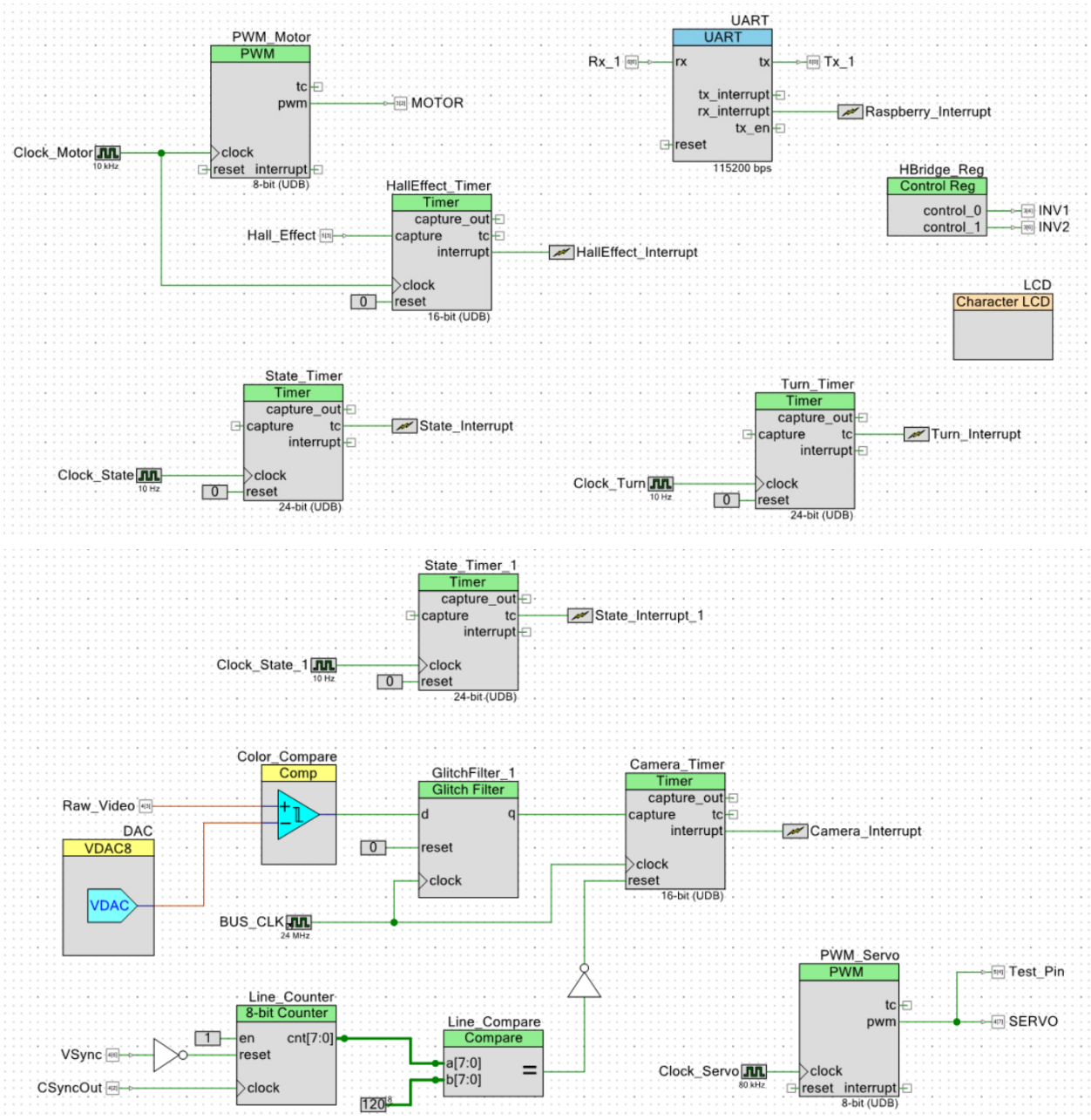
We also set some extra variables, such as “wentForward” and “turning” so that we can reset the car appropriately at the end of a motion. We also use a “state” variable to keep track of the current motion.

The firstInstruction variable is just a marker to print out the first command on our Xbee to ensure proper execution.

```
for(;;)
{
    // Runs the command in our buffer
    if (continueExecute) {
        continueExecute = false;
        execute();
        UART_PutString("End Action.\r\n");
        firstInstruction = false;

        // If reached the end, then reset beginning
        if (actionsComplete) {
            firstInstruction = true; // So that the next string's first char is recorded
        }
        else if (turning) {
            Turn_Timer_Start();
            Turn_Timer_ReadStatusRegister();
            Turn_Interrupt_ClearPending();
        }
        else {
            State_Timer_Start();
            State_Timer_ReadStatusRegister();
            State_Interrupt_ClearPending();
        }

        actionsComplete = false;
    }
```

Each instruction triggers a timer to hold the motion for a certain amount of time - 5 seconds for a forward motion, and 3 seconds for a turning motion, labeled as state_timer and turn_timer. This is done to ensure that motions are substantial and the car does not jerk for very short increments of commands. We also employ “state_timer_1”, an extra timer used as a delay before switching from the “read command from UART” functionality to the “execute command” functionality. This is to ensure that the robot does not begin executing while still expecting additional commands. All of this is done within the main function.

PSOC SCHEMATIC



In our top design, we maintain many of the same components as before, including a hall effect component (top left) that connects to the motor; and navigation capabilities (bottom left) that are no longer being used in our final project. Also, the “state timer” counter is no longer used, as that originally was used for the line compare

New components:

1. UART. Our UART is modified from before, as it used to only transmit from TX_1 to the XBee. Now, however, we are taking input from the Raspberry Pi, so we have the Rx_1 pin connected, as well as a Raspberry_interrupt function, explained before, that is triggered whenever a new set of directional commands is received from the Pi via UART.
2. HBridge_Reg. Unlike before with our motor board, we now also need to set the inverters of the HBridge to determine the direction that we are moving. Hence, we create two variables in this control register, INV1 and INV2, that are set in our “forward” and “backward” movement states. 1, 01 in binary for INV1 and INV2 respectively, translates to moving forward, while 2, 10 in binary for INV1 and INV2, translates to moving backward.
3. State Timer and Turn Timer. These are timers we use to ensure that each of our directional commands, forward/backward (state) or left/right (turn) are held for a certain amount of time. We initialize these timers to begin counting when one of the commands is received (done in the for loop within our main function), and they keep counting until the interrupt is triggered. At that point, the motion is complete, so we reset the timers and variables.

Final Thoughts on Final Project:

Thank you to all the professors and TAs who were with us every step of the way as we debugged code, fixed the hall effect sensor for the 3rd? 4th? time, and replaced our broken parts with new parts every 10 minutes.

In terms of future improvements, ideally, rather than just bucketing our commands into our simple commands, we would be able to have more complex movement, including potentially allowing “time” as an output from the LLM. In other words, to draw certain shapes, we should be able to control how long we are going right or how long we’re going left, which would be an exciting implementation and demonstrate the capabilities of LLMs. However, we were greatly limited by the capabilities of current LLMs as well as the UART communication to the PSOC. Another optimal improvement, both to reduce the time sink we had but also because of more features, would be to control the car’s motions solely through the Pi rather than the PSOC. This would allow us to run

more complex directions directly from the Pi, rather than having to ensure that it can pass through UART and be interpreted by the PSOC.

Overall, this final project allowed us to take something both of us are greatly interested in, LLMs, and creatively incorporate them into our robot. Thank you for a great semester!