TEAM RHEES
ECE 216: Term Project Report

Team Members:

- Ebisan Ekperigin
- George Klimiashvili
- Youssef Hussein
- Shingirai Dhoro

Abstract:

This report outlines the work done by our team to produce a remotely operated vehicle. The project consists of software written in C to handle the logic and control of the hardware components which are the Logitech F710 Wireless Gamepad, PIC32MX795F512L, Raspberry Pi, H-Bridge, a pair of DC Gearmotors and a display for the Raspberry Pi. The user sends a control signal to the Raspberry Pi by moving both analog controls (joysticks) of the gamepad forward and backward, where each one controls the motor speed and direction. Then the Raspberry Pi communicates with the PIC microcontroller through Universal Asynchronous Receiver/ Transmitter (UART). Afterwards, the PIC sends a Pulse Width Modulation (PWM) signal to the H-Bridge which controls the power going from the 9V+ battery to the motors to turn the wheels and move. Through that setup our team managed to build a robot that can navigate robustly around tight corners and handle sharp turns by having the user manage the speeds of the motors separately. Our ROV is also able to detect collisions and make sure that it does not go keep accelerating in the direction of the collision until it has moved away from the object. The program was tested on hardware and it works as it is intended as outlined in the in the project manual.

Team Members Contribution:

The work was split among team members where each person would fulfil one of the requirements, however each was helping out with the other requirement to the best of their ability. The debugging part was evenly done by all members during different lab sections and we all worked as a team to make sure that our system was working as expected.

- **George Klimiashvili** worked on the writing the Python code to interface with the Logitech F710 Wireless Gamepad on the Raspberry Pi. He modified the message produced to specify which joystick is used and how far up is it turned.
- **Shingirai Dhoro** worked on the C program that was used to control the DC Gearmotors. He used a timer to help synchronize the speeds of the motors according to the counts on each encoder.
- **Youssef Hussein** worked on the UART communication between the Raspberry Pi and the PIC32MX795F512L. Also worked on decoding the messages from UART to the values that control the output control.
- **Ebisan Ekperigin** Assembled the hardware parts. He mounted all the parts on the frame and managed some of the wiring between different components.

Overall, we are very happy with how everyone's hard work to make this project work and the unique skills and insights which every team member brought to the table.

Design Approach:

After discussing various design approaches, the group made a decision to use the two analog buttons on the gamepad to control the ROV. The rationale behind this choice was to have more flexibility on the control of speed and direction of the ROV using the gamepad. For example, if one slightly pushes both analog controls of the gamepad forward, the ROV goes forward but slowly, and it accelerates as you push the analog controls further forward. We thought this would be cool as it mimics real life vehicles that accelerate more when you press the gas pedal further down and less if you press the gas pedal slightly.

The left and right analog controls were set up to control the left and right motors of the ROV respectively. When both controls are pushed forward or backwards, the ROV moves in the respective direction. When you only push one button in a given direction, the ROV turns in that direction, for example, if you push the right analog control backwards, the motor will turn right while moving in reverse direction.

After the gamepad, we designed a program on Raspberry Pi that would send information to the PIC32 about which button was pressed and the amount by which it was pressed. The communication protocol we used was UART, and we used UART 2 (pin 49) to receive message from the Raspberry Pi. We used the baud rate to 115,200. Once the message is received, we determine which analog control was pressed by looking at the first character of the message. If it is R, then it is from the right analog control and if the first character is L then we know that the message is from the left analog control. In both cases, we take the value that appears after L or R and process it to produce a duty cycle of a PWM signal that drives the H-bridge that in turn drives our motors.

In order to process these values, we divide the duty cycle of the PWM period into 127 parts. The choice of the value 127 was based on the fact that the gamepad analog controls values range from 0 to 255 with 0 being the furthest backwards and 255 being the furthest forward and 127 being the rest position. Overall, the chosen value represents PWM period of about 1kHz. So, in order to get the value of OCxRS, we just subtract the gamepad value from 127 and divide it by PR2. If the numerator has a negative value, we invert the direction of the motor and take the absolute value of 127 - gamepad value. This approach results in a proportional increase/decrease in duty cycle depending on how further forward/backwards the analog controls are pushed. The equation that explains this relationship is as follows:

$$OCxRS = \frac{abs(127 - gamepadValue)}{PR2} \quad \ldots\ldots(1)$$

To effectively control the speed of the ROV, we realized that we needed to add another factor to our design since we had no guarantee that our motors will move uniformly. We added four Change Notification (CN) interrupts, CN13, CN14, CN15 and CN16 which correspond to digital input pins 81, 82, 83 and 84 respectively. In the Change Notification Interrupt Service Routine, we track 4 counters, forward and backward counters for each of the motor on our ROV. These counters keep track of how much each motor has moved forward and backward for 5 seconds (the time parameter can be easily adjusted from the code). To time this, we use the Core Timer (CT) interrupt at the start of our program. For the first 5 seconds, we get counts in the Change Notification ISR then we disable the CT interrupt after 5 seconds in the CT interrupt ISR. We then take the counters and compare them in pairs, that is left and right motor forward counters and left and right motor backward counters. If there is a mismatch, we multiply the mismatch factor to the value of OCxRS calculated in equation 1 to make sure that the motors are always moving with the same velocity. We also make sure that if the sample size of counts is too small, we discard it. An example code snippet of how this is implemented is shown in figure 1below:

```
128             if (reverseScaler > 1){
129                 OC2RS = -temp2 * reverseScaler ; // multiply by scaler to match velocities
130             }
131             else{
132                 OC2RS = -temp2; //go in negative direction
```

Figure 1: Implementation of matching motor velocities.

In this case, reverseScaler is just the ratio of right backward counter to left backward counter. If they are not the same, that is if the reverseScaler is greater than 1 then we multiply -temp2 (which is basically equation 1) with the reverseScaler and put it in OC2RS.

To invert the direction of the motor, we just look at whether the numerator of equation 1 is negative or not. If it is, we invert the direction pins for the motor in question for example, going forward, digital input pin 30 is high and pin 31 is low for right motor while digital input pin 60 is high and pin 61 is low for left motor. When going backwards, that is when the value numerator of equation 1 is negative (without absolute value), these digital input pins are reversed. This logic is implemented in the invertMotorDirectionL and invertMotorDirectionR methods for left and right motor respectively.

To drive the motors, we use Output Compare (OC) 2 (analog output pin 76) and 4 (analog output pin 78) for left and right motor respectively. Their duty cycles were controlled in the ways described in the previous paragraphs. Refer to Appendix I for flowcharts.

Extra Credit:

For the first part of the extra credit, we implemented a collision detecting sensors. We used external push buttons to detect when the front or the back of the vehicle collides with an object. In order to accomplish this, we connected push buttons between +3.3V terminal and the digital input terminals RA2 and RA3. This allowed us to detect when the car hits and object and stop the rotation of the motor in the direction of this object. To this end, we used a simple if statement that checks the value of the ports RA2 and RA3 within the change notification interrupt function and adjust the duty cycle of each motor accordingly.

For the second part of the extra credit, we implemented SPI communication protocol between PIC32 and the Raspberry Pi. The Raspberry Pi was set up as a master, while the PIC32 acted as a slave. The baud rate was set to 115,200 and SPI3 was used on the PIC32. Given the decoded messages received from the gamepad, we had to format the messages in order to be able to communicate between PIC32 and Pi. Instead of sending strings as in UART, we had to send lists of number when using SPI. Therefore, instead of using character 'L' and 'R' to specify which analog joystick was used, we used numbers 48 and 49 corresponding to ASCII characters '0' and '1' to specify which joystick was moved. These messages were further decoded by the PIC32 in a similar fashion to UART and further used to adjust the rotation of the motors.

Wiring & Connections:

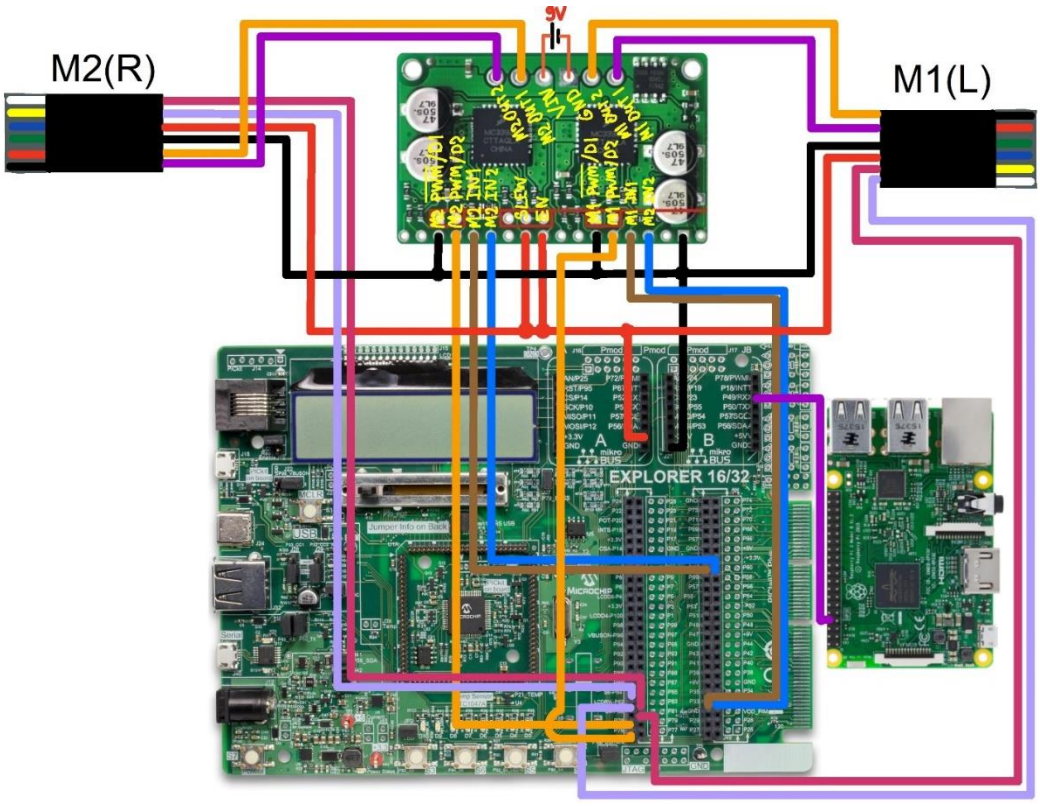


Figure 2: Schematic of ROV wiring for team Rhees.

First, we connected the +5V port on the Explorer Board to both motors' encoder Vcc ports and to H-bridge's EN and SLEW ports (shown as a red wire in Fig. 1). Next, we connected the GND port on the Explorer Board to both motors' encoder GND ports, GND port on the H-bridge and to the $M1\ \overline{PWM}/D1$ and $M2\ \overline{PWM}/D1$ ports on the H-bridge (black wire). To be

able to control the operation of the motor 1 (M1), we connected the $M1\ PWM/D2$ on the H-bridge to the port 76 (OC2) on the Explorer Board (orange wire), which allowed us to supply PWM signal to the M1. To be able to switch the direction of rotation for the M1, we used ports 32 and 33 on the Explorer Board set as digital outputs and connected them to the ports $M1\ IN2$ and $M1\ IN1$ on H-bridge (blue and brown wires). Similarly, to set up the motor 2 (M2), we connected port 78 (OC4) to $M2\ PWM/D2$ and ports 60 and 61 (set as digital outputs) to $M2\ IN2$ and $M2\ IN1$ (orange, blue and brown wires). In order to be able to read the encoder for M1, we connected encoder A port (yellow wire) to the port 81 (CN13/RD4) on the Explorer Board and encoder B port (white wire) to the port 82 (CN14/RD5). Similarly, to read encoder for M2, we connected encoder A port (yellow wire) to the port 84 (CN16/RD7) on the Explorer Boards and encoder B port (white wire) to the port 83 (CN15/RD6). To connect the H-bridge to the motors, we connected $M1\ OUT1$ and $M1\ OUT2$ to motor power wires red and black on M1, and $M2\ OUT1$ and $M2\ OUT2$ to motor power wires red and black on M2. To power up the H-bridge, we connected a 9V battery between VIN and GND pins. Finally, to allow UART communication between PIC32 and Raspberry Pi, we connected pin 49 (U2RX) to pin 8 (TXD0) on the Pi.

Wiring for Extra Credit:

In order to be able to use the collision detection sensors, we connected one push button between +3.3V port and port 58 (RA2) and another push button between +3.3V port and port 59 (RA3) on the Explorer Board.

In order to be able to use SPI communication, we connected a wire between pin 19 (MOSI) on the Raspberry Pi and pin 52 (SDI3) on the Explorer Board. Moreover, we connected pin 23 (SCLK) on the Raspberry Pi to the pin 48 (SCK3) on the Explorer Board.

Integration & Testing

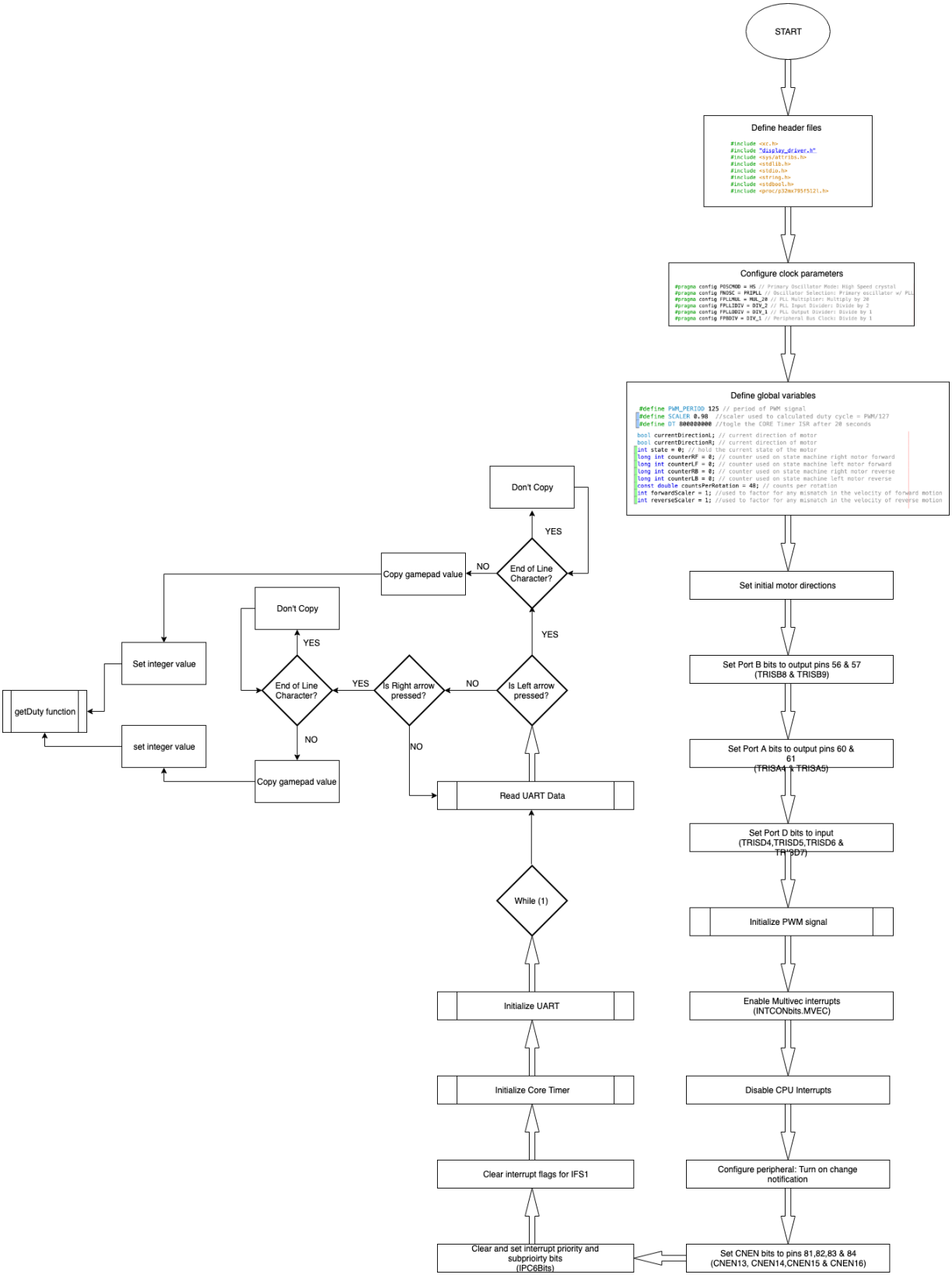
We tested each feature at a time and did incremental integrations and testing until we had the whole system up and ready. We began testing the python code to see if it is responding to pressing of gamepad buttons. To check if the UART transmition works on the Raspberry Pi, we connected Tx pin to an oscilloscope and checked if we were receiving signal as intended. After that, we tested to see if we were sending the right values to the PIC and that they there were received correctly. We then tested the decoding framework using the debugging tool in MPLABX IDE, to see if we are correctly extracting the values from the Raspberry pi. After that, we tested the code for generating the pulse width modulation (PWM) signal for each motor separately using oscilloscopes and motors themselves. Once we confirmed they were working, we assembled the whole system and tested again. At this stage, we finetuned the details regarding how we were decoding values from gamepad analog controls as well as the velocity controls as we were able to see what was not working correctly after the whole system was integrated. To test the collision detection sensor for extra credit, we drove into a wall in forward and reverse motion and adjusted our code until it was working as intended. Similar to how we tested UART communication, we first checked if we were receiving signals from the SPI MOSI pin on the Raspberry Pi and afterwards checked if the values were received on the PIC. These features were tested once all the basic functions were tested and working properly. After testing, all functions worked as they were intended to.

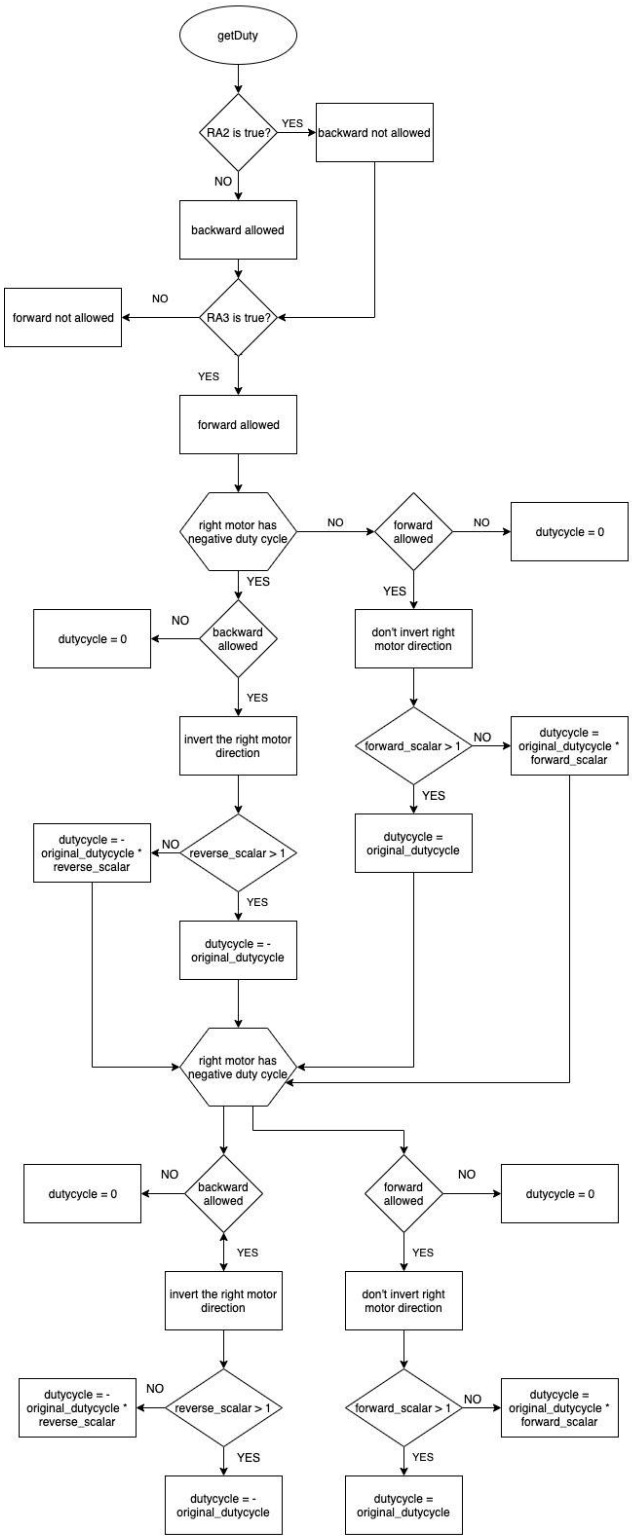
Conclusion:
That project helped the team members cultivate a deeper understanding of the material learned throughout the class. We learned to use an incremental approach to building and debugging components before putting the parts together. Also, the design decisions that were chosen in the beginning ended up affecting the workflow. After some time, it took us a lot of work to change some of the initial decision we made. In addition to that, the main lesson learned was the coordination between the team members. The busy schedules of everyone on the team made it very hard for the whole team to meet in the same time outside of class to work on the project but every member had the commitment towards the project to show up and do their part.

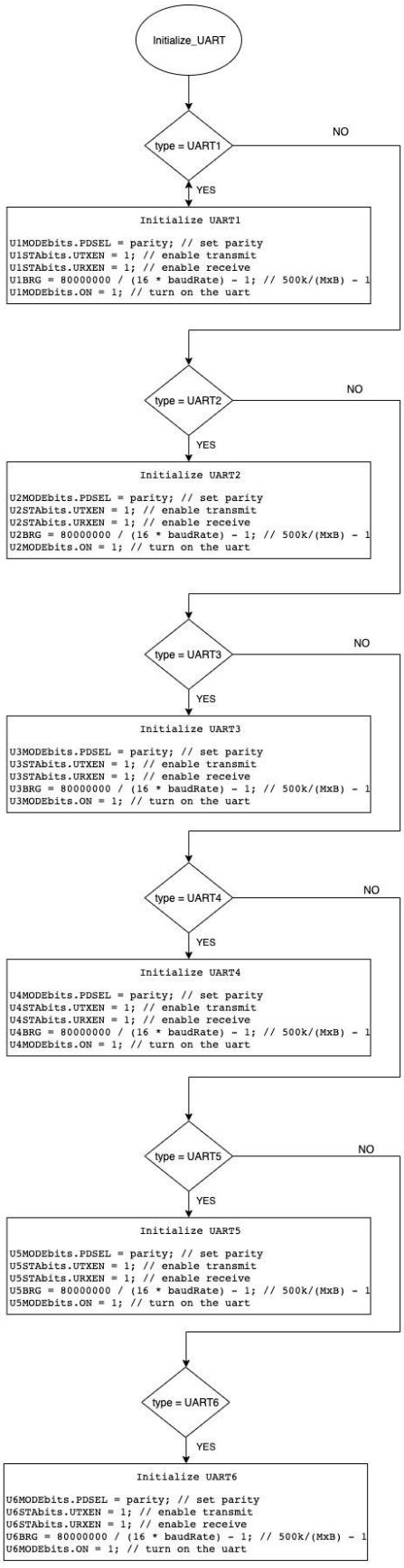
Works Cited:

- "Pololu - Dual MC33926 Motor Driver Carrier." *Pololu Robotics & Electronics*, https://www.pololu.com/product/1213.
- "Raspberry Pi." *Raspberry Pi - Electronics Club, IIT Bombay*, https://elec-club-iitb.github.io/tutorials/r_pi/.
- *Explorer 16/32 Development Kit*, https://www.microchip.com/developmenttools/ProductDetails/PartNo/DM240001-2?utm_source=MicroSolutions&utm_medium=Article&utm_content=DevTools&utm_campaign=StandAlone.

Appendix I:


START
Define header files
#include <xc.h>
#include "display_driver.h"
#include <sys/attribs.h>
#include <stdlib.h>
#include <stdio.h>
#include <string.h>
#include <stdbool.h>
#include <proc/p32mx795f512l.h>
Configure clock parameters
#pragma config POSCMOD = HS // Primary Oscillator Mode: High Speed crystal
#pragma config FNOSC = PRIPLL // Oscillator Selection: Primary oscillator w/ PLL
#pragma config FPLLMUL = MUL_20 // PLL Multiplier: Multiply by 20
#pragma config FPLLIDIV = DIV_2 // PLL Input Divider: Divide by 2
#pragma config FPLLODIV = DIV_1 // PLL Output Divider: Divide by 1
#pragma config FPBDIV = DIV_1 // Peripheral Bus Clock: Divide by 1
Define global variables
#define PWM_PERIOD 125 // period of PWM signal
#define SCALER 0.98 //scaler used to calculated duty cycle = PWM/127
#define DT 80000000 //togle the CORE Timer ISR after 20 seconds
bool currentDirectionL; // current direction of motor
bool currentDirectionR; // current direction of motor
int state = 0; // hold the current state of the motor
long int counterRF = 0; // counter used on state machine right motor forward
long int counterLF = 0; // counter used on state machine left motor forward
long int counterRB = 0; // counter used on state machine right motor reverse
long int counterLB = 0; // counter used on state machine left motor reverse
const double countsPerRotation = 48; // counts per rotation
int forwardScaler = 1; //used to factor for any mismatch in the velocity of forward motion
int reverseScaler = 1; //used to factor for any mismatch in the velocity of reverse motion
Set initial motor directions
Set Port B bits to output pins 56 & 57 (TRISB8 & TRISB9)
Set Port A bits to output pins 60 & 61 (TRISA4 & TRISA5)
Set Port D bits to input (TRISD4,TRISD5,TRISD6 & TRISD7)
Initialize PWM signal
Enable Multivec interrupts (INTCONbits.MVEC)
Disable CPU Interrupts
Configure peripheral: Turn on change notification
Set CNEN bits to pins 81,82,83 & 84 (CNEN13, CNEN14,CNEN15 & CNEN16)
Clear and set interrupt priority and subprioirty bits (IPC6Bits)
Clear interrupt flags for IFS1
Initialize Core Timer
Initialize UART
While (1)
Read UART Data
Is Left arrow pressed?
YES
NO
Is Right arrow pressed?
YES
NO
End of Line Character?
YES
NO
Don't Copy
Copy gamepad value
End of Line Character?
YES
NO
Don't Copy
Copy gamepad value
Set integer value
getDuty function
set integer value


getDuty
RA2 is true?
YES
backward not allowed
NO
backward allowed
forward not allowed
NO
RA3 is true?
YES
forward allowed
right motor has negative duty cycle
NO
forward allowed
NO
dutycycle = 0
YES
YES
dutycycle = 0
NO
backward allowed
don't invert right motor direction
YES
invert the right motor direction
forward_scalar > 1
NO
dutycycle = original_dutycycle * forward_scalar
YES
dutycycle = - original_dutycycle * reverse_scalar
NO
reverse_scalar > 1
dutycycle = original_dutycycle
YES
dutycycle = - original_dutycycle
right motor has negative duty cycle
dutycycle = 0
NO
backward allowed
forward allowed
NO
dutycycle = 0
YES
YES
invert the right motor direction
don't invert right motor direction
dutycycle = - original_dutycycle * reverse_scalar
NO
reverse_scalar > 1
forward_scalar > 1
NO
dutycycle = original_dutycycle * forward_scalar
YES
YES
dutycycle = - original_dutycycle
dutycycle = original_dutycycle


Initialize_UART
type = UART1
NO
YES
Initialize UART1
U1MODEbits.PDSEL = parity; // set parity
U1STAbits.UTXEN = 1; // enable transmit
U1STAbits.URXEN = 1; // enable receive
U1BRG = 80000000 / (16 * baudRate) - 1; // 500k/(MxB) - 1
U1MODEbits.ON = 1; // turn on the uart
type = UART2
NO
YES
Initialize UART2
U2MODEbits.PDSEL = parity; // set parity
U2STAbits.UTXEN = 1; // enable transmit
U2STAbits.URXEN = 1; // enable receive
U2BRG = 80000000 / (16 * baudRate) - 1; // 500k/(MxB) - 1
U2MODEbits.ON = 1; // turn on the uart
type = UART3
NO
YES
Initialize UART3
U3MODEbits.PDSEL = parity; // set parity
U3STAbits.UTXEN = 1; // enable transmit
U3STAbits.URXEN = 1; // enable receive
U3BRG = 80000000 / (16 * baudRate) - 1; // 500k/(MxB) - 1
U3MODEbits.ON = 1; // turn on the uart
type = UART4
NO
YES
Initialize UART4
U4MODEbits.PDSEL = parity; // set parity
U4STAbits.UTXEN = 1; // enable transmit
U4STAbits.URXEN = 1; // enable receive
U4BRG = 80000000 / (16 * baudRate) - 1; // 500k/(MxB) - 1
U4MODEbits.ON = 1; // turn on the uart
type = UART5
NO
YES
Initialize UART5
U5MODEbits.PDSEL = parity; // set parity
U5STAbits.UTXEN = 1; // enable transmit
U5STAbits.URXEN = 1; // enable receive
U5BRG = 80000000 / (16 * baudRate) - 1; // 500k/(MxB) - 1
U5MODEbits.ON = 1; // turn on the uart
type = UART6
YES
Initialize UART6
U6MODEbits.PDSEL = parity; // set parity
U6STAbits.UTXEN = 1; // enable transmit
U6STAbits.URXEN = 1; // enable receive
U6BRG = 80000000 / (16 * baudRate) - 1; // 500k/(MxB) - 1
U6MODEbits.ON = 1; // turn on the uart