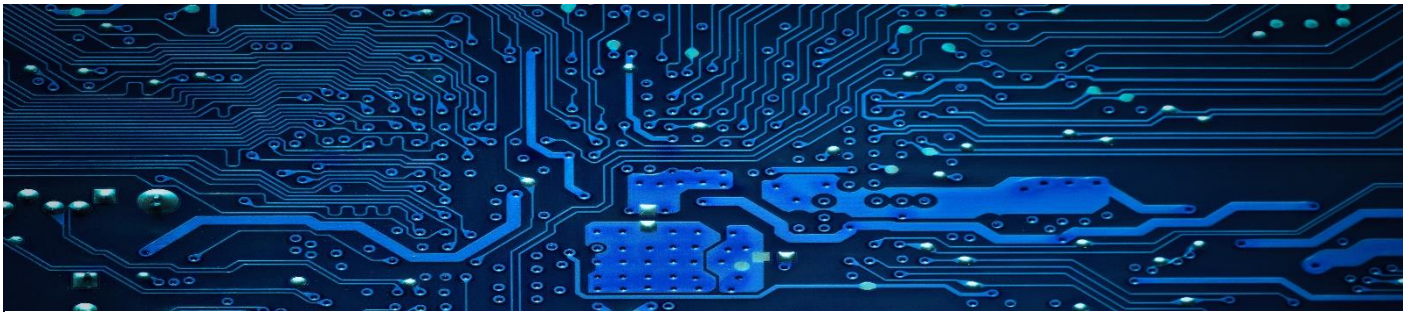
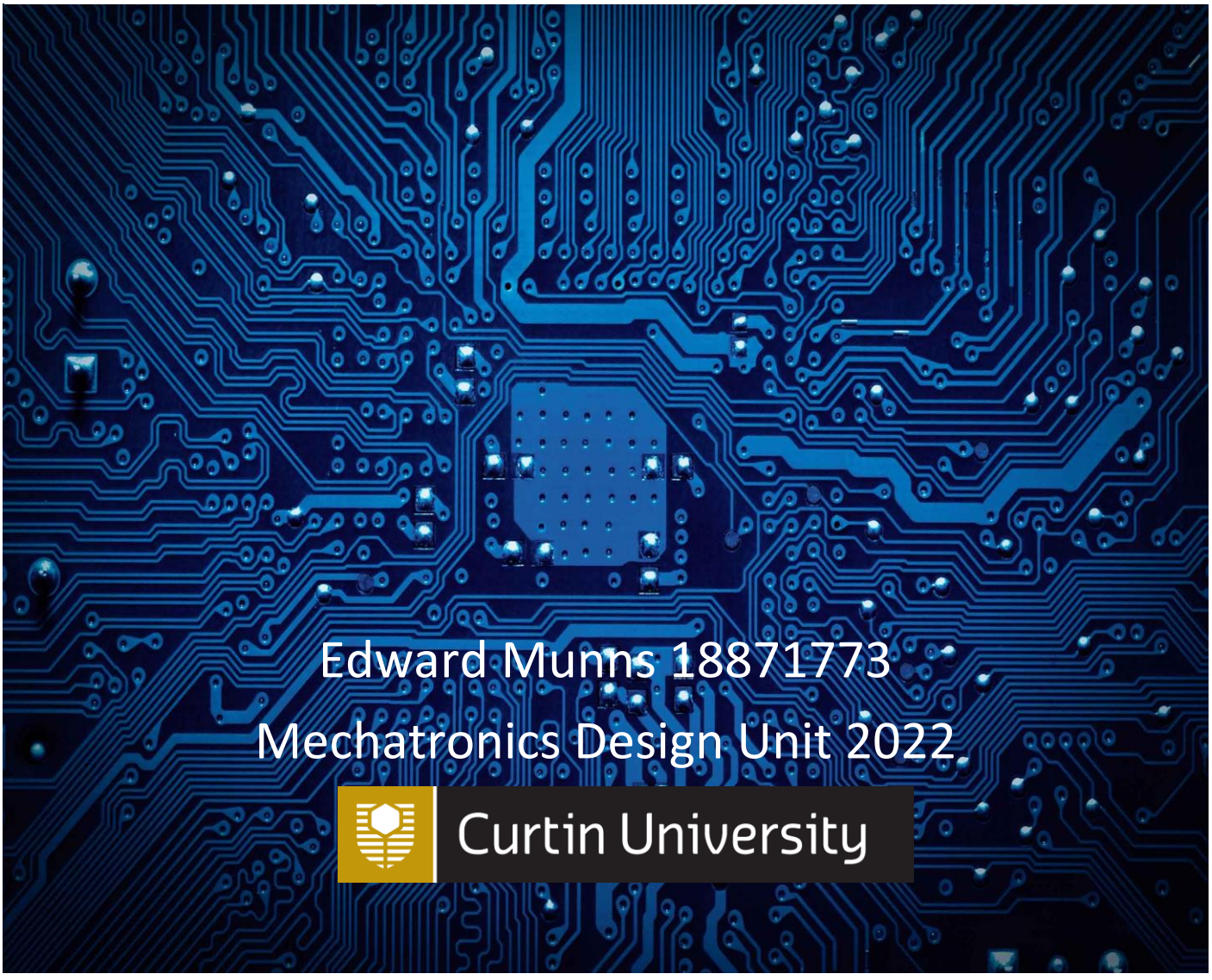


MECHATRONICS DESIGN ASSIGNMENT



Edward Munns 18871773

Mechatronics Design Unit 2022

Curtin University

Contents

Table of Figures:

1. Introduction:

This report details 2 tasks that were completed for the Mechatronics design unit at Curtin University in Semester 2 of 2022. The first task is a report on the project implementation of the unit while the second task deals with the design of a variable duty cycle PWM circuit.

2. Task 1:

This task requires detailing the creation of a line following robot throughout the 2022 2nd semester Mechatronics Design Project Unit. The report will start with design requirements and constraints, this will be followed by discussion and diagrams of the system/sub-system architecture and specifications and signal flow diagrams, a discussion on the software design and application and an analysis of alternative routes and future improvements to the system design.

2.1. Problem Domain:

This project required the mechanical and electronic construction and software development of a line following robot based on an ATMEGA 328p microcontroller, analog based variable duty cycle Pulse Width Modulated (PWM) and current amplified drive system and dual UV optical sensor feedback for error corrected control. The robotic system was required to follow a white line on a black background with feedback from the optical sensors and drive circuit control applied via software on the microcontroller. This software was also required to accept user input for control of state/mode information and provide feedback to user via serial communications.

2.2. Requirements:

After analysis of the design brief the following list of requirements/constraints was produced:

Requirements:

- Optimized for around track speed and accuracy performance
- Provide easily understood and operated user interface over serial communication
- Provide smooth and accurate control via the control algorithm
- Provide fast and easy tuning of the left and right driver circuits
- Provide method of shutdown/ESTOP

Constraints:

- PWM Waveform generation created via analog means
- User interface programmed in C# using Visual Studio IDE
- Algorithm running on 8-bit Arithmetic Logic Unit (ALU) with no hardware floating point and limited multiply/divide operations (Microchip 2015)

2.3. System/Sub-System Architecture:

The designed super-system is described in the following flow-chart diagram:

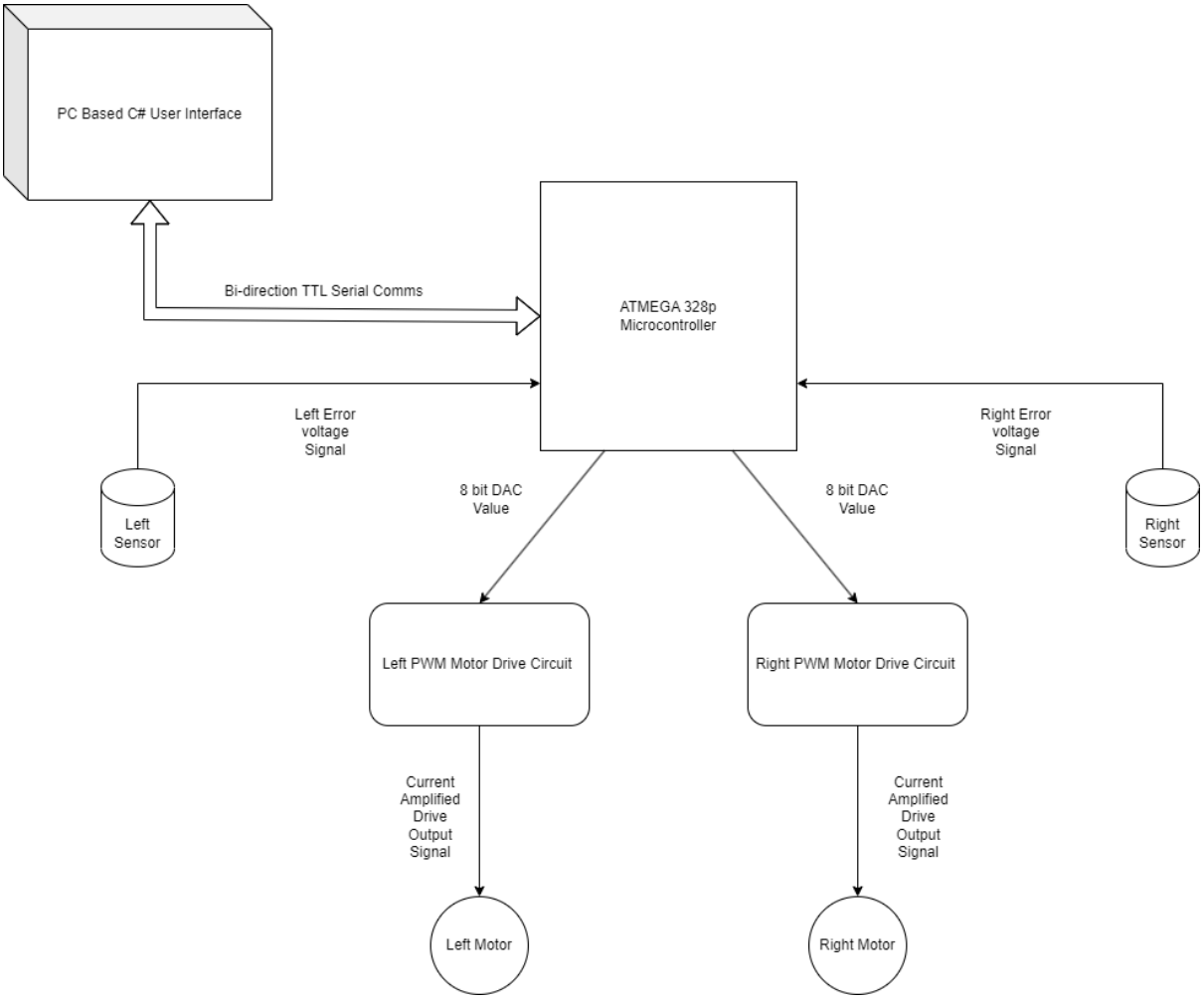


Figure 1: Super-System Overview Diagram.

After analysis this super-system was then divided into sub-systems using the “divide and conquer” system paradigm into the following sub-systems:

1. Electrical to mechanical power conversion and mechanical design
2. Current-Amplified motor output
3. Digital to Analog Converter (DAC) and LM741 OPAMP analog voltage signal generation
4. 555 Timer Based Linear Ramp Signal Generator
5. Sensor feedback system
6. Microcontroller control algorithm
7. User interface control mechanism

2.4. Sub-System Specifications:

Electrical to mechanical power conversion and mechanical design

The electrical to mechanical power transmission for the project was achieved with dual (left and right side) DC gear motors operating in a differential drive setup to provide velocity in the forward and backward directions and steering via the input power differential. Electrical power was provided to both motors from the drive systems discussed below with the motors converting this voltage and current electrical power input to torque and angular velocity mechanical power.

The mechanical design and layout of all components can be seen in the exploded view below.

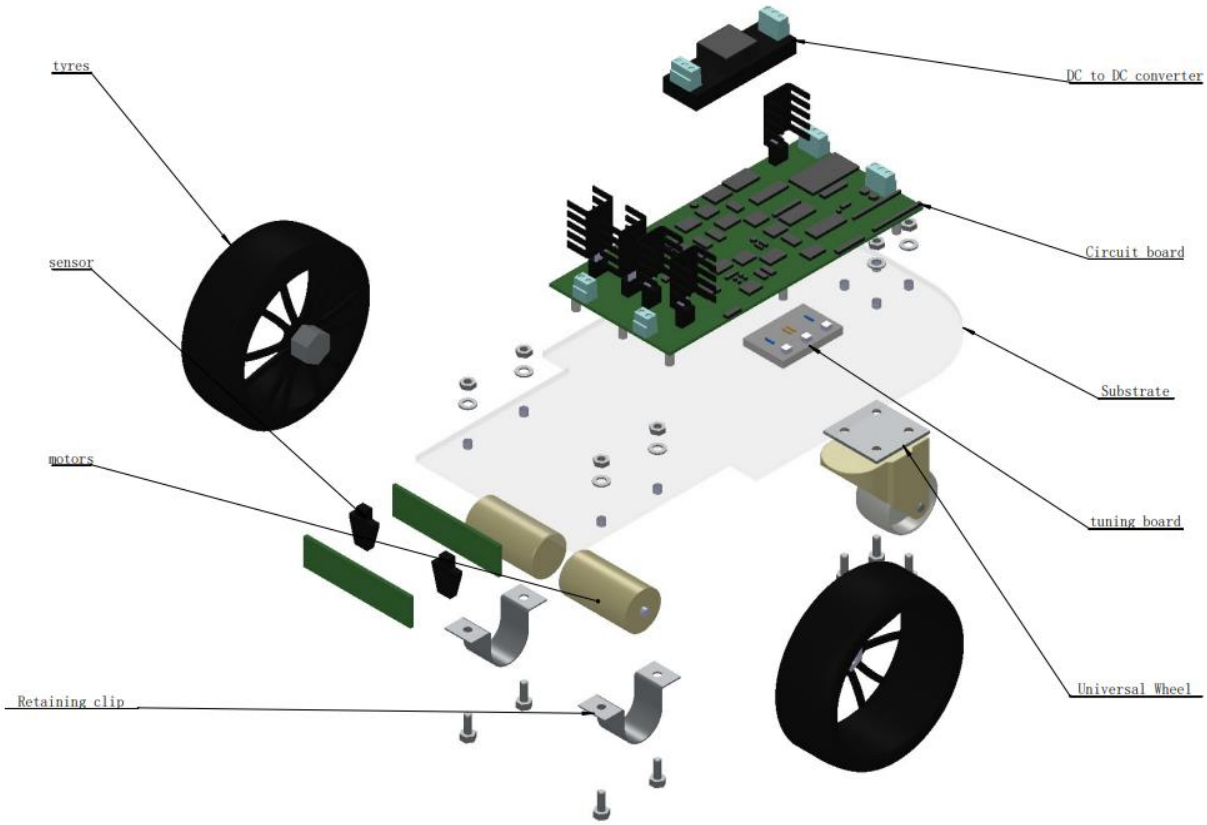


Figure 2: Mechanical Design Exploded View

Part Description List:

- Tyres: provide power transmission from axle to ground surface via traction
- Motors: Convert electrical power from drivers into mechanical power to axle
- Sensors: Provide electrical voltage feedback signal to micro-controller
- Retaining Clips: Mount motors to Substrate

- Universal Wheel: provides rotatable support to back end of substrate
- Tuning board: Provides mechanism to tune driver frequency and Left and Right Duty Cycle Symmetry
- Substrate: provides means of fastening and support of components
- Circuit board: Houses both left and right drive circuitry and microcontroller
- DC to DC convertor: Converts single swing DC voltage to -15 to 15v DC voltage

Current-Amplified motor output

This sub-system (symmetric about left and right channels) converts the Low current PWM signal generated by the output of the PWM driver to a high current PWM waveform signal via a push-pull amplifier circuit based on a TIP31 NPN and a TIP32 PNP power transistor circuit. The circuit diagram for this sub-system can be seen below.

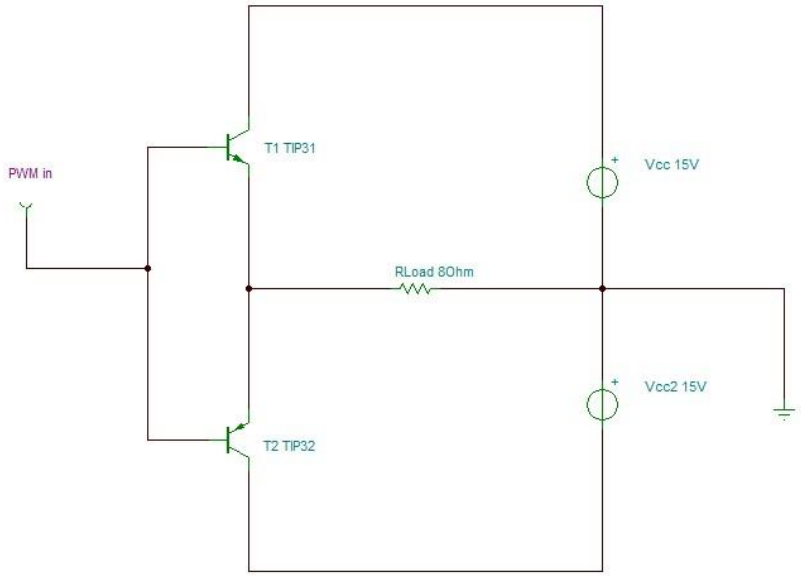


Figure 3: Push-Pull Amplifier Circuit

DAC and LM741 OPAMP signal generation

The Digital to Analog Converter and LM741 OPAMP circuit takes an 8-bit word as a control byte from the microcontroller and converts this value to a voltage output at the LM741 OPAMP output terminal. This output voltage is used as input to the positive terminal of the LM339 comparator circuit to be compared against the linear ramp signal to generate the adjustable Duty Cycle PWM moto0r control output signal. The centering of this output voltage to allow for 0 to 100% duty cycle adjustment with 50% duty cycle value falling at an input word value of 128 The 10k potentiometer on pin 14 of the DAC is used to vary the input current to pin 14 of the DAC. This adjustability allows for perfect DC matching between both left and right PWM drive circuits.

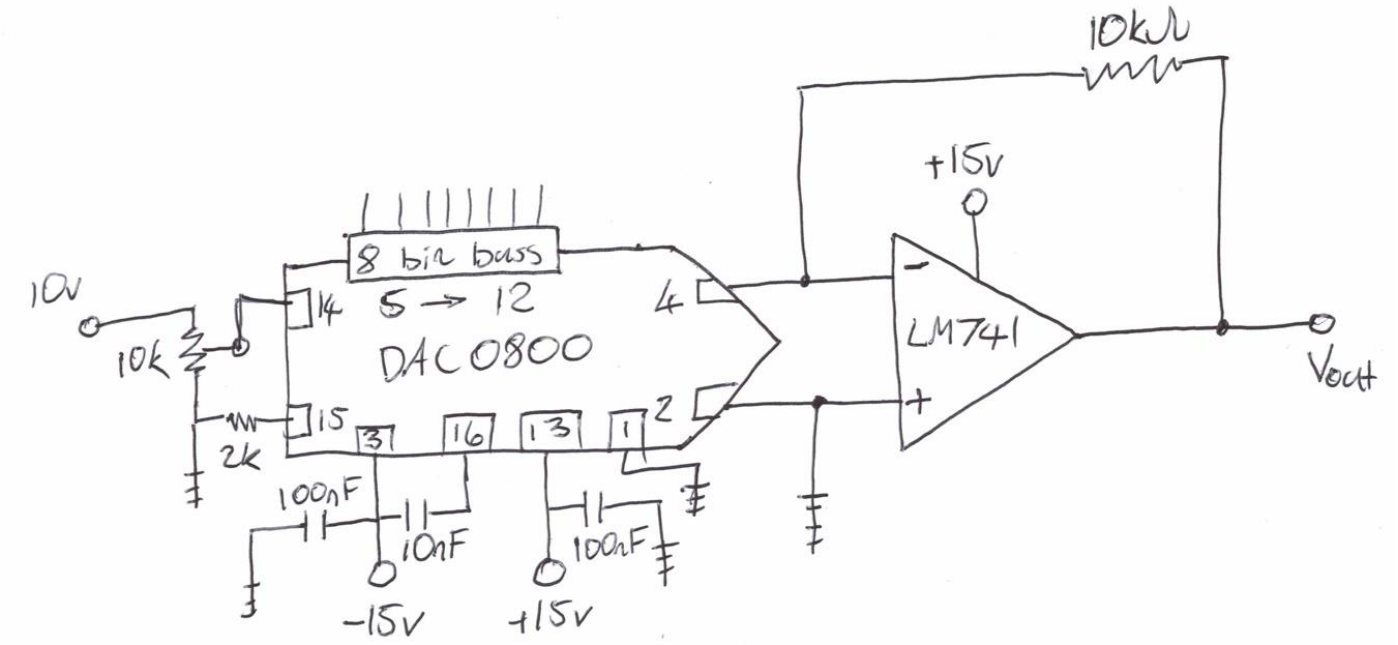


Figure 4: Digital To Analog Converter Circuit

Left and Right DAC configuration:

The diagram below shows the use of the left and right comparator circuits and the inputs of the DAC channels and single Linear ramp generator when using a single 555 circuit.

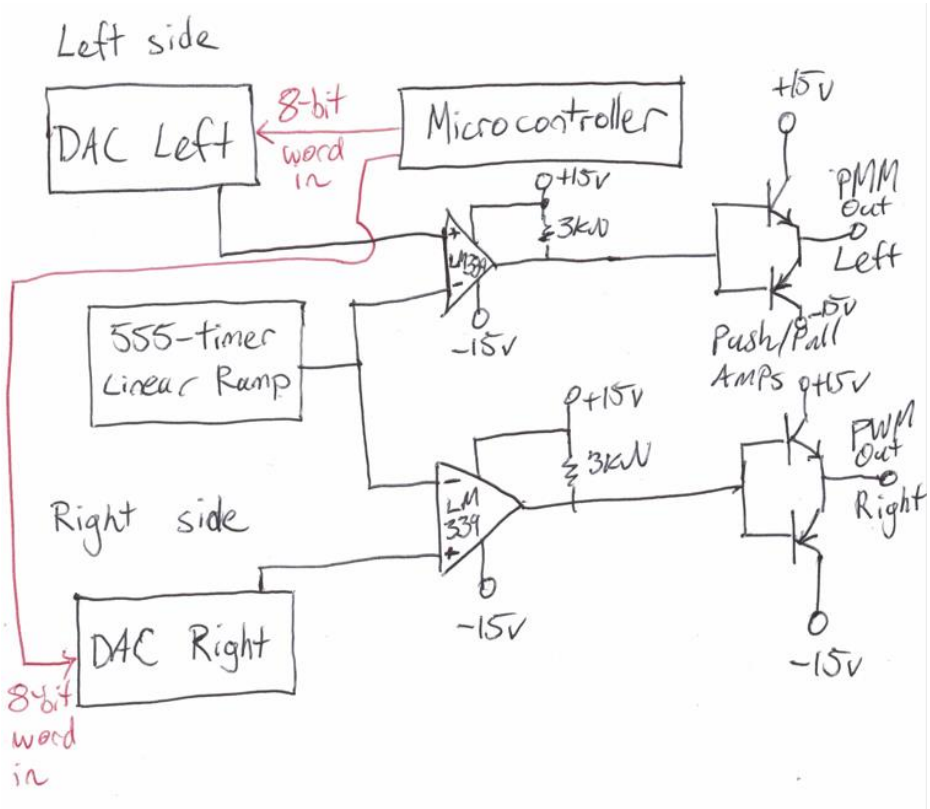


Figure 5: Comparator Circuit Configuration for Single 555 Timer Use.

555 Timer Based Linear Ramp Signal Generator

The 555-timer based Linear ramp generator is designed to be a comparison used against the DAC circuit output voltage as comparative inputs to the comparator. The linear ramp circuit also generates the PWM system frequency. The frequency can be varied vie the potentiometer P1 from ~20kHz to over 100kHz. To frequency match the left and right PWM systems it was decided to use a single 555 linear ramp circuit as input to the negative terminal of the comparators for both the left and right sides. As this comparator input has extremely high input impedance there was no detectable voltage drop in the Linear Ramp signal when doing this and the left and right PWM channels are perfectly frequency matched as they are running of the same signal. This greatly eased the PWM tuning process. A diagram of this circuit can be seen in the figure below.

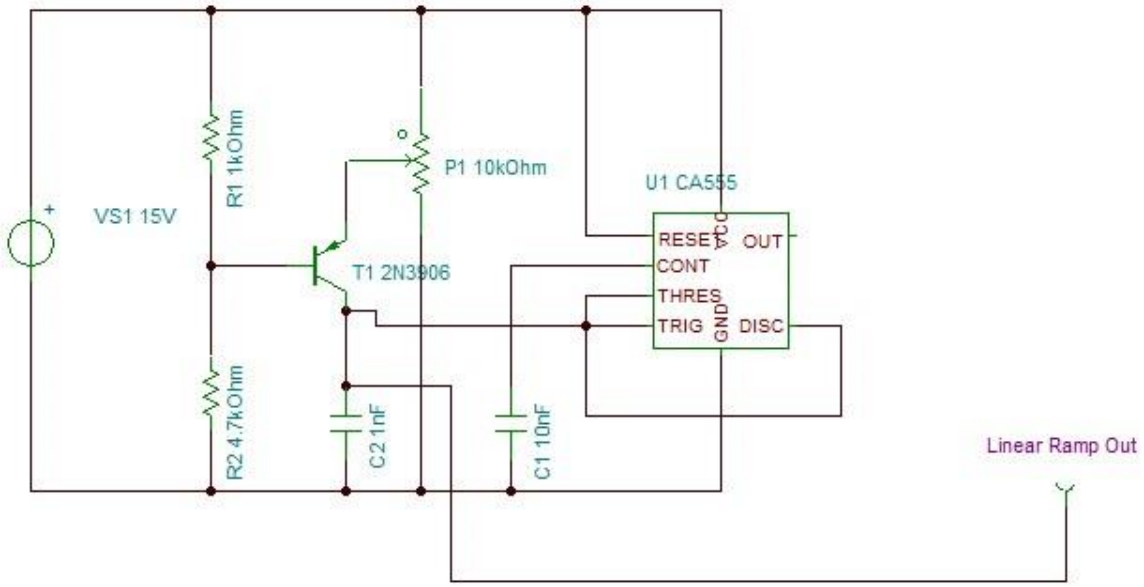


Figure 6: 555-Timer Linear ramp Generator Circuit.

Sensor feedback system

The sensor error feedback sub-system uses 2 UV reflective opto-sensors consisting of a UV LED and a phototransistor. The LED provides a UV light source that is directed towards the ground surface below the sensor mounts. This UV light is reflected of the white surface of the line the cart must follow when the sensor starts to intersect the line and is angled to be directed towards the base of the phototransistor base UV sensor. The distance from the ground surface is required to be tuned to suit this pre-set angle for optimal sensor performance. The Sensors require an input current of 40mA each for optimal LED Luminance and thus sensor performance. The following chart from the sensor datasheet shows the OPB704 distance performance curve over varying surface reflection values.

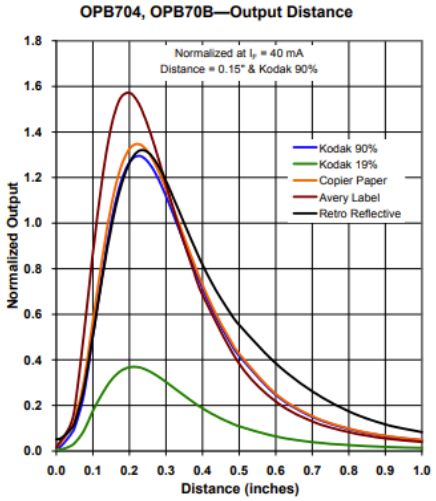


Figure 7: Opto-Sensor Distance Perform Curve (TT Electronics n.d.)

From this chart the optimal distance from surface to the sensor input is approximately 0.2 inches or 5 mm so the sensors were setup this distance above the ground surface.

The voltage generated from the sensor output is fed into the microcontroller ADC6 and ADC7 pins to be converted to an 8-bit Digital value for use in the PI control algorithm as error feedback.

Microcontroller control algorithm

The control algorithm implemented for this project uses an approximation of the continuous time PID algorithm with the derivative gain k_D set to 0 (i.e. a PI controller). This approximation was designed for implementation on low computational power 8-bit microcontrollers. The controller formula was implemented using a method described on the website Embedded.com (Embedded.com 2006). The formula in derived by first taking the derivative of the continuous time PID equation with the derivative term set to 0:

$$\dot{u}(t) = \frac{d}{dt}(k_P e(t) + k_I \int_0^t e(t)dt) = k_P \dot{e}(t) + k_I e(t)$$

This derivative is then approximated using the finite difference approach where:

$$\dot{u}(t) \cong \frac{u_{k+1} - u_k}{\Delta T}, \quad \dot{e}(t) \cong \frac{e_{k+1} - e_k}{\Delta T}$$

Resulting in a final discrete time approximation calculation of:

$$u_{k+1} = u_k + k_P(e_{k+1} - e_k) + k_I \Delta T(e_k)$$

This equation requires storing only the previous error and 8-bit DAC value to calculate the next output and can be performed with minimal calculation required and so is well suited to this project application.

User interface control mechanism

The user interface was created in the C# language within the Visual Studio IDE. This IDE is designed specifically for creation of Windows applications and is a very high level and feature rich IDE with many built-in features designed for this purpose. This system has inputs of cart controller messages of both data bytes and strings and has outputs of cart controller control messages as byte arrays sent over the serial interface.

2.5. System Signal Flowchart:

The diagram below shows the flowchart and all input and output signals from the physical sub-systems making up the super-system.

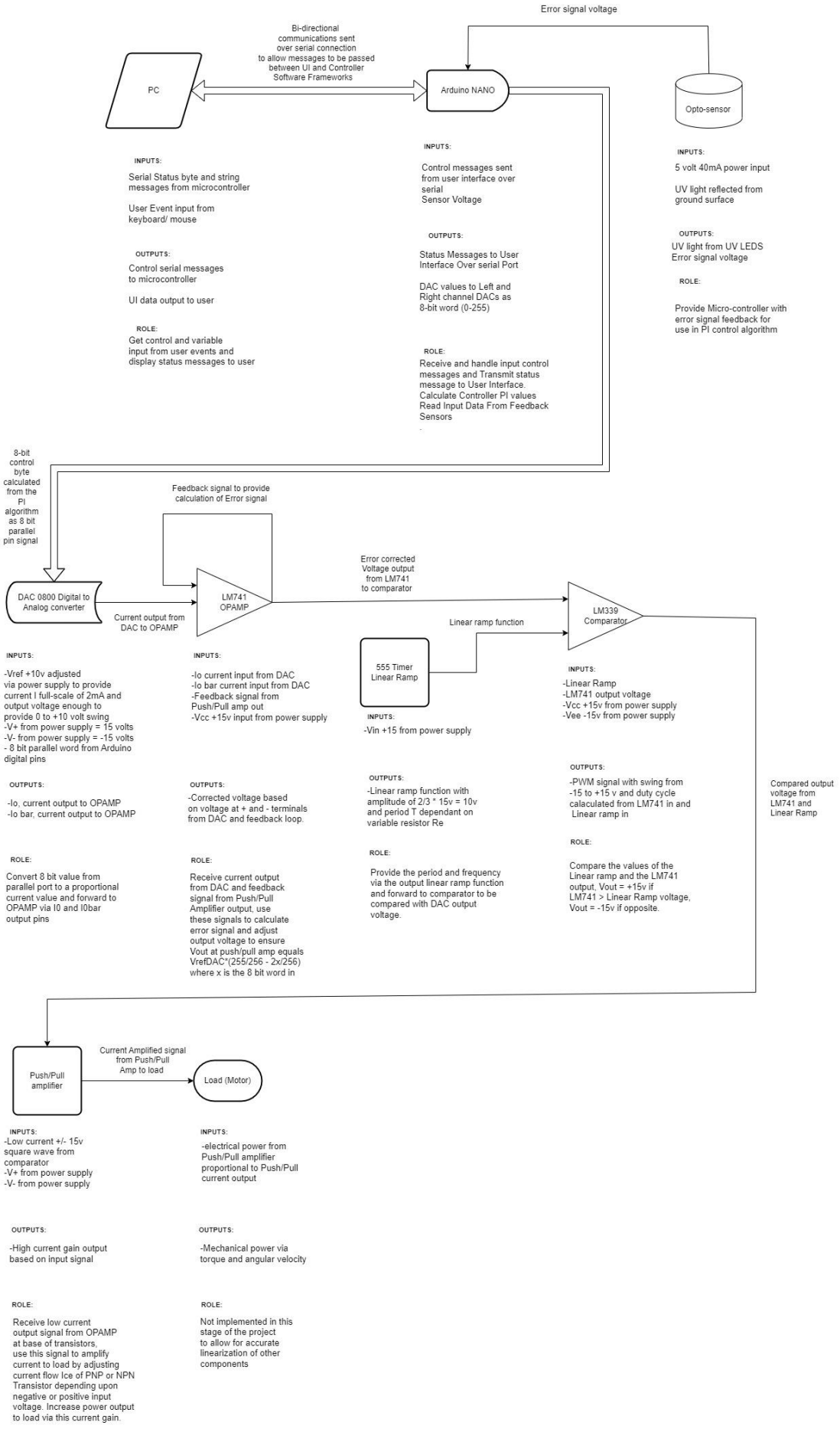


Figure 8: Sub-System Flow Chart and Input/Output Signal Diagram.

2.6. System Control Software Design:

Microcontroller Software Design

State Machine and Data configuration:

The microcontroller software for this project was written in the Microchip Studio IDE with all driver files being created for this project. The software is based on a fully interrupt driven finite state machine architecture. All relevant data of the system mode state, the left and right-side controller PID states and event Boolean flags are stored in a single control struct named State. The format of this struct and the child PI(D) struct and CartMode Enum can be seen in the following images.

```
/*
 * data.h
 *
 * Created: 5/08/2022 3:11:56 PM
 *  Author: ed
 */

#ifndef DATA_H_
#define DATA_H_

#include <stdbool.h>
#include <inttypes.h>
#include "eeprom.h"
#include "usart.h"
#include "PID.h"
#include "timer1.h"
#include "board.h"
#include "adc.h"

// this enum for cart control mode
typedef enum CartMode{
    unavailable = 0x00,
    standby = 0x01,
    manual = 0x02,
    autoLineFollow = 0x03,
    tuning = 0x04,
    saveData = 0x05,
    loadData = 0x06
}CartMode;

// State flags and enum
typedef struct State{
    CartMode cartMode;
    CartMode previousCartMode;
    PID leftPID;
    PID rightPID;
    uint16_t currentMemAddress;

    bool calcPID;
    bool messageRecieved;
    bool messageSend;
    bool cartModeChanged;
    bool saveDataToEEPROM;
    bool loadDataFromEEPROM;
}State;

// function definitions
void data_init(void);
void loadDataFromMem(void);
void saveDataToMem(void);

#endif /* DATA_H_ */
```

Figure 9: Structure of state data passing throughout Microcontroller State machine.

```c
/*
 * PID.h
 *
 * Created: 5/08/2022 3:50:31 PM
 *  Author: ed
 */


#ifndef PID_H_
#define PID_H_

#include <inttypes.h>

#define LEFT_SIDE 0
#define RIGHT_SIDE 1
#define PID_UPDATE_DT 64   // PID update in ms, 64 chosen as power of 2
#define ERROR_SAMPLES_PER_PID_UPDATE 16 // read the sensors 16 times per PID update
#define STOP 127
#define FULL_FORWARD 255
#define SLOWEST_FORWARD 155
#define FULL_BACKWARD 0
#define SLOWEST_BACKWARD 90 // these values may change


// we have 91 different backwards values (0 to 90) , stop at 127 and 100 forward values (155 to 255)
// these will be used as limits in the PID calculation

// create 2 of these structs called leftPID and rightPID
typedef struct PID{
    uint8_t newDACValue; // current control signal (uk+1)
    uint8_t previousDACValue; // uk
    int16_t newError; // ek+1
    int16_t previousError; //ek
    uint8_t kP; // proportional constant
    uint8_t kI; // integral constant
    uint8_t kD; // derivative constant
    uint8_t Dt; // sample time
    uint16_t kIDt; // this is only calculated when Dt or Ki updated to save processing
    uint8_t sensorReadingArray[ERROR_SAMPLES_PER_PID_UPDATE];
    uint8_t sensorReadingArrayIndex;
    uint16_t summedErrorSamples;
}PID;

// function definitions
void init_PID();
void calc_PID();
void send_PID_data_report();
void getError();
void setOutput();
void setDACLeft_Value(uint8_t inDACDValue);
void setDACRight_Value(uint8_t inDACValue);
void defaultPID(bool side);

#endif /* PID_H_ */
```

Figure 10: Structure of the PI(D) data for both left and right channels

Using this State data-structure defined as a global extern variable in main.c all software systems have direct access to all information describing the state of the entire system. This allows the state data to be initialized once before entering the continuous state machine loop and exist in the same RAM block memory location throughout the entire program without the need to copy this data.

The state machine is driven from the value of the CartMode enumeration with state switching performed via a mode switching function defined in main.c. This function controls the settings of all microcontroller peripherals and

interrupts. Upon a serial request from the user interface software first the previous mode is set to the current mode, then current mode is updated to the newly requested mode selection and the stateModeChanged flag is set to true. On the next cycle of the main loop the mode switching function is executed. This function can be seen below.

```
void handle_cartMode_change(){

    // this switch statement to turn off
    // any state related hardware
    switch(state.previousCartMode){
        // this case not used on controller side, added just for PC code clarity
        case unavailable:
        break;

        case standby:
        break;

        case manual:
            setDACLeft_Value(STOP);
            setDACRight_Value(STOP);
        break;

        case autoLineFollow:
            stop_PID_timer();
            adc_off();
            setDACLeft_Value(STOP);
            setDACRight_Value(STOP);
            // stop_PID_timer();
        break;

        case tuning:
            setDACLeft_Value(STOP);
            setDACRight_Value(STOP);
        break;

        case loadData:
        break;

        case saveData:
        break;
    }


    // this switch statement to turn on new
    // state related hardware etc
    switch(state.cartMode){
        // this case not used on controller side, added just for PC code clarity
        case unavailable:
        break;

        case standby:
        break;

        case manual:
        break;

        case autoLineFollow:
            adc_init();
            start_PID_timer();

        break;

        case tuning:

        break;

        case loadData:
        break;

        case saveData:
        break;
    }

    state.cartModeChanged = false;
}
```

Figure 11: State Machine Mode Switching Function.

This mode switching function along with the system being completely interrupt driven allows for the main function to initialize all systems and data on startup while the main program loop only needs to check conditional Boolean flag states for system control requirements. The main function can be seen below.

```
int main(void)
{
    // initialize board
    board_init();
    usart_init(9600);
    data_init();
    timer1_PID_init();
    adc_init();
    _delay_ms(500);

    // enable interrupts
    sei();

    /* Main loop */
    while (1)
    {

        // handle state flags
        // these should be organized in order of
        // importance to program flow
        if(state.messageRecieved){
            usart_message_recieved();
        }

        if(state.cartModeChanged){
            handle_cartMode_change();
        }
        if(state.calcPID){
            calc_PID();
            state.calcPID = false;
        }
        if(state.loadDataFromEEPROM){

        }

        if(state.saveDataToEEPROM){

        }

    }

}
```

Figure 12: Main function and Infinite Loop

Serial User Input and Cart Data Output Control Structure and Protocol:

user interface → cart:

As the microcontroller software is directly instructed on tasks to be performed via user input from the User interface software the system is dependent on user input events. To facilitate the implementation of these control instructions and send data and message feedback back to the user interface an interrupt driven usart communications protocol was created. The controller serial input protocol (user interface → cart) consists of a series of bytes sent over the serial port following the format below:

[controlByte, messageLengthByte, controlMode, byte[], checkSumByte]

Where:

- The control byte is a constant defining the operation to be performed according to the following constants defined in both the user interface and controller software:

```
// serial communications definitions must match C# code
// these are instruction sent in the first byte of the serial
// message
#define BYTE_START 0xFF
#define STRING_START 0xFE
#define EMERGENCY_STOP 0xFD
#define CHECKSUM_OK 0xF1
#define CHECKSUM_ERROR 0x1F
#define PID_DATA_SAVED 0xFC
#define PID_DATA_LOADED 0xFB
#define AUTO_MODE_DATA_REPORT 0xFA
```

Figure 13: Serial Control Byte Constant Definitions.

- The message length byte allows this dynamic message length passing by informing the microcontroller of the incoming message length.
- The cartMode byte is the byte representation of the cartMode Enum setting (defined in figure 5) requested by the user and is used to change the cartMode in the state machine.
- The byte[] represents an array of bytes dependent upon the control mode requested. This array varies in length from 0 bytes for autoLineFollow mode and standby mode to 5 bytes for tuning mode. In tuning mode the byte array passes the following additional 5 bytes to allow for user variable tuning:

[DACLeftValue, DACRightValue, kp, ki, kd]

The only exception to this serial format is if the EMERGENCY_STOP byte is sent as the control byte. This special case is a single byte message that informs the microcontroller to immediately set both DAC value to STOP (128) and enter standby mode. Entering this mode tuns off any active interrupts and timers except the serial comms interrupts to allow for serial communications to remain active.

The receive interrupt handling this data transfer triggers whenever a new byte is received to the serial receive register with these bytes being stored into a receive byte ring buffer. Upon receiving the final byte of the message, the messageRecieved flag is set and the message is handled in the next iteration of the main loop. This interrupt implementation can be seen below.

```
ISR(USART_RX_vect){

    // 0 position used for message error checking
    if(rxPosition == 0){
        recieveBuffer[rxPosition] = (uint8_t)UDR0; // save first byte

        switch(recieveBuffer[rxPosition]){
            case BYTE_START:
                //usart_transmit(0xFF);
                rxPosition++;
            break;

            // emergency stop button hit
            // set DAC values to stopped
            case EMERGENCY_STOP:
                setDACLeft_Value(STOP);
                setDACRight_Value(STOP);
                rxLength = 0; // reset counters
                rxPosition = 0;
                usart_send_string("EMEGENCY STOP\r\n");
                state.previousCartMode = state.cartMode;
                state.cartMode = standby;
                state.cartModeChanged = true;
            break;

            case CHECKSUM_OK: // received conformation of last message OK, clear receive buffer
                rxLength = 0; // reset counters
                rxPosition = 0;
            break;

            case CHECKSUM_ERROR: // last message was error, resend
                rxLength = 0; // reset counters
                rxPosition = 0;
                //usart_transmit(0x1F);
                //state.messageSend = true; // resend message from main
            break;
        }

    }

    // case for second byte to get message length
    else if(rxPosition == 1){
        recieveBuffer[rxPosition] = (uint8_t)UDR0; // save length byte
        rxLength = recieveBuffer[rxPosition]; // update message length
        //usart_transmit(recieveBuffer[rxPosition]);
        rxPosition++;

    }

    // case for bytes higher than 1
    else{
        if(rxPosition < rxLength - 1){
            recieveBuffer[rxPosition] = (uint8_t)UDR0; // save next byte
            //usart_transmit(recieveBuffer[rxPosition]);
            rxPosition++;

        }
        // case for last byte of message
        else{
            recieveBuffer[rxPosition] = (uint8_t)UDR0; // save checksum
            //usart_transmit(recieveBuffer[rxPosition]);
            rxPosition = 0;
            state.messageRecieved = true; // update state for full message done
        }
    }
}
```

Figure 14: Serial Receive interrupt

cart → user interface:

The cart to user interface serial protocol can send 2 message types, the byte message for data transmission and the string message for string message transmission to the user interface. The functions that control this message sending can be seen in the figure below.

```
39  void usart_send_string(char* inString){
40      state.messageSend = true;
41      txLength = strlen(inString) + 2;
42      txPosition = 0;
43      transmitBuffer[0] = STRING_START; // send string start byte
44      transmitBuffer[1] =  txLength; // send length of string
45
46      // load transmit buffer
47      for(size_t ii = 2; ii < txLength; ii++){
48          transmitBuffer[ii] = (uint8_t)*inString++;
49      }
50      UCSR0B |= (1 << TXEN0);
51      UCSR0B |= (1 << TXCIE0); // setup transmit interrupt
52      UDR0 = transmitBuffer[0]; // send string start byte
53      txPosition++;
54  }
55
56  void usart_send_bytes(uint8_t* inDataArray, uint8_t inLength, uint8_t controlByte){
57      state.messageSend = true;
58      txLength = inLength + 3;
59      txPosition = 0;
60      transmitBuffer[0] = BYTE_START;
61      transmitBuffer[1] = txLength;
62      transmitBuffer[2] = controlByte;
63
64      for(int ii = 0; ii < inLength; ii++){
65          transmitBuffer[ii + 3] = inDataArray[ii];
66      }
67
68      UCSR0B |= (1 << TXEN0);
69      UCSR0B |= (1 << TXCIE0); // setup transmit interrupt
70      UDR0 = transmitBuffer[0]; // send first byte
71      txPosition++;
72  }
```

Figure 15: String and Byte Message Sending Functions

The string messaging implementation allows for message text message transmission to the user interface for display in the incoming serial message box (see section on User Interface Software) and takes a C string (char*) as an argument.

The byte messaging implementation is used primarily for the optical sensor and left and right controller DAC value passing for display in user interface during autoLineFollow mode.

The interrupt for message transmission works much like the receive interrupt with the outgoing message first being copied into the transmission buffer. This interrupt then fires on each byte being read from the transmit register and copies the next byte to be transmitted. This interrupt can be seen below.

```
322  ISR(USART_TX_vect){
323
324      // send next byte
325      if(txPosition < txLength){
326          UDR0 = transmitBuffer[txPosition];
327          txPosition++;
328      }
329
330      // all bytes sent, turn off TX interrupt and reset position
331      else{
332          txPosition = 0;
333          txLength = 0;
334          UCSR0B &= ~(1 << TXCIE0);
335          state.messageSend = false;
336      }
337  }
```

Figure 16: Serial Transmission Interrupt

PID timer implementation:

The PID control algorithm is implemented with the 16-bit timer1 available on the ATMEGA 328p.

This timer is configured for a PID update rate of 64ms (chosen as power of 2 to allow for use of bit-shifting operations over multiplication). The timer controls the reading of the UV sensor error input values utilizing the onboard analog to digital converter. It was decided to average these sensor readings over 16 samples to decrease the effect of any outlier readings from the sensors so the compare interrupt was setup to trigger in 64/16 = 4ms. This resulted in a required compare register value of 64000 - 1 = 63999 = 0xF9FF which is less than the top of 2^16 so the timer clock required no pre-scaling to achieve this dt value with the system clock set at 16MHz.

On interrupt trigging a request is sent to the ADC to start a read with the PID calculation being performed every 16 interrupt triggers.

This timer implementation can be seen below.

```
/****************************************
 * PID and sensor reading timer functions
 *
 * PID update rate = 64 ms
 * 16 Error samples taken each PID update per channel
 * Error sample taken every 4ms (interrupt frequency for this timer)
 * error sample frequency = 250 Hz
 * PID update frequency 15.625 Hz
 * F_cpu = 16MHZ
 * F_cpu/F_error = 64000 so we don't need a pre scaler as top is 2^16
 * set interrupt on compare A to 64000-1
 * CTC mode
 ****************************************/

void timer1_PID_init(){
    // CTC mode, TOP is OCR1A
    TCCR1B |= (1 << WGM12);

    // set the compare register to 64000 - 1
    // this is a 16 bit register but compiler handles writing to high and low bytes
    OCR1A = 0xF9FF; // 63999
}


 void start_PID_timer(){

    // start conversion on first error values
    adc_request_read();

    calcPIDcounter = 0;

    TCCR1B &= ~(1<<CS12) | (1<<CS11) | (1<<CS10)); // reset clock select bits
    TIMSK1 |= (1<<OCIE1A); // enable interrupt on compare match
    TCCR1B |= (1<<CS10); // start timer with system clock no divider
 }


 void stop_PID_timer(){
    TCCR1B &= ~((1<<CS12) | (1<<CS11) | (1<<CS10)); // set timer clock select to no source, stop timer
    TIMSK1 &= ~(1<<OCIE1A); // disable interrupt
 }


  // this interrupt updates error samples until we have 16 samples, then we calculate the PID values
 ISR (TIMER1_COMPA_vect){

    if(calcPIDcounter == 15){

        // set the calculate PID flag in the state struct
        state.calcPID = true;
        // reset PID counter
        calcPIDcounter = 0;


    }

    else{
        adc_request_read();
        calcPIDcounter++;
    }
 }
```

Figure 17: PID Control Timer Implementation.

Sensor Feedback ADC Reading Implementation:

As seen in the PID timer implementation the timer interrupt requests a read from the ADC ever 4ms. This request begins a binary FIFO queue operation to read both the left and right sensor values. Once again this is an interrupt driven process with the ADC interrupt trigging on the completion of an ADC conversion. These ADC conversions are quite time consuming compared to many other operations in this system so handling of these with interrupts is

essential to program efficiency. The onboard ADC is a 10-bit device with the results of a conversion being presented in 2 registers, ADCH and ADCL. By default these registers are presented as follows:

ADCH = [X, X, X, X, X, X, bit 9, bit 8]

ADCL = [bit 7, bit 6, bit 5, bit 4, bit 3, bit 2, bit 1, bit 0]

It was decided to read the sensor values with 8-bit resolution to simplify calculations, so the data presentation was changed with the left present control bit in the ADMUX register to present the data as follows:

ADCH = [bit 9, bit 8, bit 7, bit 6, bit 5, bit 4, bit 3, bit 2]

ADCL = [bit 1, bit 0, X, X, X, X, X, X]

Now a single read can be performed on just the high byte with the low byte discarded.

The ADC request read function performs the following operation:

1. Sets the next channel to read flag to the left side channel
2. Sets the ADC multiplexer register to the left side channel
3. Enables the ADC interrupt and requests a read.

When this left side read is complete the interrupt then adds the converted value to the

summedLeftError value, sets the multiplexer and the sensor side flag to the right side and requests a read.

When the right-side conversion is complete the interrupt then adds this value to the summedRightError value, sets the sensor side flag to left side and turns off the ADC interrupt as this set of 2 readings has been completed. This implementation can be seen below.

```
// this function sets up a read request from the ADC
// the conversion ready interrupt will fire when the
// conversion data is complete
// first the left channel will be read and added to the error
// array, then the right channel will be read and the interrupt
// turned off.
void adc_request_read(){

    // reset the array indexes if the last error value has been read

    if(state.leftPID.sensorReadingArrayIndex == 16){
        state.leftPID.sensorReadingArrayIndex = 0;
        state.rightPID.sensorReadingArrayIndex = 0;
    }

    // set the ADC control flag to Left side
    nextADCChannel = ADC_LEFT_CONVERSION_NEXT;

    ADMUX = ((ADMUX & 0xF0) | (LEFT_ADC_PORT & 0x0F)); // setup ADC MUX for input on Left ADC channel

    // enable the ADC interrupt and start conversion
    ADCSRA |= (1 << ADIE) | (1 << ADSC);

}

  // This interrupt working as a binary First In First Out queue
  // The ADC sensor values are read with an 8 bit resolution
  // so the 2 least significant bits are discarded
  ISR(ADC_vect){

    // read the high ADC byte
    adcResult = ADCH;

    // if the left channel is next conversion in line
    if(nextADCChannel == ADC_LEFT_CONVERSION_NEXT){

        // set the read result to the current index
        state.leftPID.summedErrorSamples += adcResult;

        ADMUX &= ~((1<<MUX3)|(1<<MUX2)|(1<<MUX1)|(1<<MUX0)); // clear the left multiplexer bits
        ADMUX = ((ADMUX & 0xF0) | (RIGHT_ADC_PORT & 0x0F)); // setup ADC MUX for read on right channel
        ADCSRA |= (1 << ADSC); // ADC start conversion on right side sensor

        // flip the ADC control flag
        nextADCChannel = ADC_RIGHT_CONVERSION_NEXT;
    }
    else{

        state.rightPID.summedErrorSamples += adcResult;

        ADMUX &= ~((1<<MUX3)|(1<<MUX2)|(1<<MUX1)|(1<<MUX0)); // clear the multiplexer bits

        // flip the ADC control flag
        nextADCChannel = ADC_LEFT_CONVERSION_NEXT;

        // disable the ADC interrupt
        // the 2 error values for this error sample period have been read.
        ADCSRA &= ~(1 << ADIE);
    }

  }
```

Figure 18: Feedback Sensor ADC Implementation

PI Approximation Algorithm Implementation:

As the control algorithm is running on an 8-bit microcontroller it was decided to attempt to perform the calculations using fixed point mathematics to vastly decrease calculation instruction code length. If the calculations can be performed using only 16-bit operations and maximizing the use of bit-shifting operations in place of multiplication and division the controller update frequency could be vastly increased leading to a lower feedback sensor to drive axis lookahead distance. It was thought this may improve the performance of the control system by a large margin. Note that as the project encountered hardware issues with the feedback sensor system and this is a first attempt at implementing a fixed-point control algorithm this implementation was unfortunately never completed or tuned and still likely contains integer overflow/fixed point resolution errors. The algorithm design so far set the following desirable traits to pursue:

1. A power of 2 controller update rate, 64ms chosen.
2. Error samples averaged over a power of 2 samples, 16 samples chosen for a feedback sensor reading rate of 4ms.
3. Ki and Kp values represented as single byte values between 0 and 255.
4. Removal of any unnecessary calculation steps from the algorithm

The first thing that was noted when looking at the control algorithm formula:

$$u_{k+1} = u_k + k_P(e_{k+1} - e_k) + k_I \Delta T(e_k)$$

Was that the ki x dt multiplication only needs to be performed when the ki value is changed, so this multiplication step was removed from the equation and calculated on when the ki value is changed. This value is then stored in the kiDt variable in the PID structs for when calculations are performed.

The steps to this process are as follows:

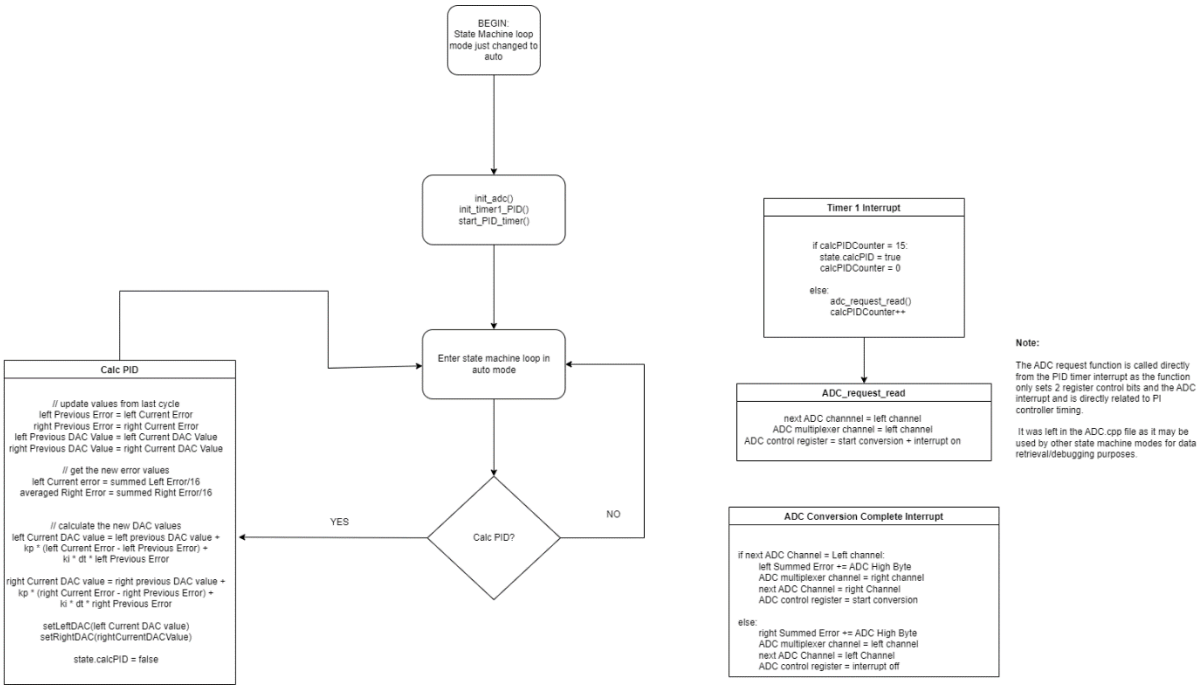
1. The previous error values are set to the current error values (this current error being the error from the previous iteration)
2. The previous 8-bit DAC value is set ton the current DAC value;
3. The 16 summed error readings as divided by 16 using a right shift (summedError >> 4) and the current error value is set to this averaged error
4. The kp term for both sides is calculated using kp*(current error – previous error)
5. The ki term for both sides is calculated using kidt*(previous error)
6. The fixed-point calculated terms are scaled by the fixed point scale value (unfortunately undetermined at this time due to lack of testing/tuning)
7. The terms are summed with the previous DAC value using previous DAC value + ki term + kp term
8. The DAC channels are set to these new control values
9. A PID data report message is sent back to the user interface software to provide user feedback of variable states
10. The sensor read count is reset to 0 for the next 16 error samples to be read in the next PI update.

This calculation function as it currently stands in an untested and untuned state can be seen below.

```
void calc_PID(){

    // copy old error and duty cycle values into previous variables
    state.leftPID.previousError = state.leftPID.newError;
    state.rightPID.previousError = state.rightPID.newError;
    state.leftPID.previousDACValue = state.leftPID.newDACValue;
    state.rightPID.previousDACValue = state.rightPID.newDACValue;

    // divide the summed error samples by the 16 samples this PID update
    uint8_t averagedLeftErrorSample = state.leftPID.summedErrorSamples >> 0x04;
    uint8_t averagedRightErrorSample = state.rightPID.summedErrorSamples >> 0x04;

    pidMessageArray[0] = averagedLeftErrorSample;
    pidMessageArray[1] = averagedRightErrorSample;

    // zero the summed errors
    state.leftPID.summedErrorSamples = 0;
    state.rightPID.summedErrorSamples = 0;

    // left side
    // kp term:
    int16_t l_kp_term = (int16_t)((state.leftPID.kP * (state.leftPID.newError - state.leftPID.previousError)));
    int16_t l_ki_term = (int16_t)(state.leftPID.kIDt * state.leftPID.previousError);
    int16_t l_summed_terms = (l_kp_term + l_ki_term);
    int8_t l_scaled = l_summed_terms >> 8; // divide the output by the fixed point scaling term, in this case 64
    state.leftPID.newDACValue = (uint8_t)(state.leftPID.previousDACValue + l_scaled);

    pidMessageArray[2] = state.leftPID.newDACValue;

    // right side
    // kp term:

    int16_t r_kp_term = (int16_t)((state.rightPID.kP * (state.rightPID.newError - state.rightPID.previousError)));
    int16_t r_ki_term = (int16_t)(state.rightPID.kIDt * state.rightPID.previousError);
    int16_t r_summed_terms = (r_kp_term + r_ki_term);
    int8_t r_scaled = (uint8_t)(r_summed_terms/64);
    state.rightPID.newDACValue = state.rightPID.previousDACValue + r_scaled;

    pidMessageArray[3] = state.rightPID.newDACValue;

    //state.rightPID.newDACValue = 127;
    setOutput();

    messageTimerByte++;

    // use this if statement to change serial report update rate
    if(messageTimerByte == 10){
        send_PID_data_report();
        messageTimerByte = 0;
    }

}
```

Figure 19: Control Algorithm Implementation.

A flow chart of the PID program flow in auto line follow mode can be seen in the diagram below.



User Interface Software Design

The user interface software system is primarily an event driven system relying on user input events to control program flow. As the system is designed to be implemented alongside the microcontroller software elements of the interrupt driven state machine controller paradigm are incorporated into this user interface software as they must be consistent between both systems. These include:

1. The definition of the microcontroller CartMode enumeration for state machine control
2. The Serial Control Byte definitions

The user interface software design had the following design requirements:

1. Bi-directional serial communication protocol with the cart software to provide state and variable feedback to user and provide user with cart control functions.
2. Follow an easy to understand and use interface for user interaction with all controls designed to accept only valid inputs and only be accessible when appropriate to limit the requirement of exception handlers in program software. Design should only implement exception handling when absolutely necessary, generally these situations relate to unpredictable events such as serial port cable removal. C# allows internal control of range limiting of user input values for all input widgets.
3. Implement means for emergency stopping of the moving components of the cart in a clear and always accessible means.
4. Allow user to set both DAC and control variables (kp, ki, kd) values on the cart by sending of serial messages.
5. Set the current cart mode easily from the interface with only relevant controls available to the user in each mode.

The following diagram shows the layout of the designed user interface with each menu explained in the corresponding text block.

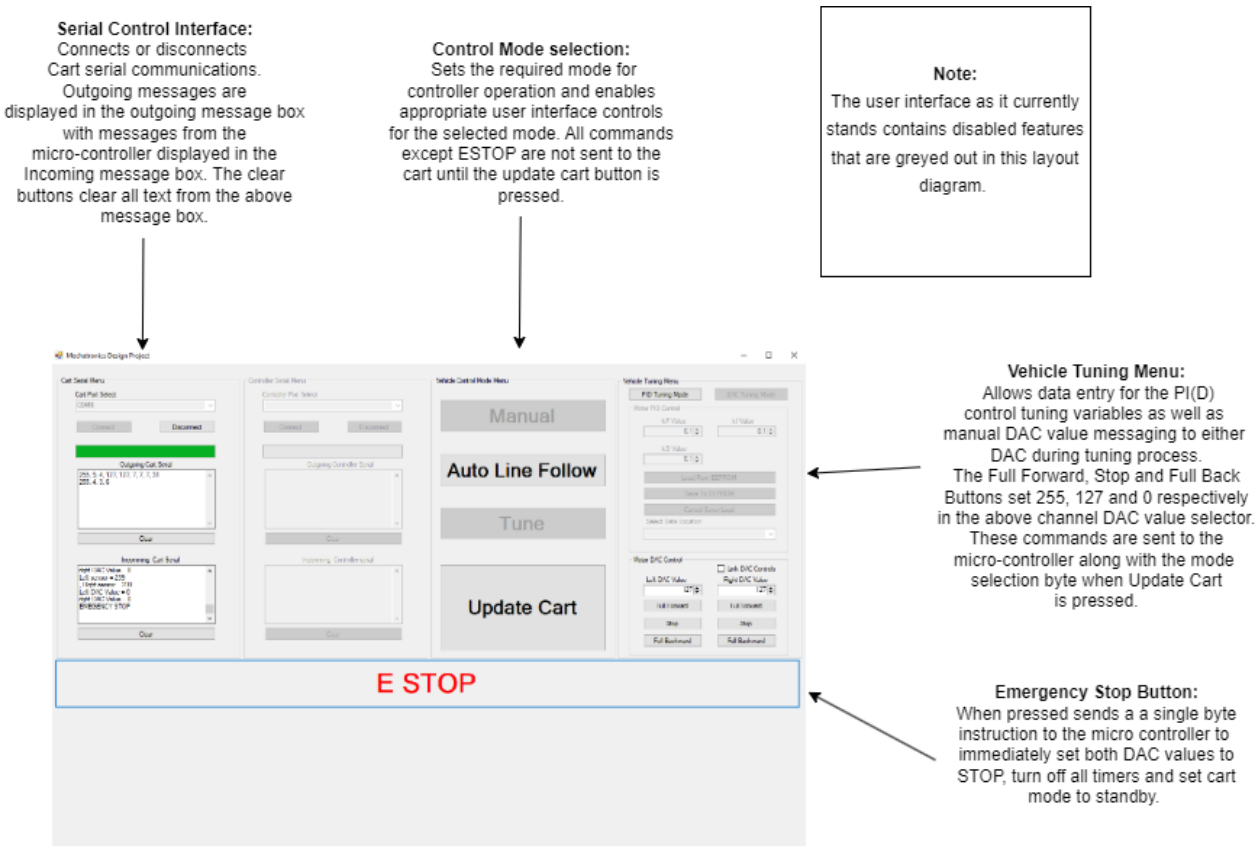


Figure 20: User interface functional description.

The following set of diagrams shows the controls accessible to the user in each cart mode:

Before Serial Connection Established:

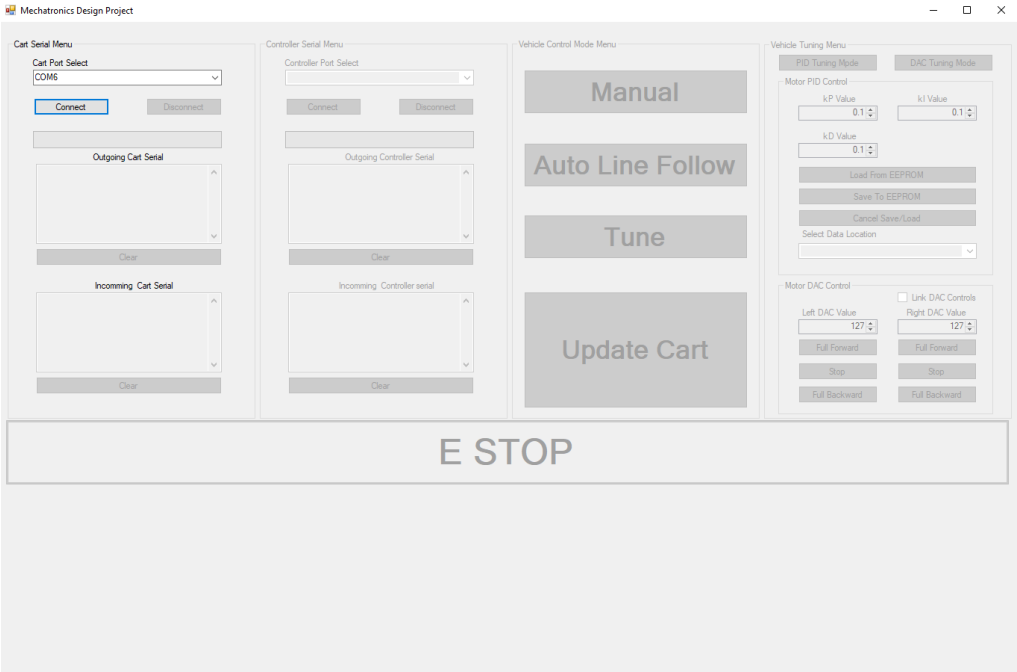


Figure 21: User Interface Before Serial Connection

DAC Tuning Mode:

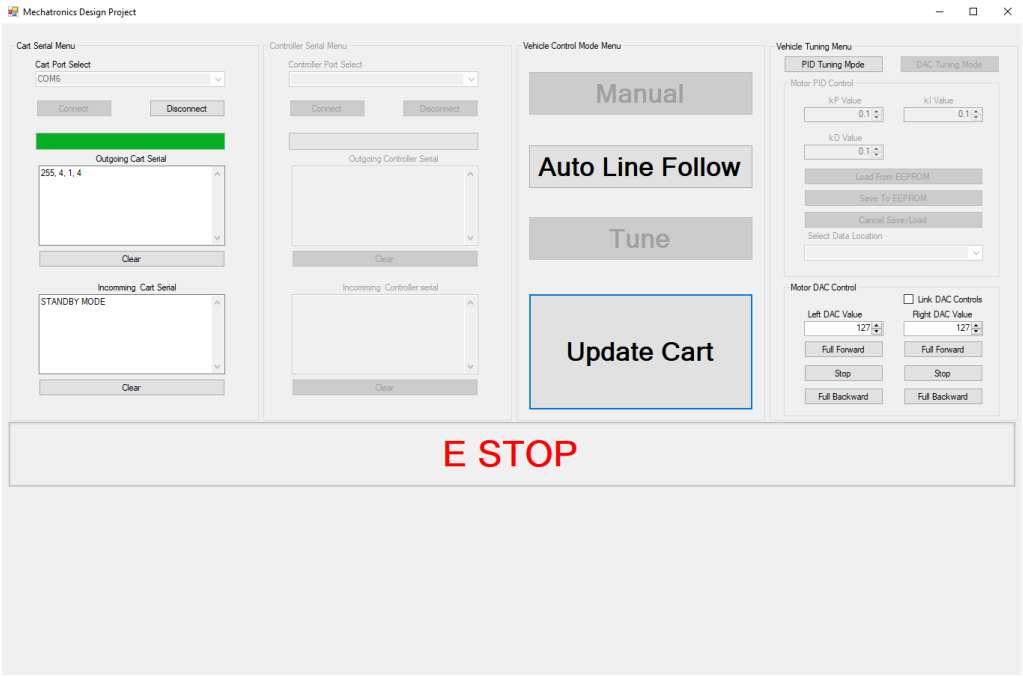


Figure 22: User Interface in DAC Tuning Mode.

PID Tuning Mode:

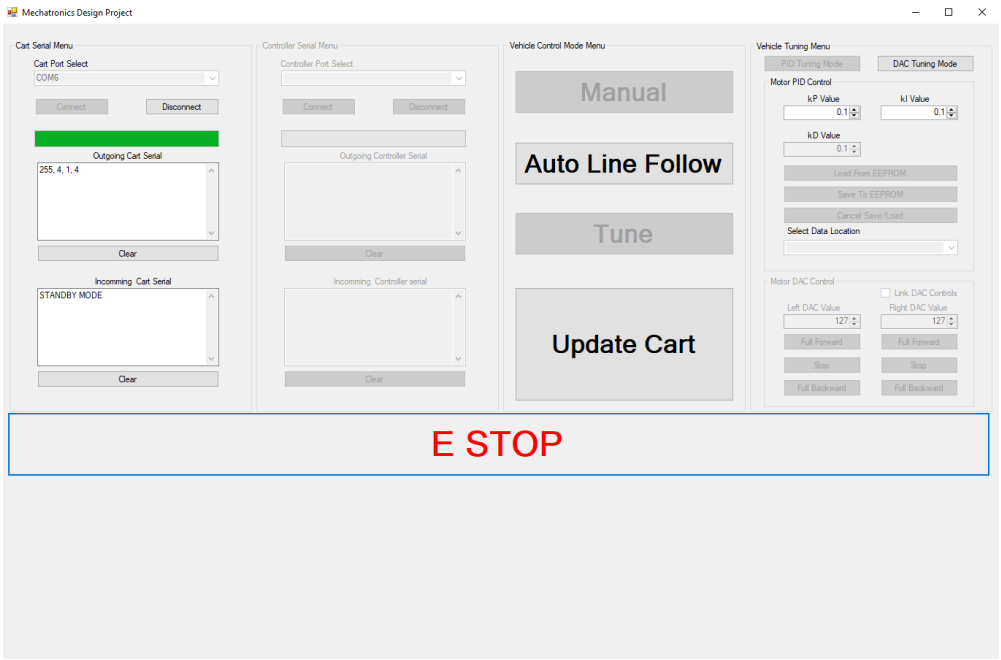


Figure 23: User Interface in PID Tuning Mode

Auto Line Follow Mode:

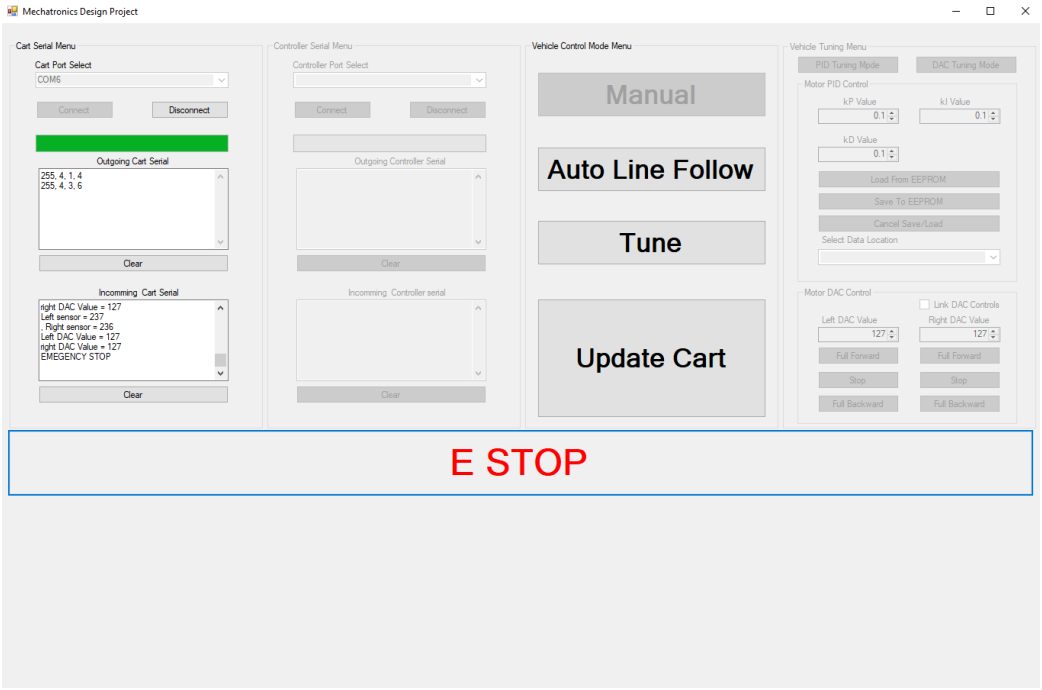


Figure 24: User Interface in Auto Line Follow Mode Showing Variable Feedback to User

2.7. Alternative PWM Waveform Generation Routes:

2.7.1. Microcontroller Timer based PWM Waveform Generation:

The PWM waveform generation for this project could be achieved using the PWM waveform generation of the ATMEGA 328p microcontroller. This would require setting up one of the onboard timers available on the ATMEGA 328p to the desired frequency and using an interrupt on the compare register to trigger a pin bit-flip on interrupt triggering. This signal could then be fed straight to the comparator positive terminal with the negative terminal set to half of the microcontroller pin voltage.

2.7.2. Dual 555 Timer Based PWM Generation:

This method of PWM waveform generation would require the replacement of the linear ramp/DAC waveform generator with dual 555 timers, with 1 running in astable mode generating the system “clock” frequency and the other in monostable mode with DAC input to the control voltage pin to set the output duty cycle. For An in-depth look at this waveform generation system see task 2 of this report.

2.8. Alternative Use-Cases for this System:

The analog PWM generation via the DAC/Linear Ramp/comparator circuit used in this project implementation could be used in many different motor/actuator control systems with the applications for alternative use-cases possible being greatly expanded if the sensor input and control algorithm sub-systems could be implemented in such a way as to allow for this error feedback and control system could be easily adjusted to match any type of feedback input sensor. For example, if the system could support RPM or angular velocity rotary encoder input instead of optical reflection for error input the system could finds many uses in motor speed control systems. The system could also be adapted to applications like temperature control.

2.9. Performance and Limitations in Project Application:

The performance of this project was unfortunately hindered by a hardware issue that went unresolved until just before the final testing of the unit. Voltage and current supplied to the 2 UV LED sensors was supplied through an output terminal on the PCB with the LEDs requiring an input current of ~40mA each to be supplied for optimal operation. The 2 sensors were tested individually and worked as expected with predictable ADC values sent to the PC software but when connected simultaneously to the power supply terminal results were very unpredictable and seemed like random noise. Upon investigation it was found that the voltage between the positive and ground input terminals would drop below the required 5-volt supply voltage with both LEDs connected and fluctuate between ~1 to 2.5 volts. The current was then directly measured across this terminal with nothing connected with a DMM and was found to only be supplying 30mA, which explained the unpredictable behavior. This resulted in the implementation of the control algorithm being unable to be tested or tuned without this feedback mechanism. While this resulted in failure to have a working system on test day it was an important lesson in electrical system diagnosis.

2.10. Future Improvements:

Future improvements to the designed system used in this project were discussed throughout the project design and implementation phases. Discussed improvements included:

- Addition of “Auto Line Seek” mode to drive in a spiral pattern until a line is detected and then automatically change mode to follow the found line.
- Addition of manual joystick control mode implemented on a second microcontroller and serial connection to allow single stick manual joystick control by the operator of the cart.
- Addition of control variable saving to microcontroller EEPROM to allow for auto variable restore functionality on startup.
- Moving of all serial communication functions to class-based system to allow for simpler serial messaging functionality
- Implementation of heartbeat signal between PC control software and cart to automatically place cart in ESTOP mode on signal loss.

3. Task 2:

3.1. Problem Domain:

This task requires the design and simulation of a circuit to produce a variable duty cycle PWM signal to be applied to the base if a Bi-polar junction transistor (BJT) operating as an amplifier to a load. The effective voltage seen at the load will be the average voltage of the PWM square wave output generated by the BJT emitter. The method of approach to this problem is to use a circuit comprised of twin 555 timers and a comparator as the signal source to the base of the BJT. The first timer is used in astable mode to generate a repeated pulse signal (Low active) to be used as the system oscillator. This signal will be feed into the trigger pin of the second timer configured in monostable mode. This pulse signal will dictate the system base frequency. The second timer has a voltage divider setup via a variable resistor to the control voltage pin. Tuning of this resistor changes the duty cycle of the output signal. This signal is then supplied to the positive terminal of a comparator and compared with $0.5(V_{cc\ Timer\ Circuit})$ generated by a voltage divider to the negative terminal of the comparator. The comparator V_{cc} is connected to the (likely higher voltage) BJT Collector voltage to allow voltage matching of the amplifier circuit. This setup allows any voltage in the 555-timer input range (4.5v to 18v) to be used for $V_{cc\ Timer\ Circuit}$ and any voltage in the 393 comparator input range (2v to 36v) to be used for $V_{cc\ Amplifier\ Circuit}$. Datasheets for all active components used can be found in the appendix of this report.

3.2. Astable Oscillator Circuit:

First a frequency was chosen considering the response time of the 393 comparator (1uS) and the preference to run the system above audible frequency to eliminate audible system noise during operation. Maximum audible frequency ~20kHz is the lower limit with higher frequencies increasing the effect of the response time of the comparator on waveform rise time and thus average PWM wave voltage. Taking both these factors into account a frequency of ~35kHz was selected.

$$T = \frac{1}{f} = \frac{1}{35x10^3} = 28.6\mu s$$

The astable circuit layout and associated equations from the 555-timer datasheet can be seen in the figure below:

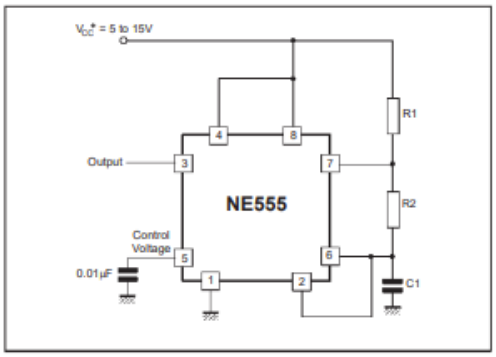


Figure 14 shows actual waveforms generated in this mode of operation.
The charge time (output HIGH) is given by :

eq 1 $t_1 = 0.693\ (R_1 + R_2)\ C_1$

and the discharge time (output LOW) by :

eq 2 $t_2 = 0.693\ (R_2)\ C_1$

Thus the total period T is given by :

eq 3 $T = t_1 + t_2 = 0.693\ (R_1 + 2R_2)\ C_1$

The frequency ofoscillation is them :

eq 4 $f = \frac{1}{T} = \frac{1.44}{(R_1 + 2R_2)\ C_1}$

and may be easily found by figure 15.
The duty cycle is given by :

eq 5 $D = \frac{R_2}{R_1 + 2R_2}$

Figure 25: Astable 555 Timer Circuit and Equations

$$let\ C_1 = 1nF,$$

$$from\ eq\ 3:\ T = 0.693(R_1 + 2R_2)C_1 \rightarrow \frac{T}{0.693C_1} = R_1 + 2R_2 \rightarrow \frac{28.6\ x\ 10^{-6}}{0.693(1\ x\ 10^{-9})} \approx 41k\Omega$$

$$from\ eq\ 5:\ DC = \frac{R_2}{R_1 + 2R_2}\ and\ DC\ (Low\ Active) = \frac{t_2}{T},$$

$$to\ achieve\ good\ control\ resolution\ a\ small\ value\ of\ \frac{R_2}{R_1 + 2R_2}\ is\ desirable,$$

$$\therefore Values\ chosen\ from\ resistors\ on\ hand, R_1 = 38k\Omega, R_2 = 200\Omega, \quad for\ DC\ (Low) \approx 0.5\%$$

The circuit diagram for this oscillator can be seen in the figure below:

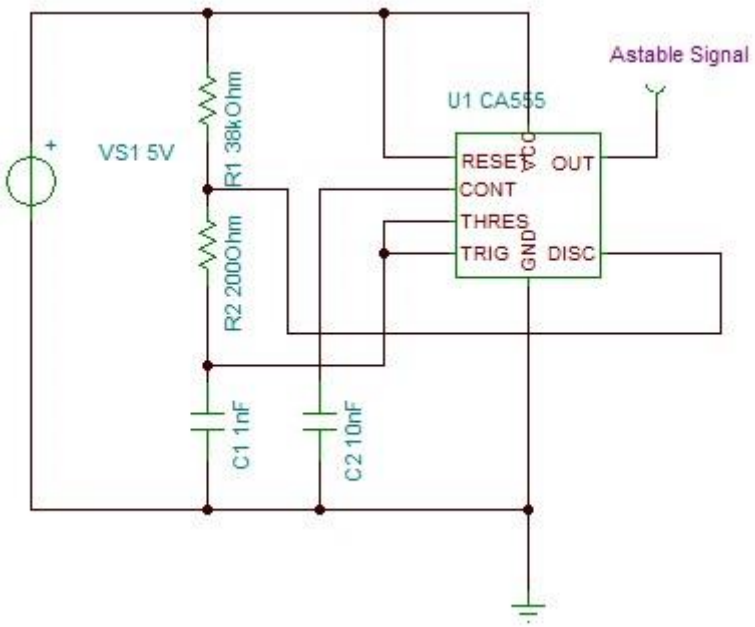


Figure 26: Astable Circuit Diagram.

3.3. Monostable PWM generation Circuit:

The astable low active “clock” signal is then fed into another 555-timer setup in monostable mode.

This 555 has the variable resistor control mechanism P1 to allow for mechanical input from the operator. This variable resistor is connected to the control voltage pin of the timer and acts as a voltage divider to adjust this pin voltage. As this voltage is raised the duty cycle of the circuit will increase. The 2 resistor R5 and R6 are provided to produce a 0.5 voltage divider to the amplification circuit. The value for R3 and C3 were chosen using the equation:

$$t_{high} = 1.1(R_3C_3)$$

taken from the 555-timer datasheet. A value of 1nF was chosen for C3 to match the astable circuit and a value of 2.5kΩ for the variable resistor and 10kΩ for R3 showed good results in simulations.

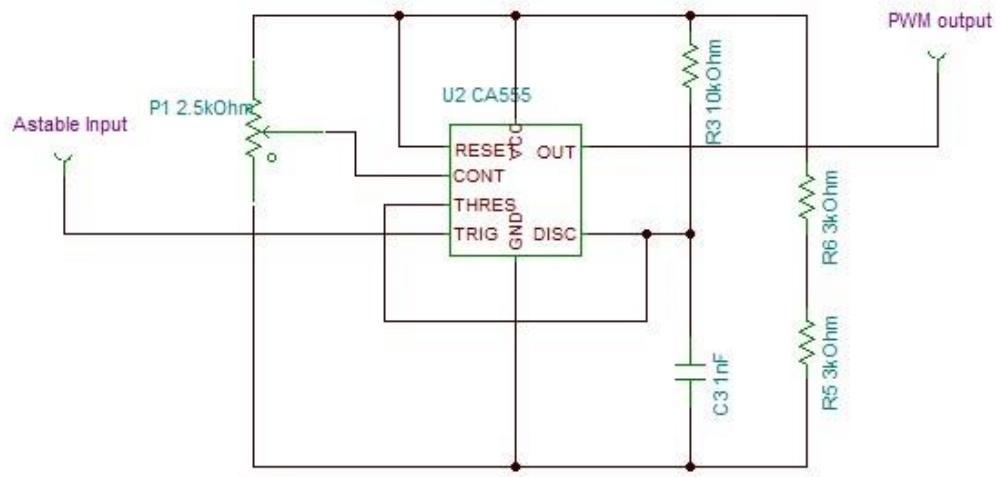


Figure 27: Monostable 555 timer circuit.

3.5. Addition of Current and Voltage Amplifier, Entire Circuit Diagram and Simulated Results:

To achieve high input power to output power amplification and allow low voltage operation of the PWM signal generation a 2-stage amplification circuit was implemented consisting of an LM393 Comparator and a TIP31 NPN BJT power transistor. The comparator provides voltage amplification of the input PWM signal from the monostable 555 timer circuit and compares this to 0.5 x $Vcc_{signal\ generator}$ generated via the voltage divider R5-R6. If the PWM input signal is below

0.5 x $Vcc_{signal\ generator}$ the output will be driven to ground of the load circuit, if the input PWM signal is greater than this value the output voltage will be driven to $Vcc_{load\ supply}$. The current of the circuit is then amplified via the TIP31 NPN transistor to supply this amplified power to the load. A diagram of the full circuit can be seen below.

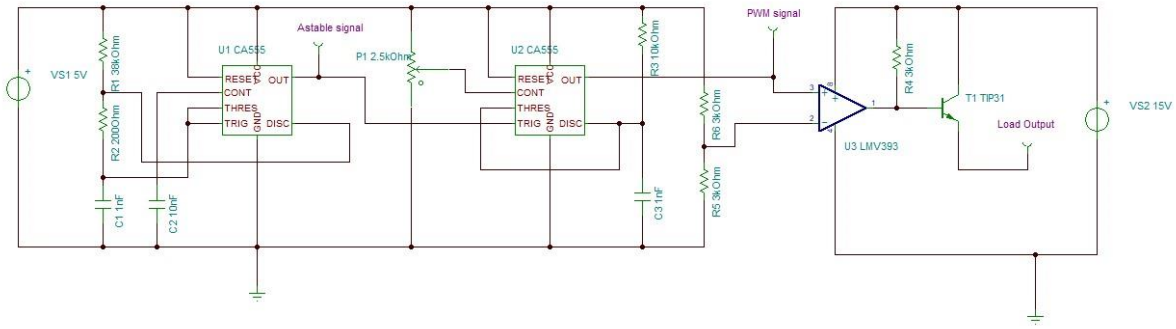


Figure 28: Entire Variable Duty Cycle and Voltage and Current Amplifier Circuit

The next set of diagrams shows the simulated results of the circuit as the % value of the variable resistor is adjusted (Note, not the PWM duty cycle percentage). Simulations were conducted at 0, 10, 50, 90 and 100%. The 0 and 100% simulation results show a small fluctuation in modelling results but do settle at the appropriate values (note the scale on the Y axis in these plots).

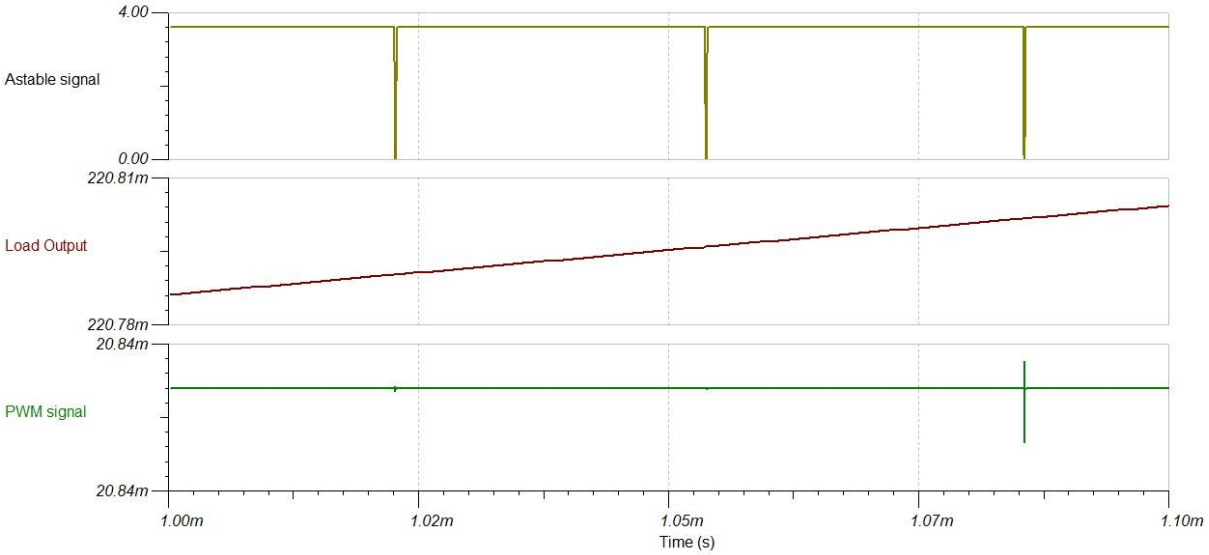


Figure 29: Output at 0% Variable Resistor Value

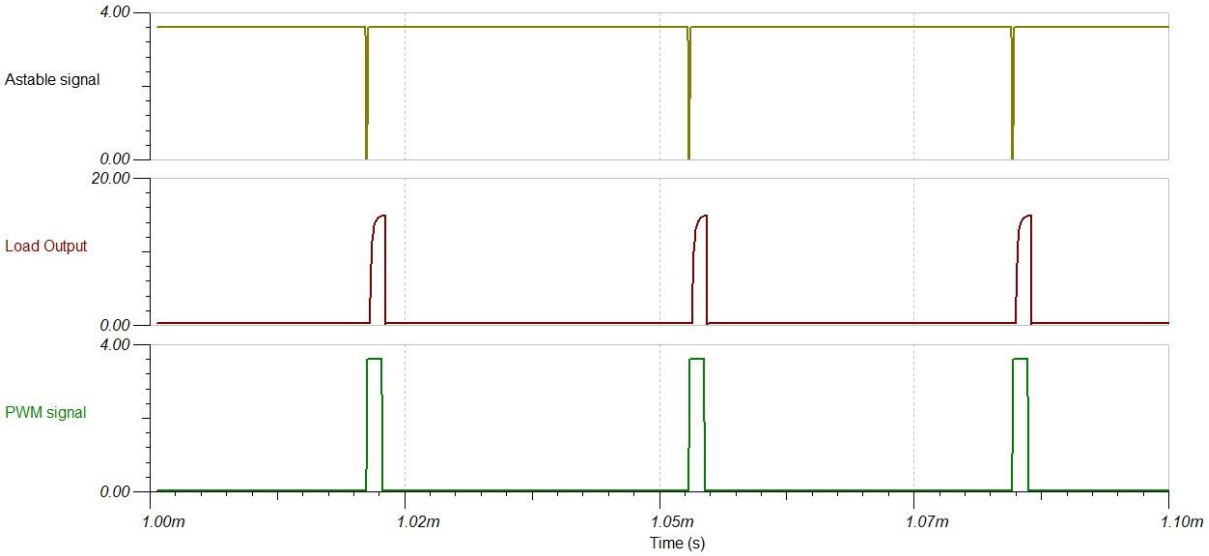


Figure 30: Output at 10% Variable Resistor Value

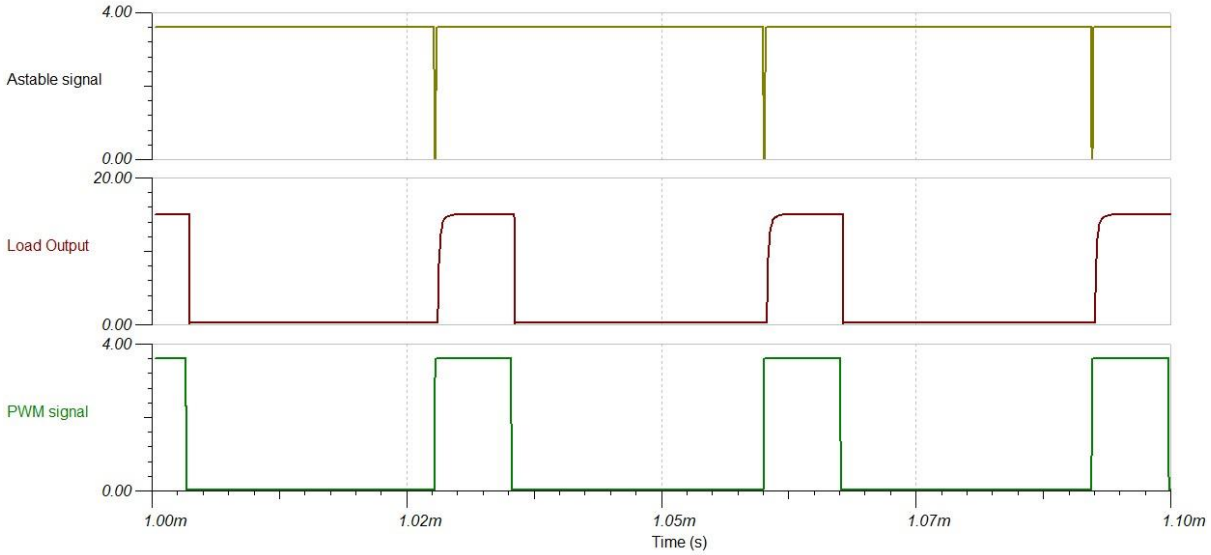


Figure 31: Output at 50% Variable Resistor Value

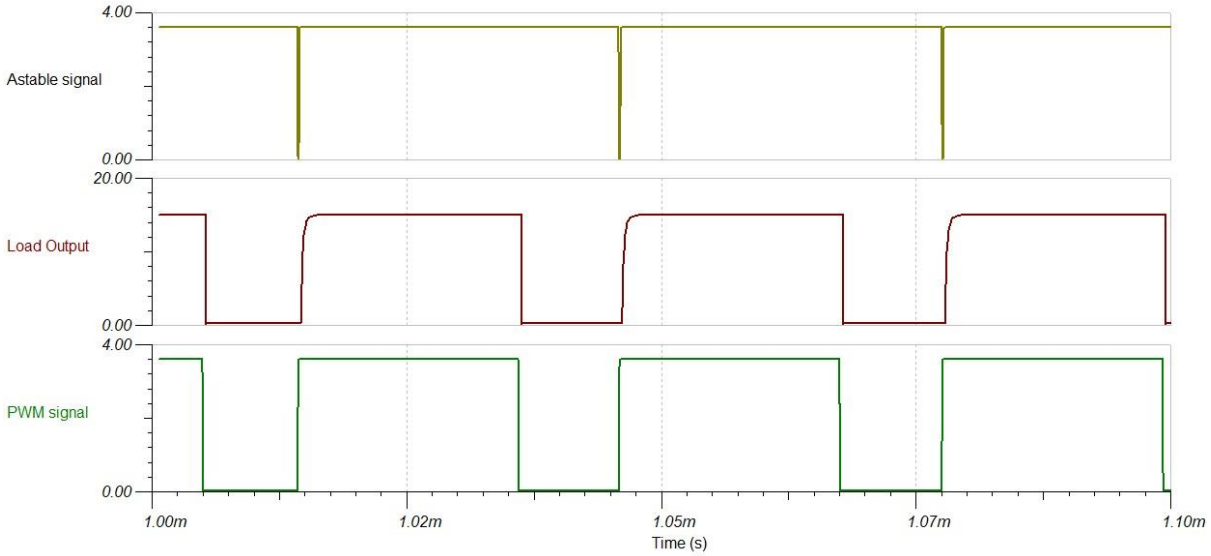


Figure 32: Output at 90% Variable Resistor Value

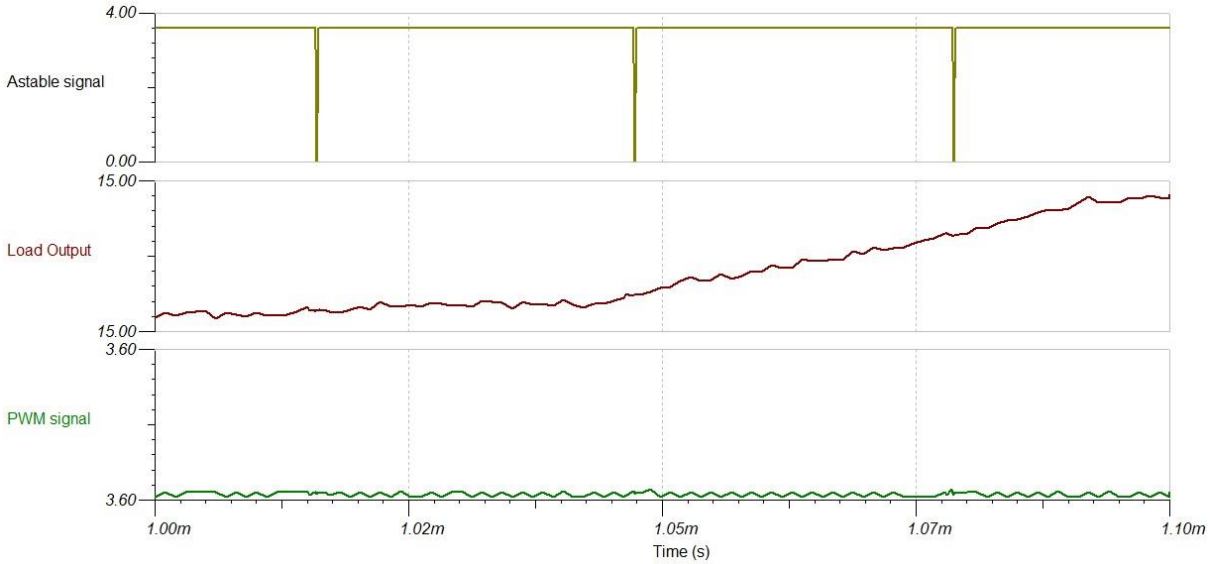


Figure 33: Output at 100% Variable Resistor Value

The results obtained through this simulation show that the system design can achieve the requirement of 0-100% duty cycle adjustment via tuning of the control variable resistor. The selection of ~35kHz waveform frequency seems an appropriate compromise between the low frequency cut off requirement of above audible frequency of ~20kHz and the impact of the comparator rise time on overall waveform generation. The worst case for this effect in these diagrams can be seen in the 10% variable resistor simulation results as the signal response rise time shows non-linearity. This effect on the overall waveform will increase as the frequency increases leading to lowering of the actual average voltage seen at the load as frequency is increased. Another thing to note about the generated PWM signal is that the control input from the variable resistor is non-linear with respect to the output PWM duty cycle generated with % increase in duty cycle increasing exponentially at higher variable resistor range. This effect can best be evidenced in the 90% variable resistor simulation results with the actual generated PWM duty cycle being closer to 75% so the last 10% in resistor adjustment produces a change in duty cycle of 25%.

4. Conclusion:

This report has detailed the project completed in the mechatronic design unit and the proposed circuit to solve the variable duty cycle via potentiometer control task. It was unfortunate the unit project was not in a te3stable state on the testing day however a great deal was learned by both members of the group in the fields of electronic design, analysis and debugging and on embedded software development during the course of the unit.

Bibliography

Embedded.com. 2006. "Embedded.com." *Make a PI controller on an 8-bit micro.* https://www.embedded.com/make-a-pi-controller-on-an-8-bit-micro-2/.

Microchip. 2015. *ATMEGA 328P Datasheet.*

TT Electronics. n.d. *OPB704 Reflective Object Sensor Datasheet .*