NATCAR

Department of Mechanical and Aerospace Engineering

ME106 Fundamentals of Mechatronics

Andrew Nguyen

Ryan Nunn-Gage

Amir Sepahmansour

Maryam Sotoodeh

May 16, 2006

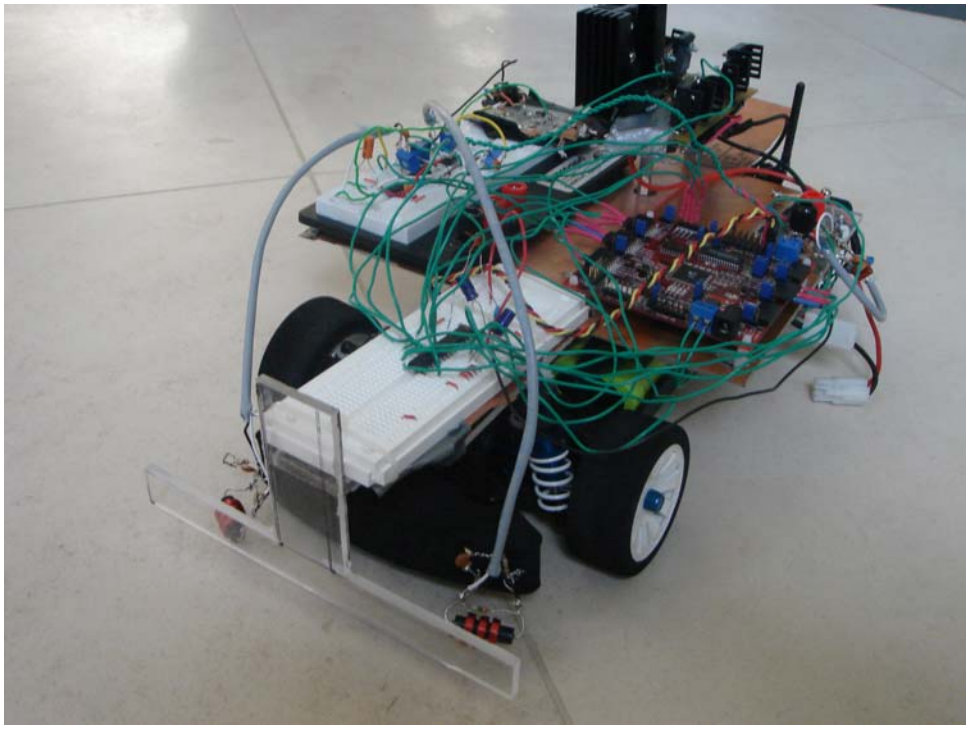
Table of Contents

Summary

As a group we chose to enter in the NATCAR competition for many reasons. The NATCAR is project, which integrates many different aspects of engineering, such as analog design, digital design, system integration, and mechatronics. In order to have a successful racecar, we needed to design a car that will be able to navigate a preset course as fast as possible.

The NATCAR competition required us to build a fully autonomous racecar, which meets the given constraints of the competition. For the analog portion of the car we designed a sensor network, which would inductively pick up the signal from the track. In addition, we needed additional circuitry to regulate the power that would be used for the analog components. In order to autonomously control the car we used a CEREBOT microcontroller to program the steering and speed control.

After months of hard work on the project, we successfully built a car that will follow a wire autonomously. We learned a lot about the analog design during testing of the racecar. We learned that the analog design could not handle the amount of current flowing through it and after approximately five minutes of the car running the analog components would get so hot that all the chips would burn out. In addition, we learned a lot about how to program microcontrollers.



Introduction

The NATCAR competition is an autonomous 1/10th scale car race that is held in the spring every year. The competition is held on the UC Davis campus. The autonomous car must traverse through the track as fast as possible without hitting any border cones.

The key to this competition is simplicity. The problem is just trying to get a complicated project such as this to work to its best before adapting fancier solutions to the car. Keep it simple. The closest competition in the past has been UC Berkley and California State University of Sacramento. Competition has been tight sometimes the teams have been within .5 seconds to 122 seconds of each other.

The NATCAR project was a comprehensive project that included skills of all the members in the group, which included analog and digital design, system integration, and testing. We had to work together to get all of our components integrated successfully in

order to build a successful car. The car uses inductive sensors to follow a wire with a 100mA and 75 kHz signal. A microcontroller controls the wheel speed and steering angle of the car. We were able to complete the car successfully whit the analog sensing portion of the car on a breadboard.

Analog Design

Power Supply

The NATCAR rules specify that the car must run off of one 7.2V NiCAD battery to supply the power for both the RC car and the digital and analog circuits added. In order to adequately power the analog and digital circuitry a +5V and -5V rail must be established.

The positive and negative power supplies were chosen to conserve overall power consumption for the circuits. Since the induced RMS voltage of the signal varies the power consumption of the sensors will end up being equal to,

$$P_{SENSOR}=(V_{RMS}^2)/R_{LOAD}$$

Whereas, if we chose a 0 to 10V power supply, the sensor would have to bias the signal with +5V to keep it from being clipped by the power rails. In that scenario the power being consumed will end up being,

$$P_{SENSOR}=(V_{RMS}^2)/R_{LOAD}+(5^2)/R_{LOAD}$$

Since the battery's voltage will change as it goes from being fully charged to being empty a solution was needed to provide a consistent 5V source to the rest of the circuit. A Low Drop Out regulator or commonly referred to as "LDO" was used as it will step down the battery's voltage and regulate it to a desired 5V. The LDO will be more than sufficient to

power the circuits given that it outputs a maximum 200mA. In the event that circuits require more current 200mA multiple LDOs can be connected to supply the required current.

To generate the -5V supply, a charge pump was used to convert the +5V output of the LDO and invert it to a -5V supply. A charge pump works by utilizing strategically placed switches to steer the charging and discharging of a capacitor to obtain the desired output. In order to maintain the charge over the capacitors, the switches must be constantly turned on and off with an internal oscillator. Since the oscillation frequency may introduce noise or EMI to the rest of the circuit, charge pumps are available at different switching frequency. In our application the MAX889T inverting charge pump was chosen since its frequency of oscillation is 2 Megahertz, given that our sensor/filter networks cutoff frequency is at 100kHz, it will reasonably attenuate any noise from the charge pump itself.

With the +5V and -5V power supplies established, the NATCAR motor will be driven directly off the battery given that it requires about 10-20A to operate given the load applied to it. The microcontroller will also be driven directly off the battery, because the microcontroller has a built in voltage regulator. The microcontroller will be supplying power to the servo.

Sensor and Filter Network

The designing of the input sensors is crucial to the success of the NATCAR. The track consists of a wire carrying a sinusoidal 100mA$_{RMS}$ signal at 75 kHz, covered in white tape. The designer can choose to sense the track either optically or inductively.

Our group chose to sense the track inductively primarily for two reasons. The first being that since we know the frequency of the signal is 75 kHz we can filter out unwanted noise. Second, is if the car veers off the track, it can still sense the signal and return to the track.

The design begins with the selection of a good inductor to sense the track. A cylindrical inductor with a strong ferrite core is best suited for this application. Toroidal and air core inductors should be avoided, as they are not suitable for this application. For our application a 1mH RF choke with a ferrite core was chosen. This inductor provided very good results in sensing the signal as with a 3M ohm shunt resistor, the inductor was able to pick up the signal with a magnitude around the order of 100mVp-p.

However, the inductor was also picking up other high frequency components. Originally intended as a quick fix, a RC lowpass filter was added in parallel to the RL network to filter out the high frequency noise, the RC components of R=1kOhm and C=1.5nF would exhibit a -3dB roll off at approximately 106k. We ended up utilizing this design as we realized this network creates a 2nd order bandpass filter, and through simple testing of this sensor in the lab, it was able to filter out high frequency component. This network was not analyzed other than to obtain the order of the circuit, since modeling is quite

complex, being that the inductor both represents the source and frequency component in this circuit.

With the sensor and filter circuit design, the signal was then amplified with an op amp utilizing a non-inverting configuration. Due to mismatches in the components, R1 was chosen as a potentiometer so that the sensors could be calibrated to match each other. R2 was chosen to be 100k ohm, so that by tuning the inductor we can achieve a gain of approximately 25-40(V/V), which works out well with our filters since it is know that a 2nd order filter attenuates on the order of 40dB in the stopband region. The schematic of the filter is shown below.

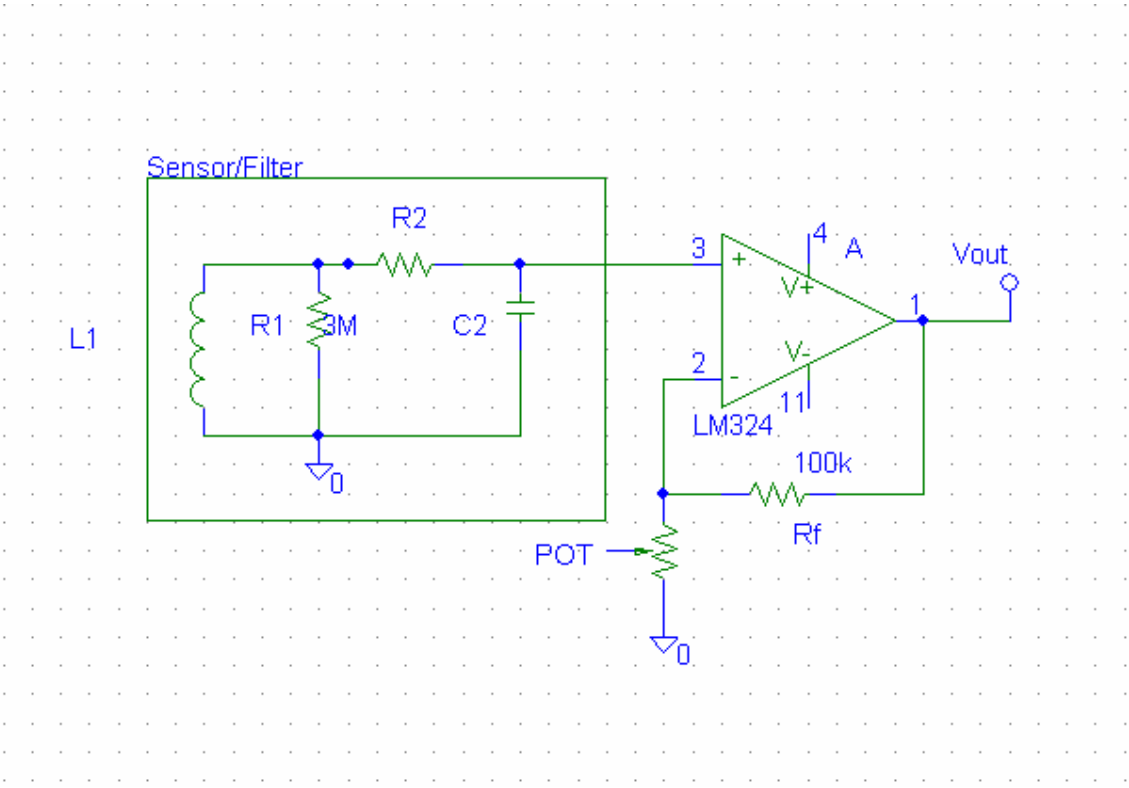


Figure 1: Sensor/Filter Design

H-Bridge Design

The expected current draw for the selected motor was around 30 Amps. The H-Bridge components have a maximum power handling capability of 60 Amps. Figure 2 is a diagram of the H-bridge. The optoisolators were added to protect the microcontroller from kickback voltage from the H-Bridge.

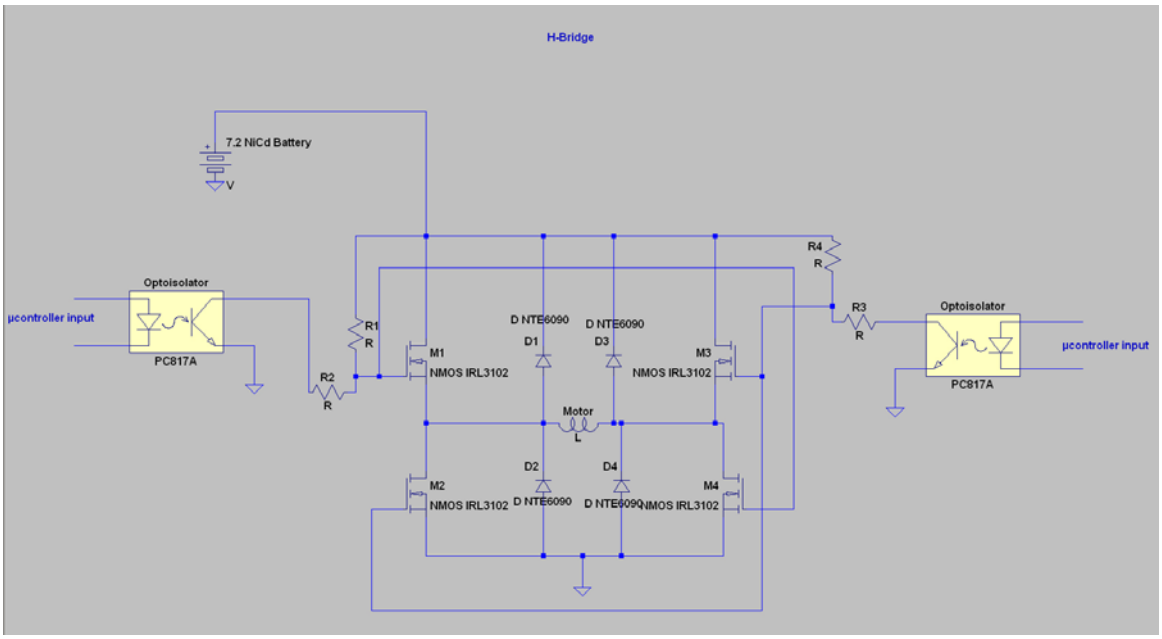


Figure 2: H-Bridge Schematic

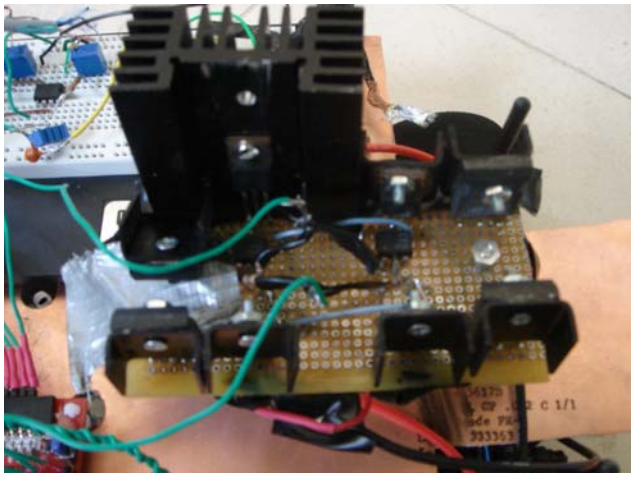


Figure 3: Actual H-Bridge Circuit

Digital Design

For our project we decided to use the CEREBOT microcontroller from DIGILENT, INC. We chose this microcontroller because it required low operating power and it had servo headers with supply power. Initially working with the microcontroller was very difficult, because we had to learn every aspect of how the programming operates with it, which took a great deal of time to learn. Attached is a copy of the program written using the C language.

ADC Interfacing

In order to convert the signals from the inductors, a multi-channel bidirectional analog to digital converter was used to digitize the analog signals from the inductors for the microcontroller. We chose to use a parallel output bidirectional Maxim MAX196A ADC. The sampling rate for this ADC is 100ksps. In order to use this ADC we needed to control the write and read signals being sent to the ADC. In addition we needed to monitor the interrupt signal from the ADC, which would allow us to read the converted data. In order to select each channel to be converted from the ADC, a control byte was sent to the ADC via the bidirectional bus. Once the data was read from the ADC it was stored in a variable to be used later for the servo and motor control. The schematic for the ADC is attached in the report.

Servo Control

Once the signals from the ADC were converted their values were compared to identify which inductor had a greater signal. Depending on which inductor signal was greater it would cause the wheels on that side to turn. Next, the differences between the signals

were calculated. This value was used to create a range of values which would determine the angle for which the servo would turn. We determined if the difference between the signals was less than .2V then the wheels should be center, which was a 1.5ms pulse to the servo. If the difference between the signals was greater than 2.25V the wheels need to be turned approximately 90°. We also determined the ranges needed to turn the servo for 15°, 30°, 45°, 60°, and 75°. With this range of values it would allow for very precise turning for the wheels.

Another issue for controlling the servo motor was determining a method of creating the pulse-width modulation. The microcontroller had libraries with pulse-width modulation programs already written but they only used one of the timers, and we needed multiple timers to create multiple PWM signals for both the servo and to the H-bridge. Therefore, we created the pulse-width modulation from a timer on the microcontroller. By modifying the timer control settings we were able to create the pulse-width modulation needed for the servo motor. The period needed for the servo motor was 20ms or a frequency of 50Hz. A 1.5ms pulse to the servo would cause the wheels to be centered, a 1.1ms pulse would make the wheels turn to the left, and a 1.9ms pulse would make the wheels turn to the right.

Motor Control

To control the motor, the same methodology was used that we implemented to control the servo motor. We created the pulse width modulation in the same manner as we did for the servo motor, by modifying a different timer on the microcontroller. However, this timer allowed us to create two different duty cycles for the same period. This allowed us to

control the forward and reverse motion of the motor with the two duty cycles. The frequency used was 1KHz. Due to the H-Bridge Circuitry, a short duty cycle (20%) would cause the motor to go fast, a 50% duty cycle would cause the motor to go at a moderate speed, and a long duty cycle (80%) would make the motor go slow. Another issue, with the circuitry was that one of the duty cycles must always be at 100% otherwise it would cause the H-Bridge Circuit to fail.

We utilized the same control logic to determine the duty cycle for the motor control. When the difference between the signals between the inductors was small, we would send a short pulse, which would cause the motor to go fast. If the difference were large then we would send a long pulse to cause the motor to slow down.

Outcome

This project was a good experience and a challenge to work on. We are happy that the racecar works with the sensor circuit built on a breadboard. In the future, we would like to improve the sensor circuit by adding a buffer on the output of the sensors; this would increase the current that is coming out of the sensors and give more sensitivity. In addition, we would include more gain stages instead of just one gain stage. This would reduce the overall power through each individual op-amp and causing them not to burn out. In addition, we would adjust the programming to obtain the best performance, currently this is not possible due to the short lifespan of the amplifiers in the sensor section. An additional improvement would be to include a feedback control system for wheel speed. Currently the cars speed needs to stay slow so it stays on the track, since

the car has no idea what speed it is going we have to set the wheel speed to work for the worst possible turning radius.

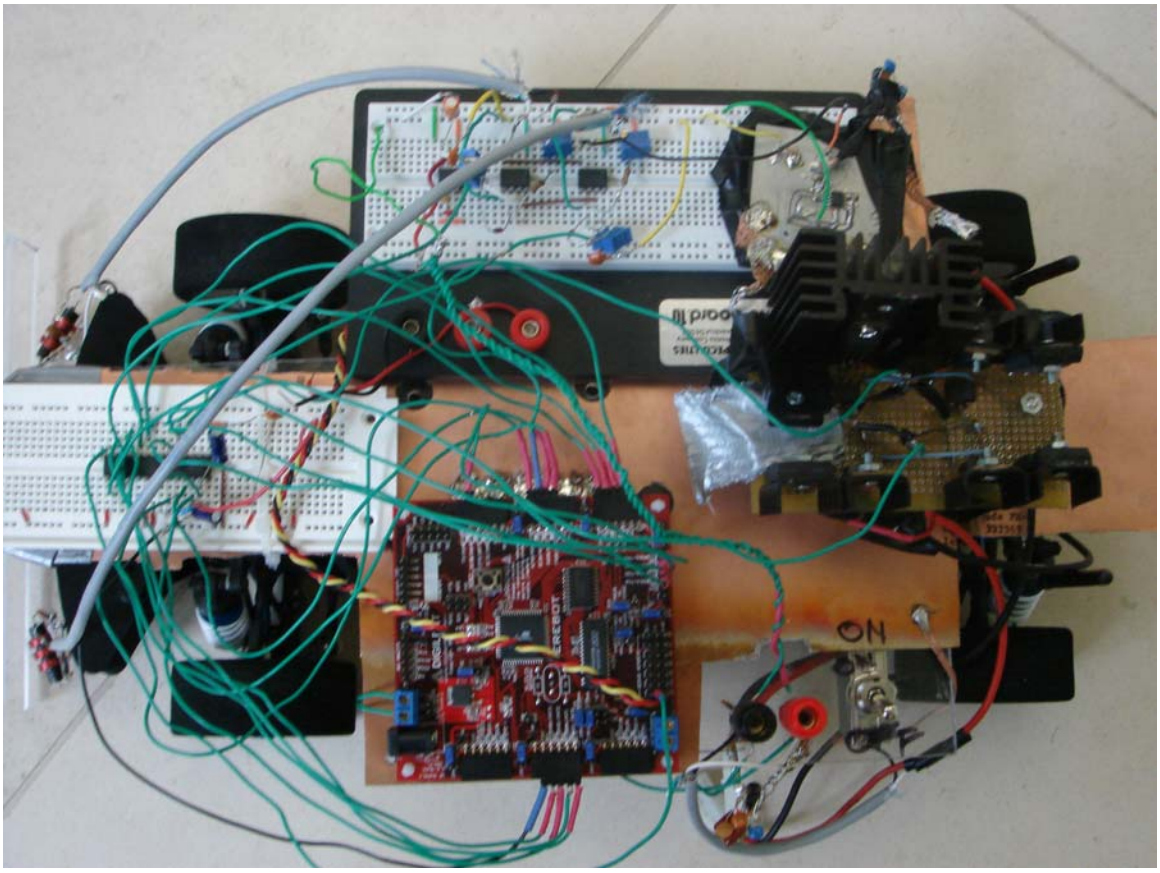
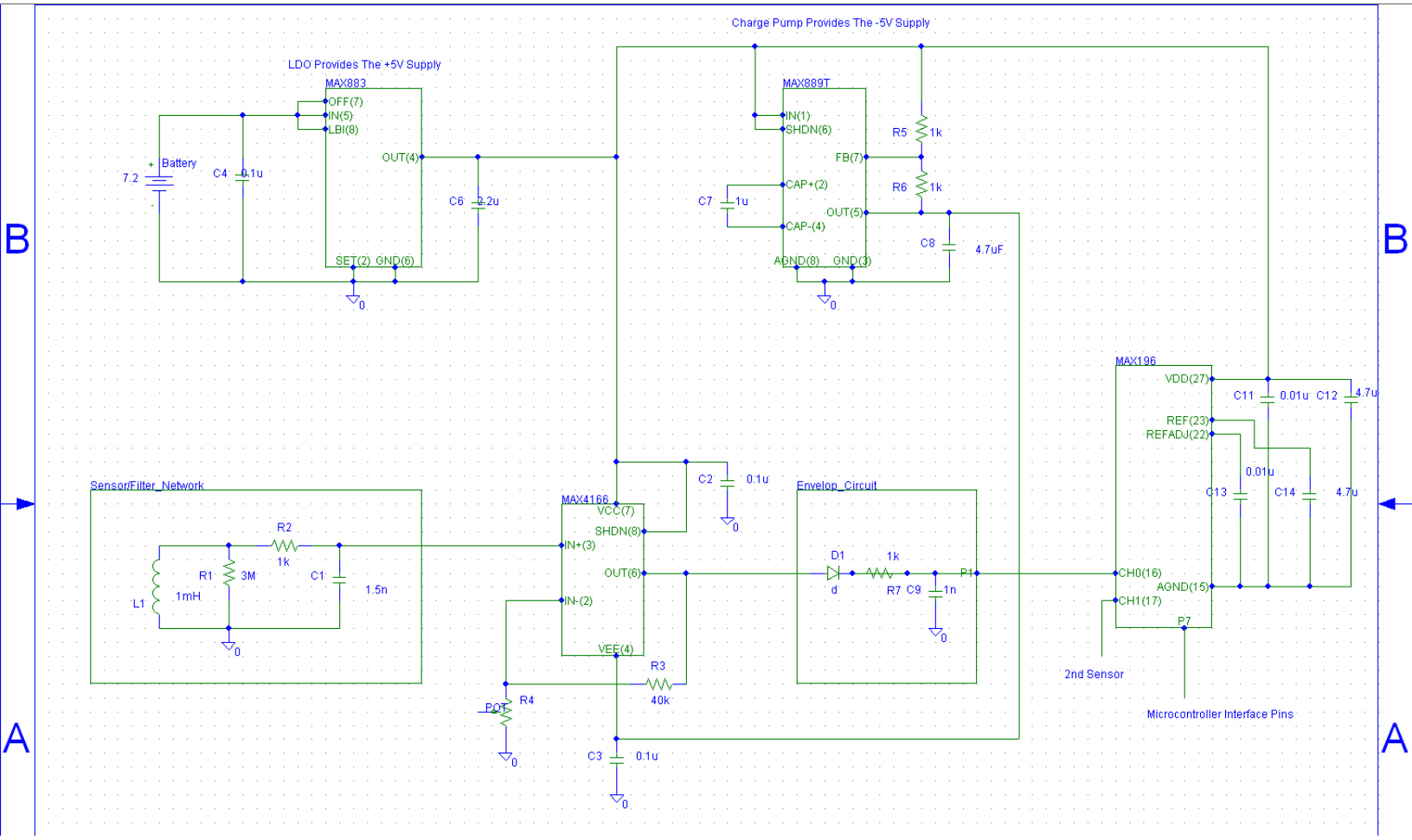


Figure 4: Components Integrated

References

Kachroo, Pushkin & Mellodge, Patricia. (2005). *Mobile Robotic Car Design.* New York: McGraw-Hill.

Pardue, J. (2005). *C Programming for Microcontrollers*. Smiley Micros.


Charge Pump Provides The -5V Supply
LDO Provides The +5V Supply
MAX883
OFF(7)
IN(5)
LBI(8)
OUT(4)
SET(2)
GND(6)
Battery
7.2
C4
0.1u
C6
2.2u
MAX889T
IN(1)
SHDN(6)
FB(7)
CAP+(2)
CAP-(4)
OUT(5)
AGND(8)
GND(3)
R5
1k
R6
1k
C7
1u
C8
4.7uF
MAX196
VDD(27)
REF(23)
REFADJ(22)
CH0(16)
AGND(15)
CH1(17)
C11
0.01u
C12
4.7u
C13
0.01u
C14
4.7u
P7
2nd Sensor
Microcontroller Interface Pins
Sensor/Filter_Network
R2
1k
R1
3M
C1
1.5n
L1
1mH
MAX4166
VCC(7)
SHDN(8)
IN+(3)
OUT(6)
IN-(2)
VEE(4)
C2
0.1u
R3
40k
POT
R4
C3
0.1u
Envelop_Circuit
D1
d
1k
R7
C9
1n
P1
A
B
A
B

```
//-------------------------------------------------------------//
//------------Program For NATCAR-----------------------------//
//-------------------------------------------------------------//
#include <avr/io.h> // include I/O definitions (port names, pin names, etc)
#include <avr/signal.h> // include "signal" names (interrupt names)
#include <avr/delay.h>
#include <avr/interrupt.h> // include interrupt support
#include "global.h" // include our global settings
#include "timer.h" // include timer function library (timing, PWM, etc)
#include "pulse.h" // include pulse output support


void pulsefunc(u08 A, u08 B, u08 W, u08 X, u08 Y, u08 Z);
u08 ADC1return (void);
u08 ADC2return (void);
u08 ADCfuncA(u08 ADC1, u08 ADC2);
u08 ADCfuncB(u08 ADC1, u08 ADC2);
u08 ADC1greaterA(u08 ADC0);
u08 ADC1greaterB(u08 ADC0);
u08 ADC2greaterA(u08 ADC0);
u08 ADC2greaterB(u08 ADC0);


int main(void)
{
    u08     W=0x03;         //Controls OC3A --99% duty cycle
    u08     X=0xE7;         //Controls OC3A --99% duty cycle
    u08     Y=0x01;         //Controls OC3B --75% duty cycle
    u08     Z=0xF4;         //Controls OC3B --75% duty cycle
    u08     ADC1temp, ADC2temp, ADC1, ADC2;
    u32     i,k;

while (1)
{
    ADC1 = 0b00000000;                                  //Initialize signal
from left inductor to zero
    ADC2 = 0b00000000;                                  //Initialize signal
from right inductor to zero

    for(i = 0; i < 50; i++)
    {
    ADC1temp = ADC1return();                            //Assign signal
from left inductor to a variable
    ADC2temp = ADC2return();                            //Assign signal
from right inductor to a variable


    if (ADC1temp > ADC1 && ADC1temp <= 0b01111111)      //Compare ADC
signals to make sure the signal is positive
        {
        ADC1 = ADC1temp;
        }
    if (ADC2temp > ADC2 && ADC2temp <= 0b01111111)
        {
        ADC2 = ADC2temp;
        }
    }
```

```
    pulsefunc(ADCfuncA(ADC1, ADC2), ADCfuncB(ADC1, ADC2), W, X, Y, Z);

    for(k =0; k<4000000000; k++)                                    //Delay
Function for pulses to servo motor to work
    {
    _delay_loop_2(10000000000000000);
    }
}
return 0;
}

void pulsefunc(u08 A, u08 B, u08 W, u08 X, u08 Y, u08 Z)        //SERVO PWM
Control
{
    u32 i;

#if 0
   // OC1B DDR- make servo output
   DDRB = (1<<PB6);

   // Timer/Counter 1 initialization
   // Clear on compare, Mode 14, /8 prescaler to make 1MHz timer
   TCCR1A = (1<<COM1A1) | (1<<COM1B1) | (1<<WGM11);
   TCCR1B = (1<<WGM13)  | (1<<WGM12) | (1<<CS11);

   ICR1  = 0x4E20;                                              // 20ms Duty
Cycle
   OCR1B = A&B;                                                 // Changes size
of PWM depending on signals from ADCs
#else
   // Port B initialization-Make servo an output
   PORTB=0x00;
   DDRB=0x40;

   // Timer/Counter 1 initialization
   TCCR1A=0xA2;
   TCCR1B=0x1A;
   TCNT1H=0x00;
   TCNT1L=0x00;
   ICR1H=0x4E;                                                  // 20ms Duty
Cycle (high byte signal)
   ICR1L=0x20;                                                  // 20ms Duty
Cycle (low byte signal)

   OCR1BH=A;                                                    // Size of PWM
(high byte)
   OCR1BL=B;                                                    // Size of PWM
(low byte)
#endif

#if 0                                                           //MOTOR PWM
Control
   // OC3A & OC3B DDR-Make signals for h-bridge, one signal for forward one
for reverse
   DDRE = (1<<PE3)|(1<<PE4);
```

```
   // Timer/Counter 3 initialization
   // /8 prescaler to make 1MHz timer
   TCCR3A = (1<<COM3A1) | (1<<COM3B1) | (1<<WGM31);
   TCCR3B = (1<<WGM33)  | (1<<WGM32) | (1<<CS31);

   ICR3  = 0x03E8;                                    // divide by 1000
to get 1kHz
   OCR3A = W&X;                                       // Pulse to one
side of h-bridge(forward) - OC3A
   OCR3B = Y&Z;                                       // Pulse to other
side of h-bridge(reverse)-OC3B

#else
   // Port E initialization
   PORTE=(0<<PE3)|(0<<PE4);
   DDRE=(1<<PE3)|(1<<PE4);                             // Makes pins PE4
and PE5 outputs for motor

   // Timer/Counter 1 initialization
   TCCR3A=0xA2;
   TCCR3B=0x1A;
   TCNT3H=0x00;
   TCNT3L=0x00;
   ICR3H=0x03;                                        // 1 ms duty cycle
for motor (high byte signal)
   ICR3L=0xE8;                                        // 1 ms duty cycle
for motor (low byte signal)

   OCR3AH=W;                                          // length of PWM
for motor for forward (high byte)
   OCR3AL=X;                                          // length of PWM
for motor for forward (low byte)

   OCR3BH=Y;                                          // length of PWM
for motor for reverse (high byte)
   OCR3BL=Z;                                          // length of PWM
for motor for reverse (low byte)

#endif

    for (i = 0; i<5; i++)                             // Delay loop for
pulses to motor and servo
    {
    return 0;
    }
}




u08 ADC1return (void)                  //Control ADC chip for writing
signal to select CH0 and reading data from CH0
{
    u08 data1;
    u08 ADC1temp;
```

```
    DDRE = (1<<PE0) | (1<<PE1) | (0<<PE2);                          //Direction
of RD and WR as outputs & INT as input
    PORTE = (1<<PE0) | (1<<PE1);                                        //READ
= 1 and WRITE = 1

    DDRF = 0xFF;                                                        //Set
Port F to all outputs (D11-D4)
    DDRD = (1<<PD0) | (1<<PD1) | (1<<PD2) | (1<<PD3);                   //Set
PD0-PD3 as outputs (D3-D0)

    PORTF = (0<<PF3) | (1<<PF2) | (0<<PF1) | (0<<PF0);              //WRITE
Control Byte for CHANNEL 0
    PORTD = (1<<PD3) | (0<<PD2) | (0<<PD1) | (0<<PD0);              //WRITE
Control Byte for CHANNEL 0
    cbi(PORTE, 1);                                                      //WRITE
= 0 (write pulse)
    sbi(PORTE, 1);                                                      //WRITE
= 1

    DDRD = (0<<PD0) | (0<<PD1) | (0<<PD2) | (0<<PD3);                   //Set
PD0-PD3 as inputs (D3-D0)
    DDRF = 0x00;                                                        //Set
Port F to all inputs (D11-D4)


    do                                                                  //A
loop for the INT signal to wait until the INT signal
    {                                                                   // to
go low to get the data from the ADC CH0
    }
    while (PINE & 0X04);

    cbi(PORTE, 0);                                                      //READ
= 0 (read pulse and read data)

    data1 = (PIND && 0x0F);                                         // Takes
the lower 4 bits of the ADC CH0
    ADC1temp = PINF;                                                    //
Takes the upper 8 bits of the ADC CH0

    sbi(PORTE, 0);                                                      //READ
= 1

    return (ADC1temp);

}

u08 ADC2return (void)          //Control ADC chip for writing signal to
select CH1 and reading data from CH1
{

    u08 data2;
    u08 ADC2temp;
    DDRE = (1<<PE0) | (1<<PE1) | (0<<PE2);                  //Direction of
RD and WR as outputs & INT as input
    PORTE = (1<<PE0) | (1<<PE1);                                    //READ = 1
and WRITE = 1
```

```
    DDRF = 0xFF;                                                    //Set Port
F to all outputs (D11-D4)
    DDRD = (1<<PD0) | (1<<PD1) | (1<<PD2) | (1<<PD3);               //Set
PD0-PD3 as outputs (D3-D0)

    PORTF = (0<<PF3) | (1<<PF2) | (0<<PF1) | (0<<PF0);      //WRITE Control
Byte for CHANNEL 1
    PORTD = (1<<PD3) | (0<<PD2) | (0<<PD1) | (1<<PD0);      //WRITE Control
Byte for CHANNEL 1

    cbi(PORTE, 1);                                                  //WRITE = 0
(write pulse)
    sbi(PORTE, 1);                                                  //WRITE = 1

    DDRD = (0<<PD0) | (0<<PD1) | (0<<PD2) | (0<<PD3);               //Set
PD0-PD3 as inputs (D3-D0)
    DDRF = 0x00;                                                    //Set Port
F to all inputs (D11-D4)


    do                                                              //A loop
for the INT signal to wait until the INT signal
    {                                                               // to go
low to get the data from the ADC CH1
    }
    while (PINE & 0X04);

    cbi(PORTE, 0);                                                  //READ = 0
(read pulse and read data)

    data2 = (PIND && 0x0F);                                 //Takes the
lower 4 bits of the ADC CH1
    ADC2temp = PINF;                                                // Takes
the upper 8 bits of the ADC CH1
    sbi(PORTE, 0);                                                  //READ = 1

    return (ADC2temp);
}

u08 ADCfuncA(u08 ADC1, u08 ADC2)        //COMPARES ADC values to determine
A, the length of the pulse needed (high byte)
{
    u08 A;
    u08 ADC0;

    if (ADC1 > ADC2)                        //Determines if signal from
left inductor is greater than right inductor
    {
        ADC0 = ADC1 - ADC2;                 // Calculates the difference
between the signals of the inductors
        A = ADC1greaterA(ADC0);             // "A" Determines the high byte
for the PWM when ADC1 is greater
        return (A);
    }
    else if (ADC1 < ADC2)                   //Determines if signal from
right inductor is greater than left inductor
```

```
    {
        ADC0 = ADC2 - ADC1;                 // Calculates the difference
between the signals of the inductors
        A = ADC2greaterA(ADC0);             // "A" Determines the high byte
for the PWM when ADC2 is greater
        return (A);
    }
    else if (ADC1 == ADC2)                  // When signals are equal make
the servo sent to center
    {
//      ADC = 0;
        A = 0x05;                           //SET sERvO CENTER at 1.5ms
pulse (high byte for pulse)
//      B = 0xDC;                           // (low byte)
        return (A);
    }
    else return (0);

}

u08 ADCfuncB(u08 ADC1, u08 ADC2)            //COMPARES ADC values to
determine B, the length of the pulse needed (low byte)
{
    u08 B;
    u08 ADC0;

    if (ADC1 > ADC2)                        //Determines if signal from
left inductor is greater than right inductor
    {
        ADC0 = ADC1 - ADC2;                 // Calculates the difference
between the signals of the inductors
        B = ADC1greaterB(ADC0);             // "B" Determines the high byte
for the PWM when ADC1 is greater
        return (B);
    }
    else if (ADC1 < ADC2)                   //Determines if signal from
right inductor is greater than left inductor
    {
        ADC0 = ADC2 - ADC1;                 // Calculates the difference
between the signals of the inductors
        B = ADC2greaterB(ADC0);             // "B" Determines the high byte
for the PWM when ADC2 is greater
        return (B);
    }
    else if (ADC1 == ADC2)
    {
//      ADC0 = 0;
//      A = 0x05;                       //SET sERvO CENTER at 1.5ms pulse
        B = 0xDC;                       // (high byte)
        return (B);
    }
    else return (0);

}

u08 ADC1greaterA(u08 ADC0)          //  If ADC1 > ADC2 values for pulse
length (A-high byte) is calculate from range
```

```
{
    u08 A;
    u08 LastA;

        if (ADC0 >= 0b00000001 && ADC0 < 0b00000101)            // 0
degrees (1.5ms pulse)
    {                                                           //ADC0
greater than 0 & less than .2V
        A = 0x05;
//      B = 0xDC;
        LastA = A;
        return (A);
    }
        else if (ADC0 >= 0b00000101 && ADC0 < 0b00001100)       // -15
degrees (1.44ms pulse)
    {                                                           //ADC0
greater than .2V & less than .5V
        A = 0x05;
//      B = 0xA0;
        LastA = A;
        return (A);
    }
        else if (ADC0 >= 0b00001100 && ADC0 < 0b00011001)       // -30
degrees (1.38ms pulse)
    {                                                           // ADC0
greater than .5V & less than 1V
        A = 0x05;
//      B = 0x64;
        LastA = A;
        return (A);
    }
        else if (ADC0 >= 0b00011001 && ADC0 < 0b00100101)       // -45
degrees (1.3ms pulse)
    {                                                           // ADC0
greater than 1V & less than 1.45V
        A = 0x05;
//      B = 0x14;
        LastA = A;
        return (A);
    }
        else if (ADC0 >= 0b00100101 && ADC0 < 0b00110011)       // -60
degrees (1.24ms pulse)
    {                                                           // ADC0
greater than 1.45V & less than 2V
        A = 0x04;
//      B = 0xD8;
        LastA = A;
        return (A);
    }
        else if (ADC0 >= 0b00110011 && ADC0 < 0b00111001)       // -75
degrees (1.18ms pulse)
    {                                                           // ADC0
greater than 2V & less than 2.25V
        A = 0x04;
//      B = 0x9C;
        LastA = A;
        return (A);
```

```
    }
        else if (ADC0 >= 0b00111001 && ADC0 <= 0b01111111)  // -90 degrees
(min pulse 1.1ms)
    {                                                       // ADC0
greater than 2.25V & less than 5V
        A = 0x04;
//      B = 0x4C;
        LastA = A;
        return (A);
    }
        else                                                // return
last A calculated to keep the same pulse applied
    {
        return (LastA);
    }
}

u08 ADC1greaterB(u08 ADC0)      //If ADC1 > ADC2 values for pulse length
(B-low byte) is calculate from range
{
    u08 B;
    u08 LastB;

        if (ADC0 >= 0b00000001 && ADC0 < 0b00000101)            // 0
degrees (1.5ms pulse)
    {
//      A = 0x05;
        B = 0xDC;
        LastB = B;
        return (B);
    }
        else if (ADC0 >= 0b00000101 && ADC0 < 0b00001100)       // -15
degrees (1.44ms pulse)
    {
//      A = 0x05;
        B = 0xA0;
        LastB = B;
        return (B);
    }
        else if (ADC0 >= 0b00001100 && ADC0 < 0b00011001)       // -30
degrees (1.38ms pulse)
    {
//      A = 0x05;
        B = 0x64;
        LastB = B;
        return (B);
    }
        else if (ADC0 >= 0b00011001 && ADC0 < 0b00100101)       // -45
degrees (1.3ms pulse)
    {
//      A = 0x05;
        B = 0x14;
        LastB = B;
        return (B);
    }
        else if (ADC0 >= 0b00100101 && ADC0 < 0b00110011)       // -60
degrees (1.24ms pulse)
```

```
    {
//      A = 0x04;
        B = 0xD8;
        LastB = B;
        return (B);
    }
        else if (ADC0 >= 0b00110011 && ADC0 < 0b00111001)       // -75
degrees (1.18ms pulse)
    {
//      A = 0x04;
        B = 0x9C;
        LastB = B;
        return (B);
    }
        else if (ADC0 >= 0b00111001 && ADC0 <= 0b01111111)  // -90 degrees
(min 1.1ms pulse)
    {
//      A = 0x04;
        B = 0x4C;
        LastB = B;
        return (B);
    }
        else                                                        //
return last B
    {
        return (LastB);
    }
}

u08 ADC2greaterA(u08 ADC0)      //If ADC1 > ADC2 values for pulse length
(A-high byte) is calculate from range
{
    u08 A;
    u08 LastA;

        if (ADC0 >= 0b00000001 && ADC0 < 0b00000101)            // 0
degrees (1.5ms pulse)
    {
        A = 0x05;
//      B = 0xDC;
        LastA = A;
        return (A);
    }
        else if (ADC0 >= 0b00000101 && ADC0 < 0b00001100)       // 15
degrees (1.56ms pulse)
    {
        A = 0x06;
//      B = 0x18;
        LastA = A;
        return (A);
    }
        else if (ADC0 >= 0b00001100 && ADC0 < 0b00011001)       // 30
degrees (1.62ms pulse)
    {
        A = 0x06;
//      B = 0x54;
        LastA = A;
```

```
        return (A);
    }
        else if (ADC0 >= 0b00011001 && ADC0 < 0b00100101)       // 45
degrees (1.7ms pulse)
    {
        A = 0x06;
//      B = 0xA4;
        LastA = A;
        return (A);
    }
        else if (ADC0 >= 0b00100101 && ADC0 < 0b00110011)       // 60
degrees (1.76ms pulse)
    {
        A = 0x06;
//      B = 0xE0;
        LastA = A;
        return (A);
    }
        else if (ADC0 >= 0b00110011 && ADC0 < 0b00111001)       // 75
degrees (1.82ms pulse)
    {
        A = 0x07;
//      B = 0x1C;
        LastA = A;
        return (A);
    }
        else if (ADC0 >= 0b00111001 && ADC0 <= 0b01111111)  // 90 degrees
(max pulse 1.9ms pulse)
    {
        A = 0x07;
//      B = 0x6C;
        LastA = A;
        return (A);
    }
        else                                                        //
return last A
    {
        return (LastA);
    }
}

u08 ADC2greaterB(u08 ADC0)      //If ADC1 > ADC2 values for pulse length
(B-low byte) is calculate from range
{
    u08 B;
    u08 LastB;

        if (ADC0 >= 0b00000001 && ADC0 < 0b00000101)            // 0
degrees (1.5ms pulse)
    {
//      A = 0x05;
        B = 0xDC;
        LastB = B;
        return (B);
    }
        else if (ADC0 >= 0b00000101 && ADC0 < 0b00001100)       // 15
degrees (1.56ms pulse)
```

```
    {
//      A = 0x06;
        B = 0x18;
        LastB = B;
        return (B);
    }
        else if (ADC0 >= 0b00001100 && ADC0 < 0b00011001)       // 30
degrees (1.62ms pulse)
    {
//      A = 0x06;
        B = 0x54;
        LastB = B;
        return (B);
    }
        else if (ADC0 >= 0b00011001 && ADC0 < 0b00100101)       // 45
degrees (1.7ms pulse)
    {
//      A = 0x06;
        B = 0xA4;
        LastB = B;
        return (B);
    }
        else if (ADC0 >= 0b00100101 && ADC0 < 0b00110011)       // 60
degrees (1.76ms pulse)
    {
//      A = 0x06;
        B = 0xE0;
        LastB = B;
        return (B);
    }
        else if (ADC0 >= 0b00110011 && ADC0 < 0b00111001)       // 75
degrees (1.82ms pulse)
    {
//      A = 0x07;
        B = 0x1C;
        LastB = B;
        return (B);
    }
        else if (ADC0 >= 0b00111001 && ADC0 <= 0b01111111)  // 90 degrees
(max pulse 1.9ms pulse)
    {
//      A = 0x07;
        B = 0x6C;
        LastB = B;
        return (B);
    }
        else                                                        //
return last B
    {
        return (LastB);
    }
}
```