CECS 347 Spring 2022 Project #2

# Self Driving Hardware PWM Car

By

Bryce Cheung, Jackie Huang

04.06.2022

A robot car that is controlled with the use of IR sensors, ADC conversions and hardware PWM signals.

**Introduction:** This project demonstrates the creation of a robot car that has the ability to perform multiple different functions. The car's wheels are controlled with the use of PWM signals controlled by the hardware of the TM4C123 microcontroller. IR sensors are used to control the direction the car turns/drives which allows it to stay within the track. The robot car is able to traverse through a simple track with barriers, where each wheel's duty cycle is controlled in order to achieve a turn.

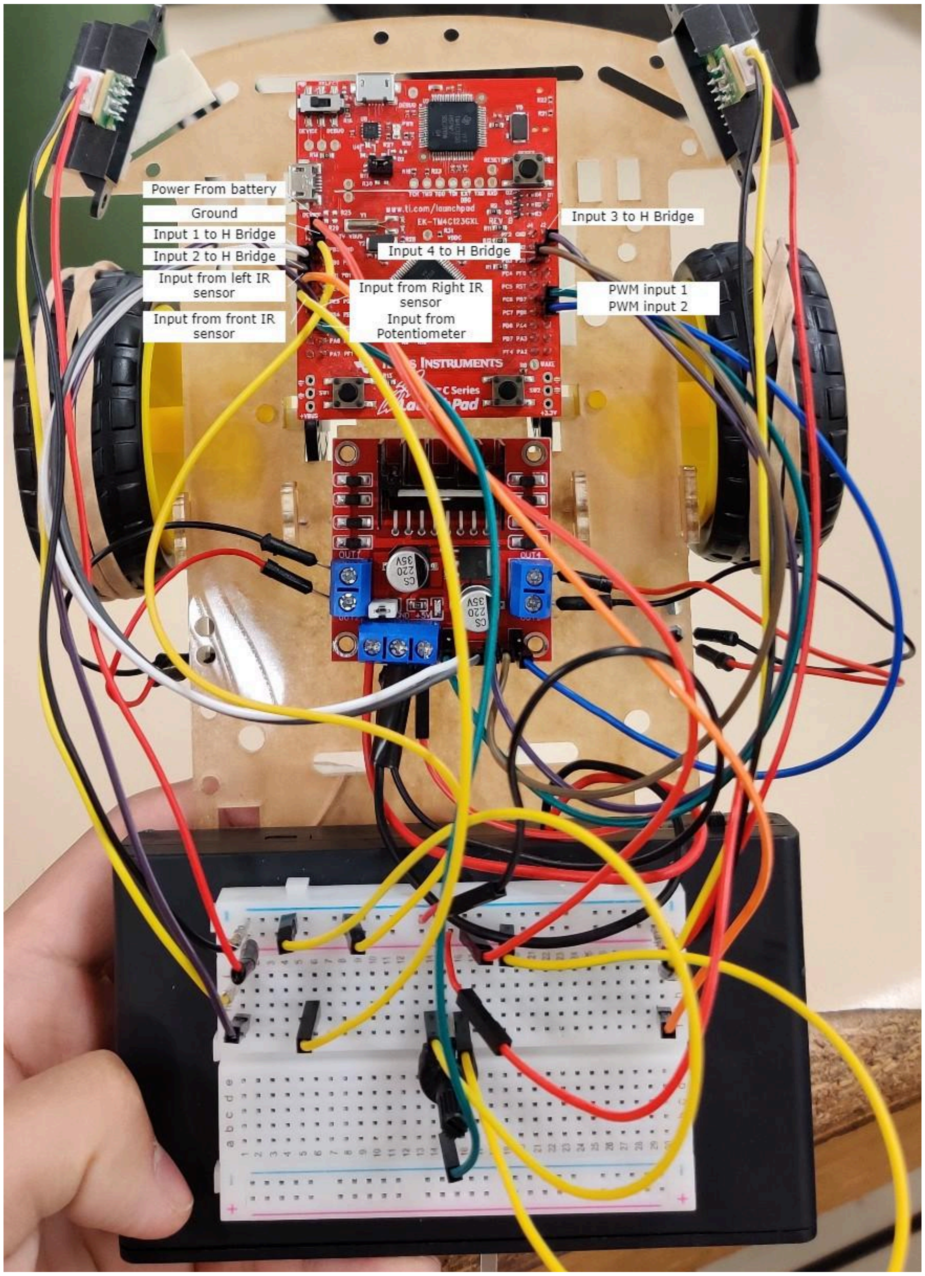
**Operation:** In order to start the car, the battery pack must first be turned on so that the TM4C microcontroller as well as the h-bridge voltage regulator connected to the motors have power. The car will be at its default(yellow) state which will last for two seconds, after which the car will determine the next state based on the IR sensor's inputs. When either of the sensors detects that it is within a range of 50cm and 15cm, the onboard LED will turn either green or blue(green for left, blue for right), and the respective wheel will engage and move forward. If a sensor detects that the car is within 15cm of a wall, the onboard LED will turn red and the entire car will stop, move backwards until the sensor does not detect that it is within 15cm of a wall, then resume the process. The car will make a full stop, and the onboard LED will turn purple if both sensors at the same time have inputs of it being more than 50cm away from any wall. The car also has a potentiometer that allows for the change of base speed at which it starts.
VIDEO

**Theory:** This base of this project uses a hardware generated PWM in order to drive two motors. The PWM is modified by changing the duty cycles based on which state the car is currently in. The car's states are determined by the inputs of the IR sensors. The IR sensors' inputs are analyzed after an ADC is used to convert the data. Two channels on one sample sequencer are being used in order to have two inputs from two different IR sensors. A filter is being used in order to clean up the sampling by the IR sensors to the TM4C MCU. Two interrupts are being used: one PORTF handler to control the change of state to the "off" state and the "white" state, and a SysTick handler to control the different duty cycles for each PWM/motor/wheel based on the different states the car is in.

## GPIO Port Table:

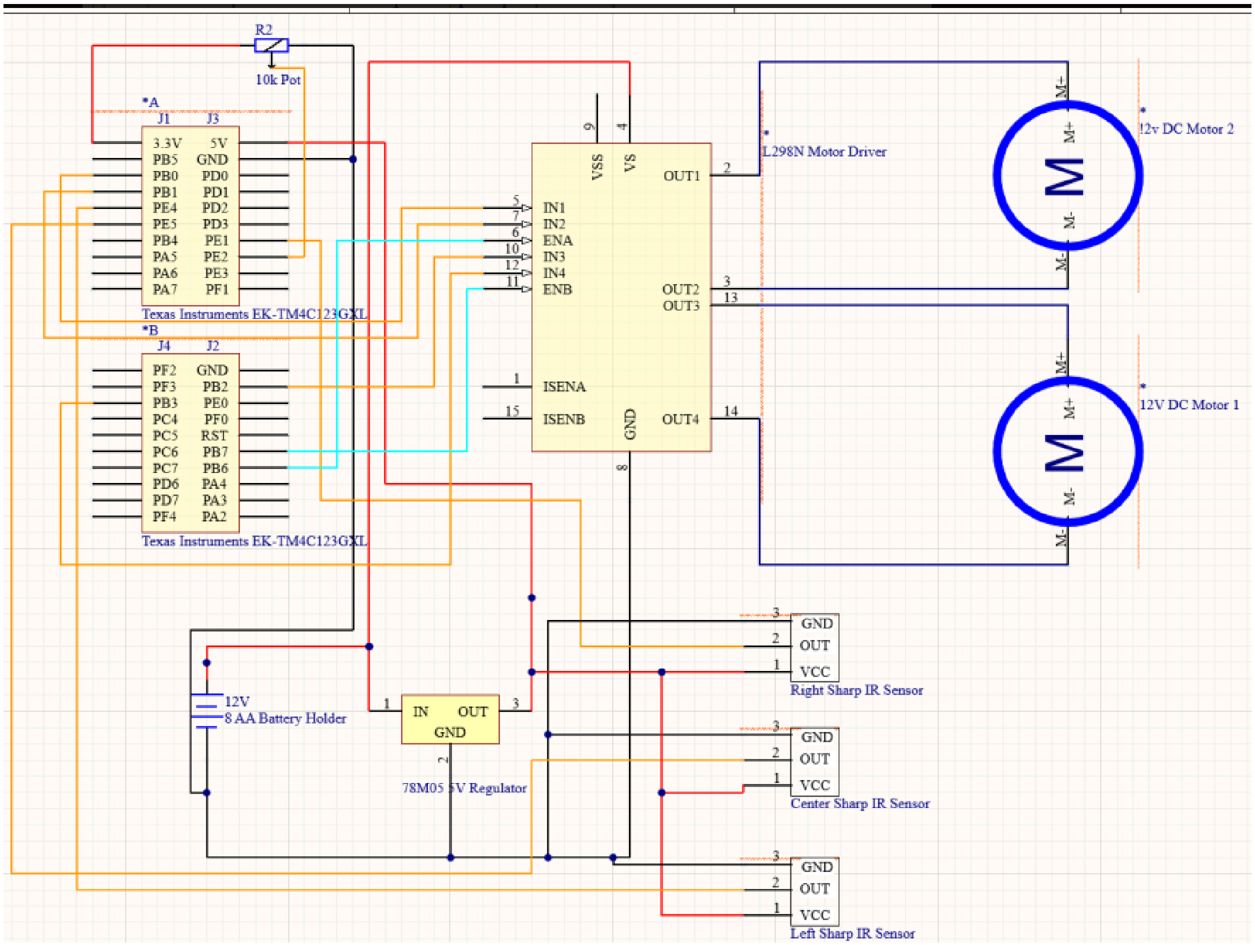
| Port Name | Input/ Output | Description |
| --- | --- | --- |
| PF0 | Input | Gets value from onboard button SW2 input bit 0 |
| PF4 | Input | Gets value from onboard button SW1 input bit 1 |
| PB0 | Output | Gets value from GPIO_PORTB_DATA_R output bit 0 |
| PB1 | Output | Gets value from GPIO_PORTB_DATA_R output bit 1 |
| PB2 | Output | Gets value from GPIO_PORTB_DATA_R output bit 2 |
| PB3 | Output | Gets value from GPIO_PORTB_DATA_R output bit 3 |
| PB6 | Output | PWM 1 Output from TM4C MCU |
| PB7 | Output | PWM 2 Output from TM4C MCU |
| PE1 | Input | Input from right IR sensor |
| PE2 | Input | Variable resistance from potentiometer |
| PE4 | Input | Input from left IR sensor |
| PE5 | Input | Input from center IR sensor |

**Picture:**




**Video: [Link]**

# Hardware design:



# Software Design

## PWM Init:

```
// period is 16-bit number of PWM clock cycles in one period
// Output on PB6/M0PWM0
void PWM0A_Init(uint16_t period){
  SYSCTL_RCGCPWM_R |= 0x01;             // 1) activate PWM0
  SYSCTL_RCGCGPIO_R |= 0x02;            // 2) activate port B: 000010
  while((SYSCTL_RCGCGPIO_R&0x02) == 0){};
  GPIO_PORTB_AFSEL_R |= 0x40;           // enable alt funct on PB6: 0100 0000
  GPIO_PORTB_PCTL_R &= ~0x0F000000;     // configure PB6 as PWM0
  GPIO_PORTB_PCTL_R |= 0x04000000;
  GPIO_PORTB_AMSEL_R &= ~0x40;          // disable analog functionality on PB6
  GPIO_PORTB_DEN_R |= 0x40;             // enable digital I/O on PB6
  GPIO_PORTB_DR8R_R |= 0xC0;    // enable 8 mA drive on PB6,7
  SYSCTL_RCC_R = 0x00100000 |           // 3) use PWM divider
    (SYSCTL_RCC_R & (~0x001E0000));   //    configure for /2 divider: PWM clock: 80Mhz/2=40MHz
  PWM0_0_CTL_R = 0;                     // 4) re-loading down-counting mode
  PWM0_0_GENA_R = 0xC8;                 // low on LOAD, high on CMPA down
  // PB6 goes low on LOAD
  // PB6 goes high on CMPA down
  PWM0_0_LOAD_R = period - 1;           // 5) cycles needed to count down to 0
  PWM0_0_CMPA_R = 0;            // 6) count value when output rises
  PWM0_0_CTL_R |= 0x00000001;           // 7) start PWM0
  PWM0_ENABLE_R |= 0x00000001;          // enable PB6/M0PWM0 0100 0000
}
```

This initializes the PWM for PB6. A nearly identical piece of initialization code is used for the PWM for PB7

## H-Bridge Control Initialization:

```
//Initalizes PB0~3 for H-Bridge Control
void Control_Int(void){
  GPIO_PORTB_DIR_R |= 0x0F; // sets pins PB0-3 to be output
  GPIO_PORTB_AFSEL_R &= ~0x0F; //disable alt fun for PB0-3
  GPIO_PORTB_DEN_R |= 0x0F; // Enable digital I/O for PB0-3
  GPIO_PORTB_PCTL_R &= ~0x0000FFFF; //Configure PB0-3 as GPIO
  GPIO_PORTB_AMSEL_R &= ~0x0F; //Disable analog fun for PB0-3
}
```

This code sets the outputs and inputs of the H-Bridge.

## LED Output Initialization:

```
// Initialize Port F LEDs
void PortF_LEDInit(void) {

  SYSCTL_RCGCGPIO_R |= SYSCTL_RCGCGPIO_R5;          // 2) activate port F: 000010
  while((SYSCTL_RCGCGPIO_R&SYSCTL_RCGCGPIO_R5) == 0){};

  GPIO_PORTF_AMSEL_R &= ~0x0E;        // disable analog function
  GPIO_PORTF_PCTL_R &= ~0x0000FFF0;   // GPIO clear bit PCTL
  GPIO_PORTF_DIR_R |= 0x0E;           // PF3-PF1 output
  GPIO_PORTF_AFSEL_R &= ~0x0E;        // no alternate function
  GPIO_PORTF_DEN_R |= 0x0E;           // enable digital pins PF3-PF1
}
```

This code allows the use of PF3-PF1 as LED outputs onboard.

**Switch Initialization:**

```
// Initialize edge trigger interrupt for PF0 and PF4 (falling edge)
void Switch_Init(void) {
  if ((SYSCTL_RCGCGPIO_R & SYSCTL_RCGCGPIO_R5) != SYSCTL_RCGCGPIO_R5){
    SYSCTL_RCGCGPIO_R |= SYSCTL_RCGCGPIO_R5;      // activate F clock
    while((SYSCTL_RCGCGPIO_R&SYSCTL_RCGCGPIO_R5) == 0){}; // wait for the clock to be ready
  }
  GPIO_PORTF_LOCK_R = 0x4C4F434B;     // unlock PortF PF0
  GPIO_PORTF_CR_R |= 0x1F;            // allow changes to PF4-0 :11111->0x0F
  GPIO_PORTF_DIR_R &= ~0x11;          // PF0 & PF4 input
  GPIO_PORTF_AFSEL_R &= ~0x11;        // no alternate function
  GPIO_PORTF_DEN_R |= 0x11;           // enable digital pins PF0
  GPIO_PORTF_PCTL_R &= ~0x000F000F;   // GPIO clear bit PCTL
  GPIO_PORTF_AMSEL_R &= ~0x11;        // disable analog function
  GPIO_PORTF_PUR_R |= 0x11;           // enable pullup resistors on PF0

  GPIO_PORTF_IS_R &= ~0x11;           //PF0 & PF4 are edge-sensitive
  GPIO_PORTF_IBE_R &= ~0x11;          //PF0 & PF4 are not both edge sensative
  GPIO_PORTF_IEV_R &= ~0x11;          //PF0 & PF4 are falling edge event
  GPIO_PORTF_ICR_R = 0x11; //clear flag 4, 0
  GPIO_PORTF_IM_R |= 0x11; //arm interrupt on PF4,0
  NVIC_PRI7_R = (NVIC_PRI7_R & 0xFF1FFFFF)| 0x00400000; //PF has priority 2
  NVIC_EN0_R = 0x40000000;  //enable interrupt 30(PORT F) in NVIC
}
```

This code allows the use of PF0 and PF4 as the onboard button 1 and button 2

**PLL Initialization:**

```
void PLL_Init(void){
    SYSCTL_RCC2_R |= SYSCTL_RCC2_USERCC2; //enable the use of advance clock control
    SYSCTL_RCC2_R |= SYSCTL_RCC2_BYPASS2; //bypass PPL while initializing
    // 2) select the crystal value and oscillator source
    SYSCTL_RCC_R &= ~SYSCTL_RCC_XTAL_M;   // clear XTAL field
    SYSCTL_RCC_R += SYSCTL_RCC_XTAL_16MHZ;// configure for 16 MHz crystal
    SYSCTL_RCC2_R &= ~SYSCTL_RCC2_OSCSRC2_M;// clear oscillator source field
    SYSCTL_RCC2_R += SYSCTL_RCC2_OSCSRC2_MO;// configure for main oscillator source
    // 3) activate PLL by clearing PWRDN
    SYSCTL_RCC2_R &= ~SYSCTL_RCC2_PWRDN2;
    // 4) set the desired system divider and the system divider least significant bit
    SYSCTL_RCC2_R |= SYSCTL_RCC2_DIV400;  // use 400 MHz PLL
    SYSCTL_RCC2_R = (SYSCTL_RCC2_R&~0x1FC00000) // clear system clock divider field
                  + (SYSDIV2<<22);      // configure for 50 MHz clock
    // 5) wait for the PLL to lock by polling PLLLRIS
    while((SYSCTL_RIS_R&SYSCTL_RIS_PLLLRIS)==0){};
    // 6) enable use of PLL by clearing BYPASS
    SYSCTL_RCC2_R &= ~SYSCTL_RCC2_BYPASS2;
}
```

This code enables the use of a PLL to generate a 50Mhz clock frequency

**Systick Initialization:**

```
// Initialize SysTick timer with interrupt enabled
void SysTick_Init(void){

  NVIC_ST_CTRL_R = 0x00; //disable the SysTick tmer
  NVIC_ST_RELOAD_R = 80000 - 1; //value to generates a 0.005s delay
  NVIC_ST_CURRENT_R = 0; //clear the value
  NVIC_SYS_PRI3_R = (NVIC_SYS_PRI3_R & 0x1FFFFFFF) | 0xA0000000; //SysTick Timer has priority 5
  NVIC_ST_CTRL_R = NVIC_ST_CTRL_CLK_SRC + NVIC_ST_CTRL_INTEN + NVIC_ST_CTRL_ENABLE;
}
```

This code sets the sampling rate of the ADC.

**ADC Initialization:**

```
void ADC_Init298(void){
  volatile unsigned long delay;
//  SYSCTL_RCGC0_R |= 0x00010000; // 1) activate ADC0 (legacy code)
  SYSCTL_RCGCADC_R |= 0x00000001; // 1) activate ADC0
  SYSCTL_RCGCGPIO_R |= SYSCTL_RCGCGPIO_R4; // 1) activate clock for Port E
  delay = SYSCTL_RCGCGPIO_R;      // 2) allow time for clock to stabilize
  delay = SYSCTL_RCGCGPIO_R;
  GPIO_PORTE_DIR_R &= ~0x32;      // 3) make PE1, PE4, and PE5 input
  GPIO_PORTE_AFSEL_R |= 0x32;     // 4) enable alternate function on PE1, PE4, and PE5
  GPIO_PORTE_DEN_R &= ~0x32;      // 5) disable digital I/O on PE1, PE4, and PE5
                                  // 5a) configure PE4 as ?? (skip this line because PCTL is for digital only)
  GPIO_PORTE_PCTL_R = GPIO_PORTE_PCTL_R&0xFF00FF0F;
  GPIO_PORTE_AMSEL_R |= 0x32;     // 6) enable analog functionality on PE1, PE4, and PE5
  ADC0_PC_R &= ~0xF;              // 8) clear max sample rate field
  ADC0_PC_R |= 0x1;               //    configure for 125K samples/sec
  ADC0_SSPRI_R = 0x3210;          // 9) Sequencer 3 is lowest priority
  ADC0_ACTSS_R &= ~0x0004;        // 10) disable sample sequencer 2
  ADC0_EMUX_R &= ~0x0F00;         // 11) seq2 is software trigger
  ADC0_SSMUX2_R = 0x0892;         // 12) set channels for SS2
  ADC0_SSCTL2_R = 0x0600;         // 13) no D0 END0 IE0 TS0 D1 END1 IE1 TS1 D2 TS2, yes END2 IE2
  ADC0_IM_R &= ~0x0004;           // 14) disable SS2 interrupts
  ADC0_ACTSS_R |= 0x0004;         // 15) enable sample sequencer 2
}
```

Initializes the ADC converter with each needed sample sequencer and functionalities.

**ADC_In:**

```
void ADC_In298(unsigned long *ain2, unsigned long *ain9, unsigned long *ain8){
  ADC0_PSSI_R = 0x0004;            // 1) initiate SS2
  while((ADC0_RIS_R&0x04)==0){};   // 2) wait for conversion done
  *ain2 = ADC0_SSFIFO2_R&0xFFF;    // 3A) read first result
  *ain9 = ADC0_SSFIFO2_R&0xFFF;    // 3B) read second result
  *ain8 = ADC0_SSFIFO2_R&0xFFF;    // 3C) read third result
  ADC0_ISC_R = 0x0004;             // 4) acknowledge completion
}
```

Controls the functionality of the ADC, uses busy-wait sampling.

**PORTF Handler:**

```
void GPIOPortF_Handler(void) {
  for (int time = 0; time < debouncing_number; time = time + 1){} //deboucning

  if (GPIO_PORTF_RIS_R & 0x10) { // switch 1
    GPIO_PORTF_ICR_R = 0x10; // ackniwkege flag 0
    car_go = car_go ^ 1; //inverts the bit
    if (car_go){
      CONTROL = NO_MOVE;
    }
    else{
      CONTROL = FORWARD;
    }
  }

  // Controls the direction the car moves in
  else if (GPIO_PORTF_RIS_R & 0x01){ //swtich 2
    GPIO_PORTF_ICR_R = 0x01; // acknowledge flag 4

    //Checks which direction the motors are current spining change accordingly
    if( LED != WHITE){
      LED = WHITE;
    }
    else{
      LED = YELLOW;
    }
  }
```

Port F handler that controls the function of the two onboard push buttons.

**Systick Handler:**

```
void SysTick_Handler(void){
  //controls when to sample
  sample = 1;
  time++;
  // if dist<=15cm, blink red LED here
  if (too_close_right) {
    R_Wheel_Duty(RW_Speed[2]);
    L_Wheel_Duty(RW_Speed[0]);
  }
  else if (too_close_left){
    L_Wheel_Duty(LW_Speed[2]);
    R_Wheel_Duty(RW_Speed[0]);
  }
  else if (CONTROL == BACKWARD & LED == RED){
    L_Wheel_Duty(LW_Speed[1]);
    R_Wheel_Duty(RW_Speed[1]);
  }
  else if (LED == PURPLE | LED == RED){
      L_Wheel_Duty(LW_Speed[0]);
      R_Wheel_Duty(RW_Speed[0]);
  }
  else{
    L_Wheel_Duty(LW_Speed[1]);
    R_Wheel_Duty(RW_Speed[1]);
  }
}
```

Handler that controls the actions of the motors based on certain states/conditions.

**ADC FIR Filter:**

```
void ReadADCFIRFilter(unsigned long *ain2, unsigned long *ain9, unsigned long *ain8){
  //                     x(n-1)
  static unsigned long ain2previous=0; // after the first call, the value changed to 12
  static unsigned long ain9previous=0;
  static unsigned long ain8previous=0;
  // save some memory; these do not need to be 'static'
  //              x(n)
  unsigned long ain2newest;
  unsigned long ain9newest;
  unsigned long ain8newest;
  ADC_In298(&ain2newest, &ain9newest, &ain8newest); // sample AIN2(PE1), AIN9 (PE4), AIN8 (PE5)
  *ain2 = (ain2newest + ain2previous)/2;
  *ain9 = (ain9newest + ain9previous)/2;
  *ain8 = (ain8newest + ain8previous)/2;
  ain2previous = ain2newest; ain9previous = ain9newest; ain8previous = ain8newest;
}
```

Filter for the ADC that accounts for the occasional data spikes of IR sensors.

**Table Estimation:**

```
unsigned char tb_estimation(unsigned int ADC_Value){
  unsigned char dist=0;

  int run = 1;
  int count = 0;
  while(run == 1){
    if( distance2[count] > ADC_Value ){
      count++;
    }
    else{
      run = 0;
    }
  }
  if (count >= 1){
    float slope = ( measurement2[count - 1] - measurement2[count] ) / ( distance2[count-1] - distance2[count] );
    float y_intercept = -slope * distance2[count-1] + measurement2[count-1];
    dist = slope * ADC_Value + y_intercept;
  }
  else{
    dist = measurement2[count];
  }

  return dist;
}
```

Function that calculates the respective distance that the IR sensors are reading using a table estimation technique.

**Equation Calculation:**

```
unsigned char eq_calcution(unsigned int ADC_Value){
  unsigned char dist=0;
  //dist = A + (B / ADC_Value);
  dist = A + B / ADC_Value;
  return dist;
}
```

Function that calculates the distance that the IR sensors read using a given equation.

**Distance Calculation:**

```
// Saves the distance for the left and right sensor
int tabledist_left = tb_estimation(left_measuremnet);
int eqdist_left = eq_calcution(left_measuremnet);
int tabledist_right = tb_estimation(right_measurement);
int eqdist_right = eq_calcution(right_measurement);
int combine_avg_left = (tabledist_left + eqdist_left)  / 2;
int combine_avg_right = ( tabledist_right + eqdist_right) / 2;
// Save the distance for the front sensor
int tabledist_front = tb_estimation(front_measurment);
int eqdist_front = eq_calcution(front_measurment);
int combine_avg_front = ( tabledist_front + eqdist_front) / 2;
//calculates the differencae between the two
int diff = combine_avg_left - combine_avg_right;
```

**Software Filter:**

```
while (count<NUM_SAMPLES) {
  while (!sample){} // sample one value every 1/20=0.05 second
  sample = 0;
  //ADCvalue += ADC0_InSeq3();
  ReadADCFIRFilter(&left_measuremnet, &right_measurement, &front_measurment);
  count++;
}
```

## State Controller:

```
//Speed setting mode
if (LED == WHITE){
  while(LED == WHITE){
    float a = 90;
    float divide = a / 2771;
    float pot_speed = ADC0_InSeq3();
    pot_speed = pot_speed * divide / 100;
    int pot_duty = pot_speed * PERIOD;
    L_Wheel_Duty(pot_duty);
    R_Wheel_Duty(pot_duty);
  }
}
else if (CONTROL == NO_MOVE){
  L_Wheel_Duty(LW_Speed[0]);
  R_Wheel_Duty(RW_Speed[0]);
  LED = OFF;
}
else if (combine_avg_left < 15 | combine_avg_right < 15 | combine_avg_front < 10){
  LED = RED;
  // 1 second wait
  while(time < 200){}
  too_close_right = too_close_left = 0;
  time = 0;
  LED = OFF;
  CONTROL = BACKWARD;
}
else if (CONTROL == BACKWARD){
  while(time < 50){}
  time = 0;
  CONTROL = FORWARD;
}
else if (combine_avg_left > 50 & combine_avg_right > 50){
  LED = PURPLE;
  too_close_right = too_close_left = 0;
}
// Detects if the car is closer to the right side, then sets the flag that activates the right wheel
else if (diff > 6){
  LED = BLUE;
  too_close_right = 1;
}
// Detects if the car is closer to the left side, then sets the flag that activates the left wheel
else if( diff < -6){
  LED = GREEN;
  too_close_left = 1;
}
else{
  LED = OFF;
  too_close_right = too_close_left = 0;
}
count = 0;
```

## Conclusion:

This project was a challenge, but very informative and crucial for understanding embedded systems. Our group had issues in organizing as well as implementing the different functions required in this project such as using the ADC to control the settings of the PWM. Overall the project allowed us to get a better understanding of using IR sensors and motors to create a working car with inputs and outputs needed to be managed and controlled.