Cover Page

Lab 6
Car walks along the wall
DMU
Spring 2024

Yuhan Wang
Wankang Zhai
2220213830

contents

1. OVERVIEW 3

2. DETAILED DESCRIPTION AND RESULTS 3

TASK1: LET THE CAR MOVE FIRST 3
1) USING PWM_INIT TO SET THE MOTOR ACTIVE 3
2) FINISH ADC INIT AND TIMER INIT FUNCTION 7
3) CONFIGURE 2 HANDLER FUNCTION 8
TASK2: USING PID TO CONTROL THE ERROR TO ZERO, KEEP THE CAR STABLE. 10

3. CONCLUSION 11

APPENDIX: VIDEOS OF THE CAR RUNNING CAN BE FOUND ON THE FOLLOWING WEBSITE 13

1. Overview

In this experiment, we used PID control to let the car complete the task of walking along the wall. Our specific contributions are as follows:

1. Set the car interface and activate the PB7 and PB8 interfaces to make the car move forward.

2. Set PWM and change the duty cycle through distance to make the car turn within a certain range.

3. Reasonably adjust the duty cycle through PID control

4. Repeatedly set the parameters so that the car's error converges within 2m from the target and can drive in a straight line.

5. Set the timer cycle timing. When the cycle reaches the target value, call ADCProcessorTrigger to successfully convert the ADC function into discrete values. used to calculate errors.

2.Detailed Description and Results

Task1: Let the car move first

1）Using PWM_init to set the motor active

First, we need to activate the PWM module. We first activate the required peripheral. Since we need to control the steering, our left and right wheels need to set different duty values, so we activate two PWM peripherals. At the same time, we also need to activate the GPIO ABD port to control the wheel motors respectively. of rotation. We referred to the car's port explanation diagram. As shown in Figure 1 and Figure 2.

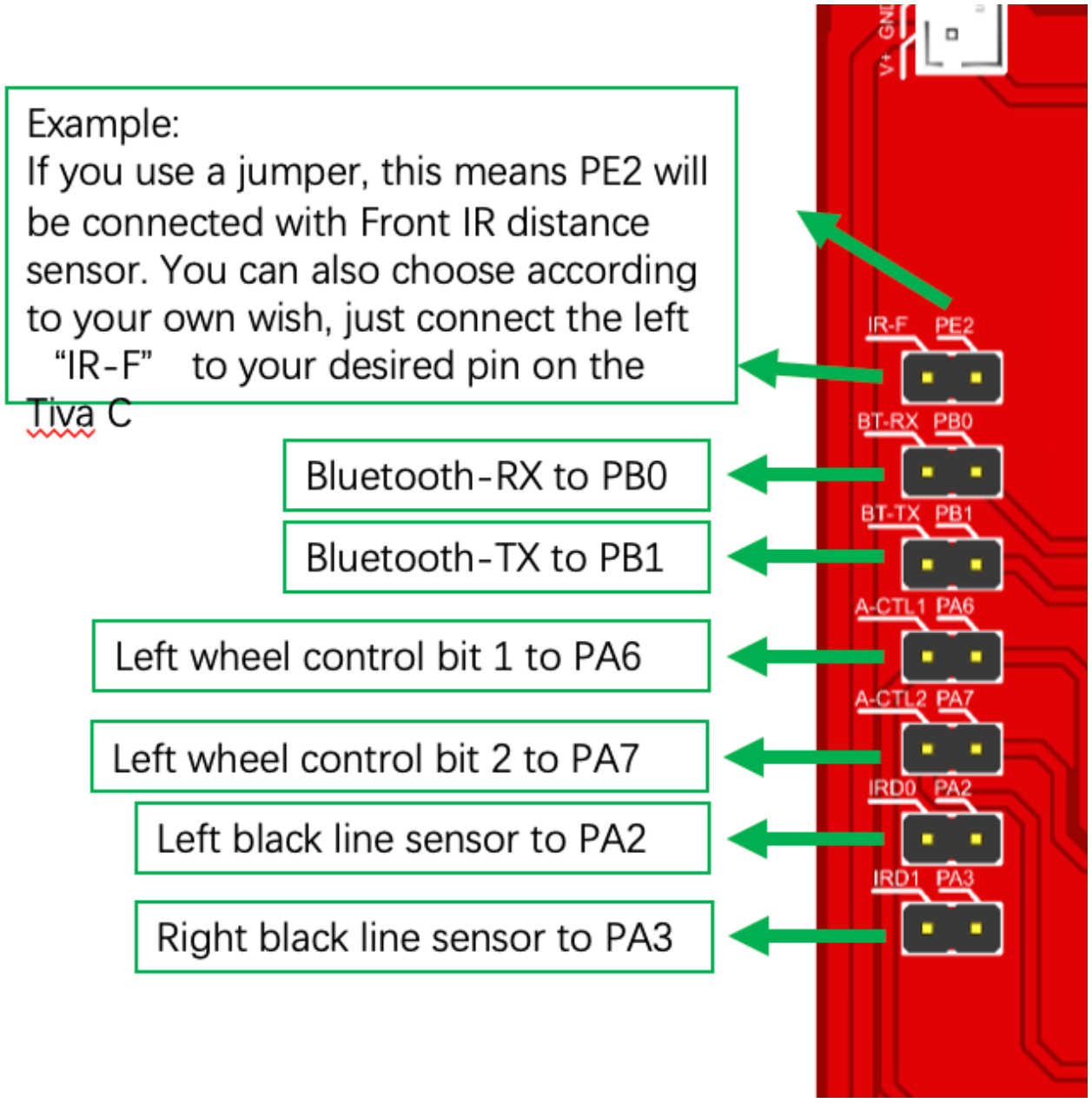


Figure 1

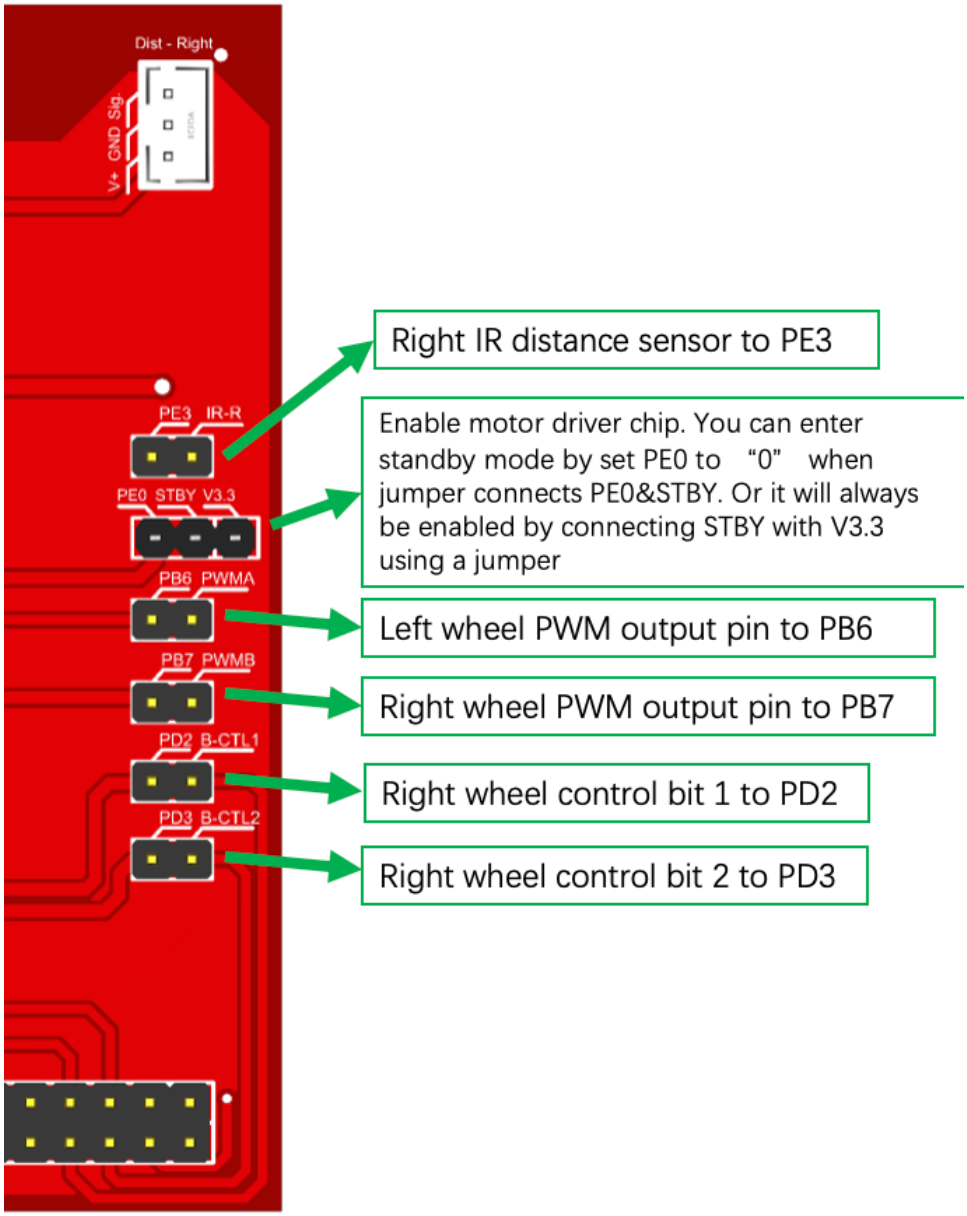

Dist - Right
V+ GND Sig
PE3 IR-R
PE0 STBY V3.3
PB6 PWMA
PB7 PWMB
PD2 B-CTL1
PD3 B-CTL2
Right IR distance sensor to PE3
Enable motor driver chip. You can enter standby mode by set PE0 to "0" when jumper connects PE0&STBY. Or it will always be enabled by connecting STBY with V3.3 using a jumper
Left wheel PWM output pin to PB6
Right wheel PWM output pin to PB7
Right wheel control bit 1 to PD2
Right wheel control bit 2 to PD3

Figure2

We clearly found that Figure 1 and Figure 2 indicate that the PA port controls the left wheel and the PD port controls the right wheel. Control the PWM of the left and right wheels respectively through PB 6 and PB7. Therefore, our code first activates five peripherals, and then waits for all peripherals to be set. According to the clock, we set the PWM clock, and then set the load. According to load we initialize PWM. And wrap it into PWM_init() function. As shown in Figure 3

```
388  void PWM_init(void)
389  {
390      SysCtlPeripheralEnable(SYSCTL_PERIPH_PWM1);
391      SysCtlPeripheralEnable(SYSCTL_PERIPH_PWM0);
392
393      SysCtlPeripheralEnable(SYSCTL_PERIPH_GPIOB);
394      SysCtlPeripheralEnable(SYSCTL_PERIPH_GPIOA);
395      SysCtlPeripheralEnable(SYSCTL_PERIPH_GPIOD);
396      while(!SysCtlPeripheralReady(SYSCTL_PERIPH_PWM1));
397      while(!SysCtlPeripheralReady(SYSCTL_PERIPH_PWM0));
398
399      while(!SysCtlPeripheralReady(SYSCTL_PERIPH_GPIOB));
400      while(!SysCtlPeripheralReady(SYSCTL_PERIPH_GPIOA));
401      while(!SysCtlPeripheralReady(SYSCTL_PERIPH_GPIOD));
402      SysCtlPWMClockSet(SYSCTL_PWMDIV_64);
403      GPIOPinConfigure(GPIO_PB7_M0PWM1);
404      GPIOPinConfigure(GPIO_PB6_M0PWM0);
405
406      GPIOPinTypePWM(GPIO_PORTB_BASE, GPIO_PIN_6);
407      GPIOPinTypePWM(GPIO_PORTB_BASE, GPIO_PIN_7);
408
409
410      int PWM_clock = SysCtlClockGet() / 64;
411      int PWM_freq = 55;
412      load = PWM_clock / PWM_freq - 1;
413
414      PWMGenPeriodSet(PWM0_BASE, PWM_GEN_0, load);
415      PWMGenConfigure(PWM0_BASE, PWM_GEN_0, PWM_GEN_MODE_DOWN);
416      width = load * duty*0.01;
417      PWMPulseWidthSet(PWM0_BASE, PWM_OUT_0, width);
418      PWMOutputState(PWM0_BASE, PWM_OUT_0_BIT, true);
419      PWMPulseWidthSet(PWM0_BASE, PWM_OUT_1, width);
420      PWMOutputState(PWM0_BASE, PWM_OUT_1_BIT, true);
421      PWMGenEnable(PWM0_BASE, PWM_GEN_0);
422      IntMasterEnable();
423
424      GPIOPinTypeGPIOOutput(GPIO_PORTD_BASE, GPIO_PIN_2);
425      GPIOPinTypeGPIOOutput(GPIO_PORTD_BASE, GPIO_PIN_3);
426      GPIOPinTypeGPIOOutput(GPIO_PORTA_BASE, GPIO_PIN_6);
427      GPIOPinTypeGPIOOutput(GPIO_PORTA_BASE, GPIO_PIN_7);
428
429      GPIOPinWrite(GPIO_PORTA_BASE, GPIO_PIN_6|GPIO_PIN_7, GPIO_PIN_6); // let the wheel move
430      GPIOPinWrite(GPIO_PORTD_BASE, GPIO_PIN_2|GPIO_PIN_3, GPIO_PIN_3);
431
432  }
```

Figure3

We first define the use of conv to implement the cumulative multiplication of polynomials, which is a very convenient function. We can implement the cumulative multiplication task for multiple polynomial convolutions. We first define four different polynomials, and then use roots to find the results of the polynomials. The roots function returns the roots of a polynomial, returned as a column vector. If there are multiple roots, they will appear repeatedly in the result vector. If a polynomial has complex roots, they appear as conjugate pairs. Therefore, in Figure 3, we have many conjugate complex roots. As shown in Figure3.

2) Finish ADC init and Timer Init function

```
void ADC_init(void)
{
    SysCtlPeripheralEnable(SYSCTL_PERIPH_ADC0);
    SysCtlPeripheralEnable(SYSCTL_PERIPH_GPIOE);
    GPIOPinTypeADC(GPIO_PORTE_BASE, GPIO_PIN_3);

    ADCSequenceDisable(ADC0_BASE, 1);
    ADCSequenceConfigure(ADC0_BASE, 1, ADC_TRIGGER_PROCESSOR, 0);
    ADCSequenceStepConfigure(ADC0_BASE, 1, 0, ADC_CTL_CH0);
    ADCSequenceStepConfigure(ADC0_BASE, 1, 1, ADC_CTL_CH0);
    ADCSequenceStepConfigure(ADC0_BASE, 1, 2, ADC_CTL_CH0);
    ADCSequenceStepConfigure(ADC0_BASE, 1, 3, ADC_CTL_CH0|ADC_CTL_IE|ADC_CTL_END);
    ADCSequenceEnable(ADC0_BASE, 1);
    IntMasterEnable();
    IntEnable(INT_ADC0SS1);
    ADCIntEnable(ADC0_BASE, 1);
}
```

Figure 4

We all know that the data measured by the sensor is an analog signal. When we use tivac to calculate, we cannot directly use the analog continuous signal. We need to convert it into a digital data signal, so here we use analog to digital. We need to use this tivac function. So first, we activate the

corresponding periphery, and then configure the corresponding ADXSequence. We take three measurements and activate interrupts as shown in Figure 4.

```
void timer_init(void)
{
    SysCtlPeripheralEnable(SYSCTL_PERIPH_TIMER0);
    TimerConfigure(TIMER0_BASE, TIMER_CFG_PERIODIC);
    int a = SysCtlClockGet();
    ratio = a / f1 - 1;
    TimerLoadSet(TIMER0_BASE, TIMER_A, ratio);
    IntEnable(INT_TIMER0A);
    TimerIntEnable(TIMER0_BASE, TIMER_TIMA_TIMEOUT);
    //GPIOPinWrite(GPIO_PORTF_BASE, RED_LED|GREEN_LED|BLUE_LED, RED_LED);
}
```

Figure 5

In Figure 5, we configured the timer_init function. We know that when we convert the analog signal into a digital signal, we need a fixed sampling frequency. In this experiment, we use a timer for sampling. First, I enable the H port, and then set the timer parameters. We set a ratio. The timer is sampled according to the frequency of f1, and finally the timer interrupt is enabled.Using plot to plot sin and cos figure

3) Configure 2 Handler Function

We need to complete the contents of the handler so that the sensor value can be read in time when the timer generates an interrupt. Therefore, we first set up the ADCIntHandler. According to the design as shown in the figure, we directly read the data passing through the sensor through ADCSequenceDataGet. The data we read can be stored in the ADCOVal array, and we collected data four times in a row. Therefore, the average value can be obtained by taking the average value, and converted into a formula through consensus. As shown in the figure, it is divided into red_flag according to the error distance.and different colored lights.

```
void ADC0IntHandler(void)
{
    ADCIntClear(ADC0_BASE, 1);
    ADCSequenceDataGet(ADC0_BASE, 1, ADC0Val);
    int i = 0;
    for(i = 0;i<4;i++)
    {
        mean+= ADC0Val[i];
    }
    mean = mean / 4;
    distance = -1.858203*pow(10,-9)*pow(mean, 3) + 1.3214630459*pow(10, -5)*pow(mean, 2) - 0.03356491643966*mean + 35.536853696418802;
    SB = target - distance;
    if(fabs(SB)>2)
    {
        red_flag = 1;
    }
    else if(fabs(SB)<=1)
    {
        green_flag = 1;
    }
    else
    {
        yellow_flag = 1;
    }

}
```

Figure 7

```
void TIMER0IntHandler(void)
{
    TimerIntClear(TIMER0_BASE, 1);
    ADCProcessorTrigger(ADC0_BASE, 1);
}
void start()
{
    PWM_init();
    f1 = 20;
    timer_init();
    TimerEnable(TIMER0_BASE, TIMER_A);

}
lookup table1[] = {{"STR", start}};
int len1 = sizeof(table1)/sizeof(table1[0]);
```

Figure 8

As shown in Figure 8, we clear the previously stored interrupt data by setting TimerClear. Set each Timer interrupt to ADC interrupt

execution. Therefore every time the timer interrupts we will calculate a new error value SB

Task2: using PID to control the error to zero, keep the car stable.

```
e_current = SB;
p_value = P * e_current;
d_value = D*(e_current - e_previous);
i_value = I * e_sum;
adj = p_value + d_value + i_value;
e_sum = e_sum + e_current;
e_previous = e_current;
duty1 = duty + adj;
duty2 = duty - adj;

if(duty1 > 95)
{
    duty1 = 95;
}
else if (duty1 < 77)
{
    duty1 = 77;
}
else if(duty2 > 95)
{
    duty2 = 95;
}
else if(duty2 <77)
{
    duty2 = 77;
}

width1 = load * duty1*0.01;
width2 = load * duty2*0.01;

PWMPulseWidthSet(PWM0_BASE, PWM_OUT_1, width1);
PWMPulseWidthSet(PWM0_BASE, PWM_OUT_0, width2);
SysCtlDelay(SysCtlClockGet() / (1000 * 3));
```

Figure 9

As shown in Figure 9, we completed the control module of this experiment by using PID control. The three coefficients are composed of P, I, and D respectively. Among them, P determines the direct error, I determines the cumulative error, and D determines the difference between this measurement error and the last measurement error. We complete the calculation of the overall PID error by adjusting and balancing the three parameters so that the car and the sensor are controlled at int distance.

It is worth mentioning that we have adjusted the value range of duty here. We need to ensure that when the error on the right side is detected to be too small, the car will not rotate too fast and cause instability. We set the minimum duty cycle to 77. Therefore, it can be ensured that the car can approach the right wall at a stable speed and remain stable. We place the code in Figure 10.

```
if(duty1 > 95)
{
    duty1 = 95;
}
else if (duty1 < 77)
{
    duty1 = 77;
}
else if(duty2 > 95)
{
    duty2 = 95;
}
else if(duty2 <77)
{
    duty2 = 77;
}

width1 = load * duty1*0.01;
width2 = load * duty2*0.01;

PWMPulseWidthSet(PWM0_BASE, PWM_OUT_1, width1);
PWMPulseWidthSet(PWM0_BASE, PWM_OUT_0, width2);
SysCtlDelay(SysCtlClockGet() / (1000 * 3));
```

Figure 10

3. Conclusion

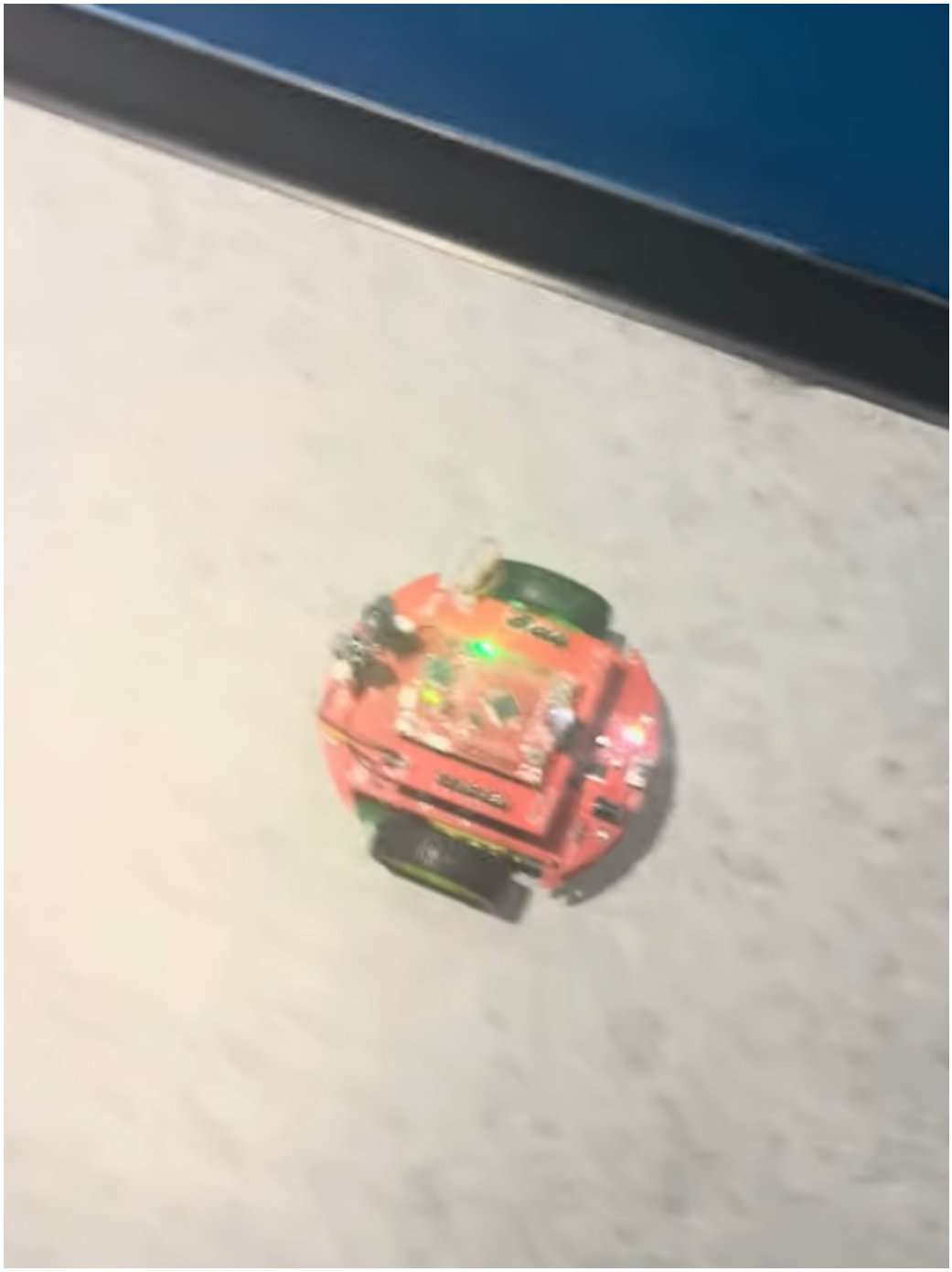
In this experiment, we successfully made the car walk smoothly along the wall. Through our tests, our car can do:

1. Walk at the fastest speed within a reasonable distance
2. We adjust our PID control so that when the distance is far/closer, we can quickly adjust the direction and eventually stabilize. Our car can complete the journey stably and ensure a 2m gap.

3. We set up the Timer and use it in combination with the ADC. In this case, we can manually adjust the information collection frequency according to the needs of the car, and combine the sensor with the error distance for PID control.

Initial

Adjust



Appendix: Videos of the car running can be found on the following website

https://www.youtube.com/shorts/igVuKryBlsU