```
A1 = Pin('B3',Pin.OUT_PP)
A2 = Pin('B5',Pin.OUT_PP)
PWMA = Pin('B8') # PB8 has TIM4, CH3
tim = Timer(4, freq=1000)
ch1 = tim.channel(3, Timer.PWM, pin=PWMA)
ch1.pulse_width_percent(abs(self.NrA1-self.NrA2))
```

Code 1 STM32 Pin Configuration for Driver

Two pins of STM32 are configured as output mode to control the corresponding "IN" pins of driver module. Another pin is set as PWM output to modulate "EN", whose timer is enabled.

3.3 Car Group: Gyroscope

Section Author	**Ziyang Long** **(UESTC ID:2018190502030, UofG ID:2429503L)** **Yuchen Yao** **(UESTC ID:2018190602001, UofG ID:2429207Y)**

3.3.1 The Choice between MPU-6050 and JY901S

There are two candidates in our choice of Gyroscope, namely MPU-6050 in Figure 16 and JY901S in Figure 17.

Figure 16 MPU-6050

The MPU-6050 devices combine a 3-axis gyroscope and a 3-axis accelerometer on the same silicon die, together with an onboard Digital Motion Processor, which processes complex 6-axis motion fusion algorithms. For us, we mainly use three angular velocities 'Gyro' around three axis 'x, y, z', where the unit is °/s. After obtaining the angular velocity, a formula is necessary to change the angular velocity into angle, and the easiest way is:

$$angle = angular\ velocity \times time$$

The process of obtaining angle can be regarded as integral calculation when adding the angular velocity per unit time together. The code below presents the main logic of calculation angle.

```
sum=0
from time import sleep
while 1:
    sum=sum+MPU.read.Gyro_z()*0.002
    if(sum>=90 or sum<=-90):
        print('Sum is ',sum)
        sum=0
        sleep(0.5)
    sleep(0.001)
```

Code 2 integral of angular velocity

Although MPU6050 performs well (the excellent ability to capture the change of position) when rapidly rotating 90 degrees, it might lead to large error if the slight deflection occurs in unit time. On the condition that the small deflection occurs, the angular velocity will be unexpected small, and due to the limitation of MPU-6050's sampling rate, (maximum value is 8KHz), some unwanted omission may happen during the accumulation of the angular velocity per unit time

```
def reset(self):
        self._write_byte(MPU_PWR_MGMT1_REG, 0x00) # Configure the
power management register openMPU6050
        self._write_byte(MPU_GYRO_CFG_REG,  config_gyro_range<<3) #
gyro sensor,?2000dps
        self._write_byte(MPU_ACCEL_CFG_REG, config_accel_range<<3)#
acceleration sensor ,?2g
        self._write_byte(MPU_SAMPLE_RATE_REG,0x01)#The     sampling
frequency >512
        self._write_byte(MPU_CFG_REG,0x00)#Set the digital low pass
filter to the first mode and the output frequency is 8KHz
        self._write_byte(MPU_INT_EN_REG,0X00) #Close all interrupts
        self._write_byte(MPU_USER_CTRL_REG,0X00) #I2C main mode off
        self._write_byte(MPU_FIFO_EN_REG,0X00) #close FIFO
        self._write_byte(MPU_INTBP_CFG_REG,0X80) #INT Pin low level
valid

        buf = self._read_byte(MPU_DEVICE_ID_REG)
        if buf != self._address:
            print("MPU6050 not found!")
        else:
            pass
```

Code 3 initialization in MPU6050

Considering the defects of MPU6050 mentioned before, we decided to give up using MPU-6050.

✓ Angle Range:±180°
✓ Stability:0.05°/s
✓ Protocol: UART

Figure 17 JY901

3.3.2 Two Communication Protocols: I2C and UART

MPU-6050 access other devices through I2C bus, which is a synchronous serial communication protocol, so data is transferred bit by bit along a single wire. I2C in Figure 18 only uses two wires to transmit data between devices, including SDA (Serial Data)-the line for the master and slave to send and receive data and SCL (Serial Clock) – the line that carries the clock signal.

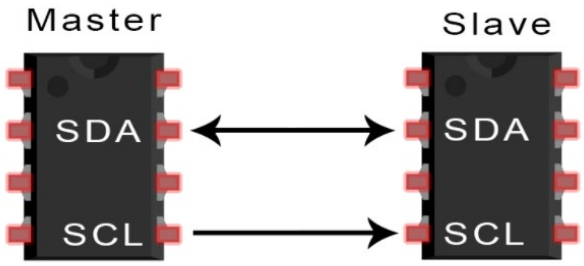

Figure 18 I2C Principle

With I2C, data is transferred in a message (Figure 19). The message is decomposed into data frames. Each message has an address frame that contains the binary address of the slave station and one or more data frames that contain the data being sent. The message also includes start and stop conditions, read/write bits, and ACK/NACK bits between each data frame.

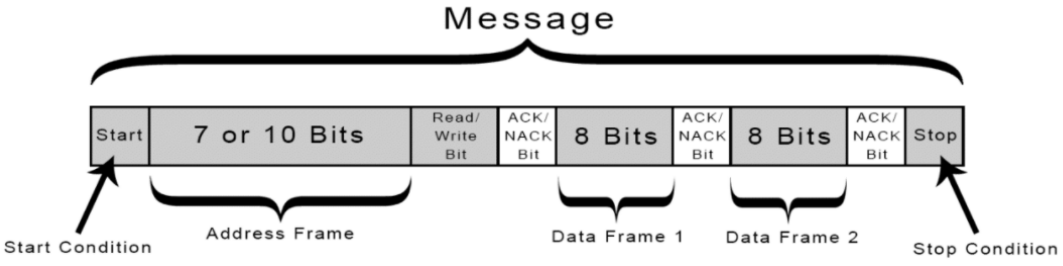

Figure 19 I2C message

Start condition: SDA line from high voltage level to SCL line from high to low voltage level.

Stop condition: SDA line is switched from low voltage level to SCL line from low to high voltage level.

Address frame: a unique 7-bit or 10-bit sequence for each slave device that identifies each slave when the master mother tongue wants to talk to it.

Read/write bits: Specifies whether the master sends data to or requests data from slave devices (low voltage levels).

ACK/NACK bit: Each frame in the message is followed by an acknowledgement/no acknowledgement bit. If the address frame or data frame is successfully received, it is returned from the receiving device to the sender's ACK bits

We refer to the relative materials and find that there are two different methods to define I2C protocol in Micro Python language. ‘pyb’ and ’machine’ are two library functions with different grammar in define and use I2C protocol. We can write code either ‘from pyb import I2C’ or ‘from machine import I2C’’. The figures below illustrate the detailed codes for the two methods respectively.

For ‘pyb’:

```
i2c=I2C(1,I2C.MASTER,baudrate=400000)
men_write()
mem_read()
```

For ‘machine’:

```
i2c=I2C(scl='PB6',sda='PB7',freq=400000)
readfrom_mem()
writeto()
```

3.3.3 Data Processing

To exploit the data generated by JY901S, we need to obtain them via UART communication and then process them to be their usable form.

The initialization of UART execution in `main` is listed as follows:

```
UART_Gyroscope = UART(2)
UART_Gyroscope.init(38400, bits=8, parity=None, stop=1,
timeout_char=100)
```

```
global z_angle
count=10000
UART_Gyroscope.irq(trigger = UART.IRQ_RXIDLE, handler =
UART_Gyroscope_ISR)
global original_angle
```

Code 4 UART initialization with Gyroscope

The global variables are defined for future applications.

Data processing is finished in the interrupt service routine (ISR), where the function `DueData` is called.

```
def UART_Gyroscope_ISR(t):
    global turn_angle
    global count
    global signal
    global command
    global original_angle
    global s_original_angle
    global angle
    global z_angle
    global y_angle
    msg_Gyroscope=UART_Gyroscope.read(UART_Gyroscope.any())
    angle = jy901.DueData(msg_Gyroscope)
    z_angle=angle[0]
    y_angle=angle[1]

    if type(z_angle)==float:
      if count==0:
          original_angle=z_angle
          s_original_angle=z_angle
      count=count+1
    return
```

Code 5 Gyroscope ISR

Here the condition `if type(z_angle)==float:` is invoked in case a null value is read. Again, global variables are defined for future application because an ISR can neither receive parameters nor return any.

The module that processes data is listed as follows:

```
from pyb import UART
from time import sleep
```

```
ACCData=[0.0]*8
GYROData=[0.0]*8
AngleData=[0.0]*8
FrameState = 0
Bytenum = 0
CheckSum = 0

a = [0.0]*3
w = [0.0]*3
Angle = [0.0]*3


def DueData(inputdata):
    global  FrameState
    global  Bytenum
    global  CheckSum
    global  a
    global  w
    global  Angle
    for data in inputdata:
        if FrameState==0:
            if data==0x55 and Bytenum==0: #Start reading at 0x55 and
increase bytenum
                CheckSum=data
                Bytenum=1
                continue
            elif data==0x51 and Bytenum==1:
                CheckSum+=data
                FrameState=1
                Bytenum=2
            elif data==0x52 and Bytenum==1:
                CheckSum+=data
                FrameState=2
                Bytenum=2
            elif data==0x53 and Bytenum==1:
                CheckSum+=data
                FrameState=3
                Bytenum=2
        elif FrameState==1: # acc

            if Bytenum<10:           # read 8 bits
                ACCData[Bytenum-2]=data
```

```
                CheckSum+=data
                Bytenum+=1
            else:
                if data == (CheckSum&0xff):
                    a = get_acc(ACCData)
                CheckSum=0
                Bytenum=0
                FrameState=0
        elif FrameState==2: # gyro

            if Bytenum<10:
                GYRODData[Bytenum-2]=data
                CheckSum+=data
                Bytenum+=1
            else:
                if data == (CheckSum&0xff):
                    w = get_gyro(GYRODData)
                CheckSum=0
                Bytenum=0
                FrameState=0
        elif FrameState==3: # angle

            if Bytenum<10:
                AngleData[Bytenum-2]=data
                CheckSum+=data
                Bytenum+=1
            else:
                if data == (CheckSum&0xff):
                    Angle = get_angle(AngleData)
                CheckSum=0
                Bytenum=0
                FrameState=0
    return Angle


def get_acc(datahex):
    axl = datahex[0]
    axh = datahex[1]
    ayl = datahex[2]
    ayh = datahex[3]
    azl = datahex[4]
    azh = datahex[5]
```

```
    k_acc = 16.0

    acc_x = (axh << 8 | axl) / 32768.0 * k_acc
    acc_y = (ayh << 8 | ayl) / 32768.0 * k_acc
    acc_z = (azh << 8 | azl) / 32768.0 * k_acc
    if acc_x >= k_acc:
        acc_x -= 2 * k_acc
    if acc_y >= k_acc:
        acc_y -= 2 * k_acc
    if acc_z >= k_acc:
        acc_z-= 2 * k_acc

    return acc_x,acc_y,acc_z


def get_gyro(datahex):
    wxl = datahex[0]
    wxh = datahex[1]
    wyl = datahex[2]
    wyh = datahex[3]
    wzl = datahex[4]
    wzh = datahex[5]
    k_gyro = 2000.0

    gyro_x = (wxh << 8 | wxl) / 32768.0 * k_gyro
    gyro_y = (wyh << 8 | wyl) / 32768.0 * k_gyro
    gyro_z = (wzh << 8 | wzl) / 32768.0 * k_gyro
    if gyro_x >= k_gyro:
        gyro_x -= 2 * k_gyro
    if gyro_y >= k_gyro:
        gyro_y -= 2 * k_gyro
    if gyro_z >=k_gyro:
        gyro_z-= 2 * k_gyro
    return gyro_x,gyro_y,gyro_z


def get_angle(datahex):
    rxl = datahex[0]
    rxh = datahex[1]
    ryl = datahex[2]
    ryh = datahex[3]
```

```
    rzl = datahex[4]
    rzh = datahex[5]
    k_angle = 180.0

    angle_x = (rxh << 8 | rxl) / 32768.0 * k_angle
    angle_y = (ryh << 8 | ryl) / 32768.0 * k_angle
    angle_z = (rzh << 8 | rzl) / 32768.0 * k_angle
    if angle_x >= k_angle:
        angle_x -= 2 * k_angle
    if angle_y >= k_angle:
        angle_y -= 2 * k_angle
    if angle_z >=k_angle:
        angle_z-= 2 * k_angle
    return angle_z,angle_y
```

Code 6 Module jy901

The data is collected as byte variables `inputdata`, which is received as an input of function `DueData`. In `DueData`, the header of the input data is removed, and the remaining data is classified according to their starting bits, which represent their address in JY901S. Once it is determined whether it represents acceleration, gyroscope reading or angle, the data is passed to corresponding specified functions. In our project, y angle and z angle are required, so we extract them only.

3.3.4 Comparison between 6-axis Scheme and 9-axis Scheme

We need the gyroscope to decide whether the car has completed its turn and reached its target angle. At first, we chose for our jy901S the 9-axis gyro sensor consisting of 3 accelerometer axes, 3 gyroscope axes, and 3 magnetometer axes additional to its 6-axis counterpart. The magnetometer measures the magnetism of the earth, which automatically determines the zero position for z-angle. Accordingly, its 180 degree and -180 degree axes, which appear to be the same one, are also decided. This brings difficulty to our operation because we cannot set 0 by ourselves, and we might encounter the problem that the car should cross the -180/180 line, which gives rise to a jump in angle value.

We solved the problem by setting a global variable *count* that counts the times of interrupt. Whenever we need to set a starting position for a turning, where the gyroscope is involved, we set *count* to 0, and record the current value read from the gyroscope. The difference between current axis and the original one is then calculated each time a new current position is read. The problem brought by the -180/180 intersection was solved through classification of scenarios. When the original angle is between -90° and -180° and the target angle is between 90° and 180°, we have:

$$delta = original\ angle - target\ angle + 360°,$$

and when the original angle is between 90° and 180° and the target angle is between -90° and -180°, we have:

$$delta = original\ angle - target\ angle - 360°.$$

In this way, our delta angle is guaranteed to be in the range of [0,360°].

However, another problem is encountered when we began to test our algorithm on the car. The car is discovered to turn to the position that is quite different from our desired one. It is then found out that the angle read by the gyroscope is far from accurate. For instance, it read 270 degrees when we turn it 180 degrees up. Still worse, the situation was not improved much after calibration.

It is then discovered that when we switched to 6-axis scheme, that is, abandoning the magnetometer axes, the behavior of the gyroscope turned out surprisingly good. The reading was accurate, and the zero position can be set. However, as the zero-position problem has been solved in our algorithm, we decided not to change the it, which can also be used in 6-axis scheme.

3.3.5 Applications

3.3.5.1 Application1: turning on the spot

An essential application involving JY901S is to assist the car to complete accurate turnings of arbitrarily given angles. To achieve this goal, we read the z angle at the very moment of instruction and record it as `original_angle`. Our target angle is calculated as

$$target\_angle = original\_angle + turn\_angle,$$

where $turn\_angle$ comes from instruction from OpenMV. We then update `z_angle` each time a new one is read in ISR, and this is where the global variable `z_angle` comes into use. The difference `delta` is also updated each time `z_angle` changes. Note that in JY901S, `delta`>0 implies a counterclockwise turning and `delta`>0 implies a clockwise one. When the absolute value of `delta` is close enough to zero, it is regarded that the turning task is accomplished. A function `turn` is defined for the car to turn, making two wheels on one side to turn forward and two wheels on the other side to turn backward. The code for function `turn` is listed below:

```
def turn(delta):
    if delta>0:                     #clockwise
        if delta>20:
            Car.run_(0,40,40,0,40,0,0,40)
            time.sleep(0.4)
```

```
        if delta<20:
            Car.run_(0,30,30,0,30,0,0,30)
            time.sleep(0.2)
            Car.run_(0,0,0,0,0,0,0,0)
            time.sleep(0.1)
    if delta<0:                      #anticlockwise
        if delta<-20:
            Car.run_(40,0,0,40,0,40,40,0)
            time.sleep(0.4)
        if delta>-20:
            Car.run_(30,0,0,30,0,30,30,0)
            time.sleep(0.1)
            Car.run_(0,0,0,0,0,0,0,0)
            time.sleep(0.1)
```

Code 7 Turning

When the absolute value of `delta` is below 10°, the pwm input for the wheels is set to be lower than normal. This is to ensure fine adjustments to be made so that the car would stop at a precise angle.

3.3.5.2 Application2: Send down-bridge signal to OpenMV

When the car goes down from the bridge, OpenMV should be informed to get back to the state of tracing. This requires a signal from stm32, which is generated when JY901S senses a large y angle. The code for this coincides with the first application in terms of UART and ISR. The different part is listed as below:

```
global y_angle
#       print("y_angle:",y_angle)
        if(y_angle>10):
            enableCamera=1
            print("sent enable camera.")
            UART_OpenMV.write(str(enableCamera))
```

Code 8 Down-bridge signal

3.3.5.3 Application3: dynamic self-adaption direction maintenance

The third application of gyroscope is dynamic self-adaption direction maintenance in order to make car go straight. The key element of this task is to ensure the running route be an approximately straight line within tolerate drift angle as 0.1°. It is critical in both patio 1 and patio 2.

The first method we tried was to use the speed feedback of the wheel to make the right and left

sides of the wheel the same speed by setting the PID with reasonable parameters. In other words, it is to walk in a straight line. Based on this, we have done work on the encoder, converting the number of pulses received by the encoder into speed, and adding the PID algorithm to achieve the same speed on both sides.

```
err = target - now #now:'count'
pwm = pwm + self.kp*(err - last_err) + self.ki*err + self.kd*(err -
last_err)
if (pwm >= self.pwm_range):
    pwm = self.pwm_range
if (pwm <= -self.pwm_range):
    pwm = -self.pwm_range
last_err = err
return pwm
```

Code 9 PID algorithm

However, the fact is not what we imagined. Even if we can make the rotation speed on both sides of the wheel exactly and quickly, due to the unevenness of the ground, the accidental idling will cause the deviation of the car. Especially on the cobblestone pavement of the second patio.

Figure 20 Cobblestone pavement

Gyroscope was then implemented to accurately perceive the angle of the current car and adjusted the output values of the PWM motors on both sides by analyzing the angle offset. The simplified version of the logic is as follows: if the angle is greater than 0.1, the car is tilted to the left, then we will increase the PWM output of the left wheel; if the angle is less than 0.1, the car is tilted to the right, then we will increase the PWM output of the right wheel.

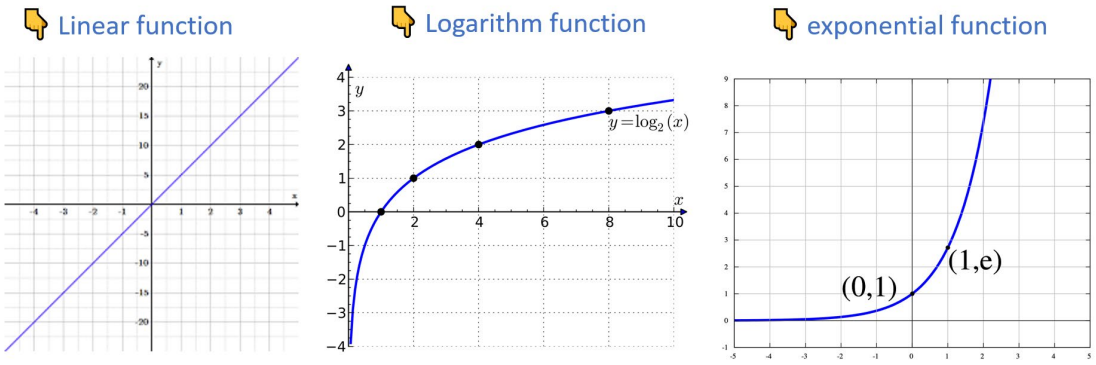

Figure 21 Three functions

The final solution: Since the change of PWM should be related to the change of angle, by comparing three functional relationships, including linear function, in function and exponential function, we find that the exponential function changes little when the angle is small, and the changes are obviously when the angle is large, this feature satisfies our idea of fine-tuning in small angles and drastically adjusting in large angles.

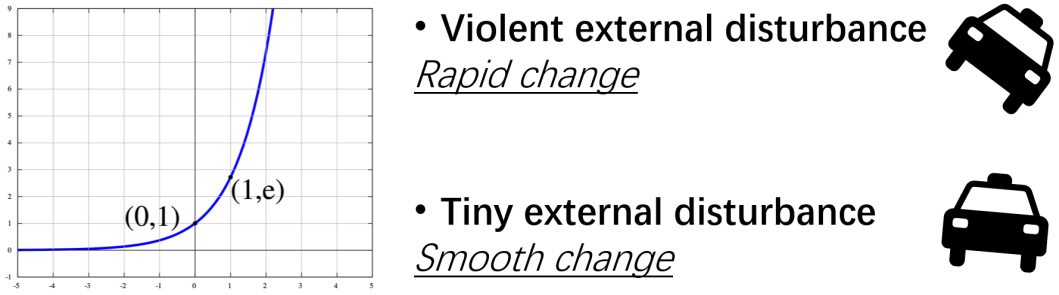

Figure 22 The merit of exponential funtion

In addition, we found that adding negative feedback (that is, the current PWM value is the last PWM value) and limiting the maximum PWM difference between the two wheels of the trolley will make the trolley adjust more quickly and the offset when the trolley goes straight will be smaller. The car has outstanding performance on the cobblestone road.

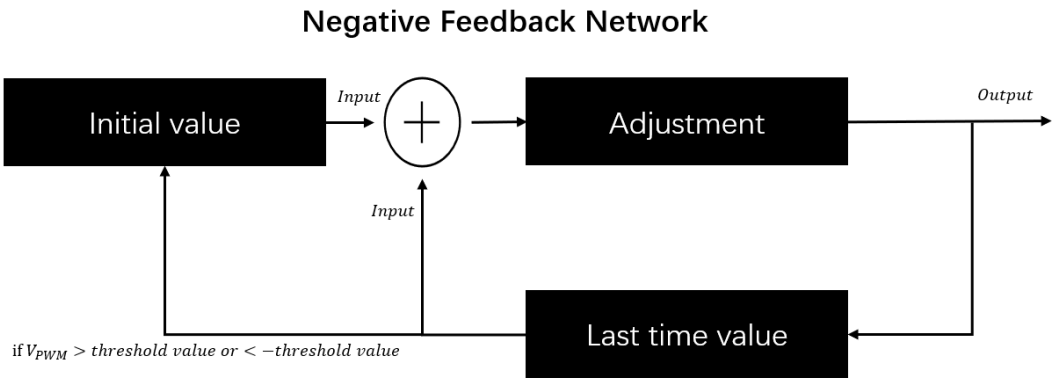

Figure 23 negative feedback network

```
def releasing():
    i=6.3 #left
    m=8.5 #right
    flag=1
    while flag:
        i+=0.1
        m+=0.1
        ch.pulse_width_percent(i)
        ch1.pulse_width_percent(m)
        if m>=12:
            m=12
        if i>=11.8 and m>=12:
            #print(i)
            #print(m)
            i=11.8
            m=12
            ch.pulse_width_percent(i)
            ch1.pulse_width_percent(m)
            time.sleep(2)
            while i>=6.3 or m>=8.5:
                if i>=6.3:
                    i=i-0.1
                if m>=8.5:
                    m=m-0.1
                print(i)
                print(m)
                time.sleep(0.3)
                ch.pulse_width_percent(i)
                ch1.pulse_width_percent(m)
                if i<6.3 and m<8.5:
                    flag=0
```

Code 12 Robotic Arm

3.6 Car group: Debugging (The Breakdown of STM32 Boards)

Section Author	**Ziyang Long** **(UESTC ID:2018190502030, UofG ID:2429503L)**
Technically Assisted by	Yuchen Yao (UESTC ID:2018190602001, UofG ID:2429207Y)

Weizhe Zhao
(UESTC ID:2018190606004, UofG ID: 2429361Z)

Preface:

As one invisible task, debugging was often neglected to demonstrate, however, bugs probably are the most troublesome thing in both software and hardware, especially in hardware. Most of the problems are proved to be simple afterwards, it took a lot of time to investigate.

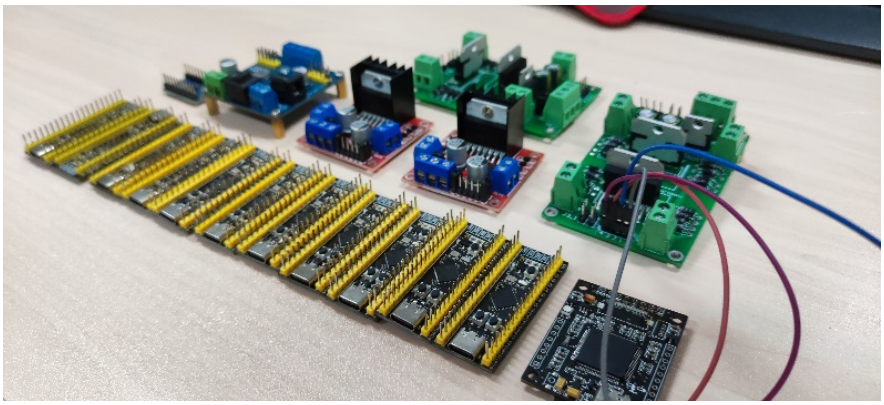

Figure 37 Boards' graveyard

3.6.1 Wrong Design in L298N PCB

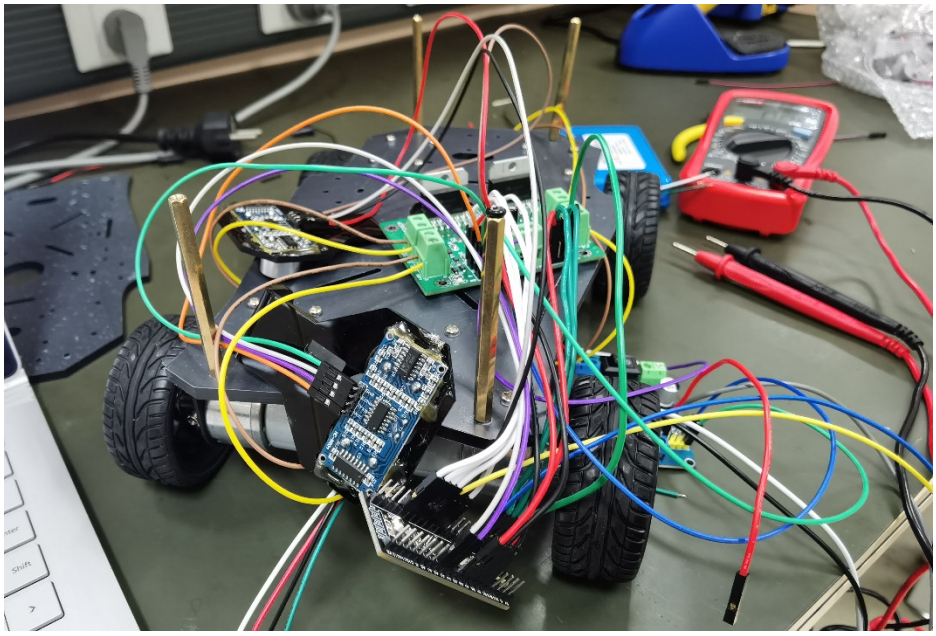

Figure 38 The first breakdown STM32 board

The first two STM32' broken was found when the power LED of STM32 board die out and cannot read the data from STM32 board when we use type C line to connect it with laptop. By checking other part of module, we focus on the problem in L298N. We design the initial L298N

PCB based on a reference file found on the internet. With only a sketchy glance of its design, we solder L298N boards, and we merely checked its power supply.

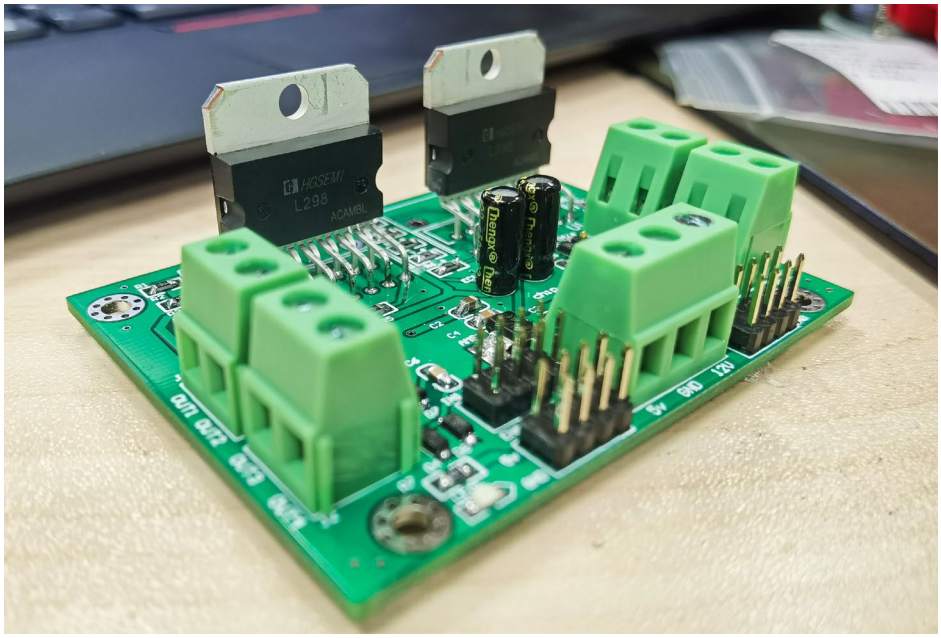

Figure 39 initial wrong design L298N module

By using Altium designer to scrutinize every detail of that PCB design in Figure 40, we found that the input pins were wrongly connect to the 5V power supply line, which is too obscure to notice. This mistake causes the reverse breakdown of STM32 board.

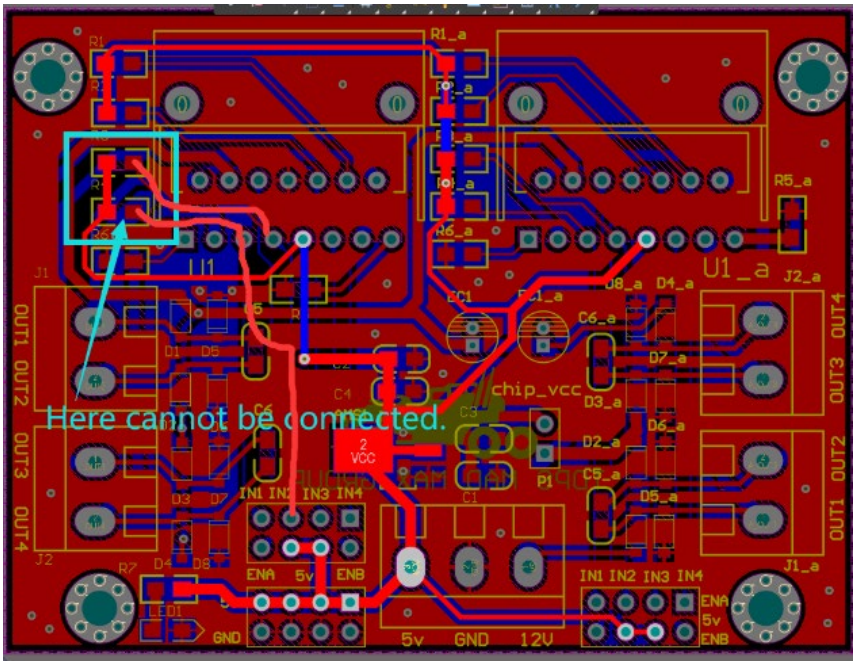

Figure 40 short circuit in PCB design

3.6.2 The Drawback of Directly Connecting L298N Module

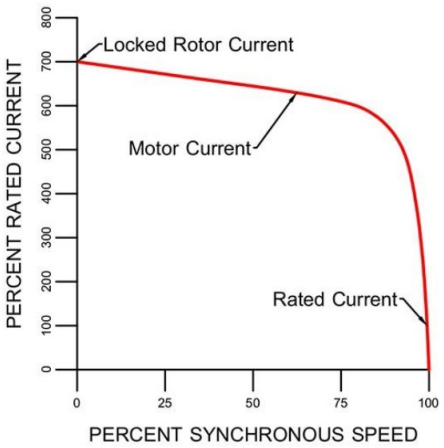

Figure 41 locked rotor current

Without any protection circuit or isolation methods, we found the STM32 will be broken by directly connecting to L298N module after using for a long time. Overcurrent and overvoltage to the Pin port are the main reason. When car's rotor is kept stationary or in other words rotor is not spinning or rotating, it will generate locked rotor current, which basically drawn by the motor at its rated voltage. The maximum current for all pins of the STM32 is 150mA[6], once the locked rotor current exceeds this value for a long period, the Pin port of STM32 will break. The rotor's frequently back and forth switch will let the voltage applied at its terminal be rated voltage of motor. This voltage sometimes high enough to breakdown the diode after the Pin port or the causing the damage of STM32.

3.6.3 Human Error-wrong Wire Connection

The 3.3V Pin port and GND Pin port is very close in STM32. When using multimeter to check the output voltage, it is common to make them short out. The 5V and 3.3V' misuse may breakdown the regulator inside the STM32 causing the damage due to negligence of team member.

3.6.4 Solution: Adding Optical Coupler Isolation between STM32 and L298N

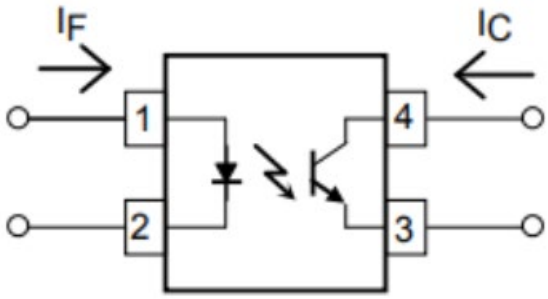

Figure 42 Circuit inside optocoupler

The structure of the optocoupler (Figure 42) is equivalent to the light-emitting diode and photosensitive triode packaged together. The working principle is the process of electricity - light - electricity. The working current drives the light-emitting diode to emit light of a certain wavelength, which is received by the photosensitive triode to produce a certain photocurrent and output after amplification. The optocoupler isolation circuit realize target that no direct electrical connection between the isolated two parts of the circuit, especially between the low voltage control circuit-STM32 and the external high voltage circuit-L298N.

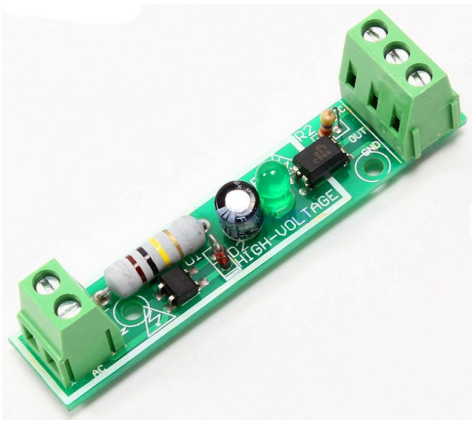

Figure 43 Optocoupler module

Snubber circuit is another potential method to avoid overvoltage, overcurrent and overheat. The inductor's storage and release of energy will remarkably decrease $\frac{di}{dt}$ and $\frac{dv}{dt}$. The switch on and switch off's voltage/current trajectory will be shaped by snubber circuit.

4 System Integration, Results and Discussion

4.1 Command Execution System (Car Group)

4.1.1 System Integration

4.1.1.1 Hardware

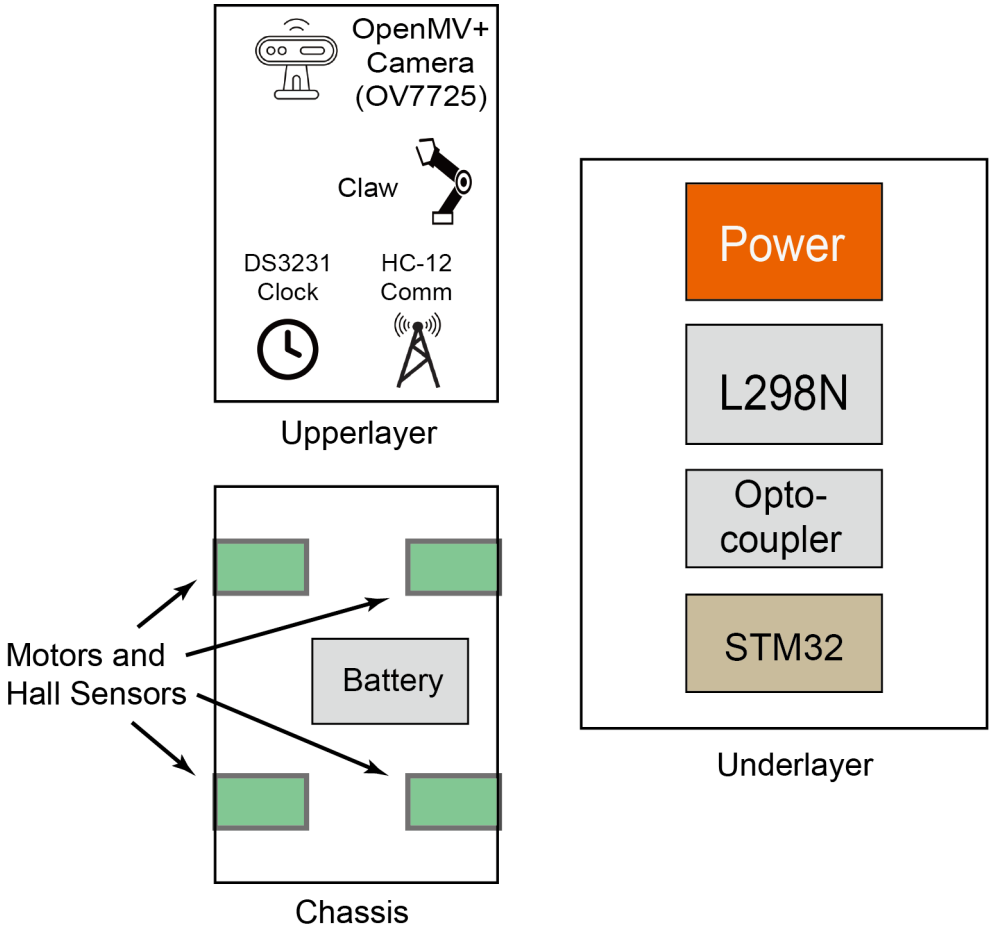

Figure 64 The Display of Modules in Three Layers

The main hardware components are illustrated in the above graph (Figure 64), including motor, hall element, L298N module, STM32 MCU, battery, HC-12 and claw. Note that one invisible module optical coupler module is between STM32 boards and L298N boards, it isolates the control circuit and drive circuit in electricity improving the safety and stability of MCU, which avoid the potential overvoltage or overcurrent in PWM value switching stage. The addition of the optical coupler module is because during our experimentation process, the high current overload easily breaks down STM32 boards. After several debugging steps and trials of changing boards and hardware, we find that the module is the optimal choice to entirely

eliminate the risk of STM malfunction.

4.1.1.2 Software: Mode Realization

The `main` function in stm32 is designed to contain several modes, each representing a different type of movement, for the convenience of external instructions. The modes are integrated in the same while loop according to their mode codes, which is illustrated in the figure below:

Bit 1: Mode Bit
- Mode = 0: STOP **0**
- Mode = 1: PWM Mode **1**210210 PWM10, Go forward
  - Continuous Control by PID in OpenMV
  - Bit 2-4: Left PWM
    - Bit 2 = 1: Negative PWM (backward)
    - Bit 2 = 2: Positive PWM (forward)
    - Bit 3-4: Magnitude(00-99)
  - Bit 5-7: Right PWM
    - Bit 5 = 1: Negative PWM (backward)
    - Bit 5 = 2: Positive PWM (forward)
    - Bit 6-7: Magnitude(00-99)
- Mode = 2: Revolving on the spot **2**1090 Turn left for 90 degree
  - Bit 2-5: Angle Value
    - Bit 2 = 1: Turn Left
    - Bit 2 = 2: Turn Right
    - Bit 3-5: 0-360
- Mode = 3: Turn while Moving **3**1020 turn left for 20 degree while moving
  - speed is defined by the car
  - Bit 2-5: Angle Value
    - Bit 2 = 1: Turn Left
    - Bit 2 = 2: Turn Right
    - Bit 3-5: 0-360
- Mode = 4: Straight Line Algorithm 1 **4**
- Mode = 5: Straight Line Algorithm 2 **5**
- Mode = 6: Robotic Arm **6**

Figure 65 Mode illustration

The decision which mode to enter is made by OpenMV, whose messages specifies the mode at the first bit. Since the messages, named `sum`, are in the syntax of string, they are easily truncated, whereby instructions are extracted. For example, the command bit is obtained as:

```
command_bit=int(sum[0:1])
```

Code 30 command_bit

a. **Mode 0: Stop**

Mode 0 brakes the car, giving all pins on L298N high input. This enables the car to make an abrupt stop.

```
i.  # mode0: stop
    1. if command_bit==0:
    2. Car.run_(0,0,0,0,0,0,0,0)
```

Code 31 Mode 0

b. **Mode 1: Receiving pwm control from OpenMV**

In this mode, the car receives all pwm controls from OpenMV, including pwm values and the angular velocity directions of each wheel. Basically, two wheels on the same side (i.e. left or right) are always given the same instructions for simplicity. Seven bits are needed in this mode, the first for mode switch and the others for car motion. Bits 2 to 4 give instructions to left wheels. A 2 on bit 2 indicates the left wheels turn forward and a 1 on this bit indicates backward movement. Bit 3 to 4 gives a pwm as a decimal figure with two digits, ranging from 0 to 99. The same rule is obeyed in bit 5 to 7 for right wheels.

```
# mode1: receiving PWM from OpenMV
    elif command_bit==1:
        if len(sum)!=7: #if wrong pattern received, move forward
            sum="1220220"
        else:
            left=int(sum[1:4])
            if left>200:
                NlB1=left-200
                NlB2=0
                NlC1=left-200
                NlC2=0
            else:
                NlB2=left-100
                NlB1=0
                NlC2=left-100
                NlC1=0
            right=int(sum[4:7])
            if right>200:
                NrA1=right-200
                NrA2=0
                NrD1=right-200
                NrD2=0
            else:
                NrD2=right-100
                NrD1=0
```

```
                    NrA2=right-100
                    NrA1=0
            Car.run_(NrA1,NrA2,NlB1,NlB2,NlC1,NlC2,NrD1,NrD2)
```

Code 32 Mode 1

c. **Mode 2: Turn on the spot**

In mode 2, the car complete turning instructions of arbitrary angle. This requires a command message of five bits, in which bit 2 implies whether the turning is clockwise or counterclockwise, and bit 3 to 5 gives the degree of turning. On entering mode 2, we set the global variable count to be zero, indicating an original angle is read and recorded from JY901S. While the turning is not completed, the program would stay in the while loop and new commands about modes are ignored, if and, during this period.

```
    # mode2:turning withour moving along
        elif command_bit==2:
            if len(sum)!=5:
                sum="21000" #if wrong pattern, stay still
            else:
                if int(sum[1:2])==1: #counterclockwise
                    turn_angle=int(sum[2:5])    # >0: anticlockwise,
<0: clockwise
                if int(sum[1:2])==2: #clockwise
                    turn_angle=-int(sum[2:5])
                count=0
                time.sleep(0.5)
                turn_complete=0
                target_angle=original_angle+turn_angle
                if target_angle<=-180:
                     target_angle=target_angle+360
                if target_angle>180:
                     target_angle=target_angle-360
                while(turn_complete==0):
                  if    z_angle<=-90    and    z_angle>=-180    and
target_angle<=180 and target_angle>=90:
                     delta=z_angle-target_angle+360
                  elif    z_angle>=90    and    z_angle<=180    and
target_angle>=-180 and target_angle<=-90:
                     delta=z_angle-target_angle-360
                  else:
                     delta=z_angle-target_angle
                     if z_angle-target_angle>=180:
                         delta=-(360-delta)
                     if z_angle-target_angle<=-180:
```

```
                    delta=-(360+delta)
                # print(delta,z_angle)
                if abs(delta)>=1:
                    print(delta,z_angle)
                    turn(delta)
                if abs(delta)<1:
                    # print(delta)
                    turn_complete=1
                    # Car.run_(0,0,0,0,0,0,0,0)
                sum="0"
```

Code 33 Mode 2

d. **Mode 3: Turning while moving**

Instead of making an accurate turning in place, in this mode, the car may move on and meanwhile adjust its direction according to the delta angle returned from OpenMV, characterizing the difference between its current orientation and the target one. The message for this mode consists of five bits, with one command bit, one bit indicating the direction of angular velocity of turning and three bits telling the absolute value of the delta angle.

```
# mode3: move while turning
    elif command_bit==3:
        if len(sum)!=5:
            sum="31000" #if wrong pattern, move forward
        else:
            if int(sum[1:2])==1:
                str_angle=int(sum[2:5])
            elif int(sum[1:2])==2:
                str_angle=-int(sum[2:5])
        Car.straight(40,40,40,40,str_angle)
```

Code 34 Mode 3

The essence of this mode is the function it calls named straight. In this function, we first decide whether the angle is valid. If it is too large, we ignore the instruction because in this mode the car is supposed to move in a relatively straight manner. If the returned angle lies between an acceptable interval, we adjust the speed of wheels to cater for the degree expected. The adjustment is based on the angle received. If the absolute value of the angle is below ten degrees, we make fine adjustments. Otherwise, we adjust one side of wheels to turn backwards and the other two to turn forward. For example, if we want the car to turn left, we instruct the right wheels to turn forward and the left ones to turn backward. In this way, our car can adjust its direction according to the instructions from OpenMV.

```
def straight(self, pwm_A, pwm_B, pwm_C, pwm_D, sum):
      self.NrA1 = pwm_A #NrA1=NrD1(same speed for wheels on the
```

```
same side)
          self.NlB1 = pwm_B
          self.NlC1 = pwm_C
          self.NrD1 = pwm_D
          step = 10
          wt=0.001
          v_base=40
          if(sum<70 and sum>-70):
             # increment=(sum/abs(sum))*math.exp(sum/2.2)
             increment=sum/2
             # increment=1

             if (sum<=10 and sum>=-10):
                 self.NrD1=v_base+increment
                 self.NrA1=v_base+increment
                 self.NlB1=v_base-increment
                 self.NlC1=v_base-increment
                 self.NrA2=0
                 self.NlB2=0
                 self.NlC2=0
                 self.NrD2=0
             if sum>10:
                 self.NrA1=v_base+increment
                 self.NrD1=v_base+increment
                 self.NlC2=v_base+increment
                 self.NlB2=v_base+increment
                 self.NrA2=0
                 self.NrD2=0
                 self.NlC1=0
                 self.NlB1=0
             if sum<-10:
                 self.NrA2=v_base-1.5*increment
                 self.NrD2=v_base-1.5*increment
                 self.NlC1=v_base-1.5*increment
                 self.NlB1=v_base-1.5*increment
                 self.NrA1=0
                 self.NrD1=0
                 self.NlC2=0
                 self.NlB2=0
          if(sum>70 or sum<-70):
             self.NrA2=0
             self.NlB2=0
```

```
            self.NlC2=0
            self.NrD2=0
          Car_motion.run_(self,   self.NrA1,self.NrA2,   self.NlB1,
self.NlB2, self.NlC1, self.NlC2, self.NrD1, self.NrD2)
          if abs(sum)>10:
            sleep(0.2)
          else:
            sleep(0.3)
          print(sum,'FL=',self.NlB1-self.NlB2,'FR=',self.NrA1-
self.NrA2,'BL=',self.NlC1-self.NlC2,'BR=',self.NrD1-self.NrD2)
          Car_motion.run_(self, 0, 0, 0, 0, 0, 0, 0, 0)
       sleep(0.5)
```

Code 35 Function straight

e. **Mode 4: Moving in a straight line on pebble ground**

In some parts in patio 2, the car is desired to move in a straight line. However, it is difficult for the car to keep direction on pebble ground, where its wheels may rotate without touching the ground, making it futile for control over wheel speeds. Thus, we designed a mode especially for this scenario, in which JY901S is invoked. To enter this mode, the message from OpenMV requires a mere command bit.

```
    # mode4: move along a stright line (patio 2)
        elif command_bit==4 and isInitialize==0:
            turn_angle=0
            count=0 #set another original angle
            time.sleep(0.5)
            a=[31,30,30,31]
            target_angle=s_original_angle+turn_angle
            isInitialize=1
            if target_angle<=-180:
                target_angle=target_angle+360
            if target_angle>180:
                target_angle=target_angle-360

        elif command_bit==4 and isInitialize==1:
            if     z_angle<=-90     and     z_angle>=-180     and
target_angle<=180 and target_angle>=90:
                delta=z_angle-target_angle+360
            elif z_angle>=90 and z_angle<=180 and target_angle>=-
180 and target_angle<=-90:
                delta=z_angle-target_angle-360
            else:
                delta=z_angle-target_angle
```

```
                if z_angle-target_angle>=180:
                    delta=-(360-delta)
                if z_angle-target_angle<=-180:
                    delta=-(360+delta)
             # print(delta,z_angle)

          a=Car.stra2(a[0],a[1],a[2],a[3],delta)
```

Code 36 Mode 4

The function `stra2` is called here to fulfill this task. This function adjusts the pwm value for wheels based on its last values. The code for it is listed below:

```
    def stra2(self, pwm_A, pwm_B, pwm_C, pwm_D, sum):
          self.NrA1 = pwm_A #NrA1=NrD1(same speed for wheels on the
same side)
          self.NlB1 = pwm_B
          self.NlC1 = pwm_C
          self.NrD1 = pwm_D
          self.NrA2=0
          self.NlB2=0
          self.NlC2=0
          self.NrD2=0
          yuzhi=0.1
          increment=abs(sum)
          rang=40#maximum differnece between speeds on two sides
          base=30
          if(sum<-yuzhi):
                self.NrA1=self.NrA1+increment
                self.NrD1=self.NrD1+increment
                self.NlC1=self.NlC1-increment
                self.NlB1=self.NlB1-increment
                if((self.NrA1-self.NlB1)>rang):
                   self.NlB1=base-(rang/2)
                   self.NlC1=base-(rang/2)
                   self.NrA1=base+(rang/2)
                   self.NrD1=base+(rang/2)
          if(sum>yuzhi):
                self.NlB1=self.NlB1+increment
                self.NlC1=self.NlC1+increment
                self.NrA1=self.NrA1-increment
                self.NrD1=self.NrD1-increment
                if((self.NlB1-self.NrA1)>rang):
                   self.NlB1=base+(rang/2)
```

```
                    self.NlC1=base+(rang/2)
                    self.NrA1=base-(rang/2)
                    self.NrD1=base-(rang/2)
          if(sum<=yuzhi and sum>=-yuzhi):
             pass
          Car_motion.run_(self,   self.NrA1,self.NrA2,   self.NlB1,
self.NlB2, self.NlC1, self.NlC2, self.NrD1, self.NrD2)
          print(sum,'FL=',self.NlB1-self.NlB2,'FR=',self.NrA1-
self.NrA2,'BL=',self.NlC1-self.NlC2,'BR=',self.NrD1-self.NrD2)
          sleep(0.1)
      return (self.NrA1, self.NlB1, self.NlC1, self.NrD1)
```

Code 37 Function stra2

f. Mode 5: Moving in a straight line across the bridge

When crossing the bridge, the car needs to follow a straight trace as well. However, our function `stra2` is designed to be sensitive to small degree changes, and the car might behave like swinging from side to side at times, making it not suitable for the bridge of a limited width. Mode 5 is designed for this task and performs smooth motion.

```
   # mode5: move along a straight line (bridge)
       elif command_bit==5 and isInitialize==0:
           turn_angle=0
           count=0 #set another original angle
           time.sleep(0.5)
           target_angle=s_original_angle+turn_angle
           if target_angle<=-180:
               target_angle=target_angle+360
           if target_angle>180:
               target_angle=target_angle-360
           isInitialize=1
           print(isInitialize)

       elif command_bit==5 and isInitialize==1:
           if     z_angle<=-90     and     z_angle>=-180     and
target_angle<=180 and target_angle>=90:
               delta=z_angle-target_angle+360
           elif z_angle>=90 and z_angle<=180 and target_angle>=-
180 and target_angle<=-90:
               delta=z_angle-target_angle-360
           else:
               delta=z_angle-target_angle
               if z_angle-target_angle>=180:
                   delta=-(360-delta)
```

```
            if z_angle-target_angle<=-180:
                delta=-(360+delta)
        a=Car.stra3(delta)
        global y_angle
#        print("y_angle:",y_angle)
        if(y_angle>10):
            enableCamera=1
            print("sent enable camera.")
        UART_OpenMV.write(str(enableCamera))
```

Code 38 Mode 5

One of the main differences between mode 5 and mode 4 is that in mode 5, stm32 returns a message to turn on the camera and informs OpenMV that it is going down the bridge. In addition, another function, namely `stra3` is called in this mode. This function is not iterative, ensuring a steady performance of the car.

```
    def stra3(self,delta):
        self.NrA1 = 36 #NrA1=NrD1(same speed for wheels on the
same side)
        self.NlB1 = 35
        self.NlC1 = 35
        self.NrD1 = 36
        step = 10
        wt=0.001
        v_base=35
        sum=delta
        if (sum==0):
           pass
        if(sum>0.1 or sum<-0.1):
           # increment=(sum/abs(sum))*math.exp(sum/2.2)
           increment=40*sum
           self.NrD1=v_base-increment
           self.NrA1=v_base-increment
           self.NlB1=v_base+increment
           self.NlC1=v_base+increment
           # self.NrA2=0
           # self.NlB2=0
           # self.NlC2=0
           # self.NrD2=0
        else:
           pass
        self.NrA2=0
        self.NlB2=0
```

```
          self.NlC2=0
          self.NrD2=0
          if(self.NrA1>50):
             self.NrA1 = 50#NrA1=NrD1(same speed for wheels on
the same side)
             self.NrD1 = 50
          if(self.NrA1<20):
             self.NrA1 = 20#NrA1=NrD1(same speed for wheels on
the same side)
             self.NrD1 = 20
          if(self.NlB1>50):
             self.NlB1 = 50#NrA1=NrD1(same speed for wheels on
the same side)
             self.NlC1 = 50
          if(self.NlB1<20):
             self.NlB1 = 20#NrA1=NrD1(same speed for wheels on
the same side)
             self.NlC1 = 20
          Car_motion.run_(self, self.NrA1,self.NrA2, self.NlB1,
self.NlB2, self.NlC1, self.NlC2, self.NrD1, self.NrD2)
          # sleep(0.2)
          print(sum,'FL=',self.NlB1-self.NlB2,'FR=',self.NrA1-
self.NrA2,'BL=',self.NlC1-self.NlC2,'BR=',self.NrD1-self.NrD2)
```

Code 39 Function stra3

g. **Mode 6: Operating the robotic arm**

If OpenMV returns a single '6', the car enters the mode of robotic arm operation. This is illustrated in detail in the parts of robotic arm. The code for mode 6 is simple, since most work is finished in the function `robotic_arm.releasing()`. After the execution of this function, we automatically set the command bit to 0 so that the car would stop for a while and wait for new instructions.

```
    elif command_bit==6:
           robotic_arm.releasing()
        sum="0"
```

Code 40 Mode 6

4.1.2 Result and Discussion

4.1.2.1 Patio1

Tracing Response

How to make car go straight is an issue that have troubled us for a long time. The key factor of the task is to ensure the running route be an approximately straight line within tolerate drift angle as 0.1. It is critical in both patio1 and patio 2.

The first solution we tried was to use the velocity feedback of the wheels to enable the right and left sides of the wheel the same speed by setting the PID with reasonable parameters. In other words, it is to walk in a straight line. Based on this, we converted the number of pulses received by the encoder into velocity as well as added the PID algorithm to achieve the same speed on both sides.

However, the fact is not what we imagined. Even if we can make the rotation speed on both sides of the wheel exactly and quickly, due to the unevenness of the ground, the accidental idling will cause the deviation of the car. Especially on the cobblestone pavement of the second patio.

Figure 66 Cobblestone pavement

The second attempt was to use a gyroscope to accurately perceive the angle of the current car, and adjusted the output values of the PWM motors on both sides by analyzing the angle offset. The simplified version of the logic is as follows: if the angle is greater than 0.1, the car is tilted to the left, then we will increase the PWM output of the left wheel; if the angle is less than 0.1, the car is tilted to the right, then we will increase the PWM output of the right wheel. The first gyroscope we tried was MPU6050, (the code above, the part with the problem). However, due to the long time required to obtain the speed of this chip, we could not get the offset of the small angle in time, so we finally gave up this scheme.

The final solution: Since the change of PWM should be related to the change of angle, by comparing three functional relationships, including linear function, ln function and exponential function, we found that the exponential function changes little when the angle is small, and the changes are obviously when the angle is large, This feature satisfies our idea of fine-tuning in small angles and drastically adjusting in large angles. In addition, we have tried two different logic methods. We found that adding negative feedback (that is, the current PWM value is the

last PWM value) and limiting the maximum PWM difference between the two wheels of the trolley will make the car adjust more quickly and the offset will be smaller when the car goes straight. The car has outstanding performance on the cobblestone road.

```
import pyb
import MPU6050
 '''
    input parameters: current enc
   output parameters: current pwm value
'''
 class PID:
    def __init__(self, pwm_range, kp_A, ki_A, kd_A, kp_B, ki_B,
kd_B, kp_C, ki_C, kd_C, kp_D, ki_D, kd_D):
        #pwm_range: maximum pwm value
        #A:B8-B7, front-right
        #B:B6-B5, front-left
        #C:B1-A6, rear-left
        #D:B0-A7, rear-right
        pwm_A = 0 #pwm_A = Nr1-Nr2
        pwm_B = 0
        pwm_C = 0
        pwm_D = 0
        err = 0
       err_A = 0
        err_B = 0
        err_C = 0
        err_D = 0
        last_err_A = 0
        last_err_B = 0
        last_err_C = 0
        last_err_D = 0
        self.pwm_range = pwm_range
        self.kp_A = kp_A
        self.ki_A = ki_A
        self.kd_A = kd_A
        self.kp_B = kp_B
        self.ki_B = ki_B
        self.kd_B = kd_B
        self.kp_C = kp_C
        self.ki_C = ki_C
        self.kd_C = kd_C
        self.kp_D = kp_D
```

```
        self.ki_D = ki_D
        self.kd_D = kd_D
        self.pwm_A = 0
        self.pwm_B = 0
        self.pwm_C = 0
        self.pwm_D = 0
        self.err = err
        self.err_A = err_A
        self.err_B = err_B
        self.err_C = err_C
        self.err_D = err_D
        self.last_err_A = last_err_A
        self.last_err_B = last_err_B
        self.last_err_C = last_err_C
        self.last_err_D = last_err_D

    # increment pid
     def incremental_pid(self, now, target):
         pwm += Kp[e(k) - e(k-1)] + Ki*e(k) + Kd[e(k) - 2e(k-1) +
e(k-2)]

        '''
         err = target - now #now:'count'
        pwm = pwm + self.kp*(err - last_err) + self.ki*err +
self.kd*(err - last_err)
         if (pwm >= self.pwm_range): # limit the amplitude
             pwm = self.pwm_range
         if (pwm <= -self.pwm_range):
             pwm = -self.pwm_range
         last_err = err
         return pwm

    def pid_A(self, now, target):
         self.err_A = target - now
         self.pwm_A = self.pwm_A + self.kp_A * ( self.err_A -
self.last_err_A ) + self.ki_A*self.err_A + self.kd_A*(self.err_A -
self.last_err_A)
         if (self.pwm_A >= self.pwm_range):
             self.pwm_A = self.pwm_range
         if (self.pwm_A <= -self.pwm_range):
             self.pwm_A = -self.pwm_range
         self.last_err_A = self.err_A
```

```
        # print("pwm_A", self.pwm_A)
        return self.pwm_A

    def pid_B(self, now, target):
        self.err_B = target - now
        self.pwm_B   =   self.pwm_B   +   self.kp_B*(self.err_B   -
self.last_err_B) + self.ki_B*self.err_B + self.kd_B*(self.err_B -
self.last_err_B)
        if (self.pwm_B >= self.pwm_range):
            self.pwm_B = self.pwm_range
        if (self.pwm_B <= -self.pwm_range):
            self.pwm_B = -self.pwm_range
        self.last_err_B = self.err_B
        return self.pwm_B

    def pid_C(self, now, target):
        self.err_C = target - now
        self.pwm_C   =   self.pwm_C   +   self.kp_C*(self.err_C   -
self.last_err_C) + self.ki_C*self.err_C + self.kd_C*(self.err_C -
self.last_err_C)
        if (self.pwm_C >= self.pwm_range):
            self.pwm_C = self.pwm_range
        if (self.pwm_C <= -self.pwm_range):
            self.pwm_C = -self.pwm_range
        self.last_err_C = self.err_C
        return self.pwm_C

    def pid_D(self, now, target):
        self.err_D = target - now
        self.pwm_D   =   self.pwm_D   +   self.kp_D*(self.err_D   -
self.last_err_D) + self.ki_D*self.err_D + self.kd_D*(self.err_D -
self.last_err_D)
        if (self.pwm_D >= self.pwm_range):
            self.pwm_D = self.pwm_range
        if (self.pwm_D <= -self.pwm_range):
            self.pwm_D = -self.pwm_range
        self.last_err_D = self.err_D
        return self.pwm_D
```

code 41 car dynamic self-adaption direction maintenance

In order to ensure that the car can patrol the route steadily, we have tried many methods. At first, we decided to let OpenMV transmit the angle between the central axis and the car to STM32 through inter-board communication (the distance between the trolley and the center

line is not transmitted), but the effect is not good: After a good number of trials, it is easy for the car to deviate from the route it needs to follow.

Next, we tried two straight-line algorithms, the exponential function algorithm with negative feedback (mode 4) and the linear function algorithm (mode 5). When we call mode4, the probability of error is small when the car is patrolling the line, but the car twists and turns when it moves forward, which affects the speed of the car and wastes a lot of time. When we call mode 5, the car can complete a short distance perfectly, but when it needs to travel a long distance, it will deviate a lot from the central axis.

After many trials by our team members, we have come up with a relatively rigorous solution: command OpenMV to transmit an 8-bit command, Bit1 is 1 means that we will use PWM Mode (continuous control by PID in OpenMV) Bit2-7 directly transfers the PWM value information to the four wheels, which has achieved good results. Since the exact PWM value makes our control of the wheels more precise, avoiding the situation that the car greatly deviates from the route.

Figure 67 the car is patrolling the line while turning

In-Position Turning

In Patio 1, there are two points where each a 90 degree in-position turn is required. One is at the beacon before the bridge, the other one is right after crossing the bridge.

At first point, once the beacon is detected, Open MV sends signal “22090” to STM32, ordering Mode 2 (in-position turning) and a clockwise 90 degree turn. Receiving the order, STM32 first gets an original angle from gyroscope and sets a target angle accordingly. In the turning process, gyroscope keeps reading current angle for STM32 to make comparison with the target one. We set the tolerance to be ±1 degree, i.e., the turning process is finished once |current - target|≤ 1. The second turn is carried out in a similar way.

In field tests, we find that the car often turns over 90 degree and figure out the reason: the area around turning points is smooth enough for the car to make turns rapidly, so even though the

gyroscope is real-time and accurate, it still turns over easily due to inertia. To fix such fault, we set the car to turn at a lower speed when the turning is about to be complete (|current - target|≤ 20).

```
def turn(delta):
    if delta>0:                         #clockwise
        if delta>20:
            Car.run_(0,50,50,0,50,0,0,50)
            time.sleep(0.4)
        if delta<20:
            Car.run_(0,30,30,0,30,0,0,30)
            time.sleep(0.2)
            Car.run_(0,0,0,0,0,0,0,0)
            time.sleep(0.1)
    if delta<0:                         #anticlockwise
        if delta<-20:
            Car.run_(50,0,0,50,0,50,50,0)
            time.sleep(0.4)
        if delta>-20:
            Car.run_(30,0,0,30,0,30,30,0)
            time.sleep(0.1)
            Car.run_(0,0,0,0,0,0,0,0)
            time.sleep(0.1)
```

Code 42 Improved Turning Program

After adopting this solution, the in-position turning orders are always executed perfectly in Patio 1.

Crossing Bridge

In field tests, we find that there is no problem with climbing up the bridge or running down the slope, since the wheels of our car have good track adhesion and the 12-Volt motors are powerful enough. Therefore, we just focus on how to move straight when crossing the bridge.

As there is no line for Open MV to carry out tracing, the car can only depend on the gyroscope to determine direction, i.e., Mode 4 and Mode 5 are to choose from.

Although Mode 4 stands for being sensitive to deviation, it also means the car would response in a dramatic way, adding risks for the slopes. As a result, we choose Mode 5. In Mode 5, the car adjusts direction in a milder way than in Mode 4. Despite possible additive deviation, Mode 5 shows great performance in a short distance, therefore is suitable to be applied in bridge crossing.

Above all, field tests have witnessed the correctness of our choice.

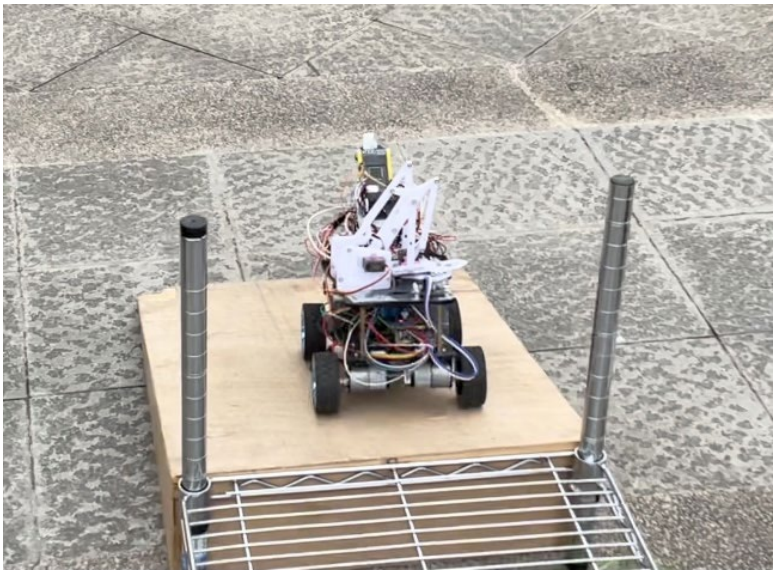

Figure 68 Crossing Bridge

4.1.2.2 Patio2

A: Self-adaptive direction control mode

In task 2, dynamic self-adaptive direction control mode is widely implemented in both the smooth tile surface and the rough cobblestone surface (Figure 69). It successfully makes the car go straight in two surfaces with quite different friction coefficients. We overcome the difficulties that the tire is easy to slip on tile surface and the wheel is likely to stuck in cobblestone.

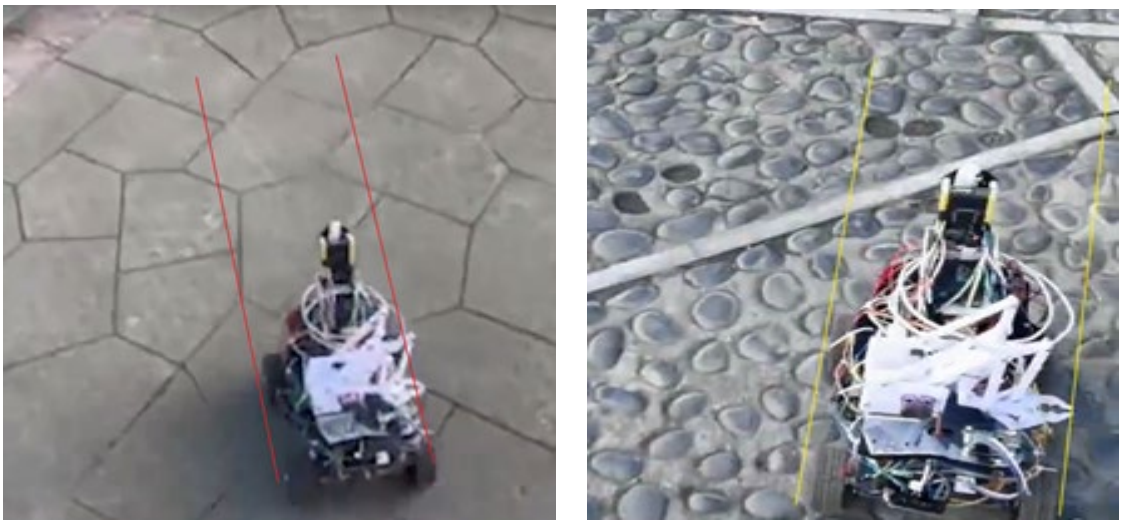

Figure 69 Running along a straight line in tile surface and cobblestone surface

Compared with fixed PWM value adjustment per second, the dynamic self-adaptive direction control mode use both exponential functions related to deviation angle

$$\text{Adjustment}=|\text{deviation angle}|\times e^{|\text{deviation angle}|/2.2}$$

and negative feedback to get better performance in going straight. The adjustment is relevant to the magnitude of disturbance, this algorithm first quantizes the influence of the uneven surface of the ground and the stall of the wheel into the deviation angle mapping to yaw axis (rotate on car's up vector)

Based on self-adaptive, the angle adjustment is divided into three stages due to the on-spot test experience, stable zone, tremor zone, and fluctuation zone. We set the different magnifications for adjustments in three zones. Moreover, to avoid the circumstance that the wheel stuck on the pebble, we amplify the PWM output value to the motor in pebble road. On the contrary, the car runs slowly in the tile surface. The car can automatically distinguish the different roads by judging the angle fluctuation in pitch axis (rotate on the cross product of the other camera's up and direction vectors).

B: Turing:

Turning on the spot plays an indispensable role in performing the tasks 2, which has been repeatedly implemented and has great influence on the positioning of the car. It is essential for car to turn on the spot at a target angle precisely. To be specific, turning at an angle precisely is the foundation for the car to track the desired route, as the car may fail to approach the desired position without a correct initial direction. We have designed algorithm to cope with the turning problem.

Our algorithm enables the car to turn at a precise angle with the tolerance of 1 degree and can revise the turning angle of the car automatically with the use of gyroscope.

We defined delta as the difference angle between the target angle received from openmv and the current angle provided by the gyroscope, where both the target angle and current angle are ranging from -180 degree to 180 degree, which determines the turning direction and the turning angle. To be specific, if delta is greater than 0, the car rotates clockwise and rotates anticlockwise when delta less than 0. However, we faced with a challenge due to the range of delta, for instance, the problem occurs when target angle=179°, while current angle=-179°, the actual result we want is to turn clockwise for 2°, the result will be turning anticlockwise for 358°. In order to solve this problem, we adjust the range of delta by dividing the turning in to several conditions, for instance, when:

$$90° \leq \text{target ange} \leq 180° \quad \text{and} \quad -180° \leq \text{current angle} \leq -90°$$
$$\text{delta} = \text{current angle} - \text{target angle} + 360°$$

With the algorithm, delta is adjusted into the range of -180° to 180°. And the car will repeatedly revise the angle until the delta angle is less than 1 degree. Moreover, the car will turn at a higher speed when delta angle is greater than 20°.

```
def turn(delta):
    if delta>0:                         #clockwise
```

```
        if delta>20:
            Car.run_(0,50,50,0,50,0,0,50)
            time.sleep(0.4)
        if delta<20:
            Car.run_(0,30,30,0,30,0,0,30)
            time.sleep(0.2)
            Car.run_(0,0,0,0,0,0,0,0)
            time.sleep(0.1)
    if delta<0:                      #anticlockwise
        if delta<-20:
            Car.run_(50,0,0,50,0,50,50,0)
            time.sleep(0.4)
        if delta>-20:
            Car.run_(30,0,0,30,0,30,30,0)
            time.sleep(0.1)
            Car.run_(0,0,0,0,0,0,0,0)
            time.sleep(0.1)
            if int(sum[1:2])==1: #counterclockwise
                turn_angle=int(sum[2:5])    # >0: anticlockwise,
<0: clockwise
            if int(sum[1:2])==2: #clockwise
                turn_angle=-int(sum[2:5])
            # global z_angle
            # UART_Gyroscope.irq(trigger = UART.IRQ_RXIDLE, handler
= UART_Gyroscope_ISR)
            # global original_angle
            count=0
            time.sleep(0.5)
            turn_complete=0
            target_angle=original_angle+turn_angle
            if target_angle<=-180:
                 target_angle=target_angle+360
            if target_angle>180:
                 target_angle=target_angle-360
            while(turn_complete==0):
              if z_angle<=-90 and z_angle>=-180 and
target_angle<=180 and target_angle>=90:
                 delta=z_angle-target_angle+360
              elif z_angle>=90 and z_angle<=180 and target_angle>=-
180 and target_angle<=-90:
                 delta=z_angle-target_angle-360
              else:
```

```
                delta=z_angle-target_angle
                if z_angle-target_angle>=180:
                    delta=-(360-delta)
                if z_angle-target_angle<=-180:                    #
取小于 180 的夹角
                    delta=-(360+delta)
            # print(delta,z_angle)
            if abs(delta)>=1:
                print(delta,z_angle)
                turn(delta)
            if abs(delta)<1:
                # print(delta)
                turn_complete=1
                # Car.run_(0,0,0,0,0,0,0,0)
                sum="0"
                 # UART_Gyroscope.deinit
```

C: Robotic Arm:

The robotic arm is necessary in task 2 to comply the task of releasing fish food. A two-degree-of-freedom robotic has been installed, which successfully performed the task in the last examination. The arm is designed to perform an action of throwing which can cast the fish food to the target position and then return to the original state in a relatively more sluggish speed. Besides, the size of the robotic arm is of great importance, the deepest diving down distance is up to 7cm, the largest arm span is up to 23cm, and the highest lifting distance is up to 15cm.

Figure 70 Size of robotic arm

The two joints of the arm are controlled by two servos (MG90S) respectively, where the rotation angle of servos can be adjusted by PWM value.

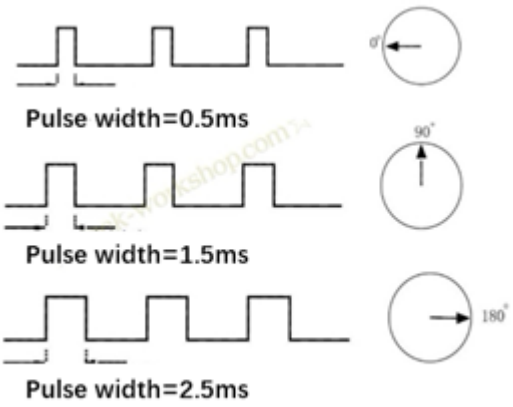

Figure 71 Working principle of servos

It is noteworthy that the PWM range is 0.25 to 12.5 under the condition that the period of the servos is 20ms. Based on measurement and trials, we set the PWM values of the two servos (which are in responsible for the control of right and left arms) to about 12. Moreover, the robotic arm is controlled by the instructions from OpenMV.

```
# main.py -- put your code here!
from pyb import UART, Pin,Timer
import time
from time import sleep

print("hello world")

A0 = Pin('PA0') #control the left steering gear engine
tim = Timer(2, freq=50)
ch = tim.channel(1, Timer.PWM, pin=A0)

B3 = Pin('PB3')# control the right steering gear engine
tim1 = Timer(2, freq=50)
ch1 = tim.channel(2, Timer.PWM, pin=B3)

ch.pulse_width_percent(6.3)
ch1.pulse_width_percent(8.5)

def releasing():
    i=6.3 #左边
    m=8.5 #右边
    flag=1
    while flag:
        i+=0.1
```

```
        m+=0.1
        ch.pulse_width_percent(i)
        ch1.pulse_width_percent(m)
        if m>=12:
            m=12
        if i>=11.8 and m>=12:
            #print(i)
            #print(m)
            i=11.8
            m=12
            ch.pulse_width_percent(i)
            ch1.pulse_width_percent(m)
            time.sleep(2)
            while i>=6.3 or m>=8.5:
                if i>=6.3:
                    i=i-0.1
                if m>=8.5:
                    m=m-0.1
                print(i)
                print(m)
                time.sleep(0.3)
                ch.pulse_width_percent(i)
                ch1.pulse_width_percent(m)
                if i<6.3 and m<8.5:
                    flag=0
```