

WRO 2024 Future Engineer



TEAM: YBRJSF

Content	Page
About our team	2
Our Robot Hardware	5
Obstacle Management	22

Portion 1: Insights into our team

Team Member:

1. Bhudit Thanaphakgosol (Right)
2. Saknun Sattham (Left)
3. Norapat Nimitkiatklai (Middle)



Team Background Information:

We are Bhudit, Saknun, and Norapat, senior students at Yothinburana School. Our journey together began in the YB Robot Club back in grade 7, and since then, we've become an inseparable team, united by our passion for robotics and innovation. Over

the years, we've cultivated a deep bond that extends beyond the walls of the classroom, driven by our shared goals and relentless dedication.

Saknun, our lead engineer, possesses an extraordinary talent for crafting intricate mechanical designs. His creations are not only innovative but also meticulously reliable, ensuring that every component functions seamlessly. Bhudit, our programming expert, breathes life into these mechanical designs with his precise and efficient coding. His expertise in programming transforms our ideas into functioning robots that operate with precision and intelligence. Norapat, the strategist of our team, ensures that our robots are not only technically sound but also optimized to excel in competitive environments. His strategic insight guides our decisions, making sure that every robot we build is positioned to win.

Together, we form Team YBR-JSF, a group of determined individuals who complement each other's strengths and share a common vision. Our journey has been filled with countless afternoons and weekends spent in the robotics lab, where we've worked tirelessly to refine our skills, explore new ideas, and push the boundaries of what we thought possible. We've faced numerous challenges in local robotics competitions, each one teaching us valuable lessons and strengthening our resolve.

Our goal for this year is clear and unwavering: we aim to qualify for the international robotics competition in Turkey. Last year, we came agonizingly close to achieving this milestone, but ultimately, we fell short. However, that experience has only fueled our determination. We've spent our summer break not only improving our robots but also expanding our knowledge, learning new techniques, and enhancing our teamwork. We are more prepared and more driven than ever before.

As we embark on our final year at Yothinburana School, we are ready to showcase the full potential of Team YBR-JSF. With the unwavering support of our classmates, teachers, and mentors, we are confident that this year will be our time to shine on the international stage. We are not just building robots; we are building a legacy, and we are ready to demonstrate our capabilities to the world.

Team Strategy:

Our strategy is deeply rooted in the belief that collaboration is the cornerstone of success. We understand that true innovation and progress come from working together, beyond the confines of individual roles. By fostering a collaborative environment, we prioritize teamwork and collective problem-solving as the key to overcoming challenges. Each member of our team brings unique strengths to the table, and we actively seek to harness these diverse perspectives. Whether it's in mechanical design, coding, or developing competition strategies, we approach every challenge as a

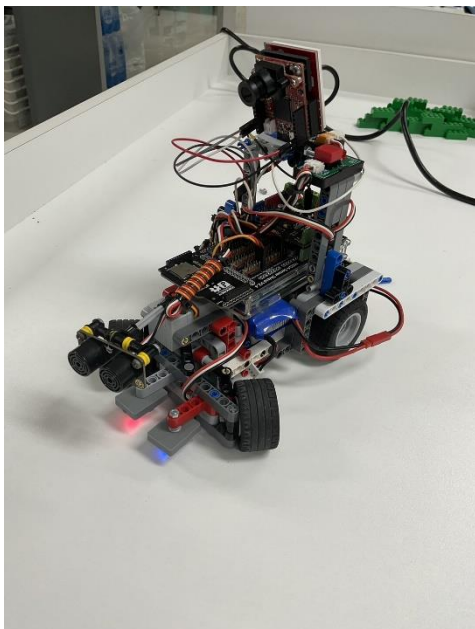
unified team, knowing that our combined efforts will lead to more creative and effective solutions.

When obstacles arise, we don't view them as roadblocks but as opportunities for growth. We gather as a team to brainstorm, explore multiple solutions, and ensure that every voice is heard. This inclusive approach not only enhances our problem-solving capabilities but also drives our innovation forward. By leveraging the different skill sets within our team, we continuously push the boundaries of what we can achieve.

In addition to our technical work, we place a strong emphasis on open communication and continuous support. We believe that a team is only as strong as its ability to communicate effectively. Outside of our formal sessions, we maintain an open line of dialogue, sharing resources, offering constructive feedback, and supporting one another in every aspect of our work. This constant exchange of ideas and encouragement helps to solidify our bond as a team and ensures that we are all aligned in our efforts to achieve our common goals.

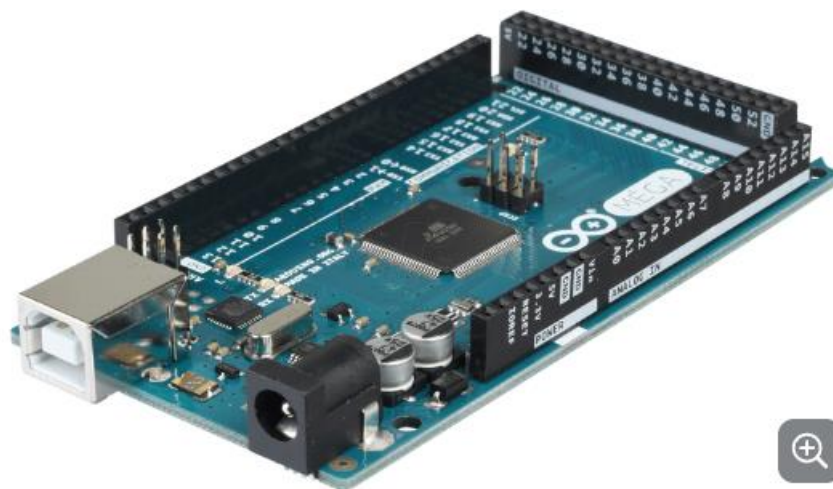
Our ultimate objective is to qualify for the international competition in Turkey, a prestigious milestone that we narrowly missed last year. However, rather than being discouraged by this setback, we have used it as motivation to refine our approach. We have renewed our focus on teamwork, innovation, and continuous improvement, knowing that these principles will be the foundation of our success. Over the past year, we have dedicated ourselves to honing our skills, improving our designs, and learning from our experiences. This commitment to growth has strengthened our resolve, and we are now more determined than ever to secure our place on the global stage.

We recognize that our journey is not just about competing; it is about demonstrating the strength of our collective effort. We are determined to show that by working together, we can achieve remarkable things. Our aim is not only to qualify for the international competition but to make a lasting impact through our innovation, teamwork, and dedication. As we move forward, we remain focused on our goal and confident that our collaborative spirit will guide us to success.



Portion 2: Our Robot Hardware

1. ARDUINO MEGA: Arduino Mega 2560



The Arduino Mega 2560 is a powerful and versatile microcontroller board that serves as a cornerstone for a wide range of electronic projects. Built around the Atmega2560 microcontroller chip, this board operates at a clock speed of 16 MHz, providing a reliable platform for both beginners and experienced developers alike. One of its standout features is its extensive memory capacity, which includes 256 KB of Flash memory, of which 8 KB is reserved for the bootloader. Additionally, it is equipped with 8 KB of SRAM and 4 KB of EEPROM, allowing for efficient storage and management of program data and

variables, making it well-suited for complex applications that require substantial memory resources.

The Arduino Mega 2560 offers an Impressive array of input and output options. It is equipped with 54 digital I/O pins, 15 of which can be used as Pulse Width Modulation (PWM) outputs. These PWM outputs enable precise control over motors, LEDs, and other components that require variable power levels. Furthermore, the board includes 16 analog input pins, allowing for the integration of a wide range of sensors that provide analog signals. This abundance of I/O pins makes the Mega 2560 ideal for projects that demand extensive interfacing with external devices, such as robotics, home automation systems, and advanced sensor networks.

Communication is another strong suit of the Arduino Mega 2560. It supports multiple communication interfaces, including UART (Universal Asynchronous Receiver-Transmitter), SPI (Serial Peripheral Interface), and I2C (Inter-Integrated Circuit). These communication protocols allow the board to interface with other microcontrollers, sensors, and peripherals, enabling the creation of intricate, multi-device systems. Additionally, the Mega 2560 features a USB interface, which is used for programming the board and for serial communication with a connected computer. This versatility in communication options makes the Mega 2560 a preferred choice for projects that require data exchange between different components or systems.

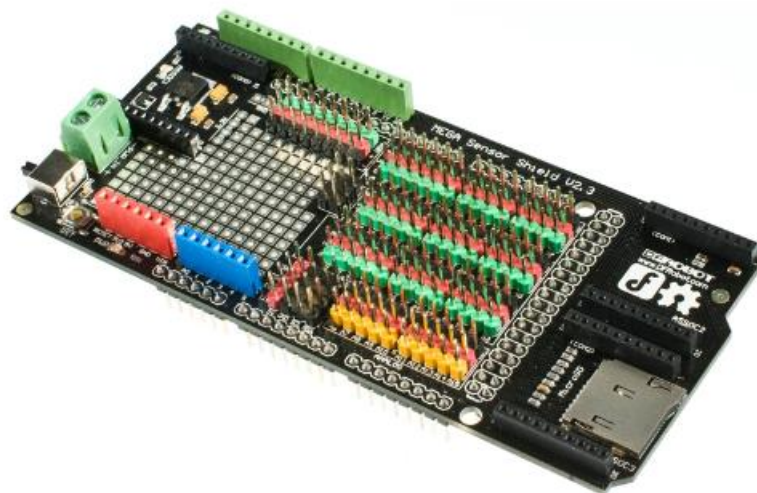
Operating at 5V, the Arduino Mega 2560 is fully compatible with a wide range of Arduino shields, expanding its functionality even further. Whether it's motor control, wireless communication, or advanced displays, the Mega 2560 can be easily extended with additional hardware to suit the needs of any project. This flexibility, combined with the extensive community support and resources available for Arduino products, ensures that users can find solutions to almost any challenge they encounter.

The Arduino Mega 2560 is widely used in fields such as robotics, where its multiple I/O pins and powerful processing capabilities enable the control of complex systems with numerous sensors and actuators. It is also popular in home automation, where it can manage a variety of tasks such as lighting, security, and environmental monitoring. Additionally, the board's ability to handle complex sensor applications makes it ideal for scientific experiments, data logging, and industrial automation.

Supported by a vibrant community and extensive library resources, the Arduino Mega 2560 is more than just a microcontroller; it's a platform for innovation. Whether you're a hobbyist looking to build your next project or a professional engineer working on a sophisticated system, the Arduino Mega 2560 provides the tools and flexibility needed to bring your ideas to life.

2. Board Extension

Gravity: IO Sensor Shield For Arduino Mega



This shield includes 3 Xbee slots, 1 microSD slot, and Arduino shield headers to plugin most Arduino Shields. It also includes a prototyping area and breakouts for Digital pins 14 to 53, Analog pins 6 to 15, and PWM pins 2 to 9.

Gravity: 2x2A Motor Shield for Arduino Twin



The motor shield allows Arduino to drive two channel DC motors, which use a L298N chip that delivers output current up to 2A on each channel. Motor motor-driven voltage is 4.8V to 35V, which is driven in a Dual full-bridge driver. The control function can be separated into two types, namely En and Mn, which are provided in the tables below.

- **Control Function Table:**

Name	Function
En	Mn Speed control(PWM)
Mn	Mn Direction Control


- **Control Signal Truth Table:**

En	Mn	State
L	X	Disable Mn
H	L	Mn Foreward(Mn+ is positive)
H	H	Mn Backward(Mn+ is negative)

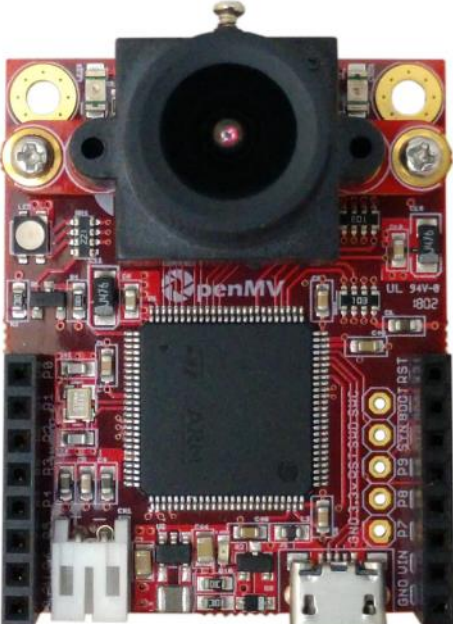
Operating Speed (no load)	0.14sec / 60° (4.8V)
Working Voltage	3.3 - 6 V
Nominal Voltage:	4.8 V
Operating Current (no load):	70 mA
Locked Rotor Current	800 mA
Max Torque:	1.8 kg-cm

3. Open Mv H7

OpenMV Cam H7 - OV7725



OpenMV Cam
By: Ibrahim Abdelkader & Kwabena W. Agyeman
<https://openmv.io>



All pins are 5V tolerant¹ with a 3.3V output
All pins can sink or source up to 25 mA²

¹ P6 is not 5V tolerant in ADC or DAC mode
² Up to 120mA in total between all pins

Max current used w/o µSD card < 150 mA
Max current used w/ µSD card < 250 mA

Micro SD Slot:
SD < 2GB Max
SDHC < 32GB Max
SDXC < 2TB Max

LEDs

- LED1 – Red
- LED2 – Green
- LED3 – Blue
- LED4 – IR

Peripherals / Timers	CPU Name	Pin Name
UART 1 RX	SP12 M25	PB15 P0
UART 1 TX	SP12 M25	PB14 P1
CAN2 TX	SP12 M25	PB13 P2
CAN2 RX	SP12 M25	PB12 P3
I2C2 SCL	UART3 TX	PB10 P4
I2C2 SDA	UART3 RX	PB11 P5
DAC	ADC	PA5 P6

3.3V Rail (250 mA supply Max)

Pin Name	CPU Name	Peripherals / Timers
Reset (Connect to GND to reset)		
BOOT 0 (Connect to 3.3V for DFU mode)		
Frame Sync (use to frame sync cams)		
P9	PD14	Servo 3
P8	PD13	Servo 2
P7	PD12	Servo 1
VIN (3.6V - 5V)		
GND Rail		

The OpenMV H7 Camera is a small, low-power, microcontroller board that allows you to easily implement applications using machine vision in the real-world. You program the OpenMV Cam in high-level Python scripts (courtesy of the MicroPython Operating System) instead of C/C++. This makes it easier to deal with the complex outputs of machine vision algorithms and working with high level data structures. But, you still have total control over your OpenMV Cam and its I/O pins in Python. You can easily trigger taking pictures and videos of external events or execute machine vision algorithms to figure out how to control your I/O pins.

The OpenMV H7 comes with an OV7725 image sensor that is capable of taking 640x480 8-bit Grayscale images or 640x480 16-bit RGB565 images at 60 FPS when the resolution is above 320x240 and 120 FPS when it is below. Most simple algorithms will run at above 60 FPS. Additionally, the OpenMV H7 now features removable camera modules which allow you to use the module with Global Shutter and FLIR Lepton sensors for serious computer vision applications. The OpenMV is perfect for applications involving frame differencing, face detection, eye and marker tracking, QR code detection/decoding, shape detection, and more.

-Quick Spec

- Full speed USB (12Mbps) interface to your computer
- μ SD Card socket capable of 100Mbps reads/writes
- SPI bus that can run up to 100Mbps
- I2C Bus, CAN Bus, and an Asynchronous Serial Bus (TX/RX) for interfacing with other microcontrollers and sensor
- 12-bit ADC and a 12-bit DAC
- Three I/O pins for servo control
- Interrupts and PWM on all I/O pins
- 640x480 8-bit Grayscale images or 640x480 16-bit RGB565 images at 60 FPS when the resolution is above 320x240 and 120 FPS when it is below - OV7725 Camera

4.Gravity: URM09 Ultrasonic Sensor(Trig)



In this setup, we're using a super bright red LED as the main light source. It's always on when the system is powered, providing a steady and strong beam of light that interacts with the environment and objects around it. The brightness of the LED is crucial because it ensures that the light is strong enough to reflect off surfaces and be detected by the system.

The system's receiver is a phototransistor, specifically the SFH310 model, which is designed to pick up the red light that bounces back from nearby objects. How well the phototransistor can detect this light depends a lot on the surface it's reflecting off. For example, smooth, white surfaces are great at reflecting red light, allowing the phototransistor to capture more light and produce a stronger signal. This results in a higher output voltage, signaling that the object has been detected.

Using red light as the main detection tool has its benefits, especially in situations where it's important to tell the difference between different surface colors. This setup is particularly useful for detecting color differences on surfaces that might be printed with special inks that resist infrared or ultraviolet light. By focusing on red light, the system can accurately distinguish between various surface colors and characteristics, providing reliable detection.

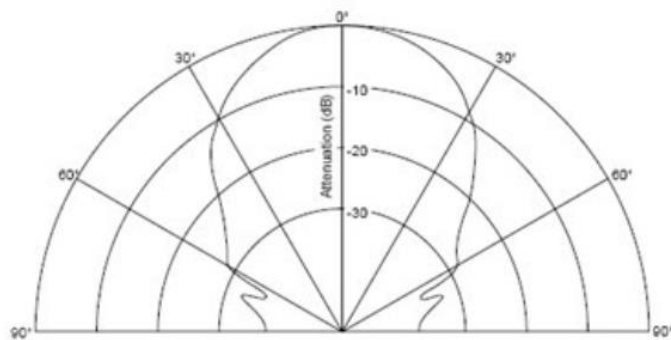
In a nutshell, combining a super bright red LED with the SFH310 phototransistor creates a solid detection system. It's capable of analyzing surfaces based on how well they reflect light, making it especially handy in applications where precise surface detection and color differentiation are needed, even in tough environments where infrared or ultraviolet resistance is important.

-Specification

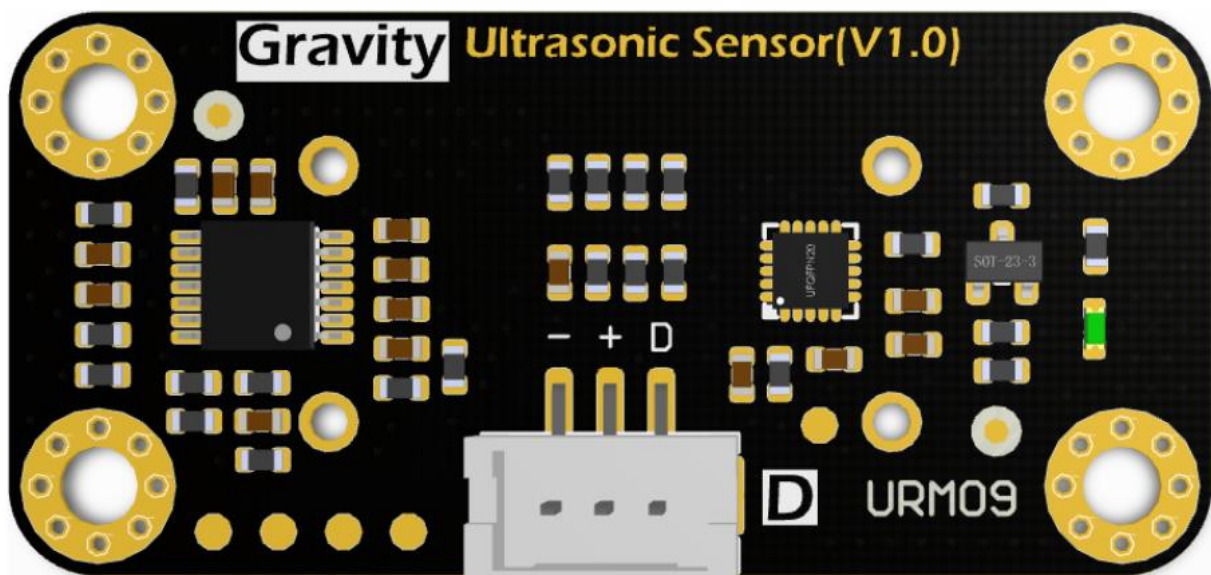
- Operating Voltage: 3.3~5.5V DC

- Max Instantaneous Working Current: <20mA
- Operating Temperature Range: -10°C ~ +70°C
- Effective Ranging Range: 2cm ~ 500cm
- Resolution: 1cm
- Accuracy: 1%
- Acoustic Frequency: 40±2KHz
- Ranging Frequency: 25Hz Max
- Dimension: 47×22mm/1.85×0.87"

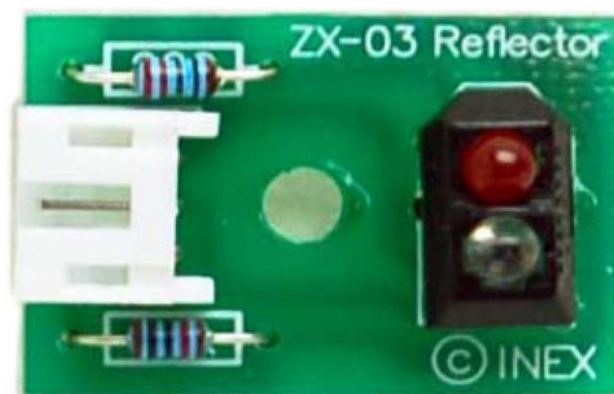
-Measuring Angle



-Board Overview



5. Inex light sensor ZX-03 Reflector



In this system, a high-brightness red LED, or super bright LED, is employed as the primary light source. This LED remains continuously illuminated when the system is powered, providing a consistent and strong light

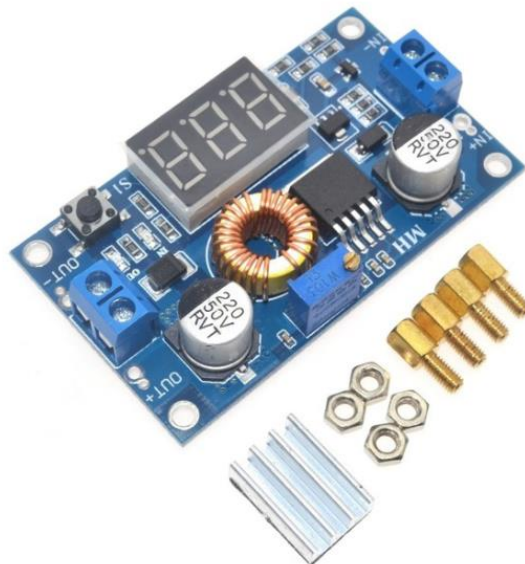
output. The role of the LED is critical in ensuring that the light emitted is sufficiently intense to interact with the surrounding environment and objects.

The receiver component of this system is a phototransistor, specifically the SFH310 model. This phototransistor is designed to detect the red light that is reflected off objects or surfaces within its range. The effectiveness of this reflection is highly dependent on the surface characteristics and color of the object in question. For instance, smooth and white surfaces tend to reflect red light more efficiently, enabling the phototransistor to capture a significant amount of reflected light. This results in a higher output voltage from the phototransistor, indicating a strong detection signal.

The use of red light as the primary detection medium is particularly advantageous in applications where surface color differentiation is essential. For example, this type of detector is well-suited for measuring color differences on surfaces that have been treated with infrared or ultraviolet radiation-resistant inks. By focusing on red light, the system can effectively distinguish between different surface colors and characteristics, providing accurate and reliable detection results.

In summary, the combination of a high-brightness red LED and the SFH310 phototransistor creates a robust detection system capable of analyzing surface characteristics based on light reflection. This makes it a valuable tool in applications that require precise surface detection and color differentiation, particularly in environments where resistance to infrared or ultraviolet radiation is a concern.

6.Step Down Module 12 V to 5 V size 5 A

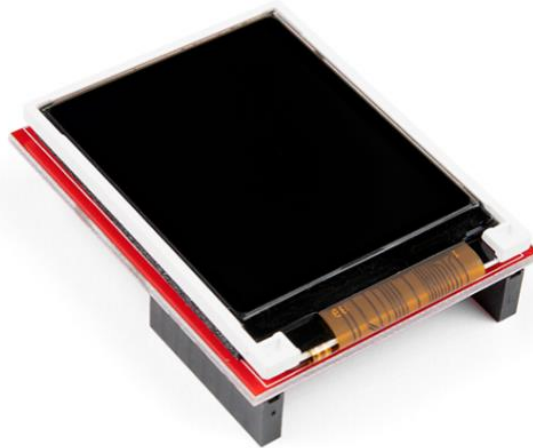


Product details 5A DC-DC Step Down Module is used to reduce the voltage to be used as a power supply for various sensors and microcontroller boards. It has a width x length of 3.2 x 6.5 cm.

Features:

- Input voltage range: 4.0-38V DC
- Output voltage range: 1.25-36V DC adjustable
- Output current: 0-5A
- Output power: 75W(max)
- Voltmeter range: 4 to 40V, error 0.1V
- High conversion efficiency up to 96%
- Built-in thermal shutdown function
- Built-in current limit function
- Built-in output short protection function
- Size: 66X39X18 mm

7. Open mv LCD shield



The LCD Shield gives your OpenMV Camera the ability to display what it sees on-the-go while not connected to your computer. This shield features a 1.8" 128x160 16-bpp (RGB565) TFT LCD display with a controllable backlight. Your OpenMV Cam's firmware already has built-in support for controlling the LCD Shield using the LCD module.

The LCD Shield is great for robotic applications where you need to debug your OpenMV Cam in the field - like when debugging a line-following robot. For example, you can use the LCD Shield to display the frame buffer after you've thresholded and drawn markups on an image.

-Quick Spec

- Screen Type - 1.8" TFT LCD
- Horizontal Resolution - 128 pixels (28.03mm), 0.18mm pixel pitch
- Vertical Resolution - 160 pixels (35.04mm), 0.18mm pixel pitch
- Display Colors - 64K 16-bit RGB565
- Power Consumption
 - Idle Backlight Off - < 1mA @ 3.3V (> 100mA w/ Cam)
 - Idle Backlight On - 30mA @ 3.3V (130mA w/ Cam)
 - Active Backlight Off - 10mA @ 3.3V (150mA w/ Cam)
 - Active Backlight On - 40mA @ 3.3V (180mA w/ Cam)

- Dimensions
 - Weight - 13g
 - Length - 48mm
 - Width - 38mm
 - Height - 13mm
- Temperature Range
 - Storage - -30°C to 80°C
 - Operating - -20°C to 70°C

8. LEGO Power Functions L-Motor



This is the Large Motor. It delivers a maximum torque of 45,4 mNm (450 mA). Without load, its rotation speed is around 380 rotations per minute.

The current consumption will depend heavily on the load it is driving. Under normal conditions, it can be around 225 mA and we do not recommend a continuous use above 450 mA.

Motorize your large LEGO builds with LEGO Power Functions elements! Bring your creations to life by powering them with the large L-Motor. Spin wheels, turn gears, and get your models moving! Designed to fit and lock onto LEGO Technic pins and axles.

9. Touch sensor Inex



When the switch is pressed, the DATA pin will have a logic “1” from R2 which is connected to the pull-up. When the switch is pressed, the DATA pin will be “0” because the switch is shorted to ground. Current flows through the LED and R1 causing the LED to light up. The DATA pin can also be used as an input, allowing the LED to be turned on or off as desired.

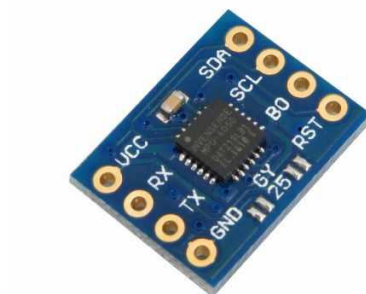
10.Li-Po HeliCoxNano 7.4V 1100 mAh 30C



Quick Spec

- 2 cells 7.4V
- Capacity 1100 mAh 30C
- Can be charged at a current of 5 times the capacity (5C)
- The connector is a JST type, easy to plug and play
- Size 32x70X12 mm.
- Weight 60 grams
- Can be used with POPBOT-XT, Robo-Creator, AT-BOT, IPST-SE robots
- Comes with a JST cable at the end that can be screwed onto various robot boards immediately

11.GY-25 sensor ZX-IMU



THOMAS

Quick Spec

- Use chip: MCU + MPU6050
- Power supply: 3-5v (internal low dropout regulator)
- Communication: serial communication (baud rate 9600,115200), IIC communication (read only raw data)
- Module size: 15.5mm * 11.5mm
- 2.54mm pitch
- Direct Data: YAW ROLL PITCH
- Heading angle (YAW) $\pm 180^\circ$
- Roll angle (ROLL) $\pm 180^\circ$
- Pitch angle (PITCH) $\pm 180^\circ$
- Angular resolution 0.01 $^\circ$

Our Car Design

Our vehicle design draws significant inspiration from real-world automotive engineering principles. One of the key aspects of this design is the placement of the steering wheel at the front of the vehicle. This decision is rooted in several critical factors that enhance the car's performance.

Firstly, positioning the steering wheel at the front of the vehicle allows for more efficient and precise maneuverability. The front wheels, which are directly controlled by the steering mechanism, can effectively guide the vehicle's direction with greater accuracy compared to a rear-steering configuration. This results in a smoother and more responsive when it driving, particularly when navigating through curves or making turns to avoid hitting the obstacle.

Next, front steering facilitates a more balanced distribution of forces, which is essential for maintaining equilibrium and stability during driving. The front-wheel steering design allows for better control of the vehicle's trajectory, especially at higher speeds, where maintaining stability is crucial for safety.

Next having the steering wheel at the front enables the vehicle to execute sharp turns more effectively, even at high speeds. This capability is vital for situations where the driver needs to swiftly avoid obstacles while maintaining control of the vehicle. The enhanced agility provided by front steering ensures that the vehicle can respond quickly and safely to sudden changes in the driving environment.

we have designed our ultrasonic sensor system with advanced mobility and strategic placement to enhance the vehicle's obstacle-detection capabilities. Specifically, the sensor is mounted above the front wheel, a position carefully chosen for its comprehensive coverage and optimal functionality.

By enabling the ultrasonic sensor to move freely, we ensure that it can scan the surrounding environment from multiple angles. This dynamic range of motion allows the sensor to detect obstacles across a wide field of view, including areas that would otherwise be difficult to monitor with a fixed sensor. The mobility of the sensor is particularly advantageous in complex driving scenarios, where obstacles may appear suddenly or from unexpected directions.

The decision to position the sensor above the front wheel further maximizes its effectiveness. This location is ideal for early detection of obstacles, as it provides a clear and unobstructed view of the road ahead and the vehicle's immediate surroundings. By detecting potential hazards from all angles, the sensor enables the vehicle to respond promptly and accurately, thereby reducing the risk of collisions.

Portion 3: Obstacle Management

3.1 Open Challenge

3.2 Obstacle Challenge

3.3 Our Code

Configuration Code:

1. Libraries and Servo Initialization

```
1  #include <Servo.h>
2  #include "Mapf.h"
3  #include <PID_v2.h>
4  Servo myservo;
5  Servo myservo2;
```

- The code includes the Servo.h library, which is used to control servos.
- The Mapf.h and PID_v2.h libraries are included, where PID_v2.h is used for the PID controller, which helps maintain stability and precision in the robot's movements.
- Two servo objects, myservo and myservo2, are created to control the servos attached to the ultrasonic sensor and the steering mechanism, respectively.

2. Motor Control

```
1 //Motor
2 const int E1Pin = 10;
3 const int M1Pin = 12;
4 typedef struct {
5     byte enPin;
6     byte directionPin;
7 } MotorContrl;
8 const int M1 = 0;
9 const int MotorNum = 1;
10
11 const MotorContrl MotorPin[] = { { E1Pin, M1Pin } };
12
13 const int Forward = LOW;
14 const int Backward = HIGH;
15
16 void initMotor() {
17     int i;
18     for (i = 0; i < MotorNum; i++) {
19         digitalWrite(MotorPin[i].enPin, LOW);
20
21         pinMode(MotorPin[i].enPin, OUTPUT);
22         pinMode(MotorPin[i].directionPin, OUTPUT);
23     }
24 }
```

- The motor is controlled through E1Pin (to enable the motor) and M1Pin (to set the motor direction).
- A MotorContrl struct is used to store the motor control pins, and an array MotorPin[] is defined to accommodate multiple motors, although currently, only one motor is being used (M1).
- The direction of the motor is set using constants Forward and Backward.

3. Sensor and Servo Setup

```
1 //Button
2 int BUTTON = A6;
3
4 // Light Sensors
5 int const RED_SEN = A7;
6 int const BLUE_SEN = A8;
7
8 // Servo
9 int const STEER_SRV = 16;
10 int const ULTRA_SRV = 23;
11
12 //Ultra
13 int const ULTRA_PIN = A9;
14
15 void setup() {
16     // put your setup code here, to run once:
17     pinMode(STEER_SRV, OUTPUT);
18     pinMode(ULTRA_SRV, OUTPUT);
19     pinMode(ULTRA_PIN, INPUT);
20     pinMode(RED_SEN, INPUT);
21     pinMode(BLUE_SEN, INPUT);
22     pinMode(BUTTON, INPUT);
23 }
```

Button

- **int BUTTON = A6:** This line defines a variable named BUTTON and assigns it to pin A6 on the Arduino. This pin will be used to connect a button.

Light Sensors

- **int const RED_SEN = A7:** This line sets up a variable named RED_SEN for the red light sensor, connected to pin A7.
- **int const BLUE_SEN = A8:** Similarly, this line sets up a variable named BLUE_SEN for the blue light sensor, connected to pin A8.

Servos

- **int const STEER_SRV = 16:** This line defines a variable named STEER_SRV for the servo that controls steering, connected to pin 16.
- **int const ULTRA_SRV = 23:** This line sets up a variable named ULTRA_SRV for the servo that adjusts the ultrasonic sensor, connected to pin 23.

Ultrasonic Sensor

- **int const ULTRA_PIN = A9;** This line creates a variable named ULTRA_PIN for the ultrasonic sensor, which is connected to pin A9.

4. PID Controller

```
1 //PID
2 PID_v2 compassPID(0.52, 0.0001, 0.047, PID::Direct);
3
4 void setup() {
5     // put your setup code here, to run once:
6     compassPID.Start(0, 0, 0);
7     compassPID.SetOutputLimits(-180, 180);
8     compassPID.SetSampleTime(10);
9 }
```

- **PID_v2 compassPID(0.52, 0.0001, 0.047, PID::Direct);:**
 - This line creates a PID controller object named compassPID.
 - The PID_v2 is a class from the PID_v2.h library used to manage the PID control algorithm.
 - The parameters in the parentheses are:
 - **0.52:** Proportional gain (Kp). It determines how much the controller reacts to the current error.
 - **0.0001:** Integral gain (Ki). It controls how much the controller reacts to the accumulation of past errors.
 - **0.047:** Derivative gain (Kd). It helps the controller react to the rate of change of the error.
 - **PID::Direct:** Indicates that the PID output will be directly used to control the system, rather than reversing the output (which is PID::Reverse).

Setup Function

- **void setup() { ... }:** This function runs once when the Arduino starts. It initializes the PID controller.
 - **compassPID.Start(0, 0, 0);:** Starts the PID controller with initial values of 0 for the proportional, integral, and derivative terms.

- **compassPID.SetOutputLimits(-180, 180);** Sets the range of the PID controller's output. This means the controller's output will be limited between -180 and 180.
- **compassPID.SetSampleTime(10);** Sets how often the PID controller should update its calculations. In this case, it's every 10 milliseconds.

Auxiliary Functions:

1. Steering_Servo

```
124 void steering_servo(int degree) {
125     myservo2.write((90 + max(min(degree, 45), -45)) / 2);
126 }
```

The `steering_servo` function controls the position of a steering servo motor based on the input angle degree. It ensures the servo position stays within ± 45 degrees of the center (90 degrees). The function adjusts the input angle to be within this range, adds 90 to center it, and then scales the result to fit the servo's range, setting the servo to a position between 0 and 90 degrees.

2. Ultra_servo

```
128 void ultra_servo(int degree, char mode_steer) {
129     int middle_degree = 0;
130     if (mode_steer == 'F') {
131         middle_degree = 135;
132     } else if (mode_steer == 'L') {
133         middle_degree = 135 - 65;
134     } else if (mode_steer == 'R' || mode_steer == 'U') {
135         middle_degree = 135 + 90;
136     } else {
137     }
138     myservo.write(mapf(max(min(middle_degree + degree, 225), 45), 0, 270, 0, 180));
139 }
140
```

The `ultra_servo` function controls the position of an ultrasonic sensor mounted on a servo motor. The function takes two parameters: `degree` (an angle adjustment) and `mode_steer` (a character indicating the steering mode).

1. Middle Degree Calculation:

- The variable `middle_degree` is initially set to 0.
- If `mode_steer` is 'F' (forward), `middle_degree` is set to 135 degrees.
- If `mode_steer` is 'L' (left), it adjusts `middle_degree` by subtracting 65, resulting in 70 degrees.

- If mode_steer is 'R' (right) or 'U' (U-turn), it adjusts middle_degree by adding 90, resulting in 225 degrees.

2. Servo Position Calculation:

- The function then calculates the final position of the servo by adding the degree input to middle_degree. It ensures this value stays between 45 and 225 degrees using max() and min() functions.
- The mapf() function is used to convert this angle from a 0-270 degree range to a 0-180 degree range, suitable for controlling the servo motor.

3. Servo Movement:

- Finally, the servo motor is commanded to move to the calculated position using myservo.write()

3. GetDistance

```
142 float getDistance() {  
143     float raw_distance = mapf(analogRead(ULTRA_PIN), 0, 1023, 0, 500);  
144     if (TURN == 'L') {  
145         raw_distance += 3;  
146     } else if (TURN == 'R') {  
147         raw_distance -= 0;  
148     }  
149     return min(raw_distance, 50);  
150 }
```

To estimate the distance to an item, the getDistance() function gets data from an ultrasonic sensor. It guarantees that the final distance stays within a predetermined maximum value of 50 units and modifies the distance according to the robot's turning direction (adding a slight offset when turning left). This aids in the robot's precise obstacle detection during turns, which can plus distance in the direction that we want.

4. Gyro Calculator

```
41 void zeroYaw() {
42     Serial1.begin(115200);
43     delay(100);
44     // Sets data rate to 115200 bps
45     Serial1.write(0xA5);
46     delay(10);
47     Serial1.write(0x54);
48     delay(100);
49     // pitch correction roll angle
50     Serial1.write(0xA5);
51     delay(10);
52     Serial1.write(0x55);
53     delay(100);
54     // zero degree heading
55     Serial1.write(0xA5);
56     delay(10);
57     Serial1.write(0x52);
58     delay(100);
59     // automatic mode
60 }
61
62 bool getIMU() {
63     while (Serial1.available()) {
64         rxBuf[rxCnt] = Serial1.read();
65         if (rxCnt == 0 && rxBuf[0] != 0xAA) return;
66         rxCnt++;
67         if (rxCnt == 8) { // package is complete
68             rxCnt = 0;
69             if (rxBuf[0] == 0xAA && rxBuf[7] == 0x55) { // data package is correct
70                 pvYaw = (int16_t)(rxBuf[1] << 8 | rxBuf[2]) / 100.f;
71                 pvPitch = (int16_t)(rxBuf[3] << 8 | rxBuf[4]) / 100.f;
72                 pvRoll = (int16_t)(rxBuf[5] << 8 | rxBuf[6]) / 100.f;
73                 pvYaw = wrapValue(pvYaw + plus_degree, -179, 180);
74                 return true;
75             }
76         }
77     }
78     return false;
79 }
```

Two functions, zeroYaw() and getIMU(), are included in the code to work with a gyroscope sensor.

1. The zeroYaw() function is in charge of effectively calibrating the gyroscope by setting its yaw angle to zero. At a baud rate of 115200, a serial communication is first established. Afterwards, it instructs the gyroscope to receive a certain byte sequence (0xA5, 0x54, 0x55, 0x52), with delays in between, in order to reset the yaw angle to zero, adjust the pitch and roll angles, and switch the sensor to automatic mode.

2. getIMU() Function: Using the serial interface, this function receives data (x,y,z) from the gyroscope. Bytes are gathered and stored in a buffer (rxBuf). It checks to see if the data is accurate after receiving a full data packet (8 bytes). If true,

5. Color_detection

```

81 void Color_detection() {
82     int blue_value = analogRead(BLUE_SEN);
83     if (TURN == 'U') {
84         int red_value = analogRead(RED_SEN);
85         if (blue_value < 605 || red_value < 480) {
86             int lowest_red_sen = red_value;
87             long timer_line = millis();
88             while (millis() - timer_line < 100) {
89                 int red_value = analogRead(RED_SEN);
90                 if (red_value < lowest_red_sen) {
91                     lowest_red_sen = red_value;
92                 }
93             }
94             if (lowest_red_sen > 480) {
95                 // Red
96                 TURN = 'L';
97                 plus_degree += 90;
98                 // ultra_servo(0, 'L');
99             } else {
100                 // Blue
101                 TURN = 'R';
102                 plus_degree -= 90;
103                 // ultra_servo(0, 'R');
104             }
105             halt_detect_line_timer = millis();
106             Line_Number++;
107         }
108     } else {
109         if (millis() - halt_detect_line_timer > 1300) {
110             Serial.println(pvYaw);
111             if (blue_value < 605) {
112                 if (TURN == 'R') {
113                     plus_degree -= 90;
114                 } else {
115                     plus_degree += 90;
116                 }
117             }
118             halt_detect_line_timer = millis();
119             Line_Number++;
120         }
121     }
122 }

```

The Color_detection() function is used to detect color from sensors and decide the robot's turning direction based on the detected color. It reads values from a blue sensor (BLUE_SEN) and a red sensor (RED_SEN). Depending on the sensor readings and the current direction of the robot (TURN), it determines whether the robot should turn left, right, or continue forward.

Main duties:

1. **Read Blue Sensor:** The function starts by reading the value from the blue sensor (blue_value).
2. **Check Current Turn Direction:**
 - If the robot's current direction (TURN) is set to 'U', it means the robot is currently going straight. The function reads the value from the red sensor (red_value) to determine if it needs to change direction based on the color detected.
3. **Detect Colors:**

- If the blue sensor reads a value less than 605 or the red sensor reads a value less than 480, it means the robot has detected a color.
- The function then records the lowest red sensor value over a short period of time (100 milliseconds) to get a more accurate reading.

4. Decide Turn Direction:

- If the lowest red sensor value is greater than 480, it is interpreted as detecting red, so the robot will turn left (TURN = 'L') and adjust its yaw angle by adding 90 degrees (plus_degree += 90).
- If the lowest red sensor value is less than 480, it is interpreted as detecting blue, so the robot will turn right (TURN = 'R') and adjust its yaw angle by subtracting 90 degrees (plus_degree -= 90).

5. Update Timers and Line Count:

- After making the turn decision, the function updates the halt_detect_line_timer and increases the Line_Number by one to track how many lines the robot has detected and passed.

6. Check for Further Turns:

- If the robot is not currently in the 'U' (going straight) direction, it checks if enough time has passed since the last turn decision (1300 milliseconds). If the blue sensor detects a line (value less than 605), it adjusts the yaw angle again based on the current turning direction and updates the timer and line counter.

Open Challenge Code:

Setup Code:

```
50 void setup() {  
51     // put your setup code here, to run once:  
52     Serial.begin(115200);  
53  
54     compassPID.Start(0, 0, 0);  
55     compassPID.SetOutputLimits(-180, 180);  
56     compassPID.SetSampleTime(10);  
57  
58     pinMode(STEER_SRV, OUTPUT);  
59     pinMode(ULTRA_SRV, OUTPUT);  
60     pinMode(ULTRA_PIN, INPUT);  
61     pinMode(RED_SEN, INPUT);  
62     pinMode(BLUE_SEN, INPUT);  
63     pinMode(BUTTON, INPUT);  
64  
65     initMotor();  
66     while (!Serial)  
67     ;  
68     myservo.attach(ULTRA_SRV, 500, 2400);  
69     myservo2.attach(STEER_SRV, 500, 2500);  
70     steering_servo(0);  
71     ultra_servo(0, 'L');  
72  
73  
74  
75     while (analogRead(BUTTON) > 500)  
76     ;  
77     zeroYaw();  
78 }
```

This **setup()** function initializes the hardware components and prepares the robot for operation by utilizing the functions explained above. It configures the necessary inputs and outputs, sets up the PID controller, and ensures that the servos, motors, and sensors are all in their correct starting positions.

Here's a breakdown of what each section does:

Serial Communication Setup:

- `Serial.begin(115200);` initializes serial communication at a baud rate of 115200. This allows the microcontroller to send and receive data through the serial port for debugging and monitoring.

Compass PID Setup:

- `compassPID.Start(0, 0, 0);` initializes the PID controller with initial values for proportional, integral, and derivative terms.
- `compassPID.SetOutputLimits(-180, 180);` sets the output range of the PID controller, limiting it to between -180 and 180 degrees.
- `compassPID.SetSampleTime(10);` sets the PID controller to update every 10 milliseconds.

Pin Mode Configuration:

- The `pinMode()` commands configure the pins for various sensors and actuators:
 - `STEER_SRV` and `ULTRA_SRV` are set as output pins for controlling the servos.
 - `ULTRA_PIN`, `RED_SEN`, and `BLUE_SEN` are set as input pins for reading sensor values.
 - `BUTTON` is also set as an input pin for reading button presses.

Motor Initialization:

- `initMotor();` is called to initialize the motors. This function is assumed to configure the necessary pins and settings for motor control.

Servo Setup:

- The code waits for the serial connection to establish (`while (!Serial);`).
- Then, the servos are attached to their respective pins (`myservo.attach()` and `myservo2.attach()`), with specific minimum and maximum pulse width values for precise control.
- `steering_servo(0);` centers the steering servo at 0 degrees.
- `ultra_servo(0, 'L');` sets the ultrasonic sensor's servo to the initial position and mode ('L' for left).

Wait for Button Press:

- The code waits in a loop until the button is pressed and the sensor value drops below 500 (`while (analogRead(BUTTON) > 500);`).

Yaw Initialization:

- Finally, `zeroYaw();` is called to set the robot's yaw (rotation angle) to zero, preparing the robot's orientation for the task.

Main Loop Code:

```
81 void loop() {  
82  
83   while (analogRead(BUTTON) > 500) {  
84  
85     motor(100);  
86     getIMU();  
87     Color_detection();  
88     float desiredDistance = 26.5;  
89     float distanceError = getDistance() - desiredDistance;  
90     float deadband = 2.0;  
91     if (abs(distanceError) < deadband) {  
92       distanceError = 0.0;  
93     }  
94     float directionFactor = (TURN == 'L') ? -1.0 : 1.0;  
95     float adjustedYaw = pvYaw + (distanceError * directionFactor);  
96     float pidOutput = compassPID.Run(adjustedYaw);  
97     steering_servo(pidOutput);  
98     ultra_servo(pvYaw, TURN);  
99  
100    if (Line_Number >= 12) {  
101      long timer01 = millis();  
102      while (millis() - timer01 < 750) {  
103        motor(100);  
104        getIMU();  
105        Color_detection();  
106        float desiredDistance = 26.5;  
107        float distanceError = getDistance() - desiredDistance;  
108        float deadband = 2.0;  
109        if (abs(distanceError) < deadband) {  
110          distanceError = 0.0;  
111        }  
112        float directionFactor = (TURN == 'L') ? -1.0 : 1.0;  
113        float adjustedYaw = pvYaw + (distanceError * directionFactor);  
114        float pidOutput = compassPID.Run(adjustedYaw);  
115        steering_servo(pidOutput);  
116        ultra_servo(pvYaw, TURN);  
117      }  
118      motor(0);  
119      while (true) {  
120      }  
121    }  
122  }  
123 }  
124 }
```

This section of the code is designed to help the robot navigate by calculating its distance from the wall and maintaining a specific distance while moving in a square pattern.

Here's a breakdown of the program:

1. Main Loop Execution:

- The function continuously runs as long as the value read from the BUTTON is greater than 500. This typically means the button has not been pressed, so the robot continues its operation.

2. Motor Control:

- `motor(100);` starts the motors at full speed (assuming 100 is the maximum speed setting). This keeps the robot moving.

3. Sensor and Control Updates:

- `getIMU();` retrieves updated orientation data from the IMU (Inertial Measurement Unit) sensor. This data is used to determine the robot's current heading and position.
- `Color_detection();` checks the color sensors to decide if the robot needs to change its direction based on the color detected.

4. Distance Calculation and PID Control:

- `float desiredDistance = 26.5;` sets the target distance from an obstacle or a reference point.
- `float distanceError = getDistance() - desiredDistance;` calculates the difference between the actual distance (obtained from `getDistance()`) and the desired distance.
- `float deadband = 2.0;` defines a range within which the distance error is considered acceptable and thus set to zero to avoid unnecessary adjustments.
- If the distance error is within the deadband, it is set to zero to stabilize the robot's behavior.

5. Direction Adjustment:

- `float directionFactor = (TURN == 'L') ? -1.0 : 1.0;` determines the direction of adjustment based on whether the robot is turning left ('L') or right ('R').
- `float adjustedYaw = pvYaw + (distanceError * directionFactor);` adjusts the yaw (orientation) based on the distance error and direction factor.
- `float pidOutput = compassPID.Run(adjustedYaw);` computes the PID output to correct the robot's heading based on the adjusted yaw.

6. Servo Adjustments:

- `steering_servo(pidOutput);` sets the steering servo position based on the PID output to correct the robot's heading.
- `ultra_servo(pvYaw, TURN);` adjusts the position of the ultrasonic sensor based on the current yaw and turning direction.

7. Special Case for Line Detection:

- If `Line_Number` is 12 or more, the robot enters a special state where it continuously runs for 750 milliseconds.
 - During this time, it performs the same actions as before: moving the motors, updating sensor data, and adjusting the steering and ultrasonic servo.
- After this period, `motor(0);` stops the motors, and the robot enters an infinite loop (`while (true) { }`), effectively halting further actions.

The `setup()` and `loop()` functions work together to enable the robot to navigate and maintain a consistent distance from walls while moving in a square pattern.

setup() Function: This function initializes the hardware components and configurations required for the robot to operate. It sets up serial communication, configures the PID controller for managing the robot's heading, and prepares various sensors and actuators. Key initialization steps include attaching servos, setting pin modes, and calibrating the gyroscope to zero. This setup ensures that the robot is properly equipped to perform its tasks once the main loop begins.

loop() Function: This function continuously controls the robot's movement and behavior. It reads sensor inputs to monitor the robot's distance from walls and adjusts its heading accordingly using a PID controller. The robot maintains a target distance from obstacles and navigates in a square pattern by adjusting its steering and ultrasonic sensor positions. Special handling is included for cases when the robot detects a certain number of lines, prompting a specific behavior before stopping the motors.

Obstacle Challenge Code:

Addition Function:

1. motor_and_steer

```
131 void motor_and_steer(int degree) {  
132     degree = max(min(degree, 45), -45);  
133     steering_servo(degree);  
134     motor((map(abs(degree), 0, 45, 50, 45)));  
135 }
```

The `motor_and_steer(int degree)` function adjusts both the steering angle and the motor speed based on the input degree. It ensures the degree is within a safe range, sets the steering servo to this angle, and varies the motor speed inversely with the steering angle. This allows the robot to steer and adjust its speed in a coordinated manner based on the degree input.

2.WrapValue

```
2  ✓ int wrapValue(int value, int minValue, int maxValue) {  
3      int range = maxValue - minValue + 1;  
4  ✓  if (value < minValue) {  
5      |   value += range * ((minValue - value) / range + 1);  
6      |   }  
7      return minValue + (value - minValue) % range;  
8  }
```

The wrapValue function ensures that a given value is constrained within the specified range [minValue, maxValue]. If the value is below minValue, it wraps around to the end of the range by adjusting it with multiples of the range size. The modulo operation then keeps the value within the bounds, effectively handling cases where values might exceed the maximum or go below the minimum.

3.OpenMV Communication Code

```
1 import time
2 import sensor
3 import display
4 from pyb import UART
5
6 # Initialize sensor
7 sensor.reset()
8 sensor.set_pixformat(sensor.RGB565)
9 sensor.set_framesize(sensor.QVGA) # QVGA is 320x240
10
11 # Enable automatic settings for a moment to calibrate, then turn off
12 sensor.set_auto_gain(True)
13 sensor.set_auto_exposure(True)
14 time.sleep(2) # Allow the camera to calibrate for 2 seconds
15 sensor.set_auto_gain(False) # Disable auto gain after calibration
16 sensor.set_auto_exposure(False) # Disable auto exposure after calibration
17 sensor.set_contrast(0)
18 sensor.set_gainceiling(16)
19
20 # Define color thresholds
21 thresholds = [(23, 54, -64, -31, -21, 127)] # General threshold
22 Red_thresholds = [(100, 100, -128, 127, -128, 127)] # Threshold for red
23
24 # Initialize UART
25 uart = UART(3, 19200)
26 clock = time.clock()
27
28 # Initialize LCD using SPI Display with width and height set to match QVGA
29 lcd = display.SPIDisplay()
30
```

The program initializes the camera for color detection by setting its resolution, configuring calibration settings, and defining color thresholds. It also establishes UART communication for interfacing with external devices like an Arduino and sets up an LCD display for visual output. This setup ensures that the system is properly configured to capture and process color information, communicate with other components, and provide feedback through the display

```

30
31 while True:
32     clock.tick()
33     img = sensor.snapshot()
34
35     # Detect and process general blobs
36     general_blobs = img.find_blobs(thresholds, area_threshold=15, pixels_threshold=15, merge=True, margin=2)
37     largest_general_blob = max(general_blobs, key=lambda b: b.area(), default=None)
38
39     # Detect and process the largest red blob
40     red_blobs = img.find_blobs(Red_thresholds, area_threshold=500, pixels_threshold=500, merge=True, margin=2)
41     largest_red_blob = max(red_blobs, key=lambda b: b.area(), default=None)
42
43     # Determine which blob is larger
44     if largest_general_blob and largest_red_blob:
45         if largest_general_blob.area() > largest_red_blob.area():
46             largest_blob = largest_general_blob
47             color = (0, 255, 0)
48             blob_type = 0
49         else:
50             largest_blob = largest_red_blob
51             color = (255, 0, 0)
52             blob_type = 1
53     elif largest_general_blob:
54         largest_blob = largest_general_blob
55         color = (0, 255, 0)
56         blob_type = 0
57     elif largest_red_blob:
58         largest_blob = largest_red_blob
59         color = (255, 0, 0)
60         blob_type = 1
61     else:
62         largest_blob = None
63
64     # If a blob was found, send its data
65     if largest_blob:
66         img.draw_rectangle(largest_blob.rect(), color=color)
67         img.draw_cross(largest_blob.cx(), largest_blob.cy(), color=color)
68         # Send x, y, width, height, and blob type (0 for general, 1 for red)
69         data = "%d,%d,%d,%d,%d\n" % (largest_blob.cx(), largest_blob.cy(), largest_blob.w(), largest_blob.h(), blob_type)
70         uart.write(data)
71
72     # Display the image directly on the LCD
73     lcd.write(img)
74
75     # Optional: read and print incoming data from UART (if needed)
76     if uart.any():
77         data = uart.readline()
78         print("Received:", data)
79

```

This program detects and processes the largest colored blobs within the camera's field of view. It distinguishes between general and red blobs based on predefined color thresholds and identifies which type of blob is the largest. The details of the largest blob, including its position and dimensions, are then sent to an Arduino via UART. This information is crucial for the Arduino to make real-time decisions, such as avoiding obstacles or navigating around detected objects.

Additionally, the program displays the live camera feed with visual annotations (rectangles and crosshairs) on an LCD screen, providing immediate visual feedback. It also monitors and prints any incoming data from the Arduino through UART, facilitating interactive communication between the camera system and the Arduino. This setup ensures that the system can not only detect and track objects effectively but also integrate with other hardware components for comprehensive robotic or automation tasks.

4. Receiving OpenMV Program

```
219 void handleIncomingData() {
220     // Initialize the largest blob variables
221     // largestBlobX = -1;
222     // largestBlobY = -1;
223     // largestBlobWidth = -1;
224     // largestBlobHeight = -1;
225     found_block = false; // Reset found_block at the start
226
227     while (Serial3.available()) {
228         char incomingByte = Serial3.read();
229         if (incomingByte != '\n') {
230             receivedString += incomingByte;
231         } else {
232             Serial.print("Raw Data: ");
233             Serial.println(receivedString);
234
235             receivedString.trim(); // Remove any leading or trailing whitespace
236
237             // Parse the received data
238             int commaIndex1 = receivedString.indexOf(',');
239             int commaIndex2 = receivedString.indexOf(',', commaIndex1 + 1);
240             int commaIndex3 = receivedString.indexOf(',', commaIndex2 + 1);
241             int commaIndex4 = receivedString.indexOf(',', commaIndex3 + 1);
242
243             if (commaIndex1 != -1 && commaIndex2 != -1 && commaIndex3 != -1 && commaIndex4 != -1) {
244                 int xValue = receivedString.substring(0, commaIndex1).toInt();
245                 int yValue = receivedString.substring(commaIndex1 + 1, commaIndex2).toInt();
246                 int widthValue = receivedString.substring(commaIndex2 + 1, commaIndex3).toInt();
247                 int heightValue = receivedString.substring(commaIndex3 + 1, commaIndex4).toInt();
248                 int blobType = receivedString.substring(commaIndex4 + 1).toInt();
249
250                 // Validate the coordinates and dimensions
251                 if (xValue >= 0 && xValue <= 320 && yValue >= 0 && yValue <= 240 && widthValue >= 0 && widthValue <= 320 && heightValue >= 0 && heightValue <= 240) {
252                     // Store the data of the received blob
253                     largestBlobX = xValue;
254                     largestBlobY = yValue;
255                     largestBlobWidth = widthValue;
256                     largestBlobHeight = heightValue;
257                     signature = (blobType == 1) ? 2 : 1; // 2 for red, 1 for general
258
259                     found_block = true; // A valid blob was received
260                     Serial.println("Valid blob detected and stored.");
261                 } else {
262                     Serial.println("Received invalid coordinates or dimensions.");
263                     found_block = false;
264                 }
265             } else {
266                 Serial.println("Data format error.");
267                 found_block = false;
268             }
269             receivedString = ""; // Clear the received string after processing
270         }
271     }
272 }
```

This program handles the reception and processing of data sent from an OpenMV camera via UART. Here's a detailed explanation:

1. Data Reception:

- The function `handleIncomingData()` reads data from the Serial3 UART interface. It collects incoming characters until a newline (`\n`) is encountered, indicating the end of a data packet.

2. Data Processing:

- **Data Accumulation:** As characters are read, they are appended to the `receivedString` variable.
- **Data Parsing:** Once a newline is detected, the complete `receivedString` is processed. The string is expected to contain comma-separated values representing the blob's position and dimensions, along with its type.
 - The string is split based on commas to extract x, y, width, height, and blobType.

- **Validation:** The extracted values are validated to ensure they fall within acceptable ranges (e.g., within the camera frame dimensions). If valid, these values are stored in variables and a flag `found_block` is set to true.

3. Handling Valid Data:

- If the data is valid, the function updates variables to store the position and size of the detected blob, as well as its type (red or general). A message is printed to indicate that a valid blob was detected and stored.

4. Error Handling:

- If the data format is incorrect or the values are out of bounds, error messages are printed. The flag `found_block` is set to false, indicating that no valid blob was detected.

5. Clearing the String:

- After processing, the received string is cleared to prepare for the next data packet.

6. Further Actions:

- The function currently does not perform any additional actions if a valid blob is detected (if `(found_block) {}`). This section is reserved for further processing based on the detected blob's data.

5. Calculating Avoidance

```
162 float calculate_avoidance() {
163     // Check if a block is found
164     if (found_block) {
165         // Reset missed block count if a block is found
166         missedBlockCount = 0;
167
168         // Constants for OpenMV H7
169         int targetHeight = 10; // Height of the object in centimeters
170         float focallength = 3.6; // Focal length of the OpenMV H7 camera lens in mm
171         float cameraFOV = 60.0; // Field of view of the OpenMV H7 camera in degrees
172
173         // Convert focal length to centimeters
174         // focallength /= 10.0;
175
176         // Use the largest blob data
177         int objectHeight = largestBlobHeight;
178         float distance = (targetHeight * focallength * 100) / objectHeight;
179
180         float blockCenterX = largestBlobX;
181         float blockCenterY = largestBlobY;
182
183         float deltaX = blockCenterX - (320 / 2); // Assuming a 640x480 image resolution
184         float deltaY = blockCenterY - (240 / 2);
185
186         float detected_degree = -deltaX * cameraFOV / 320.0; // Inverted deltaX to swap behavior
187
188         float blockPositionX = distance * sin(degreesToRadians(detected_degree));
189         float blockPositionY = distance * cos(degreesToRadians(detected_degree)) - 18;
190
191         // Adjust avoidance degrees based on blob signature
192         if (signature == 1) {
193             avoidance_degree = max(radiansToDegree(atan2(blockPositionX + 9, blockPositionY)), 5);
194             last_block = 'L';
195             Blocks_TURN == 'L';
196             before_last_block = 'L';
197         }
198         else if (signature == 2) {
199             avoidance_degree = min(radiansToDegree(atan2(blockPositionX - 14, blockPositionY)), -5);
200             last_block = 'R';
201             Blocks_TURN == 'R';
202             before_last_block = 'R';
203             avoidance_degree = 0;
204         }
205     } else {
206         // Increment the missed block count if no block is found
207         missedBlockCount++;
208
209         // If the block is missed three times in a row, set avoidance degree to zero
210         if (missedBlockCount >= MAX_MISSED_BLOCKS) {
211             avoidance_degree = 0; // or some default value
212         }
213     }
214 }
```

This function, `calculate_avoidance()`, calculates the degree to which the robot should turn to avoid a detected block based on its size, position, and type. Here's how it works:

1. Block Detection Check:

- The function first checks if a block has been detected (`if (found_block)`). If a block is found, it resets the count of missed blocks (`missedBlockCount = 0`).

2. Constants and Calculations:

- **Constants:** The function uses constants related to the camera and object dimensions:
 - **targetHeight:** The actual height of the object in centimeters.
 - **focalLength:** The focal length of the camera lens in millimeters.
 - **cameraFOV:** The camera's field of view in degrees.
- **Distance Calculation:** The distance to the object is calculated using the formula:

$$\text{distance} = \frac{\text{targetHeight} \times \text{focalLength} \times 100}{\text{objectHeight}}$$

Here, **objectHeight** is the height of the detected blob in pixels.

3. Blob Position:

- **Center Coordinates:** The function computes the blob's center coordinates in the image (**blockCenterX** and **blockCenterY**).
- **Delta Values:** It calculates how far the blob's center is from the image's center (**deltaX** and **deltaY**).
- **Detected Degree:** Converts the horizontal displacement (**deltaX**) into an angle (**detected_degree**) that represents how far off-center the blob is.

4. Block Position in Space:

- **Block Position:** Converts the 2D position into a 3D space using the distance and angle. The position is calculated using trigonometric functions:
$$\text{blockPositionX} = \text{distance} \times \sin(\text{detected_degree})$$
$$\text{blockPositionY} = \text{distance} \times \cos(\text{detected_degree}) - 18$$

Here, -18 is a vertical adjustment to the position.

5. Avoidance Degree Calculation:

- **For General Blocks:** If the blob type is general (**signature == 1**), the function calculates the avoidance degree based on the block's position. The degree is adjusted to ensure the robot turns sufficiently to avoid the block.
- **For Red Blocks:** If the blob is red (**signature == 2**), the avoidance degree is adjusted differently, often set to zero, indicating a different reaction.

6. Missed Block Handling:

- If no block is found (else block), the missed block count is incremented. If blocks are missed three times in a row (missedBlockCount >= MAX_MISSED_BLOCKS), the avoidance degree is set to zero, which might mean no avoidance action is taken.

7. Return Value:

- The function returns the calculated avoidance_degree, which tells the robot how much to turn to avoid the detected block.

Setup Code:

```
126 void setup() {
127     // put your setup code here, to run once:
128     Serial.begin(19200);
129     Serial3.begin(19200);
130
131     compassPID.Start(0, 0, 0);
132     compassPID.SetOutputLimits(-180, 180);
133     compassPID.SetSampleTime(10);
134
135     pinMode(STEER_SRV, OUTPUT);
136     pinMode(ULTRA_SRV, OUTPUT);
137     pinMode(ULTRA_PIN, INPUT);
138     pinMode(RED_SEN, INPUT);
139     pinMode(BLUE_SEN, INPUT);
140     pinMode(BUTTON, INPUT);
141
142     initMotor();
143     while (!Serial)
144         ;
145     myservo.attach(ULTRA_SRV, 500, 2400);
146     myservo2.attach(STEER_SRV, 500, 2500);
147     steering_servo(0);
148     ultra_servo(0, 'L');
149     while (analogRead(BUTTON) > 500)
150         ;
151     zeroYaw();
152 }
153
```

The setup() function initializes the hardware components and configurations necessary for the robot's operation, similar to the initial setup phase of the project. It sets up serial communication, PID control, and pin modes for sensors and actuators, and initializes servo positions. The main change in this version is related to the calculation in the loop() function, which will be discussed next.

Main Loop Code:

```

155 void loop() {
156
157   while (analogRead(BUTTON) > 500) {
158     getIMU(); // Read data from OpenMV
159     ultra_servo(pvYaw, Blocks_TURN);
160     Color_detection();
161     // Additional logic for distance tracking
162     float desiredDistance = 32.5;
163     if (TURN == 'L') {
164       desiredDistance += 5;
165     } else if (TURN == 'R') {
166       desiredDistance -= 5;
167     }
168     float distanceError = getDistance() - (desiredDistance);
169     float deadband = 2.0;
170     if (abs(distanceError) < deadband) {
171       distanceError = 0.0;
172     }
173     float directionFactor = (Blocks_TURN == 'L') ? -1.0 : 1.0;
174     float adjustedYaw = pvYaw + (distanceError * directionFactor);
175     pidOutput = compassPID.Run(adjustedYaw);
176     if (millis() - MV_timer > 20) {
177       handleIncomingData();
178     }
179     calculate_avoidance();
180     if ((avoidance_degree > 0) && largestBlobY > 0) {
181       int final_degree = mapf(min(max(getDistance(), 5), 28), 5, 28, pidOutput, 1.5 * avoidance_degree);
182       motor_and_steer(final_degree);
183       //Serial.println(final_degree);
184     } else {
185       motor_and_steer(pidOutput);
186     }
187   }
188 }
189 }

```

The loop() function in this program continuously executes the robot's main tasks, including reading sensor data, processing information, and controlling the robot's movements. Here's a breakdown of what each part of the loop() function does:

1. Button Check:

- while (analogRead(BUTTON) > 500) { ... }: This loop runs as long as the button is not pressed (i.e., its analog value is above 500). When the button is pressed, the loop() function will exit the while loop and stop executing.

2. Data Reading and Processing:

- getIMU();: Reads the data from the IMU sensor (likely to update orientation data).
- ultra_servo(pvYaw, Blocks_TURN);: Adjusts the position of the ultrasonic sensor servo based on the current yaw (pvYaw) and turn direction (Blocks_TURN).

3. Color Detection:

- `Color_detection();`: Executes color detection logic to determine the color of detected objects and update the turn direction.

4. Distance Tracking:

- `float desiredDistance = 32.5;`: Sets a default desired distance from the wall.
- If TURN is 'L' (left), it increases the desired distance by 5 units.
- `float distanceError = getDistance() - (desiredDistance);`: Calculates the error between the current distance and the desired distance.
- `float deadband = 2.0;`: Defines a tolerance range for distance error.
- If the distance error is within the deadband, it is set to 0 to minimize corrections.
- `float directionFactor = (Blocks_TURN == 'L') ? -1.0 : 1.0;`: Determines the direction of adjustment based on turn direction.
- `float adjustedYaw = pvYaw + (distanceError * directionFactor);`: Adjusts the yaw based on distance error.
- `pidOutput = compassPID.Run(adjustedYaw);`: Calculates the PID output to control the steering based on the adjusted yaw.

5. Handling Incoming Data:

- `if (millis() - MV_timer > 20) { handleIncomingData(); }`: Calls `handleIncomingData()` to process incoming data from the OpenMV camera every 20 milliseconds.

6. Avoidance Calculation:

- `calculate_avoidance();`: Computes the degree of avoidance needed based on detected blobs and updates `avoidance_degree`.

7. Motor Control:

- `if ((avoidance_degree > 0) && largestBlobY > 0) { ... }`: Checks if avoidance is needed (i.e., if `avoidance_degree` is positive and `largestBlobY` is valid).
 - `int final_degree = mapf(min(max(getDistance(), 5), 28), 5, 28, pidOutput, 1.5 * avoidance_degree);`: Maps the distance to a final steering degree based on PID output and avoidance degree.
 - `motor_and_steer(final_degree);`: Adjusts the motor and steering based on the computed final degree.
- `else { motor_and_steer(pidOutput); }`: If no avoidance is needed, it uses the PID output directly to control the motors and steering.

