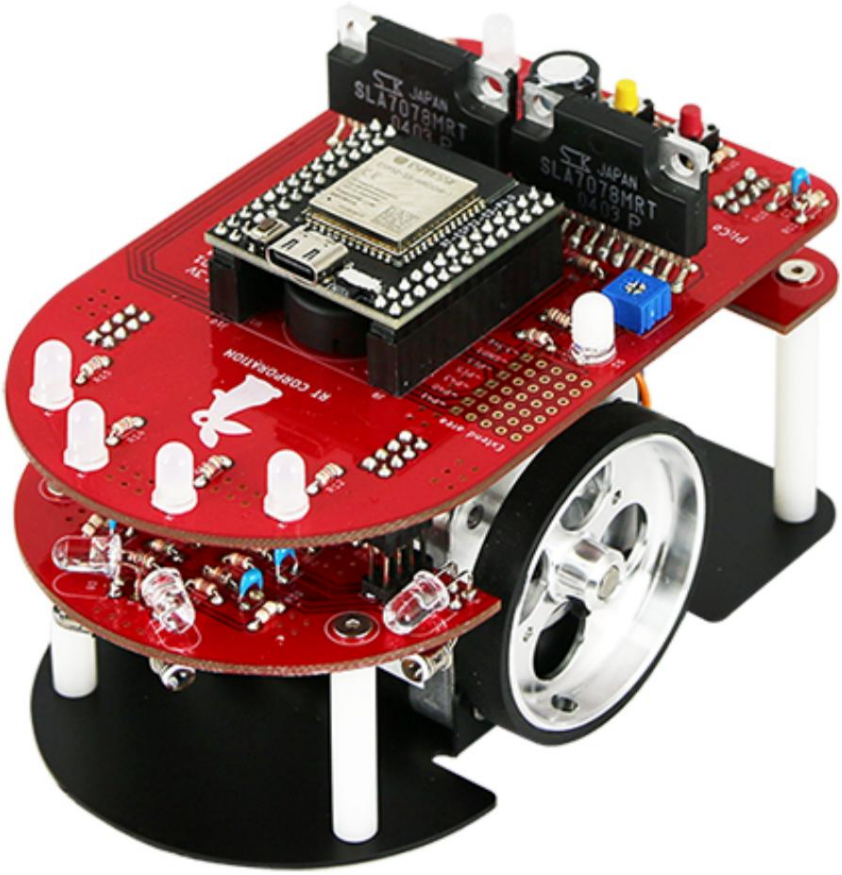
Pi:Co Classic3

Software Manual

(Arduino)



Version 1.0 RT Co., Ltd.

RT CORPORATION

table of contents

Before Use

Thank you for purchasing our "Pi:Co Classic3 (hereinafter referred to as "this product").
Thank you very much.

This manual describes the sample programs for this product.
Please read the attached Pi:Co **Classic3** Getting Started Guide before purchasing .

vinegar.

Tools used, OSS versions

Here we will provide additional information on the environment settings in the Arduino development environment construction manual.
The tools and OSS versions used at the time of writing this manual are as follows.

Main tools and OSS used in this book

Tools, OSS, etc.	Version	**URL**
Arduino IDE	2.3.2	https://www.arduino.cc/en/software
Arduino core for the ESP32 3.0.1		https://github.com/espressif/arduino-esp32/releases
micro-ROS Arduino examples for Pi:Co Classic3	2.0.0	https://github.com/rt-net/pico_micro_ros_arduino_examples/releases

Microcontroller board pin arrangement and IO connection destination

This section explains the pin arrangement and IO connection destination of the microcomputer board mounted on this product.
Please refer to the table. NC is the abbreviation for No Connect. IO number is the port number of ESP32-S3.

vinegar.

IO19 (PR12) and IO20 (PR9) are connected to USB. When USB is not in use, IO19 and
You can use IO20.

IO0 (PR6) is a mode switch. It is connected to GND at boot time, so it is not connected to GPIO.
It is not recommended to use it as such.

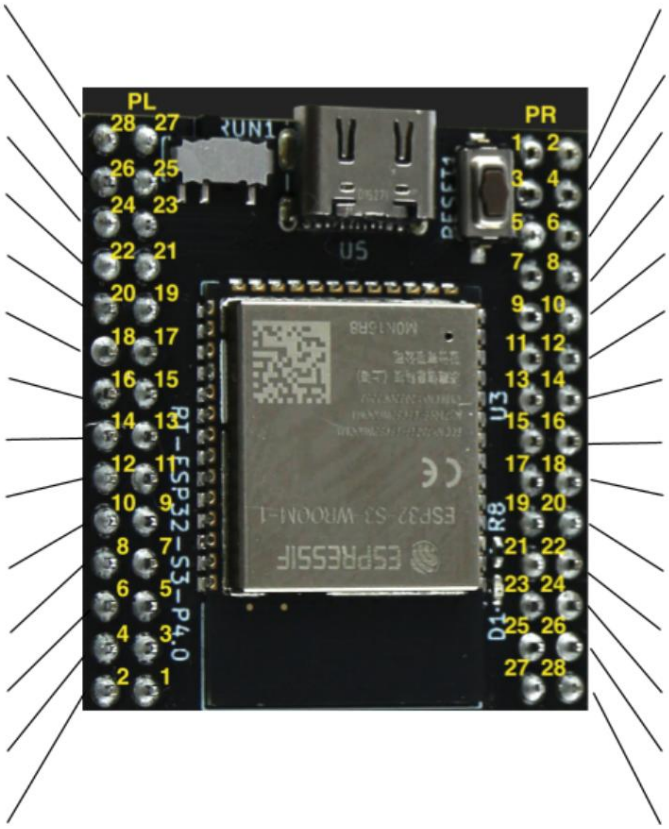
IO45 (PL15) and IO46 (PR7) are pulled down to GND with 10kÿ.

IO48(PL20), IO47(PL18), IO3(PR10), IO46(PR7) are not connected to the functions of this product.
Can be used as GPIO. Regarding IO46 (PR7), when used as GPIO, it is active HIGH.
It is recommended to use it as an output.

IO4 (PL22), IO5 (PL23), IO6 (PL24), and IO7 (PL25) are pulled to 3.3V with 1Mÿ.
It has been uploaded.

Pin Layout

Outside	Inside		Inside	Outside
NC	NC		IO12	IO11
NC	IO7		IO10	IO9
IO6	IO5		NC	IO0
IO4	IO8*		IO46	NC
IO48	GND		IO20	IO3
IO47	NC		IN	IO19
NC	IO45		NC	NC
NC	IO21		NC	RXD0
IO14	IO13		TXD0	NC
IO8*	IO38		IO17	IO18
IO39	IO40		IO15	IO16
IO41	IO42		NC	NC
IO2	IO1		NC	NC
3.3V	3.3V		NC	NC



* IO8 is located in two places.

IO connection list

IO number	Connection destination		IO number	Connection destination		IO number	Connection destination
0	MD		12	SW_R		37	No pin
1	LED0		13	PWM_R		38	BUZZER
2	LED1		14	CW_R		39	BLED1
3	Not connected		15	SLED_FL		40	BLED0
4	AD1		16	SLED_FR		41	LED3
5	AD2		17	SLED_L		42	LED2
6	AD3		18	SLED_R		45	PWM_L
7	AD4		19	USB D-		46	Not connected
8	AD0		20	USB D+		47	Not connected
9	ENGINE_EN		21	CW_L		48	Not connected
10	SW_L		35	No pin		TXD0	TXD
11	SW_C		36	No pin		RXD0	RXD

Explanation of sample programs

Here, we will explain how to develop and operate this product using the sample program collection for this product, micro-ROS Arduino examples for Pi:Co Classic3. Please install the sample programs in advance by referring to the separate Arduino development environment construction manual.

The sample program for this product can be developed with the Arduino IDE, so beginners to robot programming can learn how to operate this product in a friendly environment. For example, the Arduino IDE provides functions for commonly used initial settings, so you can initialize it just by executing functions such as Serial.begin and timerBegin. It is also helpful for beginners to know what the function names, such as digitalWrite and analogRead, do just by looking at them.

About the structure of this book

Furthermore, for those new to robot programming, this book explains how to use Arduino IDE and how to operate this product. The content is understandable even for those with no experience programming with Arduino IDE or those who have never operated LEDs, buzzers, motors, etc. The product's circuit diagram is also included, so reading it together with the program will make it easier to understand how to operate this product. The sample program used in this book is structured as follows.

Ultimately, you will be able to use this product to control a micro mouse.
Sample programs have been created to enable participants to participate in the competition.

STEP 1 to STEP 7 are sample programs that check the necessary functions of a micromouse.

This is a sample program in which STEP8 acts as a micromouse.

Sample program configuration

sample	content	Development environment
STEP1 - STEP7	Operate the necessary functions of a micromouse, such as LEDs, sensors, and motors	Windows Linux(Ubuntu) macOS
STEP8	Program for the Micromouse Competition	Windows Linux(Ubuntu) macOS

Arduino sketch notation

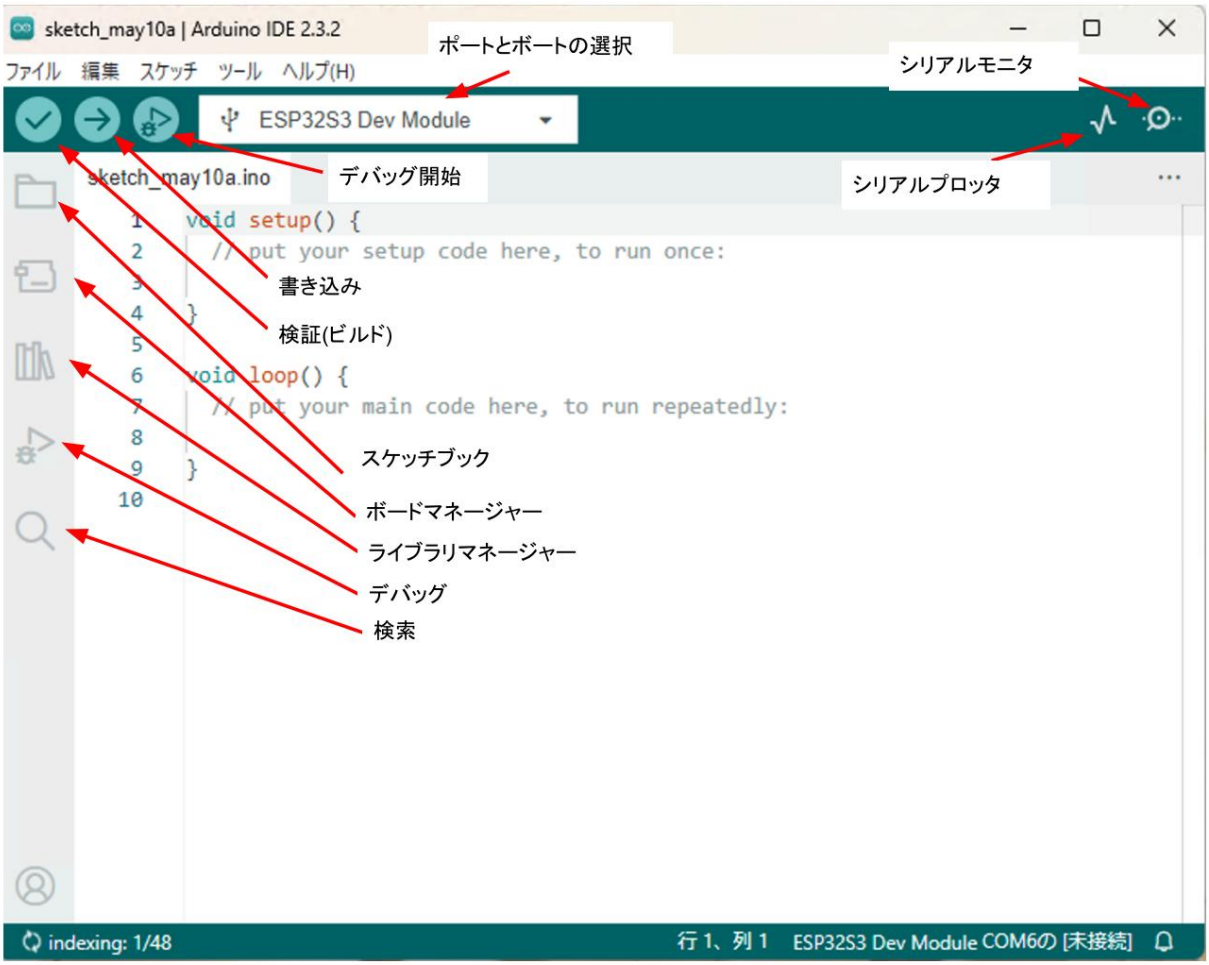
Programs handled in the Arduino IDE are called sketches. This manual provides explanations specific to sketches. Otherwise, sketches are referred to as "programs" or "codes".

An Arduino sketch is written as follows. The lines below "C/C++" are the contents of the sketch.

```
C/C++
void setup(){
  // Set the pin mode to output
  pinMode(LED0, OUTPUT); }


void loop(){
  // Blink the LED
  digitalWrite(LED0, HIGH);
  delay(500);
  digitalWrite(LED0, LOW);
  delay(500); }
```

About **Arduino** IDE icons

The icon names are as follows:



Arduino IDE icon names

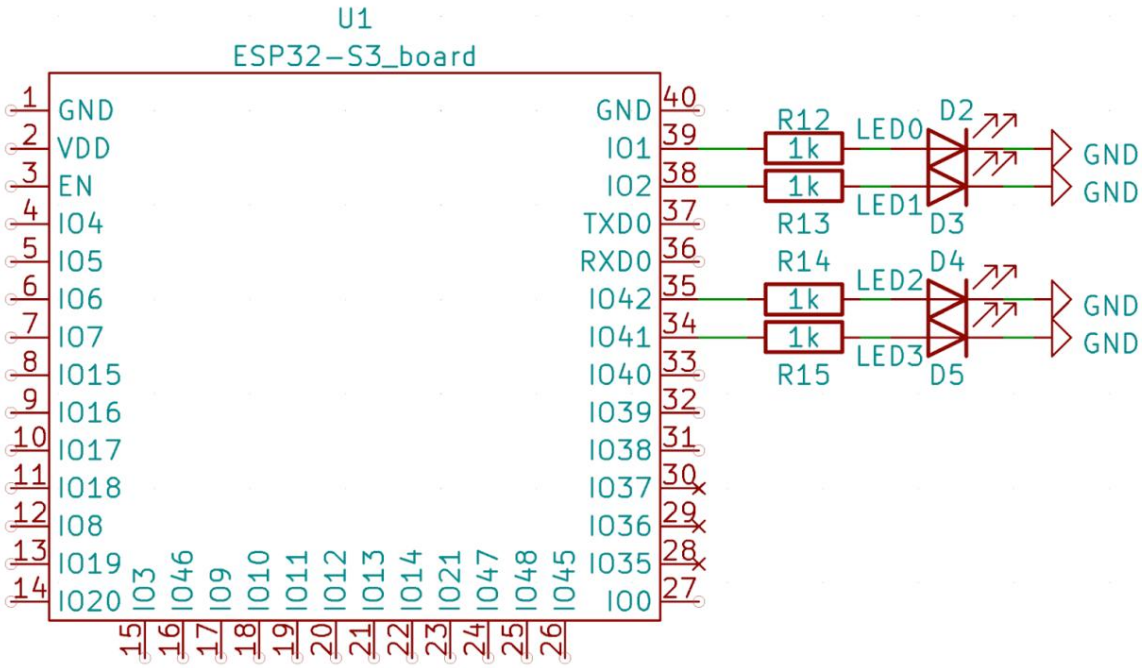
STEP 1 Make the LED blink

overview

This is a sample program that repeatedly turns all four LEDs (mode display LEDs) on in the front on for 0.5 seconds and then off for 0.5 seconds.

Circuit Diagram

The circuit diagram below shows the circuit elements related to blinking an LED.



LED driving circuit

Circuit operation explanation

When 1 is output, the GPIO (General Purpose Input/Output) port is connected
This is a circuit that turns on the LED and turns it off when it outputs 0. When the program outputs "1", GPIO outputs 3.3V, and when the program outputs "0", GPIO outputs 0V.

program

The program for STEP 1 is shown in the diagram.

Figure 1-1

```
C/C++
#define LED0 1
#define LED1 2
#define LED2 42
#define LED3 41
```

Figure 1-2

```
C/C++
void setup(){ // put your
  setup code here, to run once: pinMode(LED0, OUTPUT); pinMode(LED1,
  OUTPUT); pinMode(LED2, OUTPUT);
  pinMode(LED3, OUTPUT); }
```

Figure 1-3

```
C/C++
void loop(){ // put
  your main code here, to run repeatedly: digitalWrite(LED0, HIGH); digitalWrite(LED1,
  HIGH); digitalWrite(LED2, HIGH);
  digitalWrite(LED3, HIGH); delay(500);
  digitalWrite(LED0, LOW); digitalWrite(LED1,
  LOW); digitalWrite(LED2, LOW);
  digitalWrite(LED3,
  LOW); delay(500); }
```

Commentary

The most commonly used sample program for microcontrollers is a program that blinks an LED. The blinking of an LED is called "blinking an LED." Being able to blink an LED not only means that the microcontroller development environment is ready, but also that a simple debugging environment is ready.
From STEP 1 onwards, the LED blinking will be used to check whether the program is working.

About the main function

Arduino Sketch is a C-like programming language, so you can use C or C++.
You can write almost the same code. This sample program is written in a C-like language.

In C language, you always write a function called main function that is executed when the program starts. However, Arduino sketches do not have a main function. Instead, they have a setup function and a loop function. The loop function is executed after the setup function is executed. The loop function has the characteristic that when the processing is completed to the end, it returns to the top of the loop function and executes the processing again.

About #define

A brief explanation of #define at the beginning of a program. The define statement replaces the string to the right of define with the number or string to the right of it before compiling. These are preprocessor macro definitions that

```
C/C++
#define LED0 1
```

When you write the above define statement, the "LED0" in the program will be replaced with "1" and the They will be piled.

This product has four display LEDs. To make the LED control statements easier to understand, The strings LED0, LED1, LED2, and LED3 are defined and linked to the IO numbers of the ESP32-S3. The numbers

defined in the above define statements are the GPIO numbers of the ESP32, meaning IO1. In the Arduino sketch, you can control the GPIO by specifying the IO number of the ESP32. Please note that this is not the pin number of the ESP32 IC.

If you do not use #define, pinMode(LED0,OUTPUT) in Figure 1-2 becomes pinMode(1,OUTPUT). When you are developing, you can remember that "1" is LED0, but after some time has passed, you will forget which function "1" was assigned to. We recommend that you program using strings instead of numbers whenever possible. Defining with #define also improves the reusability of your program.

For example, if you change the specifications, When you change the port that drives the LED, you only need to change the number in this #define. There is no need to modify the program deeply.

About the setup function

Next, we will explain the contents written in the setup function. The setup function is a function that is executed only once before the loop function is executed. This is where you often write the initialization of the microcontroller's functions and variables. In this sample program, the purpose is to blink the LED, and the blinking of the LED is seen as an output from the microcontroller.

To set a GPIO as an output, use the built-in function called pinMode in the Arduino IDE. Since Arduino sketches recognize the case of function names, please write the "M" of the pinMode function in capital letters. The first argument of the pinMode function is the pin number, and the second argument is the input/output direction. In a typical C language environment, you need to include the file of the function you are using, but since the pinMode function is built into the Arduino IDE environment, there is no need to include it. Also, you can use statements such as pinMode(LED0,OUTPUT) to represent GPIO input and output as "INPUT" and "OUTPUT". You can set it manually. INPUT and OUTPUT are also built into the Arduino IDE as functions. Setting input and output using the letters INPUT and OUTPUT is convenient because it is easy to see later that this pin is set as OUTPUT. Although this is a convenient function, there is a point to be aware of. Do not use the functions and variables that are pre-installed in the Arduino IDE for purposes other than their intended use. If the user uses them for a different function, build errors may occur or variable settings may be changed unintentionally. You can check the defined functions and variables on the Arduino web page. https://www.arduino.cc/reference/en/

About the loop function

Finally, we will explain the contents of the loop function. The loop function is where you write the contents to execute a program repeatedly. It may be easier to understand if you read it as while(1){ } in C language. The display LED of this product lights up when it is set to 1 (= HIGH) and turns off when it is set to 0 (= LOW).

To output "1" or "0" to a GPIO, use the built-in Arduino IDE function digitilWrite. The first argument is the pin number, and the second argument is the value to be output. Here too, the output value is set to "HIGH" or "LOW". HIGH is defined as “1” and LOW as “0”, so they can be used with functions other than digitalWrite. After turning on the LED with digitalWrite(LED0,HIGH), delay(500) is written. Even without this description, the LED will be turned on repeatedly with digitalWrite(LED0,HIGH) and turned off with digitalWrite(LED0,LOW). However, because the processing speed of the microcontroller is so fast, the human eye cannot detect the blinking. For this reason, a time is set for the light to be turned on and off so that it can be detected by the human eye. The function that controls this delay is the built-in delay function of the Arduino IDE. When this delay function is used, the program will pause for the argument value x ms (milliseconds). In the sample, delay(500) is written, so the light will be on for 500[ms] and the light will be off for 500[ms].

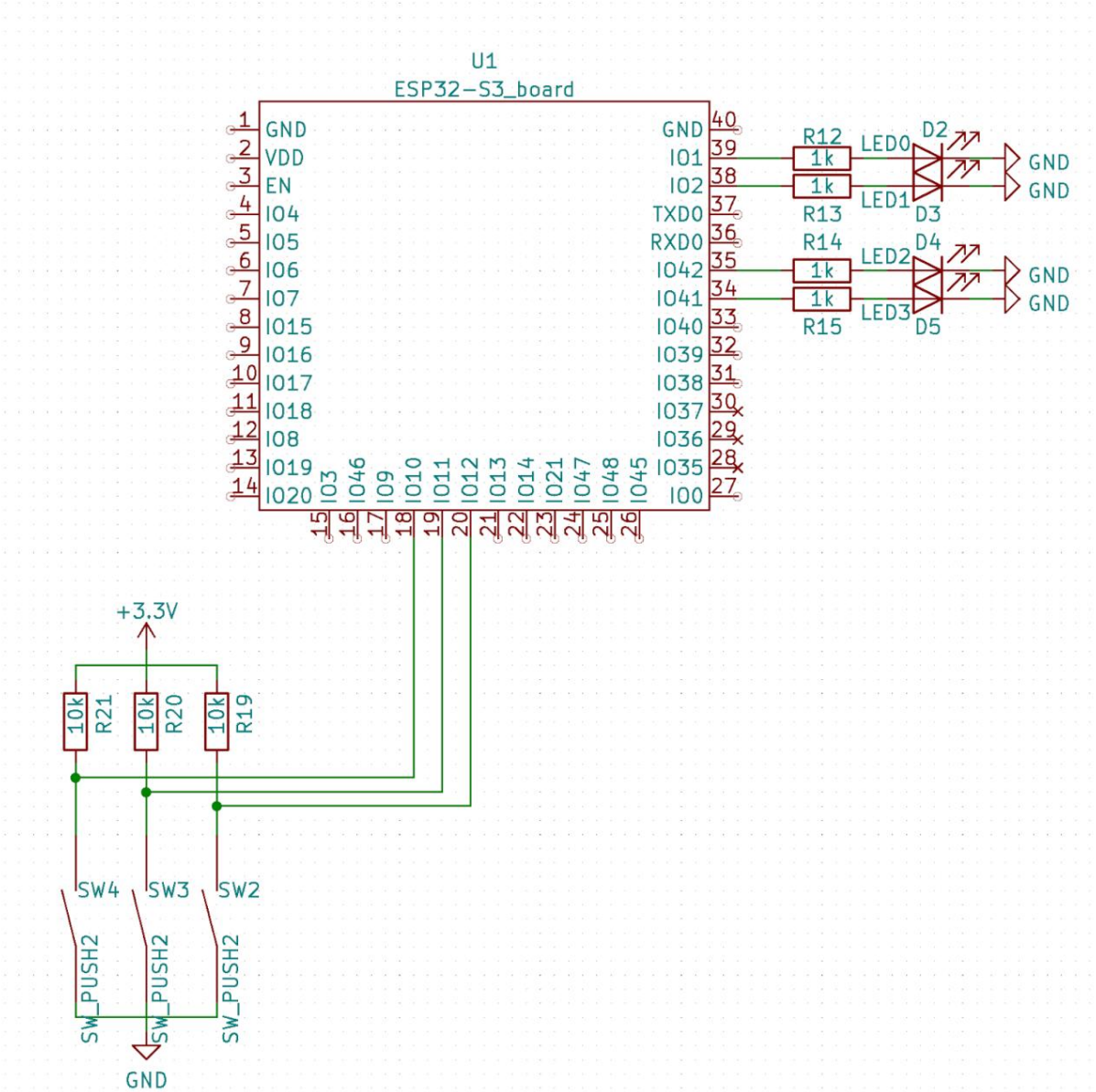
STEP 2 Press the switch to turn on the LED

overview

This is a sample program that lights up the display LEDs by pressing the switches (sample mode selection buttons). Pressing the blue switch lights up the right LED, pressing the yellow switch lights up the two middle LEDs, and pressing the red switch lights up the left LED. Pressing it again turns it off.

Circuit Diagram

The circuit diagram below shows the circuit elements related to switch input and LED lighting.



Switch status check circuit

Circuit operation explanation

A tactile switch circuit connected from IO10 to IO12 is added to the STEP1 circuit.

R19, R20, and R21 are pull-up resistors. To use the internal pull-up resistor function of the ESP32-S3, You can also remove R19 to R21 if you want.

The resistance of the tact switch is nearly 0ÿ when it is pressed, and 1Mÿ or more when it is not pressed.

The 3.3V voltage is divided by the pull-up resistor and the resistance of the tact switch and input to the ESP32-S3. When the tact switch is not pressed, the voltage is divided by the 1Mÿ resistor and the 10kÿ pull-up resistor, and approximately 3.3V is input to the ESP32-S3 and recognized as High. When the tact switch is pressed, the resistance of the tact switch is much smaller than the pull-up resistor, so the GND voltage is input to the ESP32-S3 and recognized as Low.

program

The program for STEP 2 is shown in the figure.

Figure 2-1

```
C/C++
#define LED0 1
#define LED1 2
#define LED2 42
#define LED3 41

#define SW_L 10
#define SW_C 11
#define SW_R 12

int g_state_r, g_state_c, g_state_l;
```

Figure 2-2

```
C/C++
void setup()
  { pinMode(LED0, OUTPUT);
  pinMode(LED1, OUTPUT);
  pinMode(LED2, OUTPUT);
  pinMode(LED3, OUTPUT);

  pinMode(SW_L, INPUT);
  pinMode(SW_C, INPUT);
  pinMode(SW_R, INPUT);

  g_state_r = g_state_c = g_state_l = 0; }
```

Figure 2-3

```
C/C++
void loop(){ while
  (digitalRead(SW_L) && digitalRead(SW_C) && digitalRead(SW_R))
{ continue;


  } if (digitalRead(SW_R) == 0)
      { digitalWrite(LED3, (++g_state_r) & 0x01);

  } if (digitalRead(SW_C) == 0)
      { digitalWrite(LED2, (++g_state_c) & 0x01); digitalWrite(LED1,
      (g_state_c)&0x01);

  } if (digitalRead(SW_L) == 0)
      { digitalWrite(LED0, (++g_state_l) & 0x01);

  } delay(30);
  while (!(digitalRead(SW_L) && digitalRead(SW_C) && digitalRead(SW_R)))
{ continue;


  } delay(30); }
```

Commentary

Global Declaration

In Figure 2-1, there are declarations of variables called int g_state_r, g_state_c, and g_state_l that were not included in the LED blinking program in STEP 1. These declarations are global declarations, and are convenient variables that can be read and written from any function.

About Global Variables

The global variables in the Arduino IDE are slightly different from those in C language, and can be read and written from other sketches in the tab without declaring extern. Set g_state_r, g_state_c,

and g_state_l as global variables to record the number of times the switch is pressed.
The LED is controlled so that it is off when the number is even and on when the number is odd.

About the setup function

The setup function sets the ports used to control the LED and the switch. The switch is an input from the microcontroller's perspective, so INPUT is specified as the second argument to the pinMode function, such as pinMode(SW_L,INPUT). In Figure 2-2,

g_state_r = g_state_c = g_state_l = 0, which initializes the state to 0 according to the number of times the switch was pressed and the LED blinking state.

About the loop function

Next, we will explain the processing of the loop function. The switch circuit of this product inputs a LOW signal to the port when pressed, and a HIGH signal when not pressed. The function to read the port signal uses the built-in function digitalRead(pin number) of the Arduino IDE. The while statement at the beginning of Figure 2-3 loops until one of the switches is pressed. Since one of the conditions is taken, the digitalRead function and the digitalRead function are ANDed with "&&". The return value of digitalRead is either HIGH or LOW, so the result will be the same even if you only use "&". The argument of while is the result of the conditional expression, either true or false, so the logical operator "&&" is used.

When none of the switches SW_L, SW_C, and SW_R are pressed, the return value of the digitalRead function is HIGH && HIGH && HIGH, the calculation result is HIGH, and the while statement condition is met when none of the switches are pressed. When any of SW_L, SW_C, or SW_R is pressed, the calculation result becomes LOW, so the while statement is exited. The if statement in Figure 2-3 determines which LED to turn on/off depending on the switch that was pressed. SW_R, SW_C, and SW_L do the same thing, just the LED that is turned on is different, so we will use SW_R as a representative example. digitalWrite(LED3,++g_state_r)&0x01) controls the LED on/off, but various processes are written together on one line. digitalWrite(LED3,++g_state_r)&0x01) can be broken down as follows:

```
C/C++
    g_state_r = g_state_r +1;•••ÿ int temp = g_state_r &
    0x01;•••ÿ digitalWrite(LED3,temp);•••ÿ
```

Expanding this out, I think you can understand what has been written together. ++g_state_r can be rewritten as ÿ. In ÿ, we determine whether the g_state_r variable to which 1 has been added is even or odd. In binary numbers, the determination of whether a number is even or odd is made by the least significant bit (LSB). 0 is an even number, 1 is an odd number. To keep only the least significant bit, we take a logical AND with 0x01.

You could use an if statement to write digitalWrite(LED3,LOW) when the value is 0, and digitalWrite(LED3,HIGH) when the value is 1, but the values 0 and 1 which distinguish between odd and even numbers are the same as turning the LED on and off, so just enter the values as they are.

About the delay function

The delay function used here waits for the switch chatter to subside. Mechanical switches have a spring element inside, so when the switch is pressed or released, the spring vibrates, causing repeated HIGH and LOW states. This phenomenon of repeated HIGH and LOW states is called chattering. Chattering often occurs for a few ms to a few tens of ms, and here it is set to 30 ms. The state of the switch is checked after the chattering has subsided.

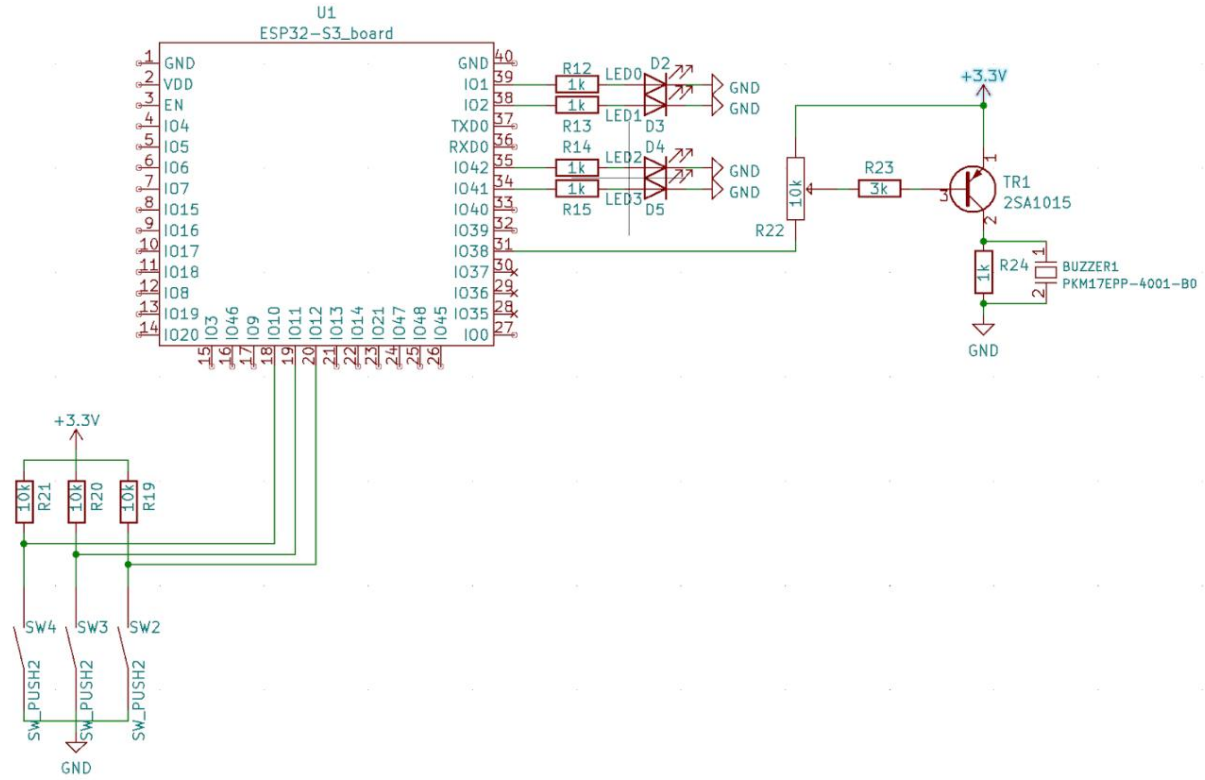
STEP 3: Use the buzzer to make a sound

overview

This is a sample program in which pressing the blue and red switches changes the LED lighting pattern, and pressing the yellow switch sounds a buzzer sound corresponding to the lighting pattern. The LED lighting pattern corresponds to a binary value, and pressing the blue switch increments (adds 1) and pressing the red switch decrements (subtracts 1). When the lighting pattern is 1, a "Do" sound is played for 1 second, when it is 2, a "Re" sound is played, and when it is 3, a "Mi" sound is played for 1 second. No sound is played for any other value.

Circuit Diagram

The circuit diagram below shows the circuit elements related to the operation of the buzzer, LED, and switch.



LED, switch, buzzer circuit

Circuit operation explanation

In STEP 3, a circuit connected to IO38 is added. A piezoelectric buzzer is used for the buzzer (BUZZER1). The sound is generated by changing the voltage across the piezoelectric buzzer. A transistor (TR1) and a resistor (R24) are required to change the voltage across the two ends.

When a LOW signal is output from IO38, TR1 turns ON and a voltage of 3.3V is applied to the piezoelectric buzzer. At this time, the electric potential of the piezoelectric buzzer changes, causing the metal plate inside to warp. After that, when IO38 is set to HIGH, TR1 turns OFF, and the charge stored in the piezoelectric buzzer is discharged through resistor R24. At this time, the electric potential of the piezoelectric buzzer changes to 0V, and the metal plate returns to its original state. By continuously changing this metal plate, sound is generated.

The base current of TR1 is adjusted by the variable resistor R22. If the base current of TR1 is large, the current flowing through the collector and emitter of TR1 will also be large, so the voltage applied to the piezoelectric buzzer will increase and the sound will become louder.

program

The program for STEP 3 is shown in the figure.

Figure 3-1

C/C++

```
#define LED0 1
#define LED1 2
#define LED2 42
#define LED3 41

#define SW_L 10
#define SW_C 11 #define
SW_R 12

#define BUZZER 38

#define INC_FREQ 2000 #define
DEC_FREQ 1000

#define FREQ_C 523 // Do #define
FREQ_D 587 // Re #define FREQ_E
659 // Mi

char g_mode;
```

Figure 3-2

```
C/C++
void setLED(char data){ if
  (data & 0x01)
     { digitalWrite(LED0, HIGH); }
  else
     { digitalWrite(LED0, LOW);

  } if (data & 0x02)
     { digitalWrite(LED1, HIGH); }
  else
     { digitalWrite(LED1, LOW);

  } if (data & 0x04)
     { digitalWrite(LED2, HIGH); }
  else
     { digitalWrite(LED2, LOW);

  } if (data & 0x08)
     { digitalWrite(LED3, HIGH); }
  else
     { digitalWrite(LED3, LOW);

}}
```

Figure 3-3

```
C/C++
void execByMode(char mode){ switch (mode)
  { case 1:

        ledcWriteTone(BUZZER, FREQ_C); delay(1000);
        ledcWrite(BUZZER,
        1024); break; case 2:


        ledcWriteTone(BUZZER, FREQ_D);
        delay(1000);
        ledcWrite(BUZZER, 1024);
        break;
    case 3:
        ledcWriteTone(BUZZER, FREQ_E);
        delay(1000);
        ledcWrite(BUZZER, 1024);
        break;
    default:
        ledcWrite(BUZZER, 1024);
        break; }
}
```

Figure 3-4

```
C/C++
void setup(){
  // put your setup code here, to run once: pinMode(LED0,
  OUTPUT); pinMode(LED1,
  OUTPUT); pinMode(LED2,
  OUTPUT); pinMode(LED3,
  OUTPUT);

  pinMode(SW_L, INPUT);
  pinMode(SW_C, INPUT);
  pinMode(SW_R, INPUT);

  ledcAttach(BUZZER,440, 10);
  ledcWrite(BUZZER, 1024);

  g_mode = 1;
  setLED(g_mode); }
```

Figure 3-5

```
C/C++
void loop(){ // put
  your main code here, to run repeatedly: while (digitalRead(SW_L)
  & digitalRead(SW_C) & digitalRead(SW_R)) { continue;




  } if (digitalRead(SW_R) == 0) { g_mode+
      +; if (g_mode
      > 15) { g_mode = 15; }
          else

      { ledcWriteTone(BUZZER, INC_FREQ);
          delay(30);
          ledcWrite(BUZZER, 1024);

      } setLED(g_mode);

  } if (digitalRead(SW_L) == 0) {
      g_mode--; if
      (g_mode < 1) { g_mode
          = 1; } else

      { ledcWriteTone(BUZZER, DEC_FREQ);
          delay(30);
          ledcWrite(BUZZER, 1024);

      } setLED(g_mode);

  } if (digitalRead(SW_C) == 0) {
      ledcWriteTone(BUZZER, INC_FREQ);
      delay(80);
      ledcWriteTone(BUZZER, DEC_FREQ);
      delay(80);
      ledcWrite(BUZZER, 1024);
      delay(300);
      execByMode(g_mode);
  }
```

```
  while (!(digitalRead(SW_L) & digitalRead(SW_C) & digitalRead(SW_R)))
{ continue;


  } delay(30); }
```

Commentary

STEP1 and STEP2 are configured only with the setup function and the loop function, but in STEP3, the LED
We will create a function to process the lights and the buzzer to ring the notes Do-Re-Mi.

About the setLED function

The function to turn on the LED is void setLED(char data). The setLED function (Figure 3-2) turns on the LED according to the argument value.
Turn on the LED. Assign the LED lighting pattern to the argument value as follows:
For example, if you enter 1 (0001 in binary) as the argument, LED0 will light up, and if you enter 6 (0110 in binary) as the argument,
When the power is turned on, LED2 and LED1 will light up.

LED lighting pattern compatibility table

7	6	5	4	3	2	1	0
invalid				LED3	LED2	LED1	LED0

In order to make it look more like the Arduino IDE, the lights are turned on and off using HIGH and LOW.
Using shift operations makes the if statement unnecessary. Writing LED3 on/off using shift operations
will look like this. This description will be used in STEP8.

```
C/C++

digitalWrite(LED3,(data>>3)&0x01);
```

About the setup function

Next, we will explain the processing of the setup function.
The first half of Figure 3-4 (GPIO settings in pinMode) is the same as STEP 2. The latter half of the ledcAttach function is
This is the initial setting function required to make a sound.
The buzzer of this product will not make a sound if you only connect power to both ends of the element.
The sound is produced by changing the voltage across both ends. The frequency of the voltage change is the frequency of the sound.
I will.

About ledcAttach function

The ESP32 built-in function ledcAttach (argument 1, argument 2, argument 3) specifies the pin to output PWM, frequency, and PWM resolution. LEDC has 8 channels. When using the ledcAttach function, unused channels are automatically assigned. The first argument specifies the IO number to output PWM.

Here, it is BUZZER (=38). The unit of frequency for the second argument is [Hz]. Here, 440 [Hz] is specified.

The third argument specifies the resolution of the PWM duty cycle as a power of 2. The maximum is 14. The unit is [bit]. For sound, a duty cycle of 50 [%] is sufficient, so high precision duty cycle resolution is not required, and frequency precision is more important, so the third argument is set to 10.

The clock division is calculated and set within the ledcAttach function. Depending on the value of this division, high and low tones may not be produced, so you need to be careful about the range of sounds that can be used. It has been confirmed that with this setting, sounds from 440 to 4000 Hz can be produced. When this ledcAttach function is executed, the pinMode function is called internally, There is no need to write pinMode(BUZZER,OUTPUT).

About ledcWrite function

PWM is output using the ESP32 built-in function ledcWrite(argument 1, argument 2). The first argument specifies the IO number. The ledcAttach function and IO number are specified. The second argument specifies the PWM duty ratio. The ledcAttach function set the PWM resolution to 10[bit], so the input range should be 0 to 1023, but the specification is that the PWM output will be HIGH when set to 1024. The PWM output will be LOW when set to 0. The buzzer circuit of this product will be muted if fixed to HIGH or LOW. In terms of power, HIGH means that no current flows through the buzzer circuit, so specify 1024 to mute. ledcWrite(BUZZER, 1024) will mute the sound.

About the global variable g_mode

The g_mode variable at the end of Figure 3-4 is a variable that records the number of times the switch is pressed. To know the number of times it is pressed, enter it into the LED function to light up the LED. There are four LEDs, and the values can be displayed in binary from 0 to 15, but since 0 is 0 even when the system is not started, we will set the specification to light up any of the values from 1 to 15. Therefore, the value of g_mode is limited to 1 to 15. Based on the specification, the initial value of g_mode is set to 1.

About the loop function

Next, we will explain the processing of the loop function. When

the blue switch (SW_R) is pressed, the value of g_mode is incremented. The upper limit is 15. When g_mode becomes 16 or higher, it is set to 15 (g_mode = 15 in Figure 3-5). Also, to let you know that it was pressed, a 2 [kHz] (INC_FREQ) sound is played for 30 [ms] (delay (30) in Figure 3-5). When the upper limit is reached, the sound is turned off. The ESP32 built-in function ledcWriteTone (argument 1, argument 2) is used as the function to play the sound. The ledcWriteTone

function specifies the IO number as the first argument. Specify the same IO number as the ledcAttach function. Specify the frequency as the second argument. When the red switch (SW_L) is pressed, the value of g_mode is decremented. The lower limit is 1. When g_mode becomes 0 or lower, it is set to 1 (g_mode = 1 in Figure 3-5). Also, to let

you know that it has been pressed, a 1[kHz] (DEC_FREQ) sound is played for 30[ms]. When the lower limit is reached, no sound is played. When the yellow switch (SW_C) is pressed, 2[kHz] and 1[kHz] are played for 80[ms] each to let you know that it has been pressed.

After the sound is played, there is a 300[ms] silence, and then the value of g_mode is passed to the execByMode function. Without this 300[ms], Do-Re-Mi would play immediately after the confirmation sound when the switch is pressed (2[kHz] and 1[kHz] are played for 80[ms each), making it difficult to distinguish the notes Do-Re-Mi. For this reason, a short silence is inserted so that Do-Re-Mi can be clearly distinguished.

About the execByMode function

Finally, we will explain the processing of the execByMode function.
This function will play one of the tones for 1 second. No sound will be produced if the value is between 4 and 15. The sound will be produced by entering the frequency of the tone in the second argument of the ledcWriteTone function.

Correspondence table between execByMode arguments and buzzer sounds

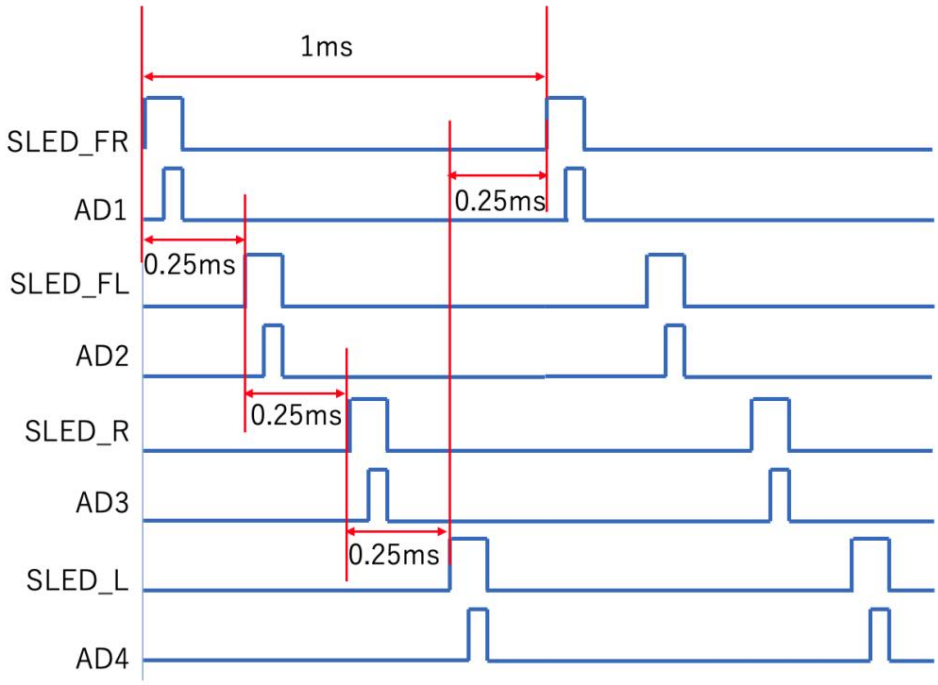
Arguments for execByMode	Buzzer sound
1	C (523[Hz])
2	Re(587[Hz])
3	Mi (659[Hz])
others	Silence

STEP 4: Display the sensor values on the serial monitor

overview

This is a sample program that outputs the wall sensor value via serial communication.
Use the monitor to view the sensor values.

The value changes depending on the sensor's light emission time and interval, so it is necessary to set the light emission to a constant interval.
The sensor emits light using a timer. It emits light in the order of front right, front left, right, and left at 0.25 ms intervals. It completes one cycle in 1 ms. The timing of the light emission and the A/D timing chart are shown below.



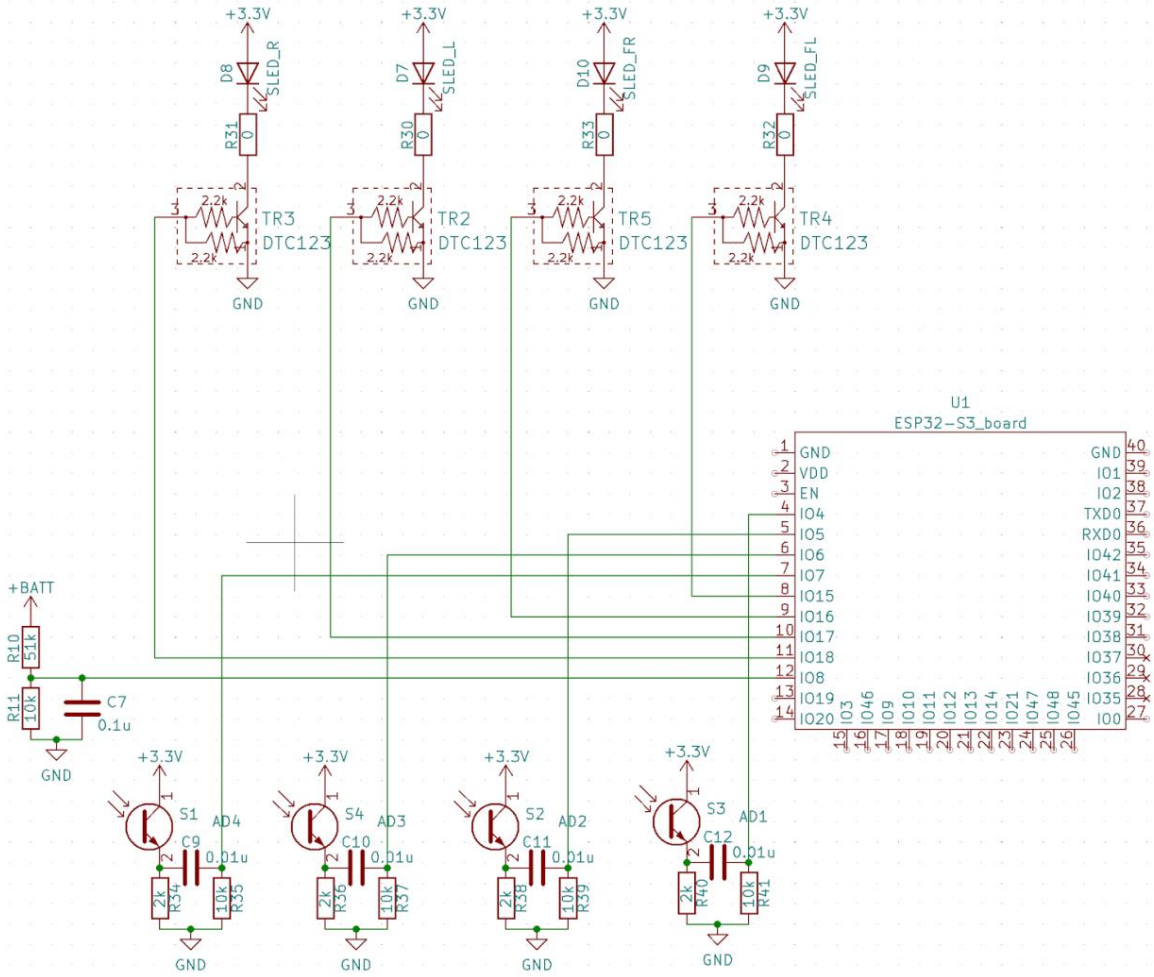
Timing chart of sensor light emission and ADC

The LEDs are turned on when the SLED_FR, SLED_FL, SLED_R, and SLED_L waveforms are high.
It's time to shine.

The periods when the AD1, AD2, AD3, and AD4 waveforms are high are the times when the ADC (analog-to-digital converter) is operating.

Circuit Diagram

The circuit diagram below shows the circuit elements related to the light emission and reception of the sensor.



Sensor light emission and light receiving circuit

Circuit operation explanation

The circuit with the SLED_R, SLED_L, SLED_FR, and SLED_FL LEDs is the light-emitting side, S1, S2, S3,
The circuit with S4 is the light receiving side.

The circuit operation will be explained from the light receiving side. The phototransistor S1 is an element that has the property of passing a current according to the intensity of the light it receives. The stronger the light, the larger the current. The output signal of the phototransistor is a continuous analog signal. The digital input of the microcontroller's GPIO recognizes it as HIGH when it is greater than the threshold and LOW when it is less than the threshold, so it is not possible to recognize the analog signal output from the phototransistor as an appropriate value. Therefore, an ADC is used to convert the analog signal to digital. The ADC converts the voltage input to the ADC port into a fine digital value according to the resolution.

The ESP32-S3 ADC returns a result of dividing 3.3V by 4095. The ADC can measure voltage, but cannot measure current. Therefore, the current from the phototransistor is converted to voltage by resistor R34. When current flows through a resistor, a voltage is generated according to Ohm's law V=IR.

The combination of the C9 capacitor and the R35 resistor is a high-pass filter. This cuts low-frequency signals such as those from direct sunlight.

In order to cut low frequency signals in the circuit, a circuit that changes the light of the sensor is required on the light emitting side. This circuit is the transistor TR3. ADC is performed only when the sensor emits light.

program

The program for STEP 4 is shown in the figure.

Figure 4-1

```
C/C++
#define SLED_FR 16
#define SLED_FL 15
#define SLED_R 18
#define SLED_L 17

#define AD4 7
#define AD3 6
#define AD2 5
#define AD1 4
#define AD0 8

volatile short g_sensor_value_fr; volatile
short g_sensor_value_fl; volatile short
g_sensor_value_r; volatile short
g_sensor_value_l; volatile short
g_battery_value;

hw_timer_t * g_timer1 = NULL;
portMUX_TYPE g_timer_mux = portMUX_INITIALIZER_UNLOCKED;
```

Figure 4-2

```
C/C++
void IRAM_ATTR onTimer1(void){ static char cnt =
  0;
  portENTER_CRITICAL_ISR(&g_timer_mux); switch (cnt) { case
  0:

      digitalWrite(SLED_FR, HIGH); //LEDÿÿ for (int i = 0; i
      < 300; i++) { asm("nop \n");


      } g_sensor_value_fr = analogRead(AD1);
      digitalWrite(SLED_FR, LOW); //LEDÿÿ break;

    case 1:
      digitalWrite(SLED_FL, HIGH); //LEDÿÿ for (int i = 0; i
      < 300; i++) { asm("nop \n");


      } g_sensor_value_fl = analogRead(AD2);
      digitalWrite(SLED_FL, LOW); //LEDÿÿ break;

    case 2:
      digitalWrite(SLED_R, HIGH); //LEDÿÿ for (int i = 0; i
      < 300; i++) { asm("nop \n");


      } g_sensor_value_r = analogRead(AD3);
      digitalWrite(SLED_R, LOW); //LEDÿÿ break;

    case 3:
      digitalWrite(SLED_L, HIGH); //LEDÿÿ for (int i = 0; i
      < 300; i++) { asm("nop \n");


      } g_sensor_value_l = analogRead(AD4);
      digitalWrite(SLED_L, LOW); //LEDÿÿ g_battery_value
      = (double)analogReadMilliVolts(AD0) / 10.0 * (10.0 + 51.0); break;
```

```
    default:
        Serial.printf("Error\n\r"); break;

  }
  cnt++; if
  (cnt == 4) cnt = 0;
  portEXIT_CRITICAL_ISR(&g_timer_mux); }
```

Figure 4-3

C/C++

```
void setup(){
  // put your setup code here, to run once:
  //Sensor ÿÿoff
  pinMode(SLED_FR, OUTPUT);
  pinMode(SLED_FL, OUTPUT);
  pinMode(SLED_R, OUTPUT);
  pinMode(SLED_L, OUTPUT);
  digitalWrite(SLED_FR, LOW);
  digitalWrite(SLED_FL, LOW);
  digitalWrite(SLED_R, LOW);
  digitalWrite(SLED_L, LOW);

  Serial.begin(115200);

  g_timer1 = timerBegin(1000000); //1MHz(1us)
  timerAttachInterrupt(g_timer1, &onTimer1); timerAlarm(g_timer1,
  250, true); //250*1us=250us(4kHz) timerStart(g_timer1); }
```

Figure 4-4

```
C/C++
void loop(){ // put
  your main code here, to run repeatedly: Serial.printf("r_sen is
  %d\n\r", g_sensor_value_r); Serial.printf("fr_sen is %d\n\r",
  g_sensor_value_fr); Serial.printf("fl_sen is %d\n\r", g_sensor_value_fl);
  Serial.printf("l_sen is %d\n\r", g_sensor_value_l); Serial.printf("VDD is
  %d\n\r", g_battery_value); delay(100); }
```

Commentary

Here, we use a timer to get the sensor values at the same interval. The timer on the ESP32-S3 is 0
The timers are numbered from 1 to 3. If you use the built-in functions of ESP32, they will be assigned to an available timer, so the user does not need to think about which timer to use and set.

The timer counts up or down in sync with the clock in the timer.
When the set target value is met, an interrupt occurs and a specified function is executed.

About timerBegin function

The timer is configured from timerBegin in the second half of Figure 4-3.

The ESP32 built-in function timerBegin inputs the frequency that will be the timer resolution. The timerBegin function is designed to use an available timer, so the user does not need to consider the timer number. Here, 1000000 is specified, so it will count up at 1 clock intervals of 1 us. The return value of the timerBegin function is a pointer to the timer to be used (address of the timer function). This pointer will be used later in setting up the timer, so it is placed in the variable g_timer1 declared in Figure 4-1.

About timerAttatchInterrupt function

The ESP32 built-in function timerAttatchInterrupt specifies the function to run when an interrupt occurs. The first

argument specifies the pointer to the timer saved in the variable earlier. The second

argument specifies the pointer to the function to run when an interrupt occurs. The onTimer1 function (Figure 4-2) that runs when an interrupt occurs is written in the sketch that contains the setup function. When writing interrupt processing in a separate sketch, you need to either call a function written in that separate sketch from the interrupt function, or move the functions and variables related to timer initialization to a separate sketch. An example of this is shown in STEP 8.

About the timerAlarm function

The ESP32 built-in function timerAlarm sets the interrupt timing. The first argument specifies

the pointer to the timer saved in the variable earlier. The value set in the second

argument is the value that will cause an interrupt when it matches the counted-up value.

In the timerBegin setting, the count-up clock of Timer 1 is 1 [MHz], which is 1 [us] in seconds. If you set the second argument to 500, an interrupt will occur after 500 us.

The third argument is whether to auto-reload or not. If set to true, auto-reloading will occur and the period If set to false, the interrupt will occur only once.

The fourth argument is the initial value for auto-reload.

About the timerStart function

When you execute the ESP32 built-in function timerStart, the timer you set will start counting. The first argument specifies the pointer to the timer you saved in the variable earlier.

About the loop function

In the loop function, the sensor values obtained within the interrupt are output from the USB port.
Since the ESP32-S3 supports USB CDC (Communications Device Class), you can directly connect the ESP32-S3 to a PC for serial communication. To initialize serial communication, use the built-in Arduino IDE function Serial.begin. The first argument is set to the communication speed [bps], but this argument is ignored when communicating via USB CDC. However, if no argument is set, a build error will occur. Therefore, set the argument to 115200 [bps] to match the communication speed of the serial monitor in Arduino IDE.

The built-in function Serial.printf in Arduino IDE has the same function as printf in C. The serial monitor is a simplified terminal unlike a general terminal, so some escape sequences cannot be used.

About the onTimer1 function

We will explain the onTimer1 function that is executed by the timer 1 interrupt. The onTimer1 function is preceded by IRAM_ATTR. This is an attribute that copies the code to be executed to IRAM (Instrunction RAM) to speed up the processing time within the interrupt.

portENTER_CRITICAL_ISR(&g_timer_mux) is a command that prohibits other interrupts.

portEXIT_CRITICAL_ISR(&g_timer_mux) cancels the prohibition of other interrupts. The switch statement in Figure 4-2 switches

the combination of LEDs and ADC that are lit each time an interrupt is received. It is possible to acquire the values of four sensors with one interrupt, but illuminating all four LEDs at the same time will affect the adjacent sensors, so the time is shifted and the values are acquired separately. Here, they are illuminated one by one, but if the sensor combination

does not affect each other, there is no problem with ADC at the same interrupt timing. For example, it is possible to acquire sensor values within the same interrupt by dividing them into the front sensor and the side sensor. However, since the main processing stops during an interrupt and the motor interrupt also stops, it is generally considered to make the processing during an interrupt as light as possible. Although there are four sensors, the processing content is the same, so the left

side sensor in case 3 is used to operate the sensor.
I will explain the work.

The LED is illuminated with digitalWrite(SLED_L, HIGH). Then, the following for statement is used to check the rising edge of the light receiving element.
Wait for the time to pass.

```
C/C++
for (int i = 0; i < 10; i++) { asm("nop \n");

}
```

As explained in the circuit operation explanation, the resistor on the emitter side of the phototransistor converts current to voltage. If this resistance is large, the response of the signal from the light receiving element will be slower, but the amplitude will be larger. Conversely, if the resistance is small, the response of the signal from the light receiving element will be faster, but the amplitude will be smaller.

The resistance is increased to a certain extent because it is easier to judge whether the sensor is near or far from the wall when the amplitude is larger. For this reason, a for statement is inserted to wait for a delay in the response from the light receiving element. Also, if no processing is performed within the for statement, the for statement itself will be deleted during optimization at compile time. Therefore, the code asm("nop \n") is required, which does nothing but consumes clocks.

About the analogRead function

After waiting for the delay of the light receiving element signal, the analog input is read using the built-in analogRead function of the Arduino IDE. Converts the signal to digital. By default, the resolution is 12 bits and the attenuator is -11db. Also, in the analogRead function, the ADC port settings are executed for the IO number specified in the first argument. Therefore, there is no need to configure the ADC in the setup function. The A/D converted value is put into the variable g_sensor_value_l and output to USB by Serial.printf executed in the loop function.

About volatile

g_sensor_value_g is declared as a short type in Figure 4-1, but there is a volatile modifier in front of it. This is a feature that completely suppresses code optimization by the compiler. In particular, when using a variable that is common to both interrupts and main, the compiler optimizes the process using data in the cache, since it is not known when the value was updated in main. This causes the value to remain unchanged for a long time.

When you declare a variable with volatile, even if the value of the variable is in the cache, the value will be obtained from the variable in external memory. However, volatile always calls the value from the variable in external memory, which slows down the execution speed. Therefore, use it only on necessary variables.

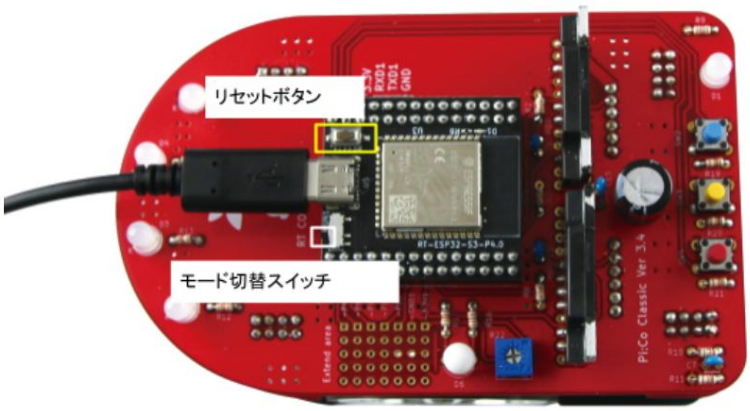
About the analogReadMilliVolts function

The analogReadMilliVolts function measures the battery voltage.
The upper limit is 3.3V, so the battery voltage is divided by 51[kÿ] and 10[kÿ] and ADC is used. When using the analogReadMilliVolts function, the return value is in units of [mv], so there is no need to worry about the ADC resolution or power supply voltage. The battery voltage is calculated from the output of this function and the value of the voltage divider resistor. In the sample, it is "/10.0*(10.0+51.0)", but if it is the opposite, "(10.0+51.0)/10.0", the value will overflow, so it is written in this order. The return type of analogReadMilliVolts is short. In order to calculate

the voltage as precisely as possible, a decimal point is used for calculation. The hardware supports float type in decimal point calculations of the ESP32-S3. However, float type calculations cannot be performed within the ESP32-S3 interrupt function. If you want to perform decimal point calculations within an interrupt, use double type. Since double performs decimal point arithmetic in software, execution speed is slow. Use of double in interrupts should be written to a minimum.

Display of values

To use the serial monitor in the Arduino IDE, click the icon in the upper right. Clicking this will add the serial monitor tag shown below. Transfer the program to this product, switch the mode switch to RUN, and press the reset button. Click the triangle to the right of the board name under the menu bar, and select the connected USB from "Select Other Board and Port".

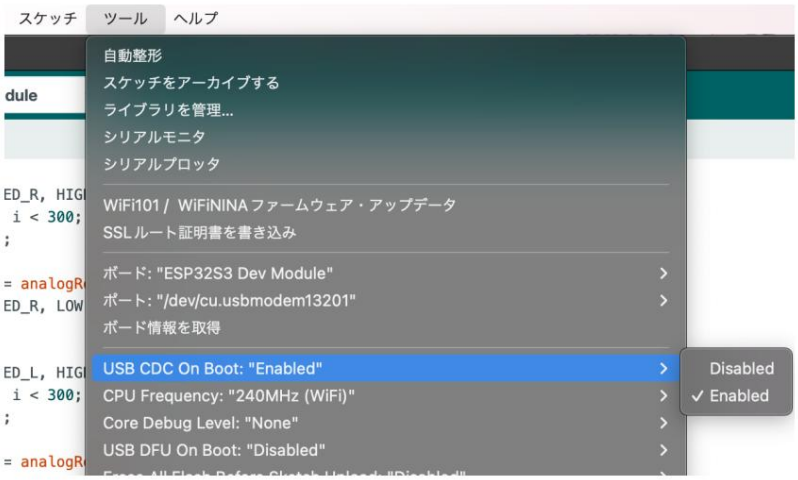


Serial monitor tag added

Board and port selected

The serial monitor tag displays the sensor value and voltage [mv]. The value changes when you hold your

hand over the sensor. If the sensor value is not displayed, the USB CDC may not be enabled. If it is not enabled, enable it and write the program again to check.



Enable USB CDC

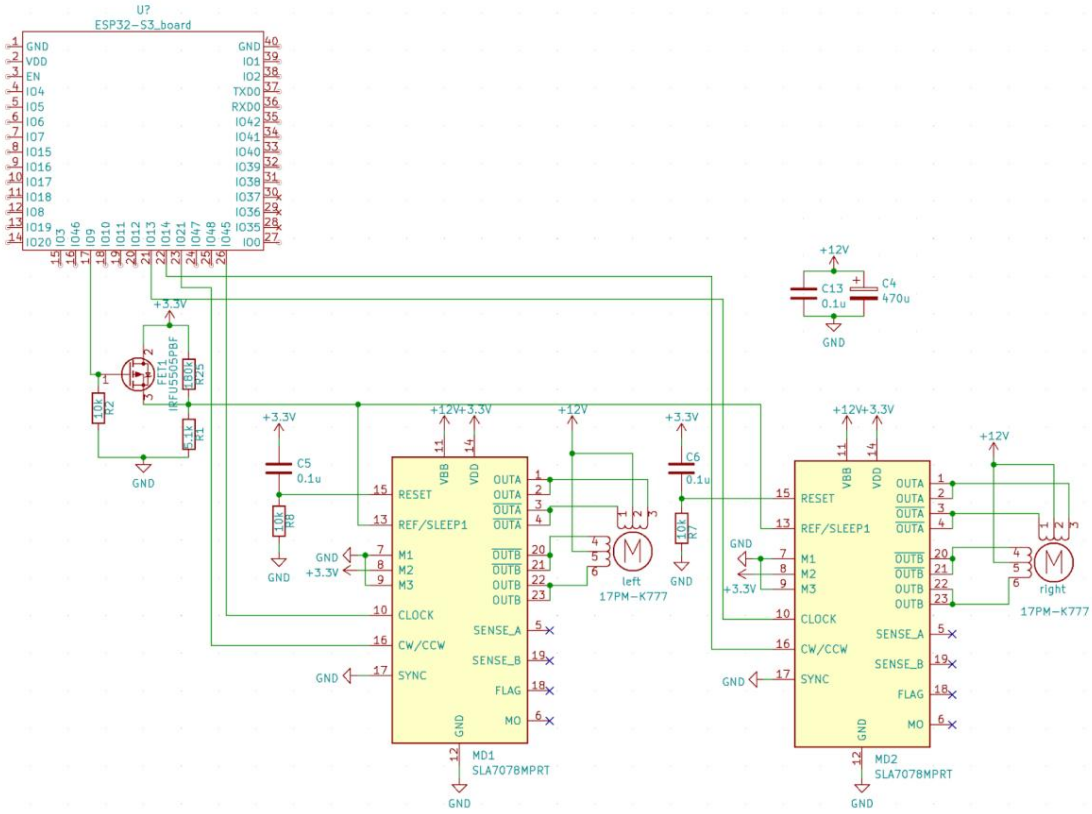
STEP 5 : Use the motor to move in a straight line

overview

A sample program that drives a vehicle in a straight line by applying the same frequency signal to both the left and right stepping motors When you press the blue, yellow, or red switch, the robot will travel a certain distance while accelerating and decelerating. The travel functions available are the accelerate function, which accelerates while traveling a specified distance, the oneStep function, which travels a specified distance at a constant speed, and the decelerate function, which decelerates while traveling a specified distance. These functions are managed in separate files so that they can be easily reused.

Circuit Diagram

The circuit diagram showing the circuit elements related to driving is shown below.



Motor Drive Circuit

Circuit operation explanation

The motor used in this product is a stepping motor. Stepping motors are also called pulse motors.
This motor rotates at a fixed angle in sync with the pulse. The motor driver (SLA7078) used in this product is set to 1-2 phase excitation, and is configured to rotate 0.9 degrees per pulse. The motor will rotate when a pulse is applied to the CLOCK pin of the motor driver.

The rotation direction is determined by the CW/CCW signal. Giving LOW to both the left and right will result in forward movement.

The current flowing through the motor is set by the voltage applied to REF/SLEEP1. In the above circuit, the combination of R25 (180kÿ) and R1 (5.1kÿ) sets the upper limit of the current flowing through the motor to approximately 600mA.
When the FET in parallel with R25 is turned ON, the voltage applied to REF/SLEEP1 becomes 3.3V and the motor driver goes into sleep mode.

program

The program for STEP 5 is shown in the diagram.

uROS_STEP5_Straight.inoFigure

5-1

```
C/C++
#define LED0 1
#define LED1 2
#define LED2 42
#define LED3 41
#define SW_L 10
#define SW_C 11
#define SW_R 12
#define MOTOR_EN 9
#define CW_R 14
#define CW_L 21
#define PWM_R 13
#define PWM_L 45

#define MIN_HZ 80
#define TIRE_DIAMETER (48.00) #define
PULSE (TIRE_DIAMETER * PI / 400.0) #define MIN_SPEED
(MIN_HZ * PULSE)

hw_timer_t * g_timer0 = NULL; hw_timer_t
* g_timer2 = NULL; hw_timer_t * g_timer3
= NULL;

portMUX_TYPE g_timer_mux = portMUX_INITIALIZER_UNLOCKED;

unsigned short g_step_hz_r = MIN_HZ; unsigned
short g_step_hz_l = MIN_HZ;

volatile unsigned int g_step_r, g_step_l; double g_max_speed;
double g_min_speed;
double g_accel = 0.0;
volatile double g_speed =
MIN_SPEED;

volatile bool g_motor_move = 0;
```

Figure 5-2

```
C/C++
void IRAM_ATTR onTimer0(void)
  { portENTER_CRITICAL_ISR(&g_timer_mux); // Disable interrupts controlInterrupt();

  portEXIT_CRITICAL_ISR(&g_timer_mux); // Enable interrupts }
```

Figure 5-3

```
C/C++
void IRAM_ATTR isrR(void)
  { portENTER_CRITICAL_ISR(&g_timer_mux); // Disable interrupt if (g_motor_move) {

      timerAlarmWrite(g_timer2, 2000000 / g_step_hz_r, true, 0); digitalWrite(PWM_R, HIGH); for (int i = 0;
      i < 100; i++) { asm("nop \n");



      } digitalWrite(PWM_R, LOW); g_step_r++;


  } portEXIT_CRITICAL_ISR(&g_timer_mux); // Enable interrupt }
```

Figure 5-4

```
C/C++
void IRAM_ATTR isrL(void)
  { portENTER_CRITICAL_ISR(&g_timer_mux); // Disable interrupt if (g_motor_move) {

      timerAlarmWrite(g_timer3, 2000000 / g_step_hz_l, true, 0); digitalWrite(PWM_L, HIGH); for (int i = 0;
      i < 100; i++) { asm("nop \n"); };
      digitalWrite(PWM_L, LOW); g_step_l++;



  }
```

```
portEXIT_CRITICAL_ISR(&g_timer_mux); // Enable interrupt }
```

Figure 5-5

```
C/C++
void setup(){
  // put your setup code here, to run once: pinMode(LED0,
  OUTPUT); pinMode(LED1,
  OUTPUT); pinMode(LED2,
  OUTPUT); pinMode(LED3,
  OUTPUT);

  pinMode(SW_L, INPUT);
  pinMode(SW_C, INPUT);
  pinMode(SW_R, INPUT);

  //motor disable
  pinMode(MOTOR_EN, OUTPUT);
  pinMode(CW_R, OUTPUT);
  pinMode(CW_L, OUTPUT);
  pinMode(PWM_R, OUTPUT);
  pinMode(PWM_L, OUTPUT);

  digitalWrite(MOTOR_EN, LOW);
  digitalWrite(CW_R, LOW);
  digitalWrite(CW_L, LOW);
  digitalWrite(PWM_R, LOW);
  digitalWrite(PWM_L, LOW);

  g_timer0 = timerBegin(1000000); //1MHz(1us)
  timerAttachInterrupt(g_timer0, &onTimer0); timerAlarm(g_timer0,
  1000, true, 0); //1000 * 0.1us =1000us(1kHz) timerStart(g_timer0);



  g_timer2 = timerBegin(2000000); //2MHz(0.5us)
  timerAttachInterrupt(g_timer2, isrR);
```

```
  timerAlarm(g_timer2, 13333, true, 0); //13333 * 0.5us = 6666us(150Hz)
timerStart(g_timer2);


  g_timer3 = timerBegin(2000000); //2MHz(0.5us)
  timerAttachInterrupt(g_timer3, &isrL);
  timerAlarm(g_timer3, 13333, true, 0); //13333 * 0.5us = 6666us(150Hz)
timerStart(g_timer3); }
```

Figure 5-6

```
C/C++
void loop() {

    // put your main code here, to run repeatedly: while
    (digitalRead(SW_L) & digitalRead(SW_C) &
digitalRead(SW_R))
        { continue;

    } digitalWrite(MOTOR_EN, HIGH);
    delay(1000);
    digitalWrite(LED0, HIGH);
    accelerate(90, 350);
    digitalWrite(LED1, HIGH);
    digitalWrite(LED2, HIGH);
    oneStep(180, 350);
    digitalWrite(LED3, HIGH);
    decelerate(90, 350);
    digitalWrite(LED0, LOW);
    digitalWrite(LED1, LOW);
    digitalWrite(LED2, LOW);
    digitalWrite(LED3, LOW);
    delay(1000);
    digitalWrite(MOTOR_EN, LOW);
}
```

interrupt.inoFigure 5-7

```C/C++
void controlInterrupt(void){
  g_speed += g_accel;

  if (g_speed > g_max_speed) { g_speed =
      g_max_speed;

  } if (g_speed < g_min_speed) { g_speed =
      g_min_speed;

  } g_step_hz_l = g_step_hz_r = (unsigned short)(g_speed /
PULSE); }
```

run.ino

Figure 5-8

```
C/C++

void accelerate(int len, int tar_speed){ int obj_step; g_max_speed =
  tar_speed; g_accel =
  1.5; g_step_r = g_step_l = 0; g_speed =
  g_min_speed =
  MIN_SPEED; g_step_hz_r = g_step_hz_l
  = (unsigned short)(g_speed / PULSE); obj_step = (int)
  ((float)len * 2.0 / PULSE); digitalWrite(CW_R, LOW); digitalWrite(CW_L, LOW); g_motor_move
= 1; while
  ((g_step_r + g_step_l) < obj_step) { continue;









}}
```

Figure 5-9

```
C/C++

void oneStep(int len, int tar_speed){
  int obj_step;
  g_max_speed = tar_speed; g_accel =
  0.0; g_step_r = g_step_l
  = 0; g_speed = g_min_speed = tar_speed;
  g_step_hz_r = g_step_hz_l = (unsigned short)(g_speed /
  PULSE); obj_step = (int)((float)len * 2.0 / PULSE); digitalWrite(CW_R, LOW); digitalWrite(CW_L,
LOW);
  while ((g_step_r + g_step_l) < obj_step) { continue;









}}
```

Figure 5-10

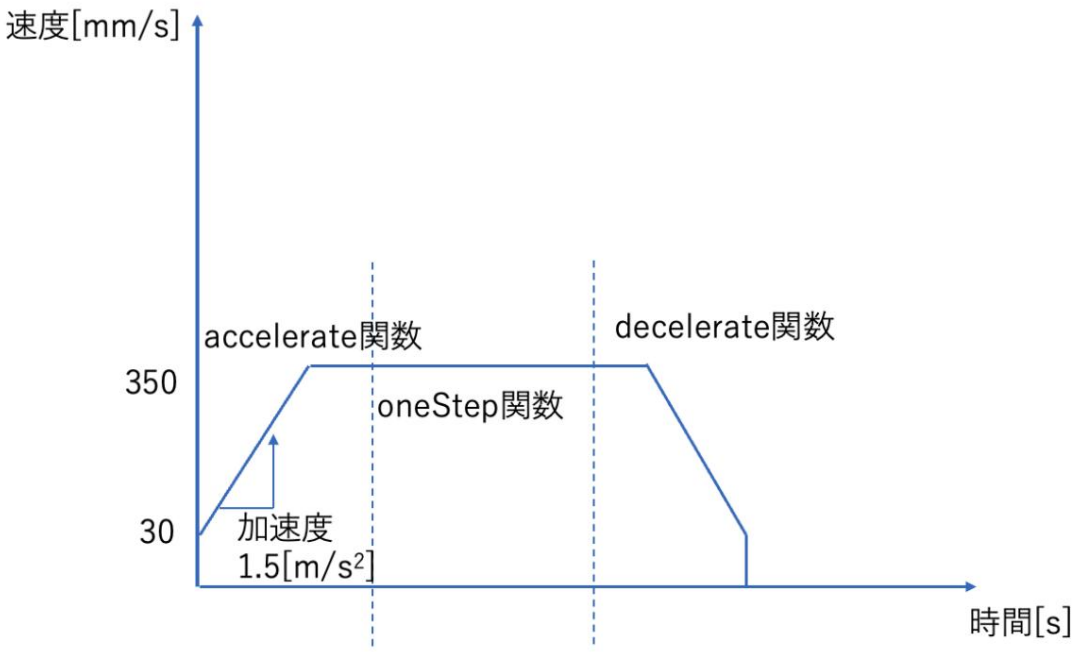
```
C/C++
void decelerate(int len, int tar_speed){ int obj_step;
  g_max_speed =
  tar_speed; g_accel = 0.0; g_step_r
  = g_step_l = 0;
  g_speed = g_min_speed =
  tar_speed; g_step_hz_r = g_step_hz_l =
  (unsigned short)(g_speed / PULSE); obj_step = (int)((float)len * 2.0 / PULSE);

digitalWrite(CW_R, LOW); digitalWrite(CW_L, LOW);



  while ((len - (g_step_r + g_step_l) / 2.0 * PULSE) >
                (((g_speed * g_speed) - (MIN_SPEED * MIN_SPEED)) /
(2.0 * 1000.0 * 1.5))) { continue;


  } g_accel = -1.5;
  g_min_speed = MIN_SPEED;

  while ((g_step_r + g_step_l) < obj_step) { continue;

  }

  g_motor_move = 0; }
```

Commentary

Before explaining the program, we will explain the characteristics of the motor used. With the combination of this product's motor and motor driver settings, it rotates 0.9 degrees per pulse. It is common to use the PWM function to generate pulses for motor control. It is also possible to calculate the distance traveled by counting the generated pulses. However, the Arduino IDE environment does not have a callback function for interrupts on the falling or rising edges of PWM. Therefore, instead of PWM, pulses are generated by timer interrupts to rotate the motor.

Unlike DC motors, stepping motors rotate in sync with pulses. If the torque is insufficient and the pulses and motor shaft become out of sync, a phenomenon known as step-out occurs and the motor stops rotating. Since motors have greater torque in the low-speed range, the interval between pulses is gradually shortened to reach the target speed.

If you want to set the target speed to 350 [mm/s], step-out will occur if you suddenly set the pulse interval to 350 [mm/s], so the initial speed starts at about 30 [mm/s]. The general speed control method for stepping motors is to then accelerate to 350 [mm/s].



Graph of stepping motor speed change

About timers

To accelerate or decelerate regularly, use a timer interrupt at regular intervals to calculate the speed.
The ESP32-S3 has four timers.

Timer 0 is used for speed calculation, Timer 2 for pulse generation for the right motor, and Timer 3 for pulse generation for the left motor.

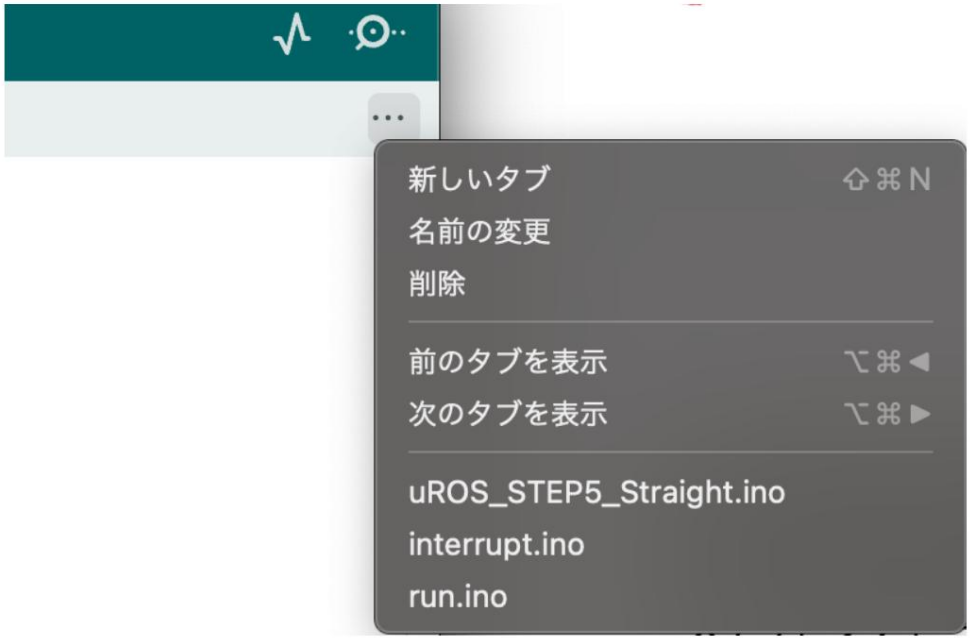
About splitting files

From STEP5 onwards, the program files are divided into multiple files. Dividing a program into files by function or hierarchy improves the readability and reusability of the program. In this sample, files are divided by function.

uROS_STEP5_Straight.ino handles motor interrupts, initial settings, and calling the straight-line function.

interrupt.ino handles timer 1 interrupts. run.ino contains the functions necessary for driving.

If you want to create a new file, click on the right side of the Arduino IDE.  Icon When you click, the following menu will appear.



Creating a new tab

Clicking "New Tab" will display the following screen. Enter the name of the file to be added and click "OK" to add the file. If you do not write an extension, the Arduino extension ".ino" will be set. You can also use ".c" or ".h" as the extension. (Do not name the file in Japanese. Compilation may not pass depending on the environment.)

Screen for entering a new file name

About the setup function

First, let's explain the program in uROS_STEP5_Straight.ino. The setup function sets the LED and switch GPIO, timer 0 for speed calculation, timer 2 for the right motor interrupt, and timer 3 for the left motor interrupt. The timer settings are the same as in STEP4, so we will omit the explanation here. The motor driver is controlled by the CW_R, CW_L, PWM_R, PWM_L, and MOTOR_EN signals.

CW_R and CW_L set the motor rotation direction.

Giving one pulse to PWM_R and PWM_L will rotate 0.9 degrees.

MOTOR_EN is a switch that passes current through the motor. This MOTOR_EN is used when passing current through the motor coil (called excitation), a characteristic of stepping motors. The power consumption of a stepping motor is highest when it is excited and stopped. When there is no need to rotate the motor, excitation is released to reduce power consumption. This control is
Use MOTOR_EN. When MOTOR_EN is LOW, excitation is OFF, and when it is HIGH, excitation is ON.

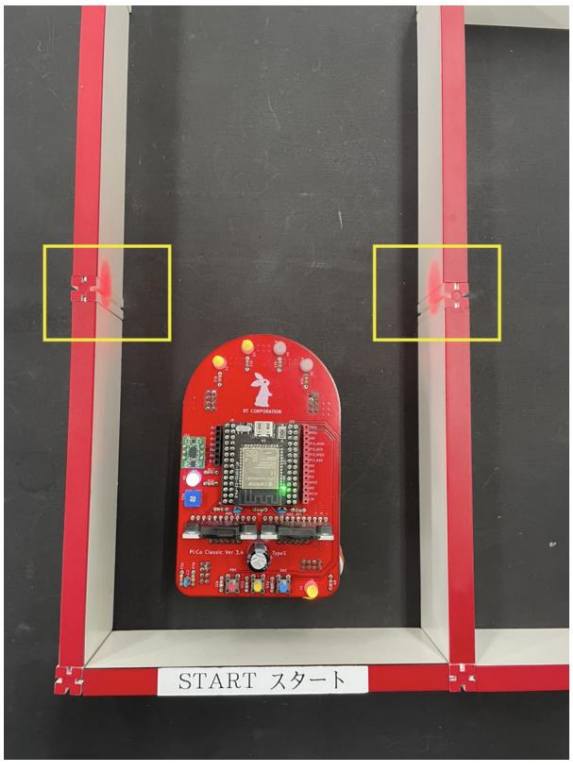
About the loop function

The loop function implements a process to start moving when a switch is pressed so that the robot does not suddenly move straight after power is turned on. When any switch is pressed, the robot accelerates to the search speed for a distance of 90 mm, moves straight at the search speed for a distance of 180 mm, and then moves 90 mm and stops.

These distances of 90mm and 180mm correspond to the maze sections in the Micromouse Classic competition. If you simply wanted to run within the distance of the section, you would run in multiples of 180mm, but the Micromouse Classic creates a map while determining whether there are walls in the section. The presence or absence of walls is determined by a sensor located at the front of the product.

When this product is in the center of a section, the sections that the sensor can detect are the section one block ahead and the center of the current section. If a wall is detected from this position, it cannot determine whether it is the wall of the current section or the wall one block ahead.

Therefore, in order to accurately determine whether or not there is a wall in each section, the presence or absence of a wall is judged after 90 mm. Since one section is 180 mm, when stopping, it moves forward 90 mm and stops in the center of the section. Due to the layout of the sensor in this product, the operation is checked using a function that moves straight ahead 90 mm and 180 mm, just like in the search.



A position that can be confirmed by the time sensor in the center of the section

Left and right motor interrupt functions

The functions that process the interrupts for the left and right motors are the isrR and isrL functions. Within each function, the PWM_R and PWM_L ports are turned ON/OFF to output a pulse. The pulse frequency is changed by adjusting the interrupt period with the timerAlarm function. Since both the left and right do the same thing, we will explain it using the isrR function.

If the timer interrupt is stopped, the isrR and isrL functions will not be executed, and the motor will stop. However, since the ESP32 in the Arduino IDE environment runs on RTOS (real-time OS), The time between executing the ESP32 built-in function timerStop and the timer stopping varies. Even if you write the following consecutively, the left and right motors will not stop at the same time.

```
C/C++
timerStop(g_timer2);
timerStop(g_timer3);
```

Therefore, when the variable g_motor_move in Figure 5-3 is false, pulse generation stops and the motor is stopped. A pulse is created with digitalWrite(PWM_R, HIGH) and digitalWrite(PWM_R, LOW). The for statement in between adjusts the pulse width. This ensures that a pulse with a minimum width of 2us (maximum frequency 250kHz, duty 50%) that can be recognized by the motor driver being used is output. The variable g_step_r is incremented each time a pulse is output. This pulse number is used by driving control functions such as accelerate and oneStep.

Next, we will explain the

program in interrupt.ino. The controlInterrupt function is the interrupt for timer 0 .
This function is called internally and integrates the acceleration g_accel to find the speed g_speed.

The relationship between velocity and acceleration is

$$= +$$
$$_0$$

0 is the initial speed, and is the time difference between the starting time and the current time. In the timer 0 interrupt, the acceleration is added to the current speed to calculate the next speed +1.

$$= ÿ +$$
$$_{+1}$$

In timer 0, an interrupt occurs every 1 ms,
so ÿ is 0.001. However, in the sample program, g_speed += g_accel, and the ÿ of 0.001 is not used. This is because the calculation of 0.001 is omitted by changing the units of speed and acceleration.

If you multiply the acceleration [m/ s2] by ÿ of 0.001 [s], the unit of speed is [mm/s]. If you leave the unit of speed in [mm/s], the acceleration will be multiplied by ÿ, so the formula will be g_speed += g_accel. In other words, the unit of speed is [mm/s] and the unit of acceleration is [m/s2] .

Speed Limits

if (g_speed > g_max_speed) and if (g_speed < g_min_speed) limit the speed to be within the upper and lower limits. The upper limit is particularly important. When the accelerate function is being executed, the acceleration is fixed at 1.5 [m/s2] and the speed continues to increase. If the initial speed is 30 [mm/s], the speed after moving 90 mm with an acceleration of 1.5 [m/s2] is as follows:

$$^2 \quad ÿ \text{ Initial velocity} \quad ^2$$

This can be calculated using the formula distance =
(velocity)/(2 * acceleration). Transforming this into the equation for velocity gives us:

$$\text{speed}^2 = (2 * \text{acceleration}) * \text{distance} + \text{initial} \quad ^2$$

velocity.

$$^2 = 2 * 1.5 * 0.09 + 0.03 \quad ^2 = 0.2709$$

$$\text{Speed Speed} = 0.52 [\ /\ ] = 520 [\ /\ ]$$

This exceeds the upper speed limit of 350 [mm/s]. As can be seen in the graph of the speed change of the stepping motor, after reaching 350 [mm/s] within the accelerate function, the limit function is used to control the speed so that the motor continues to run at 350 [mm/s].

Instructions for stepping motors

g_speed / PULSE converts speed to frequency [Hz]. The unit of rotation speed of a stepping motor is pps [pps: pulse per second]. pps is the same as frequency [Hz]. To convert from speed to frequency, use the following formula:
Frequency = Speed / = 48 * ÿ/400

48 is the tire diameter [mm]. 400 is the number of pulses required for one rotation of the motor. Since the unit of speed is [mm/s] and the unit of PULSE is [mm], the unit of mm is canceled out and the unit of frequency becomes [Hz].

About run.ino

Finally, we will explain the program in run.ino. The accelerate, oneStep, and decelerate functions set the parameters for the initial speed, acceleration, distance traveled, maximum speed, and minimum speed required for going straight.

About the accelerate function

The first argument of the accelerate function is the distance to travel, and the second argument is the maximum speed. The number of pulses required to travel the specified distance is calculated within the accelerate function. Since the number of pulses required for one rotation of a stepping motor is fixed, the distance traveled per pulse can be calculated. In the case of this product, the number of pulses required for one rotation is 400, and the tire diameter is 48 mm. The distance traveled per pulse can be calculated from these parameters as follows:

48 * ÿ/400 = 0.377 [ ]

In this sample, the first argument of the accelerate function is 90, so if you count 90/0.377 = 238.7 pulses, it will travel 90 mm. The required pulses are calculated as obj_step = (int)((float)len * 2.0 / PULSE). The travel distance is doubled to compare with the combined number of travel pulses on the left and right. The travel pulse numbers g_step_r and g_step_l are the counts when

the pulses are output by timers 2 and 3.
It will be updated.

The while ((g_step_r + g_step_l) < obj_step) compares the number of travel pulses with the number of required pulses. Here, the sum of g_step_r and g_step_l is compared. This makes it possible to reduce the travel error compared to finding the distance using only the number of travel pulses on one side. In particular, there is a difference in the number of travel pulses on the left and right during posture control.

g_max_speed sets the upper limit of the speed. In this sample, it is 350 [mm/s]. g_accel sets the acceleration used in the controlInterrupt function. In this sample, it is 1.5 [m/s2] . g_speed sets the lower limit speed

(MIN_SPEED) as the initial speed. Because a timer is used to generate pulses, the lower limit speed cannot be close to 0 [mm/s]. Therefore, it is set to a speed that can be set by the timer count value. The ESP32-S3 timer counter is 54 bits, but the lower limit speed is set to 30 [mm/s] so that it can be set using the 16-bit counter value commonly used in general-purpose microcontrollers. Enter the lower limit speed in g_min_speed as well.

g_step_hz_r = g_step_hz_l = (unsigned short)(g_speed / PULSE), Timer 2 and Timer
Sets the stepper motor speed used in interrupt 3.

digitalWrite(CW_R, LOW); and digitalWrite(CW_L, LOW) determine the direction in which the motor rotates.
The circuit is designed so that when it is set to LOW, it moves forward.

g_motor_move = 1 enables timer 2 and timer 3 to output pulses.

About the oneStep function:

The first argument of the oneStep function is the distance traveled, and the second argument is the initial speed. This function travels a specified distance at a constant speed without accelerating or decelerating. Therefore, the acceleration g_accel is 0. The oneStep function is called after accelerating to 350 [mm/s] with the accelerate function, so the same speed is entered into the variables of the current speed g_speed, the upper limit speed g_max_speed, and the lower limit speed g_min_speed.

About the decelerate function

The first argument of the decelerate function is the distance traveled, and the second argument is the initial speed. Called after executing a function or oneStep function.

When the decelerate function is called, it will run at the initial speed.

g_max_speed, g_accel, g_speed, and g_min_speed are initialized to the initial speed, just like oneStep. The timing to start

deceleration is adjusted so that the running speed becomes MIN_SPEED as soon as the specified distance is reached. The first while statement determines when to start decelerating. The distance before deceleration begins is determined by the formula that appeared in the interrupt.ino explanation:

$$\text{distance} = (\text{speed}^2 - \text{Initial velocity}^2)/(2 * \text{acceleration}).$$

When it is time to decelerate, change the acceleration from 0 to -1.5 [m/s2] . Also, change the lower limit speed to g_min_speed=MIN_SPEED. Set motor_move=0 to disable pulse output once the specified distance has been traveled.

STEP 6 Rotate using the motor

overview

This is a sample program that turns 90 degrees right, 90 degrees left, 180 degrees right, and 180 degrees left when you press the blue, yellow, or red switch. The 90 and 180 degree turns are used to change direction in Micromouse competitions.

Circuit Diagram

This is the same circuit diagram as STEP 5, so please refer to STEP 5.

program

The following shows the changes made to STEP 5. interrupt.ino is the same as STEP 5, so it is omitted here.

uROS_STEP6_rotate.ino

Figure 6-1

```
C/C++
#define MIN_HZ 80 #define
TIRE_DIAMETER (48.00) #define PULSE
(TIRE_DIAMETER * PI / 400.0) #define MIN_SPEED (MIN_HZ *
PULSE) #define TREAD_WIDTH (65.00)



typedef enum { front,
  right,

  rear,
  left,
  unknown, }
t_local_dir;
```

Figure 6-2

```
C/C++
void loop(){ // put
  your main code here, to run repeatedly: while (digitalRead(SW_L)
  & digitalRead(SW_C) & digitalRead(SW_R)) { continue;



  } digitalWrite(MOTOR_EN, HIGH);
  delay(1000);
  rotate(right, 1);
  delay(1000);
  rotate(left, 1);
  delay(1000);
  rotate(right, 2);
  delay(1000);
  rotate(left, 2);
  delay(1000);
  digitalWrite(MOTOR_EN, LOW); }
```

run.ino

Figure 6-3

```
C/C++
void rotate(t_local_dir dir, int times){ int obj_step;
  g_max_speed =
  350.0; g_accel = 1.5;
  g_step_r = g_step_l
  = 0; g_speed = g_min_speed =
  MIN_SPEED; g_step_hz_r = g_step_hz_l =
  (unsigned short)(g_speed / PULSE); obj_step = (int)(TREAD_WIDTH * PI /
4.0 *
  (float)times * 2.0 / PULSE);


  switch (dir) { case
      right:
          digitalWrite(CW_R, HIGH);
          digitalWrite(CW_L, LOW);
```

```
            g_motor_move = 1; break;
            case left:

            digitalWrite(CW_R, LOW);
            digitalWrite(CW_L, HIGH); g_motor_move
            = 1; break; default:


            g_motor_move = 0; break;

    }

    while (((obj_step - (g_step_r + g_step_l)) * PULSE) >
                    (((g_speed * g_speed) - (MIN_SPEED * MIN_SPEED)) /
(2.0 * 1000.0 * 1.5))) { continue;


    } g_accel = -1.5;
    g_min_speed = MIN_SPEED;

    while ((g_step_r + g_step_l) < obj_step) { continue;

    }

    g_motor_move = 0; }
```

Commentary

STEP6 is a sample program to check the rotation on the spot. It has almost the same structure as STEP5.
It is composed of a rotor that changes direction and rotates a fixed angle.

About the loop function

The loop function in the uROS_STEP6_rotate.ino file calls the rotation functions 90 degrees right, 90 degrees left, 180 degrees right, and 180 degrees left in order. The rotate function is a

function that rotates. The first argument is the direction of rotation, and the second argument is how many degrees from 90 degrees as the base.
Specify whether to rotate twice.

The rotation direction of the first argument is specified not as a number but as a string, either right or left. Instead of a declaration like #define right, specify the rotation direction as a string using typedef and an enumeration type, typedef enum {•••} t_local_dir. typedef gives a new name

to a data type. For example, if you write it as follows, uint32 will become unsigned
It will be the same as int.

```
C/C++
typedef unsigned int uint32;
```

The basic way to declare an enumeration is:

```
C/C++
enum tag name { member 1, member 2...};
```

When declaring a variable using an enumeration type, write it as follows:

```
C/C++
enum tag name variable name;
```

Here, the enum type is named t_local_dir using typedef, so the enum tag name is unnecessary. The front, right, rear, and left written in

this enum are also used when exploring the maze in STEP 8. If the value of the second argument is set to 1, it will rotate 90 degrees, if it is set to 2, it will rotate 180 degrees, and if it is set to 3, it will rotate 270 degrees.

About the rotate function

In STEP 6, add the rotate function to the run.ino file. The type of the first argument of the rotate function is t_local_dir.

In C language, user-created types must be defined in the header or declared as extern. However, in the Arduino IDE environment, if you declare the t_local_dir type where the setup function is located, you can use that type in other files without declaring it. The variable settings in the rotate function are the same as

in STEP 5. To explain only the differences, obj_step = (int)

(TREAD_WIDTH * PI / 4.0 * (float)times * 2.0 / PULSE) calculates the number of pulses required for a specified turning angle. TREAD_WIDTH, the distance between the left and right tires, is the turning diameter, and the circumference of diameter D is calculated as D x ÿ. As 90 degrees is one unit, 90/360=1/4, so the distance traveled in a 90

degree turn is

The result is TREAD_WIDTH*PI/4.0. This is combined with the second argument, times, to calculate the distance required to turn. To convert the distance to the number of pulses, divide by PULSE. The

switch statement switches the direction of rotation of the tires. In the default: area, the motor operates.

The pulse output is stopped to prevent this.

The condition comparison in the first while statement is the same as in the decelerate function, but the variables used are The numbers are different.

The arguments of the rotate function do not have a distance, so they cannot be compared by distance.

The number of pulses required to turn is stored in obj_step. The difference between the number of pulses required to turn and the number of pulses turned (g_step_r + g_step_l) is multiplied by PULSE (=0.377[mm]) to convert it to distance.

STEP 7: Use P control to drive along the wall

overview

This is a sample program for controlling the robot to run straight without hitting a wall. Pressing the blue or yellow switch will cause the robot to run a certain distance while accelerating or decelerating. Pressing the red switch will cause the robot to continue outputting the wall sensor value via serial communication.

The line that runs through the center of the section is used as the reference point. When the Micromouse (this product) is not at that reference point, it moves to the reference point with a speed difference between the left and right tires. A mode for checking the reference point and a mode for running without bumping into things are required, so the mode can be switched by operating a switch. Pressing the red switch outputs the sensor value. Pressing the yellow or blue switch will run without bumping into walls. The travel distance for the sample program is set to 4 sections = (180x4mm).

program

A program that combines STEP 4 and STEP 5 and adds posture control (control to keep the distance from the wall constant) The program added in STEP 7 is shown in the figure. run.ino is the same as in STEP 5, so it is omitted here.

uROS_STEP7_P_control.ino

Figure 7-1

```
C/C++
#define REF_SEN_R 352 #define
REF_SEN_L 327

#define TH_SEN_R 173 #define
TH_SEN_L 169 #define
TH_SEN_FR 145
#define TH_SEN_FL 134

#define CONTH_SEN_R TH_SEN_R
#define CONTH_SEN_L TH_SEN_L

#define CON_WALL_KP (0.5)

typedef struct{ short
  value; short p_value;
  short error; short ref;
  short th_wall;
```

```
  short th_control; bool
  is_wall; bool
  is_control; } t_sensor;


typedef struct{
  double control;
  double error;
  double p_error;
  double diff;
  double sum;
  double sum_max;
  double kp;
  double kd;
  double ki;
  bool enable;
} t_control;
```

Figure 7-2 Setup function

```
C/C++
  g_sen_r.ref = REF_SEN_R;
  g_sen_l.ref = REF_SEN_L;

  g_sen_r.th_wall = TH_SEN_R;
  g_sen_l.th_wall = TH_SEN_L;

  g_sen_fr.th_wall = TH_SEN_FR;
  g_sen_fl.th_wall = TH_SEN_FL;

  g_sen_r.th_control = CONTH_SEN_R;
  g_sen_l.th_control = CONTH_SEN_L;

  g_con_wall.kp = CON_WALL_KP;
```

Figure 7-3 Loop function

```
C/C++
void loop(){ // put
  your main code here, to run repeatedly: while (digitalRead(SW_L)
  & digitalRead(SW_C) & digitalRead(SW_R)) { continue;



  }

  if (digitalRead(SW_L) == 0) { while (1)

      { Serial.printf("r_sen is %d\n\r", g_sen_r.value); Serial.printf("fr_sen is
          %d\n\r", g_sen_fr.value); Serial.printf("fl_sen is %d\n\r", g_sen_fl.value);
          Serial.printf("l_sen is %d\n\r", g_sen_l.value); Serial.printf("VDD is
          %d\n\r", g_battery_value); delay(100);



      }

  } digitalWrite(MOTOR_EN, HIGH);
  delay(1000);
  accelerate(90, 350);
  oneStep(180 * 3, 350);
  decelerate(90, 350);
  delay(1000);
  digitalWrite(MOTOR_EN, LOW); }
```

interrupt.inoFigure 7-4

```
C/C++
void controlInterrupt(void){ double spd_r,
  spd_l;

  g_speed += g_accel;

  if (g_speed > g_max_speed) { g_speed =
      g_max_speed;

  } if (g_speed < g_min_speed) { g_speed =
      g_min_speed;
  }

  if ((g_sen_r.is_control == true) && (g_sen_l.is_control == true)) { g_con_wall.error =
g_sen_r.error
      - g_sen_l.error; } else { g_con_wall.error = 2.0 * (g_sen_r.error -

  g_sen_l.error);
  }

  g_con_wall.control = 0.001 * g_speed * g_con_wall.kp *
g_con_wall.error;

  spd_r = g_speed + g_con_wall.control; spd_l = g_speed
  - g_con_wall.control;

  if (spd_r < g_min_speed) { spd_r =
      g_min_speed;

  } if (spd_l < g_min_speed) { spd_l =
      g_min_speed;
  }

  g_step_hz_r = (unsigned short)(spd_r / PULSE); g_step_hz_l =
  (unsigned short)(spd_l / PULSE); }
```

Figure 7-5 sensorInterrupt function (right side sensor processing excerpt)

```
C/C++
    case 2:
        digitalWrite(SLED_R, HIGH); for (int
        i = 0; i < 300; i++) { asm("nop \n");


        } g_sen_r.value = analogRead(AD3);
        digitalWrite(SLED_R, LOW); if
        (g_sen_r.value > g_sen_r.th_wall) { g_sen_r.is_wall
            = true; } else { g_sen_r.is_wall
        = false;


        } if (g_sen_r.value > g_sen_r.th_control) {
            g_sen_r.error = g_sen_r.value - g_sen_r.ref;
            g_sen_r.is_control = true; } else

        { g_sen_r.error = 0;
            g_sen_r.is_control = false;

        } break;
```

Commentary

About uROS_STEP7_P_control.ino

The newly added code in uROS_STEP7_P_control.ino sets the parameters required for control.
Set this (g_sen_*** at the end of the setup function).

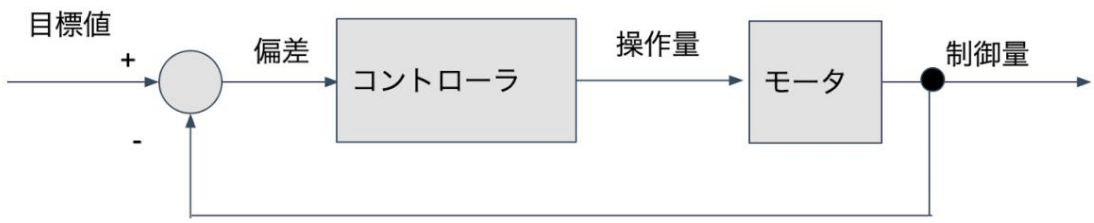
g_sen_r and g_sen_l are variables of a structure declared with the type t_sensor. g_sen_r.ref is the value of the sensor on the right side when the mouse is placed in the center of the area. This value is used as the reference, and the motor is controlled to get close to this value.

In g_sen_l.ref, enter the reference value of the left lateral sensor when the micromouse is placed in the center of the section. These reference values are written as parameters in #define REF_SEN_R and #define REF_SEN_L.

g_sen_r.th_wall, g_sen_l.th_wall, g_sen_fr.th_wall, and g_sen_fl.th_wall at the end of the setup function are thresholds beyond which it is determined that a wall exists. These will not be used in STEP7, but will be used when creating the map for STEP8.

At the end of the setup function, g_sen_r.th_control and g_sen_l.th_control are
This is the threshold that enables the control. We will explain why this threshold is necessary.

The control method used in this sample is P control (proportional control). P control is a control method that increases the manipulated variable when the controlled variable deviates from the reference value. The manipulated variable is calculated by multiplying the difference (called deviation) between the reference sensor value and the current sensor value by a coefficient. When the Micromouse moves away from the wall, the sensor value decreases and the manipulated variable increases. The method of control by comparing the target value with the current position is called feedback control.



Feedback Control

Since the amount of operation is calculated using the deviation, when there is no wall, the sensor value is 0 and the reference value remains the same.
This is the amount of control. If the sensor value becomes 0, it will mistakenly think that the wall is far away and control will be performed. To avoid this mistake, control is enabled when the sensor value is greater than

g_sen_*.rh_control. For information on how to obtain the g_sen value, please refer to the attached Pi:Co Classic3 Getting Started Guide.

About interrupt.ino The interrupt.ino

file contains the controlInterrupt function, which controls the speed, and the function that obtains the sensor value and performs biasing. There is a sensorInterrupt function that calculates the difference and determines whether a wall is present or not.

Regarding the controlInterrupt function,

the following block checks the sensor value and controls the motor to move the micromouse to the center of the section. When

g_sen_r.is_control is true, it means that the sensor on the right is the one to be controlled.

```
C/C++
if ((g_sen_r.is_control == true) && (g_sen_l.is_control == true)) { g_con_wall.error = g_sen_r.error -
g_sen_l.error; }
    else { g_con_wall.error = 2.0 * (g_sen_r.error - g_sen_l.error);



}
```

In the if statement, the deviations on the left and right are added together. g_sen_l.error is the opposite sign to g_sen_r.error. The reason for this is to accommodate the deviation calculation in the sensorInterrupt function.

In the sensorInterrupt function, the right deviation is calculated as g_sen_r.error = g_sen_r.value - g_sen_r.ref, and the left deviation as g_sen_l.error = g_sen_l.value - g_sen_l.ref. For example, when the micromouse is closer to the wall to the right of the reference position, the right sensor value will be greater than the reference value, so the deviation will be positive. Conversely, the left sensor value will be smaller than the reference value, so the deviation will be negative. To prevent the left and right deviations from cancelling each other out, the sign of the left deviation is reversed. In the else statement, the left

and right deviations are doubled. This is to accommodate situations where there is a wall on only one side. In the following code, the

sum of the left and right deviations is processed as the deviation to be input into the manipulated variable.

```
C/C++
g_con_wall.control = 0.001 * g_speed * g_con_wall.kp * g_con_wall.error;



  spd_r = g_speed + g_con_wall.control; spd_l = g_speed -
  g_con_wall.control;
```

The control amount g_con_wall.control is calculated from the deviation g_con_wall.error between the left and right, and the control amount is added to the running speed g_speed to create a speed difference between the left and right, and the car is controlled on the target line. The control amount is calculated using the P control coefficient con_wall.kp for the deviation. If the coefficient is large, the car will approach the target line quickly, but if it is too large, the car will run unsteadily on the target line.

If you add the operation amount directly to the left and right speed, when the speed is fast, it will approach the target line gradually, but when the speed is slow, it will approach the target line suddenly. To approach the target line equally at any speed, make the operation amount a ratio of the speed. That is 0.001*g_speed. 0.001 means the interrupt interval of 0.001 seconds.

spd_r = g_speed + g_con_wall.control, spd_l = g_speed - g_con_wall.control add the operation amount to the running speed g_speed. if (spd_r < g_min_speed) checks whether the speed after adding the operation amount is below the lower limit of the speed. If it is below the lower limit, the speed is set to the lower limit.

About the sensorInterrupt function The

sensorInterrupt function makes control decisions and calculates deviations based on the acquired sensor values. There are case 0 to case 3, but the processing content is the same, so we will explain it using case 2, the right side sensor. The

first half is the same as STEP 4. If (g_sen_r.value > g_sen_r.th_wall) is used to determine whether a wall exists. If it is determined that there is a wall, it sets g_sen_r.is_wall=true, and if it is determined that there is no wall, it sets g_sen_r.is_wall=false. This is not used in the STEP7 sample, but it will be used when generating the STEP8 map.

Similarly, if (g_sen_r.value > g_sen_r.th_control) determines whether to control or not. If it is decided to control, calculate the deviation by subtracting the target value from the current sensor value and set g_sen_r.is_control=true. If it is decided not to control, set the deviation to 0 and set g_sen_r.is_control=false.

STEP 8: Run through the maze

overview

This is a sample program for running through a maze in the Micromouse Classic Competition. Pressing the blue and red switches increases or decreases the running mode number, and pressing the yellow switch executes the function corresponding to the running mode number. The functions that are executed are listed in the table below. The running mode number is displayed in binary by the LED.

STEP8 Driving Mode List

Driving mode number	The function to be executed
1. Left method (no search)	
2. Legislative Exploration	
3. Shortest run	
15 Motor OFF Switch	to adjustment mode
Other Do nothing	

The adjustment modes are as follows:

Adjustment Mode List

Adjustment Mode Number	The function to be executed
1 Checking sensor values (viewAdc)	
2 Check tire diameter (straightCheck)	
3. Check the tread (rotationCheck)	
4. Check maze information (copyMap, mapView)	
5, 6 Do nothing	
7 Return to driving mode selection	

STEP 1 to STEP 7 are sample programs for creating and verifying device drivers to operate the hardware of this product. In STEP 8, we will add an application that uses the device driver to explore and navigate a maze.

program

Because there are a large number of files, only the necessary parts are illustrated in the explanation.

Commentary

STEP8 is made up of multiple files divided by function, taking into consideration the reusability of the program.
The files are listed in the table below.

The device.ino contains hardware-dependent functions such as LEDs, buzzers, and motors.
The function to be called by the timer interrupt is also defined.

In device.h, there is a GPIO definition such as #define LED 47, which was written at the beginning of the program so far.
This file summarizes the definition of the numbers. Even if the GPIO function assignments change, this file will remain the same.
You just need to fix it.

search.ino contains the left-hand and foot-standby search functions used in maze searching.

The shortest running function is written in fast.ino.

SPIFFS.ino contains functions to read and write maze information to the Flash ROM.

map_manager.h and map_manager.ino contain functions for managing the maze and generating routes.
The Arduino IDE is C-like, but it also supports C++, so it can be used with C++ class functions.
The MapManager class is defined using the

In addition, the functions that appeared in STEP 1 to STEP 7 are described in separate files.

STEP8 file list

File name	Implementation details
uROS_STEP8_micromouse.ino	A setup function that calls the hardware initialization function. A loop function that determines the mode and calls the function to be executed.
adjust.ino	Displaying sensor values, Adjustment function for checking tire-related parameters, etc.
device.ino	Using the built-in functions of the Arduino IDE, Hardware control functions. The initialization function for the feature.
device.h	Definition of GPIO number.
fast.ino	The shortest running function.
interrupt.ino	Functions that process control and sensor values in interrupts.
map_manager.h, map_manager.ino	Classes for map management and calculating the shortest path.
misc.ino	Switch operation and goal appeal functions.
mytypedef.h	Declaration of enums and sensor structures.
parameter.h	Sensor reference, tire diameter, and other parameters Meter.
run.ino	A function to set driving parameters for going straight and turning.
search.ino	Left hand and foot position functions.
SPIFFS.ino	A function to read and write maze information to Flash ROM.

Explanation of uROS_STEP8_micromouse.ino

In this file, select Left Handed, Foot Standing, and Adjustment Mode.

An instance of the MapManager class that manages the

```
C/C++
MapManager g_map_control;
```

We declare this.

About the loop function

In the loop function, the mode is determined by pressing the blue, yellow, or red switches. The mode numbers range from 1 to 15. Pressing the blue switch calls the incButton function, which increases the mode number by 1. If the number becomes greater than 15, it is reset to "1". Pressing the red switch calls the decButton function, which decreases the mode number by 1. If the number becomes less than 1, it is reset to "15". Pressing the yellow switch executes the okButton and execByMode functions. The incButton, decButton, and okButton functions are written in the misc.ino file.

Once the mode has been determined, the execByMode function is used to call the function corresponding to the mode number.

```
C/C++
void loop() { //

 put your main code here, to run repeatedly: setLED(g_mode); switch
  (getSW()) { case
  SW_LM:

        g_mode = decButton(g_mode, 1, 15); break;

    case SW_RM:
        g_mode = incButton(g_mode, 15, 1); break;

    case SW_CM:
        okButton();
        execByMode(g_mode); break;


  } delay(1); }
```

About the execByMode function

The execByMode function executes the function according to the mode number determined in the loop function. When the mode number is 1, the robot runs in the left direction. In the left direction, the robot does not stop even after passing the goal because there is no maze management.

When the mode number is 2, the robot runs in a foot-standing manner. In this mode, the robot searches from the start to the goal, and when it reaches the goal, it makes a goal appeal (executes the goalAppeal function). After the appeal, the maze information acquired during the run up to that point is written to the Flash ROM. After that, the robot returns to the start, prioritizing passage through unexplored areas. When the robot returns to the start, it overwrites the acquired maze information to the Flash ROM (executes the mapWrite function).

```C/C++
    case 2: //ÿÿÿ
        g_map_control.positionInit();
        searchAdachi(g_map_control.getGoalX(),
g_map_control.getGoalY());
        rotate(right, 2);
        g_map_control.nextDir(right);
        g_map_control.nextDir(right);
        goalAppeal();
        searchAdachi(0, 0);
        rotate(right, 2);
        g_map_control.nextDir(right);
        g_map_control.nextDir(right); mapWrite();
        break;
```

When the mode number is 3, it will run the shortest distance. Before running the shortest distance, the maze information written in the Flash ROM is called with the copyMap function. After that, it executes the same process as in mode 2. The running function switches from the searchAdachi function to the fastRun function. Since the maze information is not updated in the shortest distance run, the mapWrite function is not executed even if it returns to the start. When the mode number is 15, it will switch to adjustment mode. When the mode number is other than 1, 2, 3, or 15, it will not execute anything.

Explanation of adjust.ino

It checks the sensor values, drives a straight line distance to check that the set tire diameter is correct, turns to check that the set tread is correct, and checks the maze information that was driven.

About the adjustMenu function:

Determine the mode to check with the adjustMenu function. The behavior when the mode is selected is
This is the same as the loop function in the uROS_STEP8_micromouse.ino file, so to distinguish it, the adjustMenu function blinks LED3 at 33 ms intervals.

```
C/C++
        delay(33);
        LED3_data ^= 1;
        setLED((mode & 0x7) + ((LED3_data << 3) & 0x08));
```

After the mode number is determined by the adjustMenu function, execute the execByModeAdjust function to set the mode number. Calls the corresponding function.

About the viewAdc function

The viewAdc function displays the sensor values using serial communication. The Arduino IDE serial monitor cannot display the escape sequences used in the function, so you should use a different serial monitor. We recommend the Tera Term app[1] for Windows and the screen command for Linux (Ubuntu).

About the straightCheck function

The straightCheck function uses the accelerate, straight, and decelerate functions to move forward a specified distance. The distance is specified as an argument to straightCheck(9) in the exexByModeAdjust function. Each function is explained in run.ino.

About the rotationCheck function

The rotationCheck function uses the rotate function to rotate to the right eight times.

About the mapView function

The mapView function displays maze information using serial communication.

[1] https://teratermproject.github.io/

Explanation of device.ino

Setting up GPIO, timers, etc., and using the built-in functions of ESP32 and Arduino IDE

This page contains a collection of functions that perform numerical operations.

About the IRAM_ATTR onTimer0 function

Calls the speed control's controlInterrupt function. For information on the controlInterrupt function, see I will explain this using interrupt.ino.

About IRAM_ATTR onTimer1 function

Call the sensorInterrupt function to get the sensor value. The sensorInterrupt function will be explained in interrupt.ino.

About **IRAM_ATTR isrR and isrL functions**

This function is called by the motor interrupt. The processing is the same as STEP 5.

About the controlInterruptStart function

Starts counting timer 0.

About the controlInterruptStop function

Stops timer 0 from counting.

About sensorInterruptStart function

Starts timer 1 counting.

About sensorInterruptStop function

Stops timer 1 from counting.

About the PWMInterruptStart function

Timers 2 and 3 start counting.

About the PWMInterruptStop function

Stops timers 2 and 3 from counting.

About the initAll function

Set the GPIO assigned to the LEDs and switches, the ledc function used for the Buzzer, the initial settings for the Flash ROM, the initial settings for timers 0, 1, 2, and 3, and the initial settings for serial communication. The GPIO, timer, and serial communication settings are the same as in STEP 4 and STEP 5.

BLED0 and BLED1 are GPIO outputs for controlling the two-color LED. The remaining battery power is indicated by the

two-color LED.

It initializes the wall, the control structure (g_con_wall), and the goal coordinates of the maze information.

Since the motor is stopped immediately after power is turned on, set the initial value of the g_motor_move variable to false. The initial values of the g_step_hz_r and g_step_hz_l variables are set to MIN_SPEED. After

initialization is complete, a sound is emitted at the INC_FREQ frequency for 80[ms].

To initialize the Flash ROM, use the ESP32 built-in function SPIFFS.begin. Initialization takes a few minutes the first time.

```
C/C++
  if (!SPIFFS.begin(true)) { while (1)

      { Serial.println("SPIFFS Mount Failed"); delay(100);

      }
  }
```

The values used to initialize the sensor structure and control structure are written in parameter.h.

```
C/C++
  g_sen_r.ref = REF_SEN_R; g_sen_l.ref
  = REF_SEN_L; g_sen_r.th_wall =
  TH_SEN_R; g_sen_l.th_wall = TH_SEN_L;


  g_sen_fr.th_wall = TH_SEN_FR; g_sen_fl.th_wall
  = TH_SEN_FL;

  g_sen_r.th_control = CONTROL_TH_SEN_R; g_sen_l.th_control
  = CONTROL_TH_SEN_L;

  g_con_wall.kp = CON_WALL_KP;
```

In the Micromouse Classic competition, the goal coordinates do not change, but
The goal coordinates are changed as parameters. The parameters are written in parameter.h.
Masu.

```
C/C++
  g_map_control.setGoalX(GOAL_X);
  g_map_control.setGoalY(GOAL_Y);
```

About the setLED function

The setLED function controls the LED corresponding to the bit in the argument. When the target bit is 1, it lights up, and when it is 0, it lights up.
It will turn off at time.

SetLED function lighting pattern correspondence table

7	6	5	4	3	2	1	0
invalid				LED3	LED2	LED1	LED0

About the setBLED function

The setBLED function controls the two-color LED corresponding to the bit of the argument. The two-color LED lights up when the target bit
When it is 1, it turns on, and when it is 0, it turns off.

setBLED function lighting pattern correspondence table

7	6	5	4	3	2	1	0
invalid						BLED1 (red)	BLED0 (blue)

About the enableBuzzer function

Set the frequency as an argument to the ledcWriteTone function to output the PWM that will be the source of the sound.

About the disableBuzzer function

Use the ledcWrite function to fix the PWM value to HIGH and stop the buzzer sound.

About the enableMotor function

Set the MOTOR_EN pin to HIGH to energize the motor.

About the disableMotor function

Set the MOTOR_EN pin to LOW to de-energize the motor.

About the moveDir function

Specifies the direction of rotation of the left and right motors.

About the getSW function

If the blue switch is pressed, it returns 4, if the yellow switch is pressed, it returns 2, and if the red switch is pressed, it returns 1. To prevent chattering from affecting it, it waits 20[ms] before checking that the switch is not pressed. If the switch is still pressed after waiting, it waits 20[ms] and checks again that it is not pressed. When multiple switches are pressed, it adds up the values of the pressed switches and returns the result. For example, when the blue and red switches are pressed, it returns a value of 4+1=5.

About the getSensorR function

Returns the A/D converted value of the right sensor. The processing is the same as in STEP 4.

About the getSensorL function

Returns the A/D conversion value of the left sensor. The processing is the same as in STEP 4.

About the getSensorFL function

Returns the A/D converted value of the left front sensor. The processing is the same as in STEP 4.

About the getSensorFR function

Returns the A/D converted value of the right front sensor. The processing is the same as in STEP 4.

The getBatteryVolt function returns the A/D

converted value of the battery voltage in units of [mv].

Explanation of device.h

This file defines GPIO numbers as strings.

Explanation of fast.ino

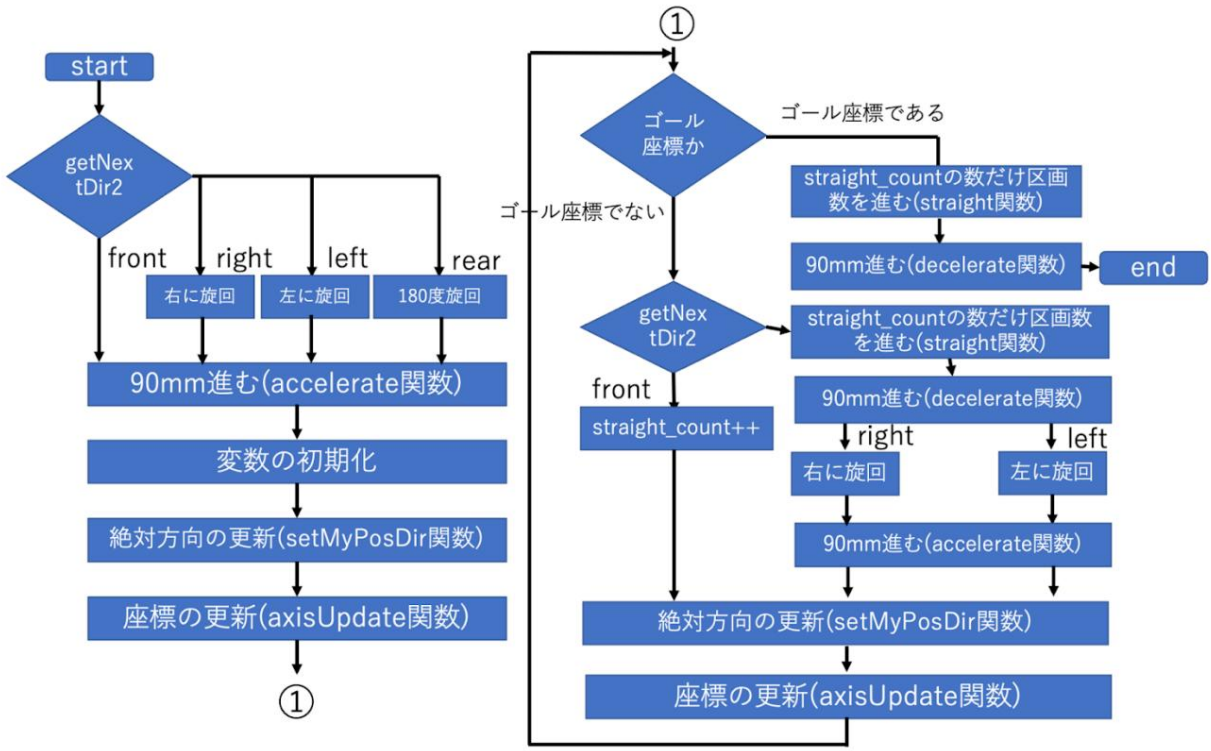
This file implements the fastRun function. When the fastRun function is executed, the robot will travel along the shortest route to the goal coordinates (gx, gy) in the maze information that has already been explored. Any coordinates can be specified as the goal coordinates. The flowchart for the fastRun function is shown below.

Use the rotate function to rotate right, left, or 180 degrees.
The straight function is written in run.ino.

Before turning, the robot moves straight through the number of blocks specified by "straight_count". After moving straight, the robot initializes

"straight_count" to 0. The getNextDir2, setMyPosDir, and axisUpdate functions are member functions of the MapManager class.
It is written in map_manager.ino.

To get the coordinates of the current position, use the member function getMyposX of the MapManager class.
Use getMyposY.



Flowchart of the shortest route

Explanation of interrupt.ino:

interrupt.ino contains the controlInterrupt function, which is called by the timer 0 interrupt, and the sensorInterrupt function, which is called by the timer 1 interrupt. The

controlInterrupt function updates the driving speed and controls straight-line driving using the wall sensor. Straight-line driving control adjusts the left and right tire speeds to maintain a certain distance from the wall. If a turn is executed while this control is enabled, the wall sensor will react during the turn, making it impossible to turn to the calculated angle. The g_con_wall.enable change

is made so that wall control is not performed during a turn, and only straight-line driving is controlled against the wall. The operation amount is added to the running speed only when g_con_wall.enable is true. The sensorInterrupt function implements the same processing as in STEP7. The functions in device.ino (getSensorR, getSensorL, getSensorFL, getSensorFR) are used to obtain the sensor values.

In STEP8, a function to display the battery voltage with a two-color LED has been added to the sensorInterrupt function. The LED lights up blue when the battery voltage is high and red when the battery voltage is low.
For a Li-Po 3-cell battery, the voltage when fully charged is 12.6V. We would like to set the upper limit (BATT_MAX) to 12.6V, but since power may be supplied from a 12V AC adapter, the upper limit is set to 12.0V. Since Li-Po batteries cannot be charged if they are over-discharged, the lower limit (BATT_MIN) is set to 10V. Therefore, when the battery voltage is 12.0V or higher, only the blue LED is lit, and when it is 10V or lower, only the red LED is lit. When the battery voltage is between 12.0V and 10.0V, the ratio of blue and red LEDs to light is

changed according to the voltage. The lighting ratio is implemented by how many times the sensorInterupt function is executed 10 times. For example, when the battery voltage is 11.6V, the blue LED will light 8 times out of 10, and the red LED will light 2 times.

Explanation of map_manager.h This file

defines the MapManager class that manages maze information and calculates the shortest route. Although the Arduino IDE is C-like, it can also be written in C++, so the C++ class function is used.

Explanation of map_manager.inoThis is a

file that implements the member functions of the MapManager class.

After initializing the wall information of all

sections of **the MapManager constructor** maze information to _UNKNOWN, set the wall information known at the start point. There are walls around the maze perimeter and on the east side of the start section, so set WALL. There is no wall on the north side of the start section, so set NOWALL.

The positionInit member function

is a function that should be executed when starting to drive. It initializes the self-position information (mypos) held by MapManager. As both the x and y coordinates of the starting area are 0, the mypos.x and mypos.y members are set to 0. As the direction of travel at the start is north, the mypos.dir member is set to north.

The setMyPosDir member function

sets the direction in the argument to mypos.dir.

getMyPosX Member Function

Returns the value of mypos.x.

getMyPosY Member Function

Returns the value of mypos.y.

The getWallData member function

returns wall information in the absolute direction of the specified partition.

The setWallData member function

Sets the wall information for the specified area in an absolute direction.

The getGoalX Member Function

Returns the x-coordinate value of the goal that has been set.

getGoalY Member Function

Returns the y coordinate value of the goal that has been set.

The setGoalX Member Function

Sets the x-coordinate of the goal.

The setGoalY Member Function

Sets the y coordinate of the goal.

The axisUpdate member

function advances the current position (mypos.x, mypos.y) by one block in the direction of mypos.dir.

The nextDir Member Function

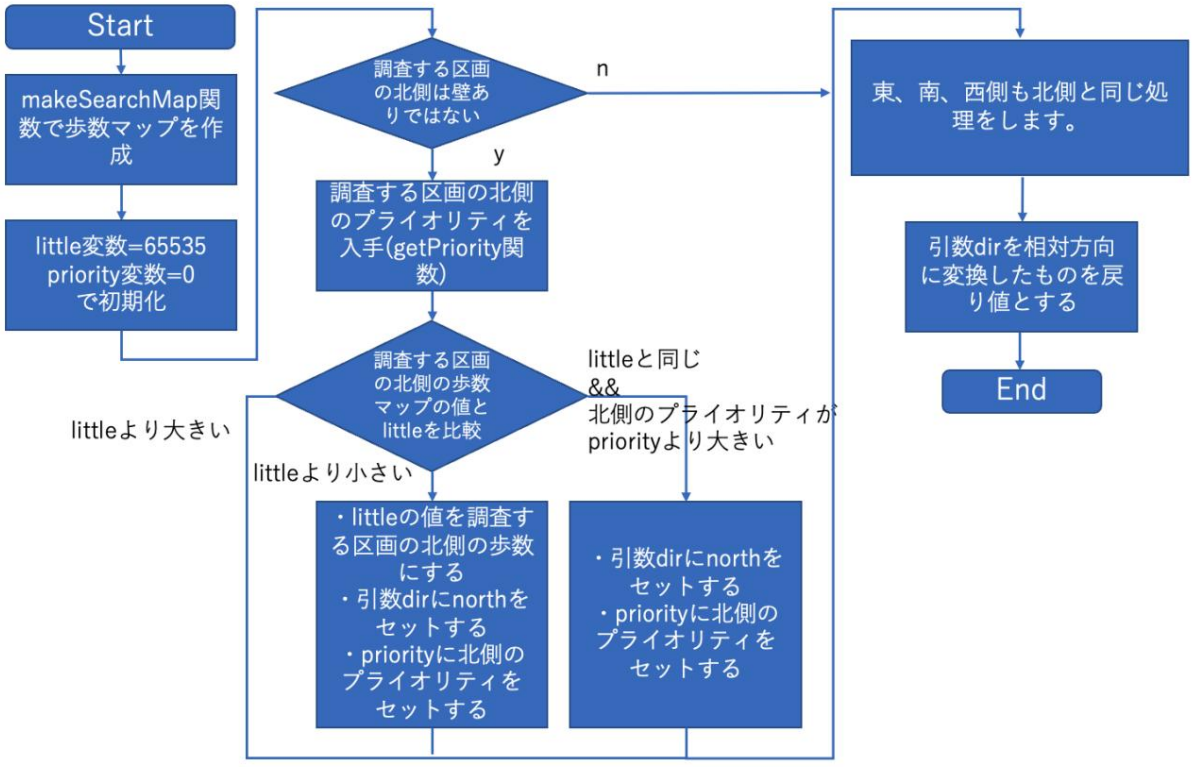
Updates the current direction (mypos.dir) to the direction after rotating 90 degrees to the direction specified by the argument (right, left).

The setWall Member Function

Based on the wall presence flag set in the argument (true for wall, false for no wall), the current position (
Updates the wall information to the north, south, east and west of the position (mypos).
In IS_SEN_L, set the flag for front, right, or left side.

Used when searching for **the**

getNextDir member function . Creates a step count map with the weight of the goal coordinates (x, y) specified by the argument set to 0, and returns the direction with the fewest steps from the current section to the goal. The absolute direction is set in the argument dir, and the relative direction from mypos.dir is set in the return value. The flow chart of this function is shown below.

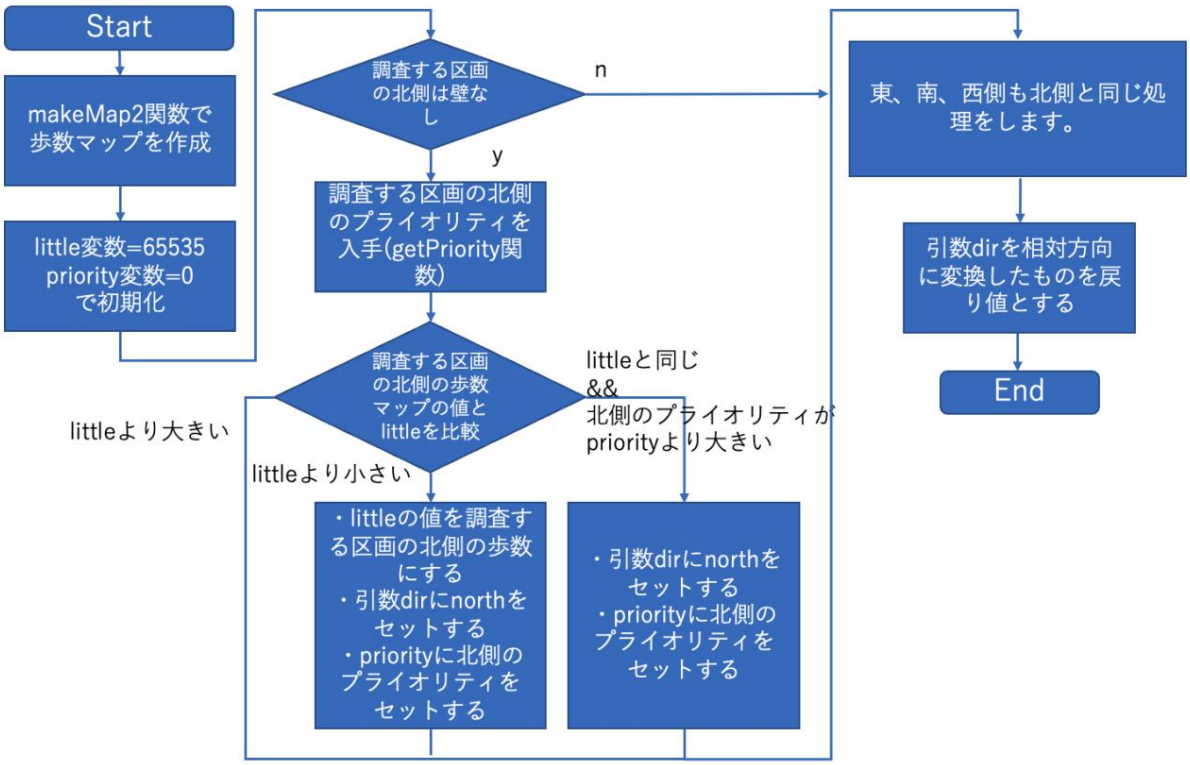


Flowchart of getNextDir function

getNextdir2 member function

Used to introduce the shortest route. The weight of the goal coordinates (x, y) specified by the argument is set to 0. It creates a step count map and returns the direction with the fewest steps from the current section to the goal.

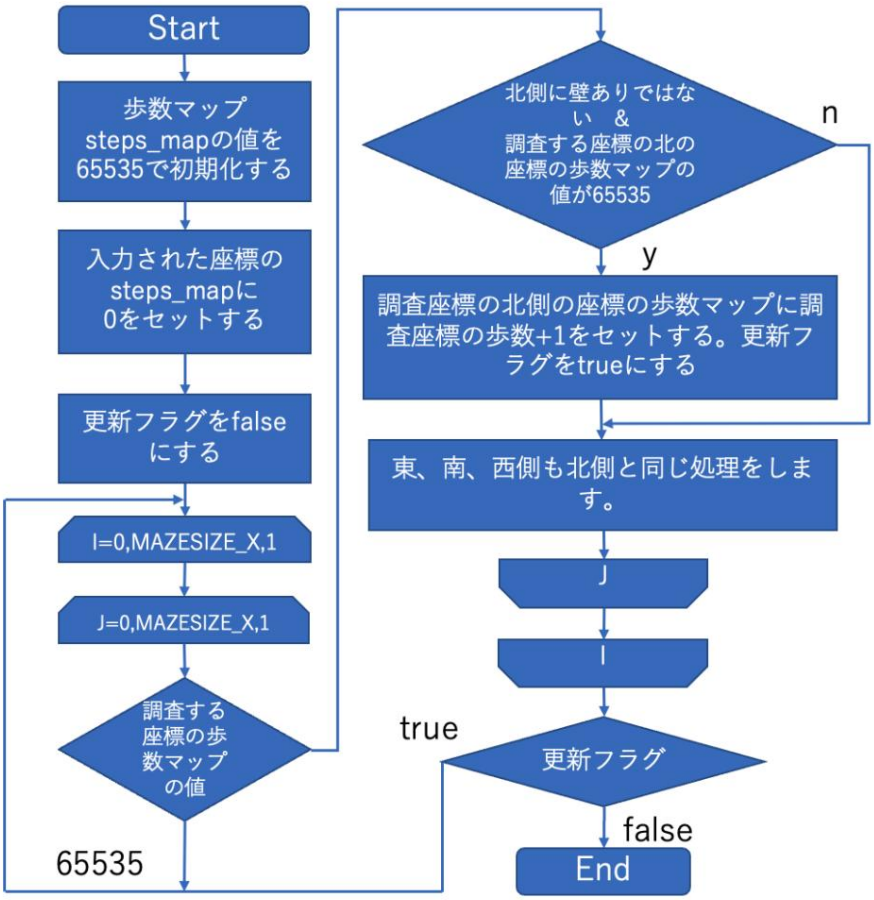
The flowchart of this function is shown below. The difference from the getNextDir function is the generation of the step count map. The difference is that the makeMap2 function is used and the condition expression during the survey is changed to check that there are no walls (NOWALL is set).



Flowchart of getNextDir2 function

This function creates a step count

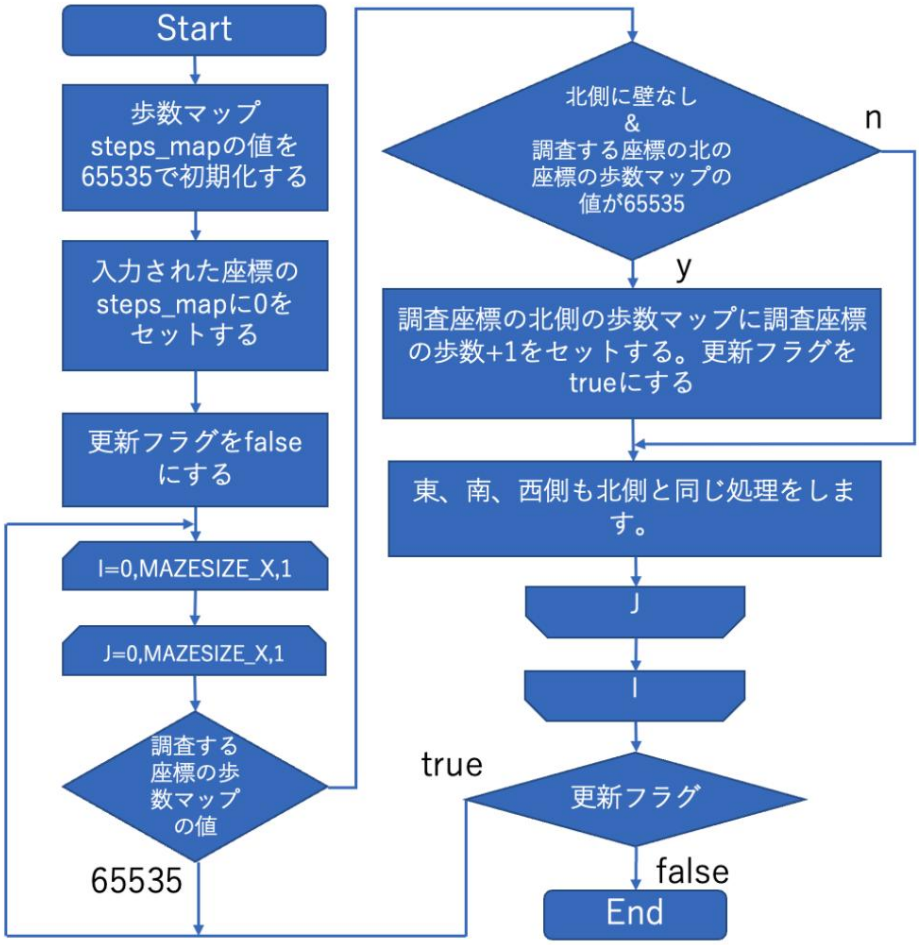
map in the member variable steps_map, with the weight of the coordinates (x, y) specified in **the makeSearchMap member function** arguments set to 0. Each block in the step count map contains a value indicating how many blocks one must travel to reach the coordinates (x, y). This function creates a map under the assumption that there are no walls in unsearched blocks. The flowchart for this function is shown below.



Flowchart of makeSearchMap function

This function creates a step

count map in the member variable steps_map with the weight of the coordinates (x, y) specified in **the makeMap2 member function** arguments set to 0. Unlike the makeSearchMap function, this function uses information about a maze that has already been explored, so it checks that there are no walls between sections (NOWALL is set). The flow chart for this function is shown below.



Flowchart of makeMap2 function

The getPriority member

function returns the priority of whether to move to the area specified by the arguments (x, y, dir).
The higher the priority number, the higher the priority.

If the direction to the partition (dir) is the same as the self direction (mypos.dir), the priority is set to 2. If they are in the opposite direction, the priority is set to 0. Otherwise, the priority is set to 1. If any of the wall information for the partition (x, y) is unknown (_UNKNOWN), the priority is increased by 4.

Explanation of misc.ino

This file implements the tactile switch processing function and the appeal function when entering the goal.

Explanation of mytypedef.h

This is a header file that defines the structures and enumerations used in STEP 8. It contains the structure t_sensor, which groups together sensor-related variables used for things like sensor values and determining whether a wall is present, the structure t_control, which groups together variables necessary for speed control, and t_CW_CCW, which is an enumeration for the motor rotation direction.

Explanation of parameter.h

This file declares constants such as the target sensor values and tire diameters.

Explanation of run.ino

This file defines the travel function used in STEP5, STEP6, and STEP7.

We have added a function "straight" that can sometimes drive through a series of consecutive straight sections at once.

The controlInterrupt function, which processes the timer 0 interrupt, controls the wall when going straight, but when turning, the g_con_wall.enable variable is turned on/off so that the wall is not controlled. Since the rotate function is a function that turns, g_con_wall.enable=false. For other functions, g_con_wall.enable=true.

In STEP8, the maze is analyzed while driving. Since maze analysis takes more than a few clocks, controlInterruptStart is executed at the end of the accelerate function or oneStep function so that driving control does not stop during maze analysis. The number of driving pulses obtained by getStepR and getStepL includes the number of pulses during maze analysis, so it is possible to drive a specified distance including the amount of movement during analysis. If driving control stops during maze analysis, the tires will lock and slip, causing the driving distance to be inaccurate.

Accelerate Function

After initializing the variables g_step_r and g_step_l, which store the number of running pulses, to 0, the running is performed in a while loop. It waits until the number of row pulses reaches the specified distance (len). After exiting the while loop, it initializes g_step_r, g_step_l, and the acceleration variable g_accel to 0. It assigns argument 2, finish_speed, to g_max_speed, g_min_speed, and g_speed. Finally, it calls controlInterruptStart to continue moving straight at a constant speed even after the accelerate function is executed.

The g_step_r and

g_step_l variables are not initialized before the while loop in order to check the increased number of travel pulses before executing the oneStep function. Once the specified distance has been traveled, the while loop is exited and the g_max_speed, g_min_speed, g_speed, g_accel, g_step_r, and g_step_l variables are set in the same way as the accelerate function .

At the end of the accelerate function and the oneStep function, the g_step_r and g_step_l variables are initialized to 0. You can also call these functions consecutively to travel the specified distance.

deceleration function

As with the oneStep function, before the while loop, the g_step_r and g_step_l variables are not initialized to 0. Also, after the while loop, the clock supply to the motor driver is stopped (g_motor_move=0), so the g_max_speed, g_min_speed, g_speed, g_accel, g_step_r, and g_step_l variables are not initialized.

rotate function

Before the while loop, initialize the g_step_r and g_step_l variables to 0. After the while loop, do not initialize the g_max_speed, g_min_speed, g_speed, g_accel, g_step_r, and g_step_l variables because the clock supply to the motor driver is stopped (g_motor_move=0).

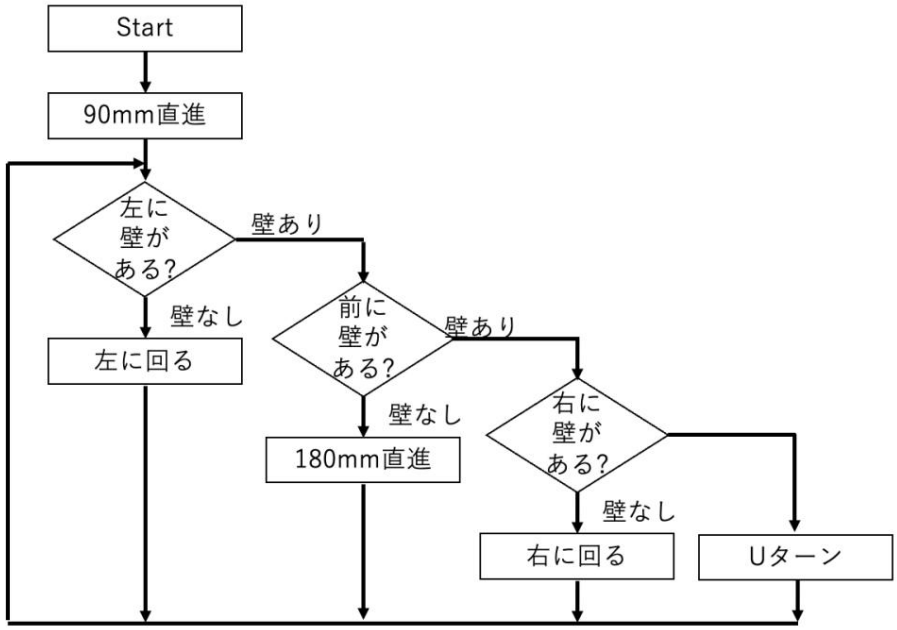
The straight

function accelerates and decelerates to travel a specified distance (len). The initial speed (init_speed), maximum speed (max_speed), and final speed (finish_speed) can be specified as arguments. If the initial speed is smaller than MIN_SPEED, the g_step_r and g_step_l variables are initialized to 0. After exiting the while loop, the clock supply to the motor driver is stopped (g_motor_move=0). If the final speed is the same as the search speed, the g_max_speed, g_min_speed, g_speed, g_accel, g_step_r, and g_step_l variables are initialized in the same way as the accelerate function, and straight-ahead driving continues.

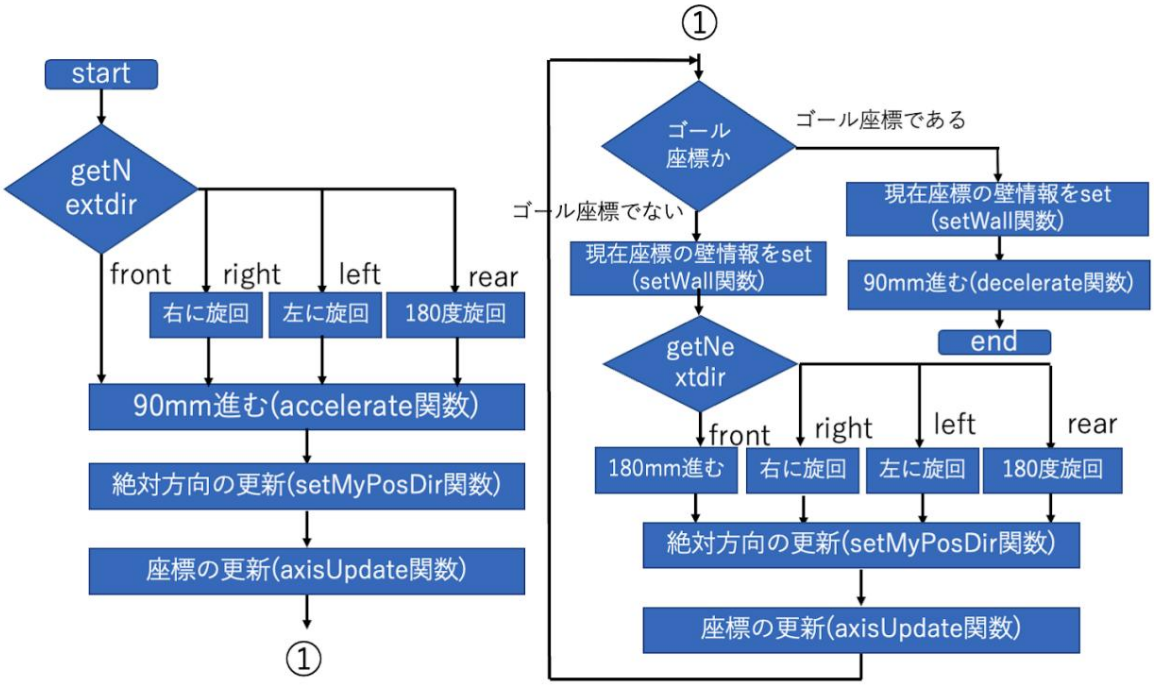
Explanation of search.ino

This is a file that implements the left-hand search function searchLefthand and the foot-hand search function searchAdachi. The flowcharts for the searchLefthand and searhcAdachi functions are shown below. The searchLefthand function does not manage coordinates and maze information, so it will continue running even if it reaches the goal. The searchAdachi

function stops running when it reaches the goal coordinates (gx, gy) specified by the arguments.



Flowchart of the searchLefthand function



Flowchart of searchAdachi function

Explanation of SPIFFS.ino

This file implements functions to read and write maze information to Flash ROM. SPIFFS (SPI Flash
It uses the .NET File System to store data.

The function for writing maze information is mapWrite, and the function for reading maze information is copyMap.

The wall information for one section is compiled into one byte and then read and written.
The format is as follows:

Maze information read/write format

7	6	5	4	3	2	1	0
Information on the Western Wall		South Wall Information		East Wall Information		North Wall Information	

When accessing the Flash ROM, executing a timer interrupt is prohibited by the ESP32 specifications.
When accessing the Flash ROM, the timer must be stopped.
I will.

In order to save data using SPIFFS, you must first create a data area in the SPIFFS system.
You need to format the Flash ROM by pressing the
No formatting will occur from the second writing onwards.

Maze information can also be stored outside of Flash ROM.
The maze information acquired by the computer is often stored in SRAM (Static Random Access Memory).
However, SRAM only stores data when the power is on.
If you turn off the power, all the information stored in the maze will be lost.
Even if a shorter path is found, the search must be started again.

The data on the Flash ROM will not be erased even if the power is turned off, so you can save the maze information during the exploration.
This allows you to resume exploration even if you get stuck and have to power down.

Adjustments to complete the maze

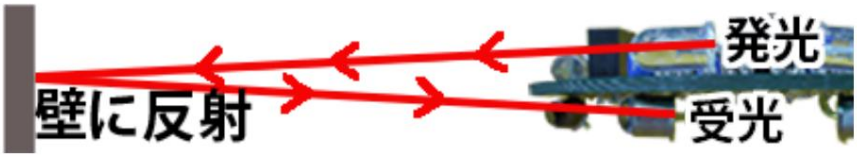
If you simply write the sample program uROS_STEP8_micormouse, you may not be able to reach the goal of the maze because the actual wheel position and wall sensor inclination will differ from the initial values that are set in advance. In order to travel stably to the goal, you will need to adjust the parameters to match the product you have assembled. The parameters to be adjusted include the wall sensor reference value, the attitude control feedback gain, the tire diameter, and the tread (the distance between the tires).

Physical Adjustment of the Wall Sensor

The presence or absence of a wall and the distance are measured by the light emitted by the wall sensor LED being reflected by the wall and then received by a phototransistor. Physically adjusting the angle of the wall sensor can be expected to improve the measurement results.

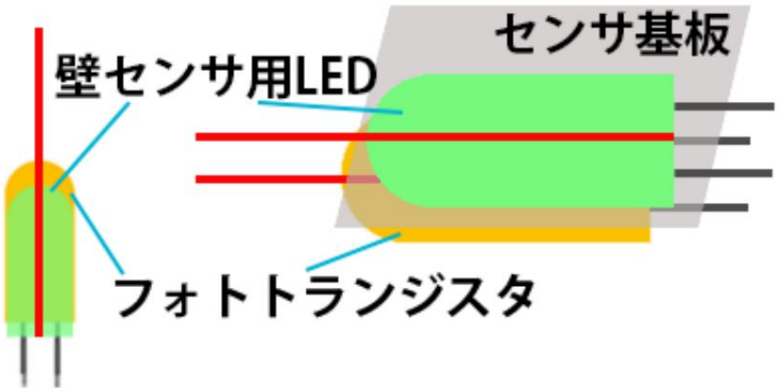
Angle between LED and phototransistor for wall sensor

As shown in the figure below, the wall sensor LED and phototransistor are not oriented perpendicular to the wall.
It's better to put it at a slight angle.



The angle of the sensor when this product is viewed from the side

In addition, as shown in the figure below, the direction (optical axis) of the wall sensor LED and the phototransistor are aligned parallel to each other.
We will adjust it accordingly.

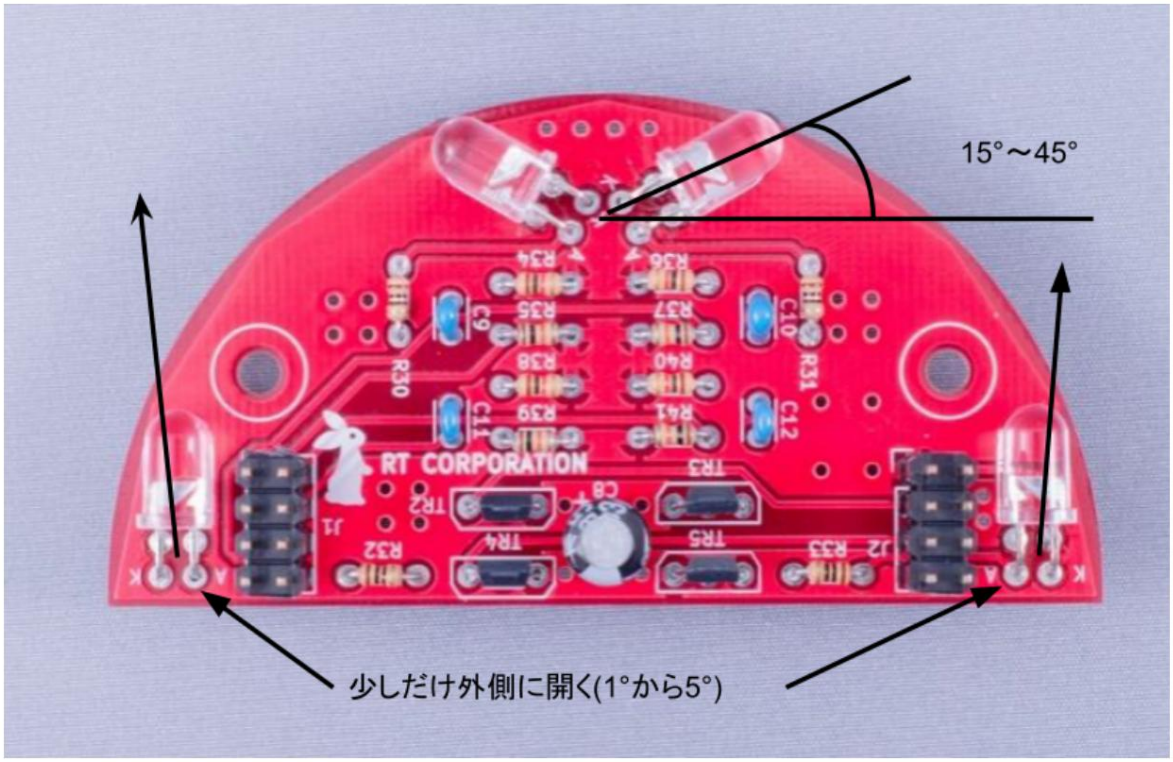


Optical axis of the sensor when viewed from above

Adjust left/right direction

The wall sensor LED and phototransistor for measuring the left and right walls should not be pointed straight ahead, but should be tilted at about 15° to 45° to the wall as shown in the figure below (it is recommended to adjust the tilt based on the silk of the board). This angle range is the controllable range for attitude control. Also, if the tilt is too large, the amount of light reflected by the wall will be reduced and distant walls will not be measured, so care must be taken. The wall sensor LED and phototransistor for measuring the front

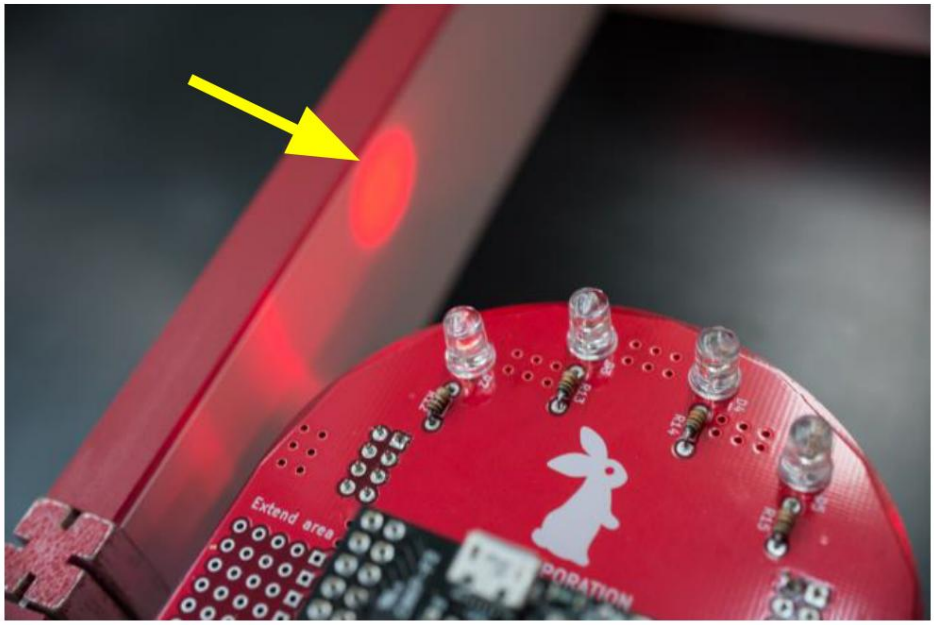
wall should not be pointed straight ahead, but should be pointed outward by about 1° to 5° as shown in the figure below. When the optical axis of the wall sensor is perpendicular to the wall, the amount of light reflected by the wall is the greatest. Conversely, the more the optical axis deviates from the perpendicular, the smaller the sensor value becomes, making it more difficult to detect the wall. If the wall sensor is pointed slightly outward, the timing at which the sensor value reaches its maximum will be different for the left and right wall sensors, so a wide range of tilts can be accommodated. The same effect can be achieved by narrowing the angle inwards, but if you are planning on driving diagonally, it is better to point it outwards.



Adjusting the sensor angle

Adjust the vertical orientation

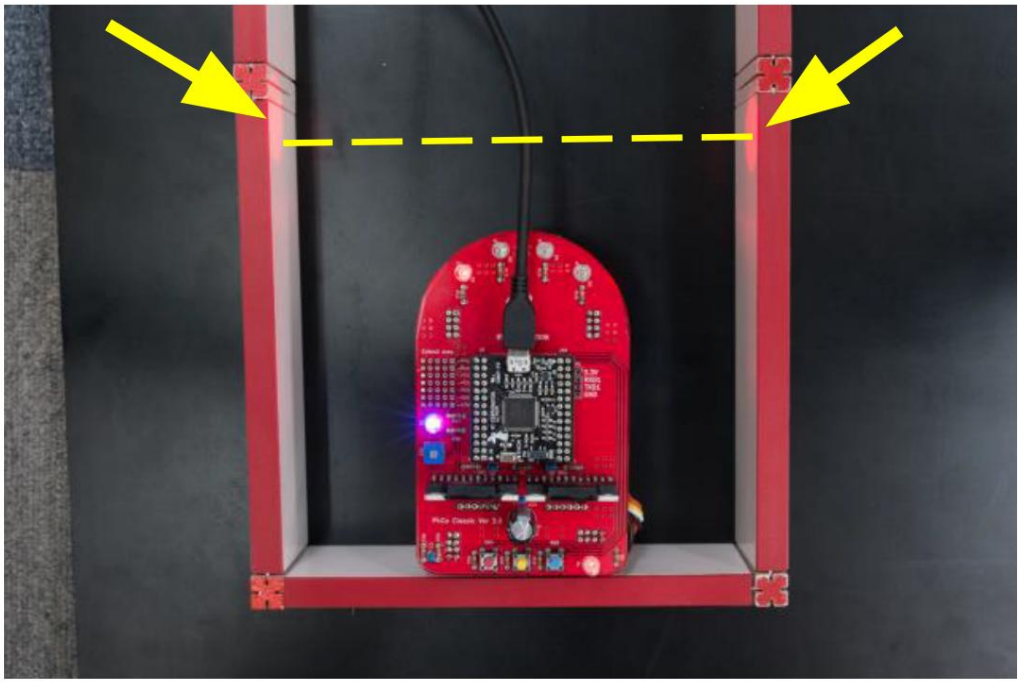
Adjust the vertical direction so that the light emitted by the wall sensor LED does not exceed the height of the wall. Basically, you can adjust it by bending it with your hands, but if that is difficult, warm it with a soldering iron and then adjust the angle.



The light from the wall sensor hitting the wall

Adjust the orientation to be symmetrical

When this product is placed in the center of the maze section, adjust the direction so that the light emitted by the wall sensor LEDs on the left and right hits the same position. In the diagram below, the light from the left wall sensor hits the pillar, but the light from the right wall sensor hits the wall in front of the pillar. Adjust the direction of the wall sensors so that they hit the same position as much as possible.



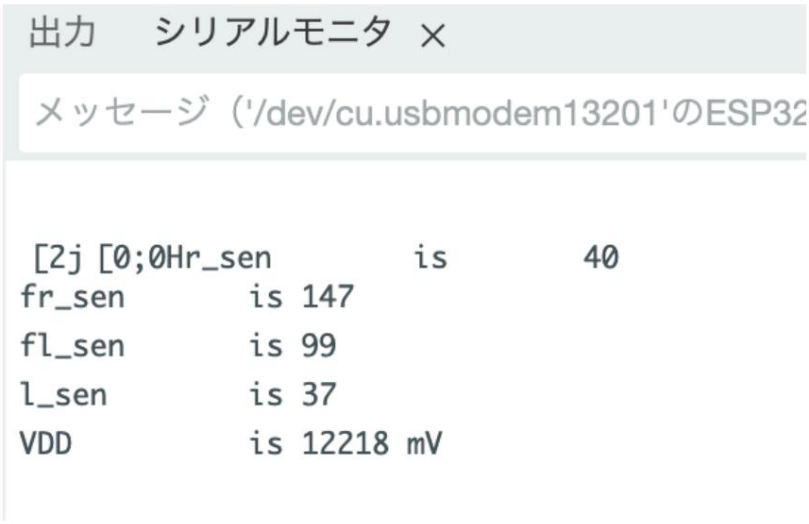
When the left and right positions are not adjusted

Set the parameters of the wall sensor

Set the parameters for the wall sensor. In the sample program, in paramter.h
Set the variables in the table below. Each threshold is used for wall detection and posture control.
In addition, the posture control during running is performed with P control with the goal of running in the center of the maze.
I am.

variable	detailed
REF_SEN_R	Target value of the right lateral sensor (measured with this product placed in the center of the section)
REF_SEN_L	Left lateral sensor target value (measured with this product placed in the center of the section)
TH_SEN_R	Threshold value for the presence or absence of a wall on the right side sensor (measured with this product attached to the left wall)
TH_SEN_L	Threshold value for the presence or absence of a wall on the left side sensor (measured with this product attached to the right wall)
TH_SEN_FR	Threshold value for the presence or absence of a wall for the right front sensor (when the product is placed 3 cm behind the pillar) (Measured at
TH_SEN_FL	Threshold value for the presence or absence of a wall for the left front sensor (when the product is placed 3 cm behind the pillar) (Measured at

Run tuning mode 1 to display the sensor values on the serial monitor of the Arduino IDE.

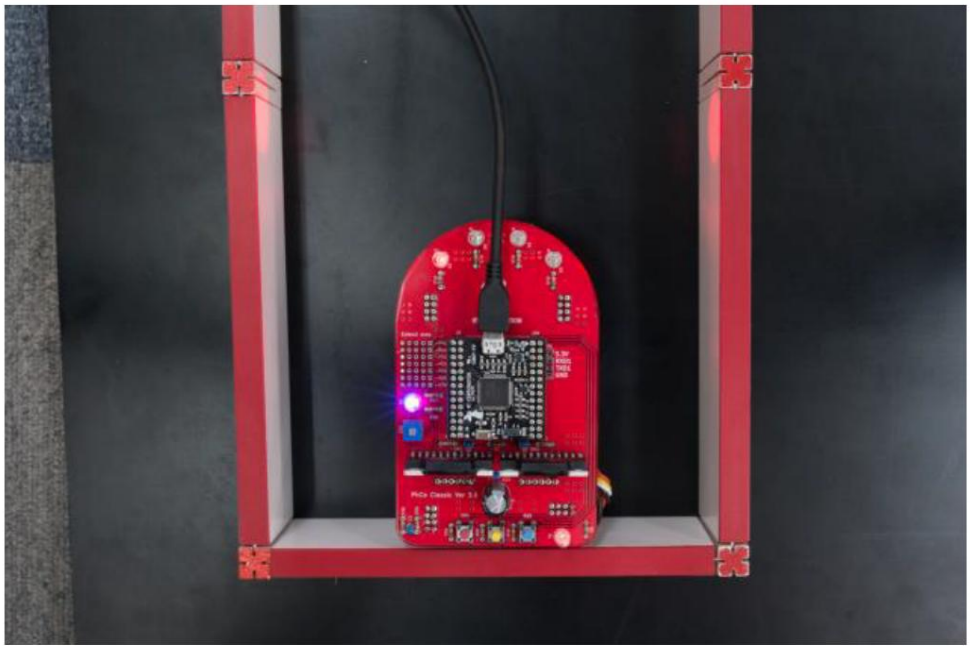


Displaying sensor values on the serial monitor

project	content
r_sen	Right lateral sensor value
fr_sen	Right front sensor value
fl_sen	Left front sensor value
l_sen	Left lateral sensor value
VDD	Battery voltage

Set the target values for the left and right wall sensors

As shown in the figure below, place this product in the center of the maze section and obtain the r_sen and l_sen values. The obtained sensor values will vary, but there is no need to measure the sensor values precisely. In paramter.h, write the obtained r_sen value in REF_SEN_R and the l_sen value in REF_SEN_L.



Arrangement for setting left and right wall sensor target values

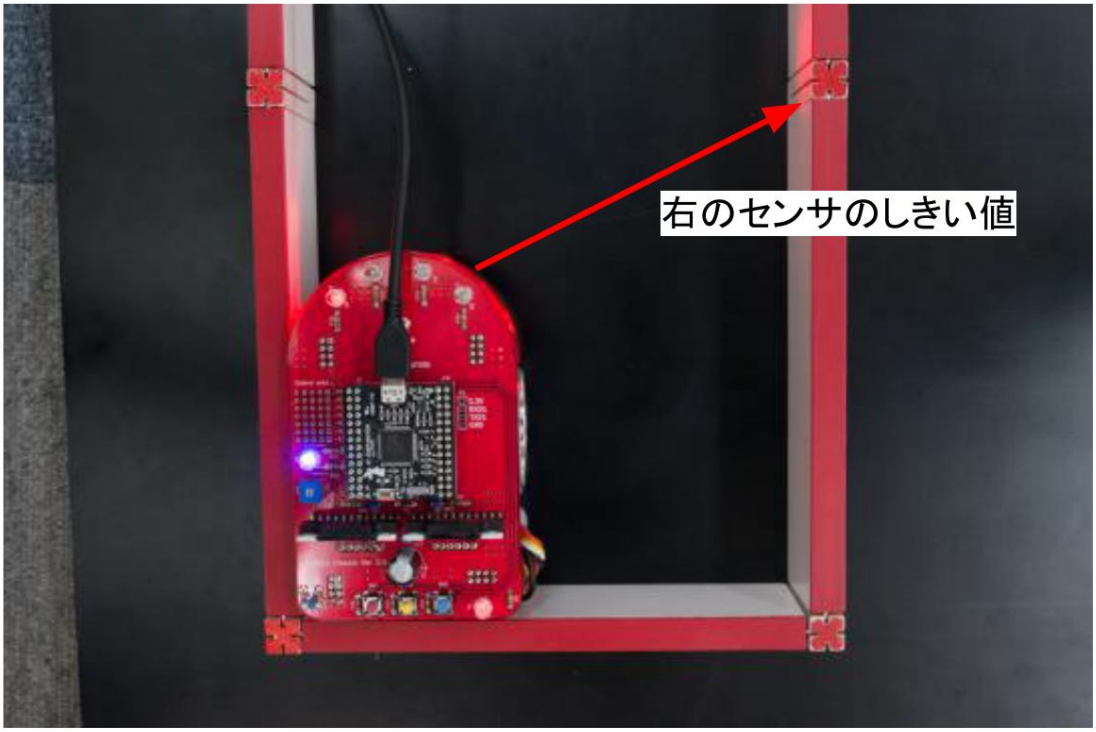
For example, if the acquired r_sen value is 550 and the l_sen value is 600, as shown in the figure below, Change 589 in REF_SEN_R to 550 and 628 in REF_SEN_L to 600.

```
11
12    #define REF_SEN_R 589 ←― 550に書き換え
13    #define REF_SEN_L 628 ←― 600に書き換え
14
```

Set the thresholds for the left and right wall sensors

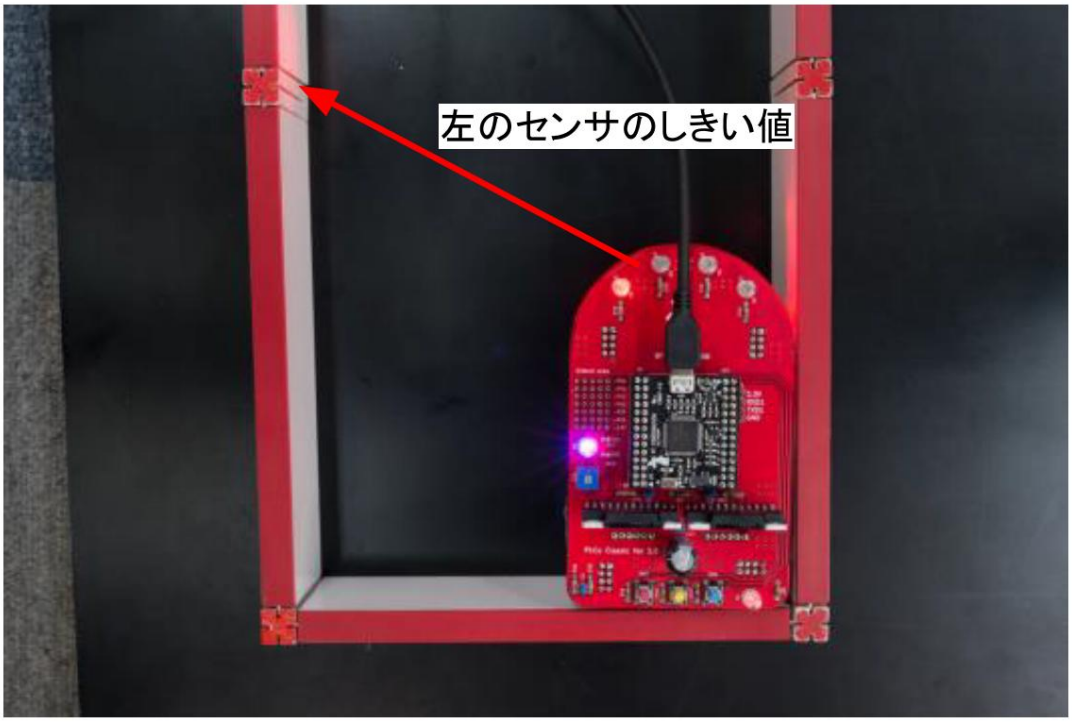
Set the threshold value for the left and right wall sensors. This threshold value is the lower limit for posture control. I will still use it.

First, place this product firmly against the wall on the left side of the section as shown in the figure below, and obtain the value of the wall sensor on the right side. At this time, check that there is a difference of 100 or more in the sensor value depending on whether or not there is a wall on the right side of the section. If there is not a difference of 100 or more, the angle of the wall sensor needs to be adjusted. If there is no difference even after adjustment, check that there is no poor solder contact and that the orientation of the element is correct. Once the r_sen value can be obtained without any problems, rewrite the value of TH_SEN_R in paramter.h.



Arrangement for setting right wall sensor threshold

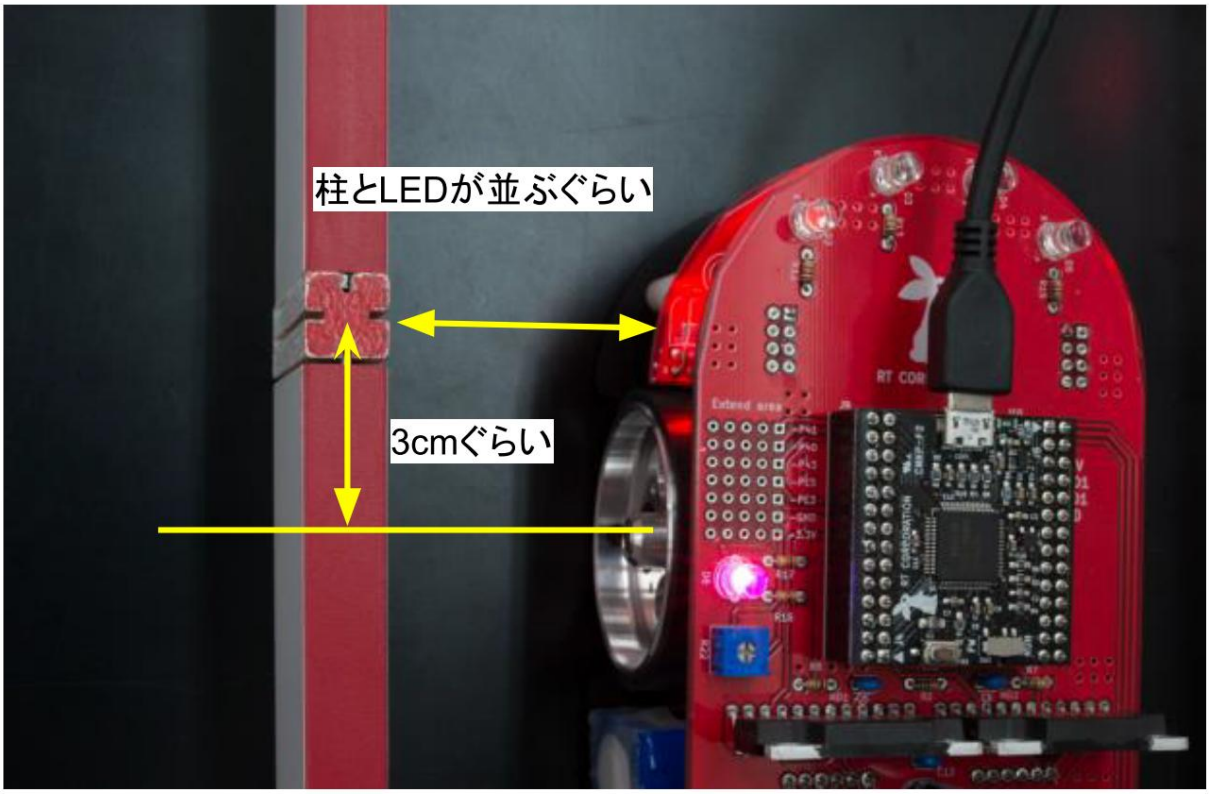
Next, use the same procedure to set the threshold for the left wall sensor. As shown in the figure below, place this product firmly against the right wall of the section and obtain the value of the left wall sensor. Once you have **obtained** the l_sen value without any problems , rewrite the value of TH_SEN_L in paramter.h.



Arrangement for setting left wall sensor threshold

Set the threshold for the wall sensor that measures the front wall

Next, use the same procedure to set the threshold value for the wall sensor that measures the front wall. As shown in the figure below, place the product so that the wall sensor LED is lined up with the pillars of the maze, and acquire the sensor value. At this time, check that there is a difference of 50 or more in the sensor value with and without a front wall. If there is no difference, you will need to adjust the angle of the wall sensor. Once you have acquired the sensor value without any problems, edit the **Write the fr_sen value in TH_SEN_FR and the fl_sen value in TH_SEN_FL.** In the sample program for this product, it is assumed that the detection of the wall ahead will be performed when the motor's rotation axis lines up with the pillars of the maze, as shown in the diagram below. However, since there is a deviation of about 3 cm between the acquisition position of the sensor value assumed by the microcontroller and the actual acquisition position while driving, it is necessary to acquire the sensor value when the wall sensor LED lines up with the pillar. In the sample program, the distance correction by the wall sensor that measures the wall ahead is not performed, so there is no need to consider the exact distance.



Arrangement for setting a wall sensor threshold for measuring a front wall

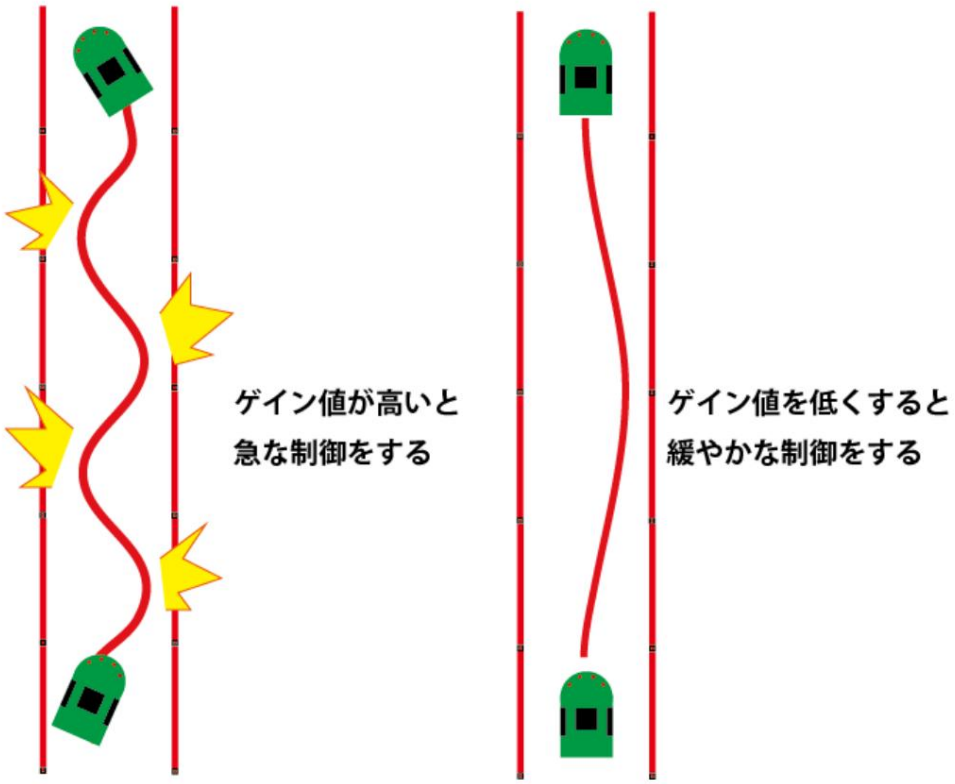
Once you have set the parameters for each sensor, build the program. Then, Let's move on to adjusting the input.

Gain Adjustment

In the sample program uROS_STEP8_micormouse, P control is used to move through the center of the maze. By adjusting the gain in this P control, you can change how fast the robot approaches the target value.

If the gain is set too high, the robot may become unstable around the target value or may suddenly lose its position at the edge of the wall. On the other hand, if the gain is too low, the robot may be controlled slowly to the target value, which may increase the possibility of hitting the wall. **The gain is set by**

CON_WALL_KP in parameter.h. The default value in the sample program is 0.3. If the gain is set small, the time to converge will be longer, and if the gain is set large, the time to converge will be faster. Also, the larger the gain, the more torque is required, and when the motor speed increases, step-out is more likely to occur, so care must be taken (stepping motors rotate the rotor in sync with pulses from the microcontroller. Step-out refers to the phenomenon in which the motor stops rotating when the motor speed and the pulses from the microcontroller are no longer synchronized).



The time it takes to converge varies depending on the gain value.

Execute adjustment mode 2 of the sample program to adjust the gain. Place the product near the left or right wall of the maze and run it across nine sections. Adjust the gain so that it converges to the target value after running three or four sections. If it is not possible to prepare a maze with nine sections, make adjustments using the longest maze that you can prepare. The running distance can be changed by specifying the number of sections as an argument to the straightCheck function in adjust.ino.

```
164  unsigned char execByModeAdjust(unsigned char mode)
165  {
166    disableMotor();
167    switch (mode) {
168      case 1:
169        viewAdc();
170        break;
171      case 2:
172        straightCheck(9);
173        break;
174
175      case 3:
176        rotationCheck();
177        break;
```

持っている迷路の
区画数に合わせる

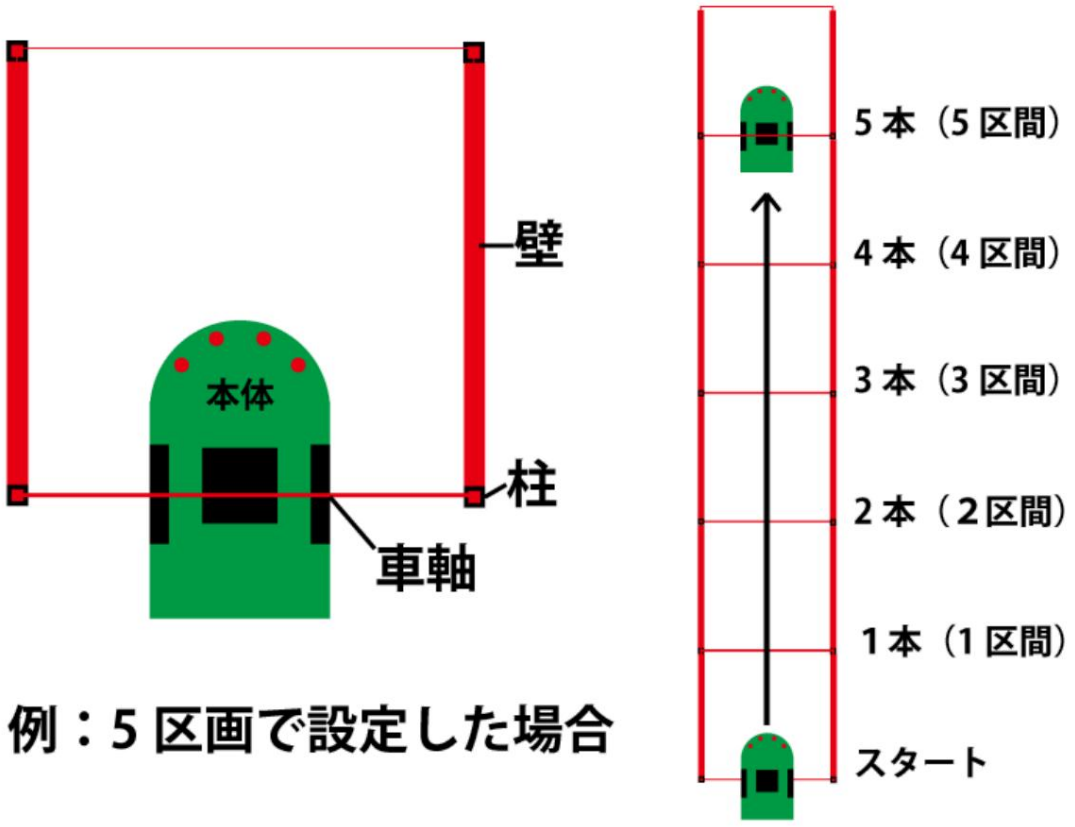
Change in mileage

Tire diameter parameter adjustment

This section describes how to adjust the tire diameter parameter. The tire diameter parameter is **It's set in paramer.h as TIRE_DIAMETER.**

First, measure the tire diameter in millimeters to two decimal places using a measuring tool such as a caliper or ruler. Then, run the product repeatedly in a straight section. At this time, fine-tune the decimal points of the tire diameter parameter so that it travels the specified distance. To reduce measurement errors, prepare a maze that is as long as possible. Also, since tire friction and the degree of slip during driving change depending on the environment, adjust the parameters each time the environment changes.

As an example, we will show you how to adjust the tire diameter parameters by running this product for 5 blocks. As shown in the figure below, align the motor's axis with the pillars of the maze, run the robot through five sections, and then check how far the robot's final position has shifted based on the motor's axis. If the robot's final position is closer to the target, reduce the tire diameter parameter. If the robot has advanced too far, increase the tire diameter parameter.



How to adjust tire diameter parameters

How should the tire diameter parameters be changed to account for the final position error? It can be found by calculation.

As an example, consider the case where the vehicle travels between five sections and stops 10 mm short of the target position. In the sample program, the default value of the tire diameter parameter is 48 mm. Also, the distance of one section is 180 mm, so the distance of five sections is 5 × 180 = 900 mm. Stopping 10 mm short of the target position means that the vehicle only traveled 900 ÿ 10 = 890 mm. This means that there is a discrepancy between the tire diameter parameter written in the program and the actual tire diameter. To

calculate the tire diameter after adjustment, set up the following equation.

$$\frac{\text{Current tire diameter Target mileage}}{\text{mileage}} = \frac{\text{Adjusted tire diameter Actual mileage}}{\text{mileage}}$$

$$\frac{48}{900} = \frac{}{890}$$

$$= \frac{48\times890}{900}$$

Calculating this formula gives us = 47. 47[mm]. Put this value in parameter.h Write this in TIRE_DIAMETER, build the program, and run the maze again. Repeat this adjustment until the error from the target running distance is within 1 mm.

Dust on the maze or tires will affect parameter adjustments, so it is necessary to remove dust before running this product through a maze. Dust on the tires can be removed with masking tape as shown in the figure below.

Removing dust with masking tape

Tread parameter adjustment

This section explains how to adjust the tread (distance between tires) parameter. The tread parameter is set as TREAD_WIDTH in paramter.h. By adjusting the parameter, you can change the turning angle during a pivot turn. If this adjustment is insufficient, the posture will shift every time you turn. In a maze with many straight routes, posture control can correct the posture to some extent, but in a maze with many turns, errors may accumulate and cause the vehicle to collide with a wall. To adjust the tread parameter, run the sample program's

adjustment mode 3 and make 90° turns clockwise x 8 times (i.e., a total of 720° turns). Then adjust so that the posture direction is the same before and after the turn. Since it is not clear at which point the tire is touching the ground to make the turn, you need to make the adjustment steadily and steadily. If the turning angle is insufficient, increase the value of TREAD_WIDTH, and if it turns too much, decrease TREAD_WIDTH.

How to change the tread parameters to account for the error in the final turning angle can be calculated. As an example,

consider the case where adjustment mode 3 is performed and the turning angle is 10° short. In adjustment mode 3, the wheel rotates twice,

resulting in a turn of 360 × 2 = 720°. The default value of the tread parameters in the sample program is 64[mm]. Therefore, the target wheel turning distance is 720 ÷ 360 × 64ÿ[mm], but

since it was 10° short, it is 710 ÷ 360 × ÿ[mm]. In this case, if the formula is 720 ÷ 360 × 64ÿ = 710 ÷ 360 × ÿ, the calculation is = 720 ÷ 710 × 64 = 64.9[mm].

Rewrite TREAD_WIDTH with the calculated tread value, build the program, and then run adjustment mode 3 again to check the turning angle. Repeat this adjustment until the error is sufficiently small.

Maze running

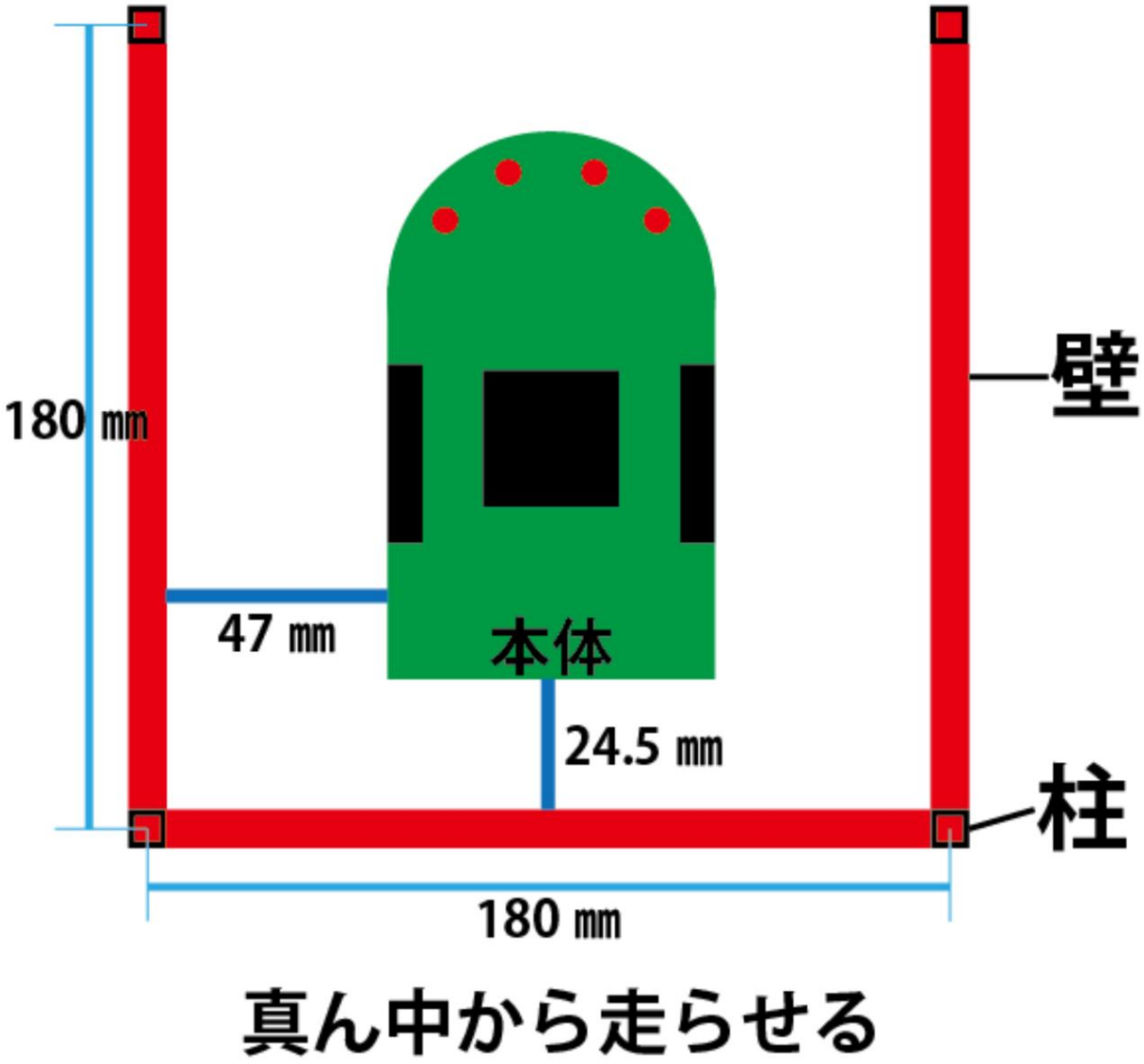
Now that we are ready to run this product through a maze, let's actually run it. In the sample program uROS_STEP8_micormouse, the goal coordinates are (x=3, y=3) by default, but you can change the goal coordinates to suit your maze. For example, for a micromouse competition, set the goal coordinates to (x=7, y=7), (x=8, y=7), (x=7, y=8), or (x=8, y=8). The goal coordinates can be set with GOAL_X and GOLA_Y in paramter.h. In a 5x4 maze like the one below, if the start is at the bottom left and the goal is at the top right, the start coordinates are (x=0, y=0) and the goal coordinates are (x=4, y=3). Please note that the counting of the sections starts from 0, based on the starting point.

5x4 maze

In the sample program's driving mode 2 (search driving using Adachi's method), when the goal coordinates are reached and when the robot returns to the start coordinates, the explored maze information is saved in the microcontroller's flash memory. This means that even if the robot hits a wall on the way back from the goal coordinates to the start coordinates and gets stuck, forcing the user to press the reset button, the robot can still use the maze information saved in the flash memory to navigate the shortest route. In the sample program, the robot

starts driving when the robot is placed in the center of the area, as shown in the figure below.

Distance calculations and other procedures are performed assuming that the course will begin at the same time.



Starting position of this product

Revision History

Issue date **(YY/MM/DD)**	Version	Revision details
24/7/8	1.0	First edition

Copyright and Intellectual Property Rights