

2023학년도

햄스터 봇

충북교육연구정보원 영재교육원 정보강사 박정진

이동하고 회전하기

앞으로 1초 이동하기

- 햄스터 로봇이 1초 동안 앞으로 이동한 후 정지하게 해봅시다.

바퀴의 속도는 -100부터 100까지의 값을 가지는데 최대 속도에 대한 % 값입니다.

양수 값이면 바퀴가 앞으로, 음수 값이면 뒤로 회전하고, 0이면 멈춥니다.

햄스터 로봇이 앞으로 이동하게 하려면 왼쪽 바퀴와 오른쪽 바퀴의 속도를 같은 양수 값으로 하면 됩니다.

wait() 함수를 사용하면 코드의 다음 줄로 넘어가지 않고 일정 시간(msec) 동안 기다릴 수 있습니다.



- 소스코드

```
#include "roboid.h"

int main(int argc, char *argv[]) {
    hamster_create();

    // 왼쪽 바퀴와 오른쪽 바퀴의 속도를 30으로 설정한다.
    hamster_wheels(30, 30);
    wait(1000); // 1초 기다린다.
    hamster_stop(); // 정지한다.

    // 통신 연결을 종료하고 메모리를 해제한다.
    dispose_all();

    return 0;
}
```

뒤로 1초 이동하기

- 햄스터 로봇이 뒤로 이동하게 하려면 왼쪽 바퀴와 오른쪽 바퀴의 속도를 같은 음수값으로 하면 됩니다.

바퀴의 속도를 0부터 -100까지 변경하면서 로봇이 이동하는 속도가 어떻게 달라지는지 관찰해 봅시다.



■ 소스코드

```
#include "roboid.h"

int main(int argc, char *argv[]) {
    hamster_create();

    // 왼쪽 바퀴와 오른쪽 바퀴의 속도를 -30으로 설정한다.
    hamster_wheels(-30, -30);
    wait(1000); // 1초 기다린다.
    hamster_stop(); // 정지한다.

    // 통신 연결을 종료하고 메모리를 해제한다.
    dispose_all();

    return 0;
}
```

제자리에서 5초 돌기 (스핀)

- 왼쪽 바퀴와 오른쪽 바퀴의 속도를 크기는 같게 하고 부호만 반대로 하면 제자리에서 회전합니다. 로봇이 회전하는 중심점은 왼쪽 바퀴와 오른쪽 바퀴의 축을 연결하는 직선의 중심입니다.



■ 소스코드

```
#include "roboid.h"

int main(int argc, char *argv[]) {
    hamster_create();

    // 왼쪽 바퀴의 속도를 -30,
    // 오른쪽 바퀴의 속도를 30으로 설정한다.
    hamster_wheels(-30, 30);
    wait(5000); // 5초 기다린다.
    hamster_stop(); // 정지한다.

    // 통신 연결을 종료하고 메모리를 해제한다.
    dispose_all();

    return 0;
}
```

한쪽 바퀴를 축으로 5초 회전하기 (피봇 턴)

- 왼쪽 바퀴를 정지하고 오른쪽 바퀴에만 속도 값을 주면 왼쪽 바퀴를 중심으로 회전합니다.



- 오른쪽 바퀴를 중심으로 회전하게 해보고, 시계 방향, 반시계 방향 등 다양한 회전을 만들어 보시다.
- 왼쪽 바퀴 또는 오른쪽 바퀴, 하나의 값만 설정하기 위해서는 hamster_left_wheel() 또는 hamster_right_wheel() 함수를 사용하여도 됩니다.

■ 소스코드

```
#include "roboid.h"

int main(int argc, char *argv[]) {
    hamster_create();

    // 왼쪽 바퀴의 속도를 0,
    // 오른쪽 바퀴의 속도를 30으로 설정한다.
    hamster_wheels(0, 30);
    wait(5000); // 5초 기다린다.
    hamster_stop(); // 정지한다.

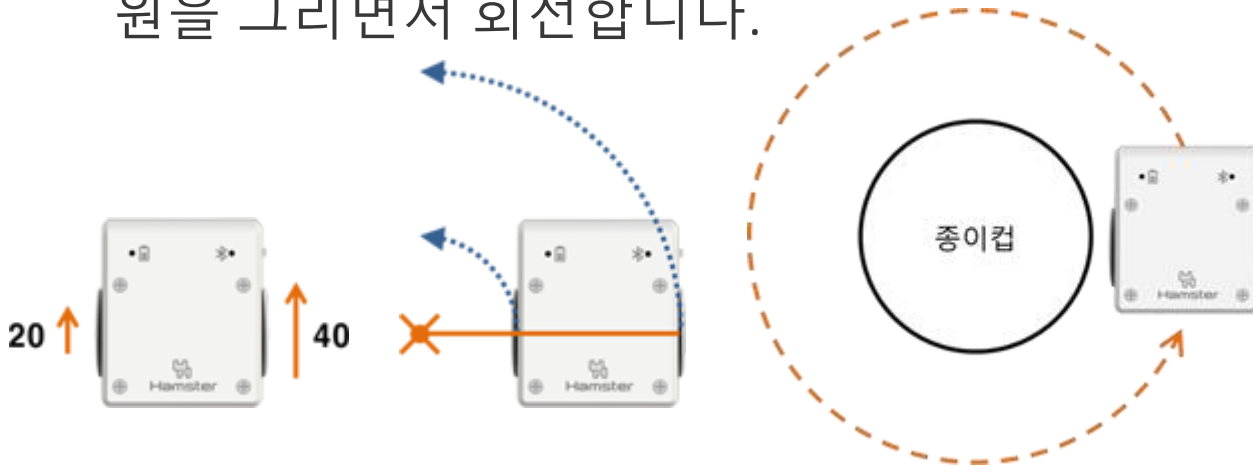
    // 통신 연결을 종료하고 메모리를 해제한다.
    dispose_all();

    return 0;
}
```

- hamster_left_wheel(30); // 왼쪽 바퀴의 속도를 30으로 설정한다.
- hamster_right_wheel(30); // 오른쪽 바퀴의 속도를 30으로 설정한다.

둥글게 5초 회전하기 (라운드 턴)

- 왼쪽 바퀴와 오른쪽 바퀴의 속도 크기를 다르게 하면 원을 그리면서 회전합니다. 두 바퀴 간의 속도 차이가 작을수록 큰 원을 그리면서 회전합니다.



- 왼쪽 바퀴와 오른쪽 바퀴의 속도를 변경하면서 로봇이 회전하는 원의 크기가 어떻게 달라지는지 관찰해 봅시다.
- 종이컵의 바깥을 돌 수 있도록 왼쪽 바퀴와 오른쪽 바퀴의 속도를 조정해 봅시다.

■ 소스코드

```
#include "roboid.h"

int main(int argc, char *argv[]) {
    hamster_create();

    // 왼쪽 바퀴의 속도를 20,
    // 오른쪽 바퀴의 속도를 40으로 설정한다.
    hamster_wheels(20, 40);
    wait(5000); // 5초 기다린다.
    hamster_stop(); // 정지한다.

    // 통신 연결을 종료하고 메모리를 해제한다.
    dispose_all();

    return 0;
}
```

LED 켜고 앞으로 1초 이동하기

- 햄스터 로봇이 양쪽 LED를 파란색으로 켜고 1초 동안 앞으로 이동하게 해봅시다.

```
#include "roboid.h"

int main(int argc, char *argv[]) {
    hamster_create();

    // 왼쪽 LED와 오른쪽 LED를 파란색으로 켜다.
    hamster_leds("blue", "blue");
    hamster_wheels(30, 30); // 앞으로 이동한다.
    wait(1000); // 1초 기다린다.

    // 왼쪽 LED와 오른쪽 LED를 끈다.
    hamster_leds("off", "off");
    hamster_stop(); // 정지한다.

    dispose_all();
    return 0;
}
```

- 햄스터 로봇의 LED는 7가지 색상을 표현할 수 있으며, LED를 끄기 위해서는 "off"를 입력하면 됩니다. 대소문자는 구분하지 않습니다.

LED 색상	설명
"off"	LED를 끈다.
"red"	LED를 빨간색으로 켜다. (R: 255, G: 0, B: 0)
"yellow"	LED를 노란색으로 켜다. (R: 255, G: 255, B: 0)
"green"	LED를 초록색으로 켜다. (R: 0, G: 255, B: 0)
"sky blue"	LED를 하늘색으로 켜다. (R: 0, G: 255, B: 255)
"blue"	LED를 파란색으로 켜다. (R: 0, G: 0, B: 255)
"purple"	LED를 보라색으로 켜다. (R: 255, G: 0, B: 255)
"white"	LED를 하얀색으로 켜다. (R: 255, G: 255, B: 255)

소리 내며 뒤로 1초 이동하기

- 햄스터 로봇이 소리를 내면서 뒤로 1초 동안 이동하게 해봅시다.

버저 소리의 음 높이를 주파수[Hz]로 입력하면 됩니다. 버저 소리를 끄기 위해서는 0을 입력하면 됩니다.

버저 소리의 음 높이는 소수점 둘째 자리까지 입력할 수 있습니다.

- 소스 코드

```
#include "roboid.h"

int main(int argc, char *argv[]) {
    hamster_create();

    // 버저 소리의 음 높이를 1000 Hz로 설정한다.
    hamster_buzzer(1000);
    hamster_wheels(-30, -30); // 뒤로 이동한다.

    wait(1000); // 1초 기다린다.

    hamster_buzzer(0); // 버저 소리를 끈다.
    hamster_stop(); // 정지한다.

    dispose_all();
    return 0;
}
```

오리걸음 경주

- 오리걸음 경주는 뒤뚱뒤뚱 달려가서 누가 먼저 목표 지점에 도착하는지 시합하는 경주입니다.

뒤뚱뒤뚱 달려가는 동작을 만들어야겠습니다.

왼쪽 바퀴를 중심으로 왼쪽으로 회전하다가 오른쪽 바퀴를 중심으로 오른쪽으로 회전하는 것을 반복하면 뒤뚱뒤뚱 앞으로 달려가게 됩니다.



방향을 바꾸는 명령은 어떻게 전달하면 좋을까요?
방향을 자주 바꾸어야 하기 때문에 컴퓨터 키보드의 스페이스 키를 눌러 명령하도록 해봅시다.

```
#include <conio.h>
#include "roboid.h"
#define ENTER 13

int main(int argc, char *argv[]) {
    int key, left = 0;
    hamster_create();

    while(1) {
        key = _getch(); // 키보드 입력을 얻는다.
        if(key == ENTER) { // 엔터 키를 눌렀으면
            break; // 종료한다.
        } else if(key == ' ') { // 스페이스 키를 눌렀으면
            left = 1 - left; // 방향을 반대로 한다.

            if(left) { // 왼쪽 방향이면
                hamster_wheels(0, 50);
            } else { // 오른쪽 방향이면
                hamster_wheels(50, 0);
            }
        }
        wait(20); // 너무 빨리 반복하지 않도록 한다.
    }

    dispose_all(); // 통신 연결을 종료하고 메모리를 해제한다.
    return 0;
}
```

햄스터 조종기

- 키보드의 방향 키를 사용하여 햄스터 조종기를 만들어 봅시다.

위쪽 방향 키는 앞으로 이동, 아래쪽 방향 키는 뒤로 이동, 왼쪽 방향 키는 왼쪽으로 회전, 오른쪽 방향 키는 오른쪽으로 회전하도록 하고, 스페이스 키를 누르면 정지하도록 합시다. 키보드의 방향 키는 특수 키이기 때문에 `_getch()` 함수를 두 번 호출해야 합니다.

```
#include <conio.h> // _getch() 함수 때문에 필요하다.  
#include "roboid.h"
```

```
#define ENTER 13  
#define UP 72  
#define DOWN 80  
#define LEFT 75  
#define RIGHT 77
```

```
int main(int argc, char *argv[]) {  
    int key;  
    hamster_create();  
  
    while(1) {  
        key = _getch(); // 키보드 입력을 얻는다.  
        if(key == ENTER) { // 엔터 키를 눌렀으면  
            break; // 종료한다.  
        } else if(key == ' ') { // 스페이스 키를 눌렀으면  
            hamster_stop(); // 정지한다.  
        } else if(key == 0xE0 || key == 0) { // 특수 키인 경우  
            key = _getch(); // 한 번 더 키보드 입력을 얻는다.  
            if(key == UP) { // 위쪽 방향 키  
                hamster_wheels(30, 30); // 앞으로 이동한다.  
            } else if(key == DOWN) { // 아래쪽 방향 키  
                hamster_wheels(-30, -30); // 뒤로 이동한다.  
            } else if(key == LEFT) { // 왼쪽 방향 키  
                hamster_wheels(-30, 30); // 왼쪽으로 회전한다.  
            } else if(key == RIGHT) { // 오른쪽 방향 키  
                hamster_wheels(30, -30); // 오른쪽으로 회전한다.  
            }  
        }  
        wait(20); // 너무 빨리 반복하지 않도록 한다.  
    }  
    dispose_all(); // 통신 연결을 종료하고 메모리를 해제한다.  
    return 0;  
}
```

라인 트레이서

개요

- 왜 내 로봇은 똑바로 가지 못할까요? 로봇의 양쪽 바퀴는 똑같지 않습니다. 양쪽 모터의 성능도 다르고 기어가 뻑뻑하거나 느슨한 정도, 바퀴와 바닥 간의 마찰력, 타이어의 마모 상태 등에 따라 실제로 바퀴가 이동하는 속도는 달라질 수 있습니다. 양쪽 바퀴가 이동하는 속도가 조금만 달라도 로봇은 똑바로 이동하지 못합니다. 이것이 실제의 로봇입니다.
- 사람은 똑바로 걸을 수 있을까요? 아무도 없는 복도를 눈 감고 걸어 가봅시다. 복도의 중앙에서 출발하면 아마 20 걸음도 채 못 가고 벽과 부딪칠 것입니다. 이번에는 눈을 뜨고 복도를 보면서 걸어 가봅시다. 복도 끝까지 문제 없이 갈 수 있을 것입니다. 사람은 걸으면서 주위의 상황을 끊임없이 인지하고 자신의 방향을 조정하기 때문입니다.
- 센서를 사용하면 똑바로 가지 못하는 로봇을 잘 조종하여 인간처럼 가게 할 수 있습니다. 하지만 지능이 없는 로봇은 이 일도 매우 어려운 일입니다. 이러한 지능을 직접 코드로 작성하여 햄스터 로봇을 똑똑하게 만들어 줍시다.
- 라인 트레이서는 일종의 지능형 로봇으로, 센서를 통해 검출된 선의 정보를 처리하여 정해진 선을 따라 이동하는 로봇을 말합니다. 무인 자동화 생산 공정에서 생산된 물건을 운반하는 용도로 사용될 수 있습니다.
우선 간단하게 햄스터 로봇의 바닥 센서 한 개를 사용하여 주행하는 방법을 알아보도록 하겠습니다.

왼쪽 바닥 센서를 사용 검은선의 오른쪽 테두리 따라가기

- A4 용지를 준비하고 햄스터 로봇이 이동할 직선을 검은색 테이프 또는 펜으로 그립니다. 미리 제작된 파일을 프린터로 인쇄해도 됩니다.
- 라인 트레이서의 원리는 의외로 간단합니다. 햄스터 로봇의 왼쪽 바닥 센서가 하얀색 종이 위에 있으면 검은색 선의 오른쪽 테두리로 다가가기 위해 왼쪽으로 움직이고, 왼쪽 바닥 센서가 검은색 선 위에 있으면 오른쪽으로 움직이면 됩니다.
- 이를 반복하면 다음 그림과 같이 검은색 선의 오른쪽 테두리를 따라 앞으로 이동할 수 있습니다.
- 원리는 간단한데 이를 실제로 구현하기 위해서는 두 가지가 필요합니다.
 - 왼쪽 바닥 센서가 하얀색 종이 위에 있는지, 검은색 선 위에 있는지 알 수 있어야 합니다.
 - 햄스터 로봇을 왼쪽 또는 오른쪽으로 움직여야 합니다.



왼쪽 바닥 센서를 사용 검은선의 오른쪽 테두리 따라가기

- 우선 왼쪽 바닥 센서가 하얀색 종이 위에 있는지, 검은색 선 위에 있는지 알기 위해 왼쪽 바닥 센서의 값을 관찰해 봅시다. 왼쪽 바닥 센서가 하얀색 종이 위에 있을 때와 검은색 선 위에 있을 때 센서 값이 어떻게 다른 지 확인해 봅시다.
- 측정된 센서 값으로부터 햄스터 로봇의 왼쪽 바닥 센서가 하얀색 종이 위에 있는지, 검은색 선 위에 있는지 판단하기 위해서는 크기를 비교하면 됩니다. 기준이 되는 숫자가 필요한데 앞에서 측정한 두 개의 센서 값의 중간 값을 사용해 봅시다. 하얀색 종이 위에 있을 때 센서 값이 100, 검은색 선 위에 있을 때 센서 값이 0이었다면 기준이 되는 숫자는 50으로 하면 됩니다.

■ 소스코드

```
#include <stdio.h>
#include "roboid.h"

int main(int argc, char *argv[]) {
    hamster_create();

    while(1) {
        printf("%d\n", hamster_left_floor());
        wait(20); // 너무 빨리 반복하지 않도록 한다.
    }

    return 0;
}
```

왼쪽 바닥 센서를 사용 검은선의 오른쪽 테두리 따라가기

- 왼쪽 바닥 센서가 하얀색 종이 위에 있으면, 즉 센서 값이 50보다 크면 왼쪽으로 회전하고, 아니면 오른쪽으로 회전하도록 코드를 작성해 봅시다.
- 소스코드

```
if(hamster_left_floor() > 50) {  
    hamster_wheels(0, 30);  
} else {  
    hamster_wheels(30, 0);  
}
```

- 다음 그림과 같이 햄스터 로봇의 왼쪽 바닥 센서를 검은색 선의 오른쪽 테두리에 올려 놓고 작성한 코드를 실행해 봅시다



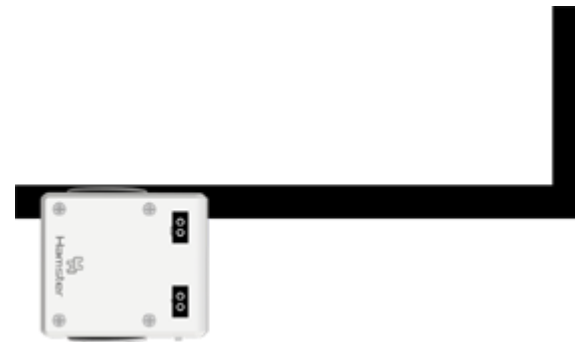
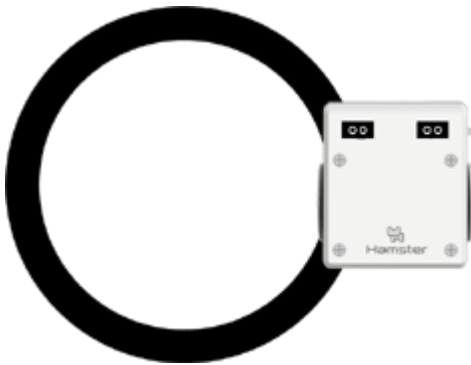
```
#include "roboid.h"  
  
int main(int argc, char *argv[]) {  
    hamster_create();  
  
    while(1) {  
        if(hamster_left_floor() > 50) {  
            hamster_wheels(0, 30);  
        } else {  
            hamster_wheels(30, 0);  
        }  
        wait(10); // 너무 빨리 반복하지 않도록 한다.  
    }  
  
    return 0;  
}
```

왼쪽 바닥 센서를 사용하여 검은색 선의 오른쪽 테두리를 따라 곡선 및 90도 주행하기

- 직선을 따라 주행하는 것은 너무 단순합니다. 곡선을 따라 주행하거나 90도 꺾어진 길도 잘 달려갈 수 있어야 합니다.
- 우리가 복도를 달려가다가 왼쪽 또는 오른쪽으로 꺾어진 길을 만나면 어떻게 하는지 생각해 봅시다. 속도를 줄여서 모퉁이를 돈 다음 다시 달려가지 않을까요? 하지만 모퉁이가 어디쯤 있는지 알 수 있어야 속도를 줄일 수 있습니다. 앞에서 실습한 햄스터 로봇은 바닥만 보면서 움직입니다. 우리도 바닥만 보면서 복도를 달려간다면 언제 모퉁이가 나타날지 알 수가 없어서 빨리 달리지는 못할 것 같습니다.
- 작성한 코드를 그대로 사용하여 곡선을 따라 주행할 수 있는지 실험해 봅시다. 선을 벗어나지 않고 잘 주행할 수 있도록 햄스터 로봇이 왼쪽 또는 오른쪽으로 회전하는 속도를 조정하면 됩니다.

왼쪽 바닥 센서를 사용하여 검은색 선의 오른쪽 테두리를 따라 곡선 및 90도 주행하기

- 큰 원에서 시작하여 원의 크기를 조금씩 줄이면서 실험해 봅시다. 원이 클 때와 작을 때 햄스터 로봇의 회전 속도를 어떻게 해야 하는지 실험하고 관찰하여 원의 크기와 회전 속도 간의 관계를 알아봅시다.
- 왼쪽으로 90도 꺾어진 길은 잘 따라갈 수 있을까요? 문제 없이 잘 따라간다면 설명을 듣기 전에, 작성한 코드와 햄스터 로봇의 동작을 관찰하면서 그 이유를 추론하고 토의해 봅시다.



센서 두 개를 사용한 라인 트레이서

- 기본적으로는 앞으로 이동하면서 햄스터 로봇의 왼쪽 바닥 센서가 검은색 선 위에 있으면 중앙으로 가기 위해 왼쪽으로 움직이고, 오른쪽 바닥 센서가 검은색 선 위에 있으면 중앙으로 가기 위해 오른쪽으로 움직이면 됩니다.



- 원리는 간단한데 이를 실제로 구현하기 위해서는 두 가지가 필요합니다.

왼쪽 바닥 센서 또는 오른쪽 바닥 센서가 검은색 선 위에 있는지 알 수 있어야 합니다.

햄스터 로봇을 왼쪽 또는 오른쪽으로 움직여야 합니다.

센서 두 개를 사용한 라인 트레이서

- ❖ 우선 왼쪽 바닥 센서와 오른쪽 바닥 센서의 값을 관찰해 봅시다. 바닥 센서가 하얀색 종이 위에 있을 때와 검은색 선 위에 있을 때 센서 값이 어떻게 다른지 확인해 봅시다.

```
#include <stdio.h>
#include "roboid.h"

int main(int argc, char *argv[]) {
    hamster_create();

    while(1) {
        printf("%d, %d\n", hamster_left_floor(), hamster_right_floor());
        wait(20); // 너무 빨리 반복하지 않도록 한다.
    }
    return 0;
}
```

- 측정된 센서 값으로부터 햄스터 로봇의 바닥 센서가 하얀색 종이 위에 있는지, 검은색 선 위에 있는지 판단하기 위해서는 크기를 비교하면 됩니다. 기준이 되는 숫자가 필요한데 앞에서 측정한 센서 값의 중간 값을 사용해 봅시다. 하얀색 종이 위에 있을 때 센서 값이 100, 검은색 선 위에 있을 때 센서 값이 0이었다면 기준이 되는 숫자는 50으로 하면 됩니다.

센서 두 개를 사용한 라인 트레이서

- 둘 다 검은색 선 위에 없는 경우가 있기 때문에 왼쪽 바닥 센서가 검은색 선 위에 있는 경우와 오른쪽 바닥 센서가 검은색 선 위에 있는 경우를 모두 검사해 주어야 합니다. 앞으로 이동하면서 왼쪽 바닥 센서가 검은색 선 위에 있으면, 즉 왼쪽 바닥 센서의 값이 50보다 작으면 왼쪽으로 회전하고, 오른쪽 바닥 센서가 검은색 선 위에 있으면, 즉 오른쪽 바닥 센서의 값이 50보다 작으면 오른쪽으로 회전하도록 코드를 작성해 봅시다.

■ 소스코드

```
#include "roboid.h"

int main(int argc, char *argv[]) {
    hamster_create();

    while(1) {
        hamster_wheels(30, 30);
        if(hamster_left_floor() < 50) {
            hamster_wheels(-30, 30);
        } else if(hamster_right_floor() < 50) {
            hamster_wheels(30, -30);
        }

        wait(10); // 너무 빨리 반복하지 않도록 한다.
    }
    return 0;
}
```

센서 두 개를 사용한 라인 트레이서 (두번째 방법)

- 다시 원점으로 돌아가서 아래 그림을 보고 다르게 생각해 봅시다. 기본적으로는 앞으로 이동하면서 햄스터 로봇의 왼쪽 바닥 센서가 검은색 선 위에 있으면 중앙으로 가기 위해 왼쪽으로 움직이고, 오른쪽 바닥 센서가 검은색 선 위에 있으면 중앙으로 가기 위해 오른쪽으로 움직이면 됩니다.
그런데 왜 바닥 센서가 검은색 선 위에 있다는 것을 기준이 되는 숫자와 크기를 비교하여 판단할까요?
바닥 센서의 값이 50보다 작으면 검은색 선 위에 있다고 판단할 필요가 있을까요?
아래 그림을 자세히 살펴보면 햄스터 로봇이 중앙에서 오른쪽에 있는 경우에는 오른쪽 바닥 센서의 값이 왼쪽 바닥 센서의 값보다 커지게 됩니다. 반대로 햄스터 로봇이 중앙에서 왼쪽에 있는 경우에는 왼쪽 바닥 센서의 값이 오른쪽 바닥 센서의 값보다 커지게 됩니다. 즉, 기준이 되는 숫자를 사용하지 않고 왼쪽 바닥 센서 값과 오른쪽 바닥 센서 값의 크기를 비교하면 됩니다.

```
#include "roboid.h"

int main(int argc, char *argv[]) {
    int left_floor, right_floor;
    hamster_create();

    while(1) {
        left_floor = hamster_left_floor();
        right_floor = hamster_right_floor();
        hamster_wheels(30, 30);
        // 중앙에서 오른쪽에 있으면 왼쪽으로 회전한다.
        if(left_floor < right_floor) {
            hamster_wheels(-30, 30);
        }
        // 중앙에서 왼쪽에 있으면 오른쪽으로 회전한다.
        else if(left_floor > right_floor) {
            hamster_wheels(30, -30);
        }
        wait(10); // 너무 빨리 반복하지 않도록 한다.
    }
    return 0;
}
```

근접 센서 사용하기

❖ 장애물 감지하기

- 햄스터 로봇의 전방에 있는 근접 센서는 적외선을 방출하는 IR-LED와 적외선을 감지하는 광 트랜지스터로 이루어져 있습니다. 광 트랜지스터는 IR-LED가 방출하는 적외선이 전방의 물체에 반사되어 들어오는 광량을 측정합니다. 장애물이 가까이 있으면 반사된 광량이 많아서 측정되는 값이 증가하고, 장애물이 멀면 반사된 광량이 적어서 측정되는 값이 감소합니다. 장애물이 없으면 반사된 빛이 없어 0의 값을 가집니다. 햄스터 로봇의 근접 센서는 IR-LED가 로봇의 앞면 좌우에 하나씩 설치되어 있으며, 중앙 아래에 광 트랜지스터가 있어서 좌우의 IR-LED에 대한 광량을 번갈아 측정합니다. 전방의 1cm 이상, 30cm 이하의 거리에 있는 물체나 장애물을 감지할 수 있으며, 0부터 255까지의 값을 가집니다.



근접 센서 사용하기

■ 장애물 감지하기

```
#include <stdio.h>
#include "roboid.h"

int main(int argc, char *argv[]) {
    hamster_create();

    while(1) {
        printf("%d, %d\n",
                hamster_left_proximity(),
                hamster_right_proximity());
        wait(20); // 너무 빨리 반복하지 않도록 한다.
    }
    return 0;
}
```

- 햄스터 로봇의 앞을 손으로 막고 손을 앞뒤로 움직여 로봇과의 거리를 다르게 하면서 왼쪽 근접 센서와 오른쪽 근접 센서의 값이 어떻게 달라지는지 관찰해 봅시다. 손을 너무 가까이 가져 가면 센서 값이 측정되지 않을 수도 있습니다. 하얀색 종이로 하였을 때, 다양한 색상의 물체로 하였을 때 어떤 변화가 생기는지도 관찰해 봅시다. 같은 거리에서 장애물의 색이 밝을수록 근접 센서의 값이 더 커진다는 것을 알 수 있는데, 밝은 색일수록 반사되는 광량이 많아지기 때문입니다. 센서의 값들을 관찰하고 분석하는 것은 매우 중요하기 때문에 앞으로 새로운 센서를 사용할 때마다 반드시 이와 같은 방법으로 관찰하도록 합시다.

근접 센서 사용하기

❖ 햄스터 로봇이 앞으로 이동하다가 앞에 장애물이 나타나면 정지하게

```
#include "roboid.h"

int main(int argc, char *argv[]) {
    hamster_create();

    while(1) {
        if(hamster_left_proximity() < 40 && hamster_right_proximity() < 40) {
            hamster_wheels(30, 30); // 앞으로 이동한다.
        }
        else {
            hamster_stop(); // 정지한다.
        }

        wait(20); // 너무 빨리 반복하지 않도록 한다.
    }
    return 0;
}
```

근접 센서 사용하기

❖ 손으로 연주하기

```
#include "roboid.h"

int main(int argc, char *argv[]) {
    double hz = 0;
    int proximity;
    hamster_create();

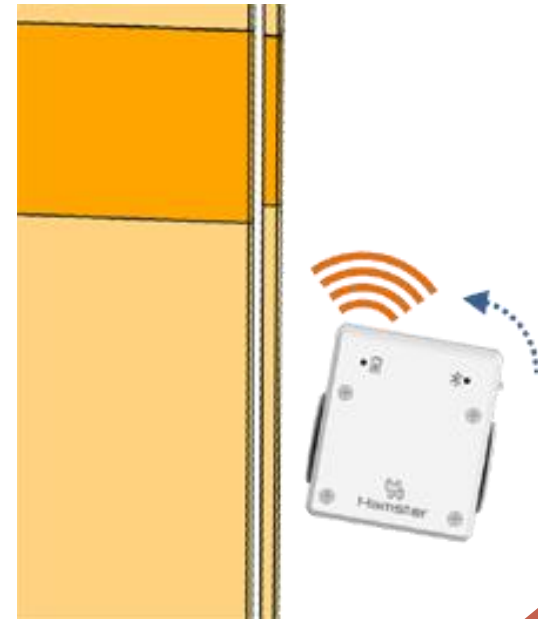
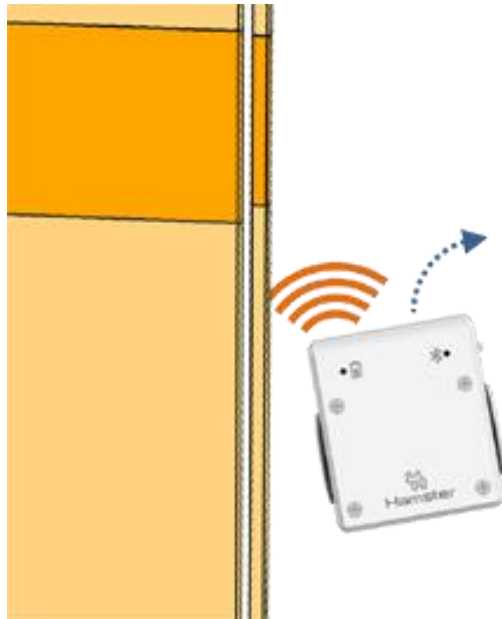
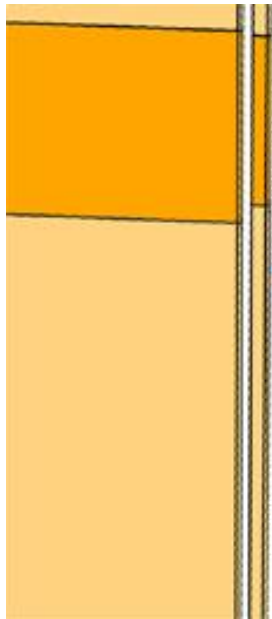
    while(1) {
        proximity = hamster_left_proximity();
        if(proximity < 10) // 거리가 너무 멀면 0으로 한다.
            proximity = 0;

        // 이전의 음정과 현재 근접 센서 값을 조합하여 음정이 부드럽게 변하도록 한다.
        hz = (hz * 5 + proximity * 50) / 10.0;
        hamster_buzzer(hz);
        wait(20); // 너무 빨리 반복하지 않도록 한다.
    }
    return 0;
}
```

벽 따라가기

❖ 왼쪽 근접 센서를 사용하여 왼쪽의 벽 따라가기

- 실험을 위해 적당한 높이의 책을 준비합니다. 책의 높이가 너무 낮으면 센서로 감지하지 못하므로 책의 높이는 햄스터 로봇의 높이와 비슷하거나 더 높게 하는 것이 좋습니다. 책 대신 A4 용지를 접어서 벽을 만들어도 되고 미로판의 벽을 사용하여도 됩니다.



벽 따라가기

- 우선 햄스터 로봇이 벽을 향해 있는지, 반대쪽을 향해 있는지 알기 위해 그림과 같이 햄스터 로봇을 벽과 나란히 놓고 왼쪽 근접 센서의 값을 관찰해 봅시다. 햄스터 로봇이 벽을 향해 있을 때와 반대쪽을 향해 있을 때 왼쪽 근접 센서의 값이 어떻게 달라지는지도 확인해 봅시다.
- 햄스터 로봇의 앞면에 있는 근접 센서는 원래 앞쪽에 있는 물체를 감지하기 위해 사용됩니다. 햄스터 로봇의 앞쪽에 물체가 있으면 센서 값이 커지기 때문에 이 실험을 위해서는 햄스터 로봇의 앞쪽에 어떠한 물체도 있지 않아야 합니다.

■ 소스코드

```
#include <stdio.h>
#include "roboid.h"

int main(int argc, char *argv[]) {
    hamster_create();

    while(1) {
        printf("%d\n", hamster_left_proximity());
        wait(20); // 너무 빨리 반복하지 않도록 한다.
    }

    return 0;
}
```

벽 따라가기

- 햄스터 로봇이 벽을 향해 있을 때는 왼쪽 근접 센서의 값이 커지고, 반대쪽을 향해 있을 때는 값이 작아진다는 것을 알 수 있습니다. 햄스터 로봇이 벽과 나란히 있을 때의 센서 값을 기준으로 크기를 비교하면 됩니다. 햄스터 로봇이 벽과 나란히 있을 때의 센서 값이 5라고 관측되었다면 햄스터 로봇이 벽을 향해 있는 경우를 다음과 같이 표현할 수 있습니다.

■ 소스코드

```
#include "roboid.h"

int main(int argc, char *argv[]) {
    hamster_create();

    while(1) {
        if(hamster_left_proximity() > 5) {
            hamster_wheels(30, 0);
        } else {
            hamster_wheels(0, 30);
        }

        wait(10); // 너무 빨리 반복하지 않도록 한다.
    }

    return 0;
}
```

벽 따라가기

❖ 왼쪽 근접 센서를 사용하여 원통 따라 돌기

- 원통을 따라 돌 때는 벽을 따라갈 때보다 센서의 값이 더 작아집니다. 왼쪽 근접 센서는 조금 앞쪽의 왼쪽에 있는 벽을 감지하게 되는데, 곡면인 경우에는 조금 앞쪽의 왼쪽에 있는 면이 햄스터 로봇에서 멀어지기 때문에 센서의 값이 작아지게 되는 것입니다.



벽 따라가기

■ 소스코드

```
#include "roboid.h"

int main(int argc, char *argv[]) {
    hamster_create();

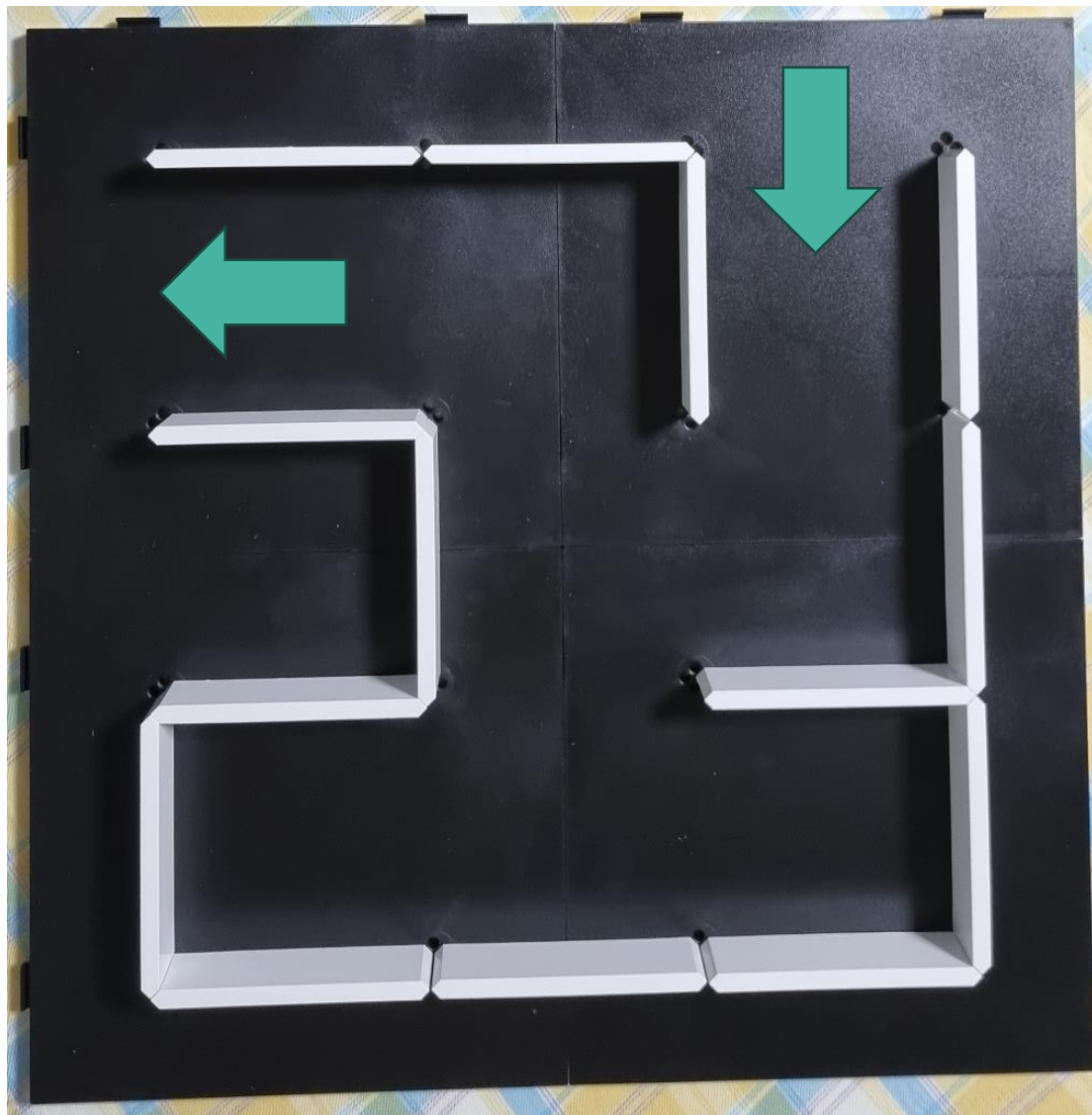
    while(1) {
        if(hamster_left_proximity() > 0) {
            hamster_wheels(30, 0);
        } else {
            hamster_wheels(0, 30);
        }

        wait(10); // 너무 빨리 반복하지 않도록 한다.
    }

    return 0;
}
```

- 조금 앞쪽의 왼쪽에 있는 면을 감지하여 따라가기 때문에 원통의 크기가 클수록 더 잘 따라가게 됩니다.
- 오른쪽 근접 센서를 사용하여 원통을 따라 돌 수 있도록 해봅시다.

미로 탈출



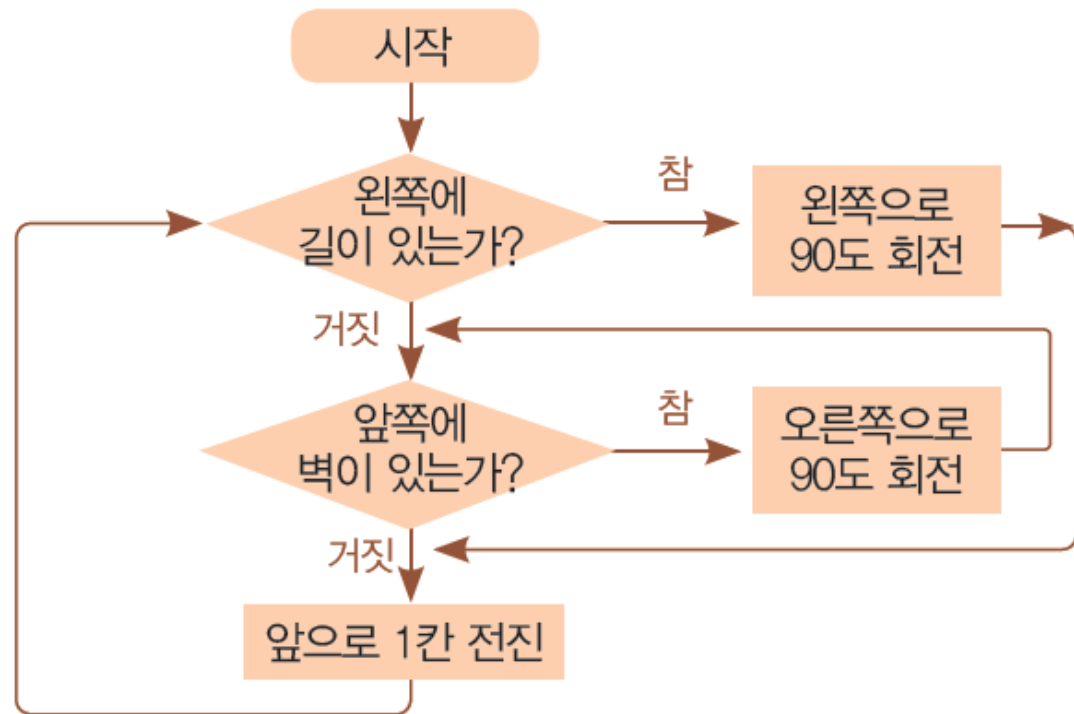
미로 탈출

❖ 좌수법 알고리즘

자연어

- 1 왼쪽으로 길이 있다면 왼쪽으로 90도 회전한 후 단계 3으로 가고, 아니면 단계 2로 간다.
- 2 앞쪽이 벽이면 오른쪽으로 90도 회전한 후 단계 2로 가고, 아니면 3으로 간다.
- 3 앞으로 1칸 전진한 후, 단계 1로 간다.

순서도



```
#include "roboid.h"

int main(int argc, char *argv[]) {
    hamster_create();

    while(1) {
        printf("%d %d\n", hamster_left_proximity(), hamster_right_proximity());
        if(hamster_left_proximity() <= 20) { // 왼쪽에 길이 있으면,
            hamster_wheels(40, 40); // 1.5초 직진 질주
            wait(1500);
            hamster_wheels(-30, 30); // 90도 왼쪽 스펀턴
            wait(1000);
            hamster_wheels(40, 40); // 1.2초 직진 질주
            wait(1200);
        }
        else if(hamster_right_proximity() >= 75) { // 앞쪽에 벽이 있으면,
            hamster_wheels(30, -30); // 우회전
            wait(1000);
        }
        else {
            hamster_wheels(30, 30); // 0.01초 직진
            wait(10);
        }
    }
    return 0;
}
```

```
// 미로탈출 (좌수법) ver1.0
```

```
#include "roboid.h"
```

```
void pivot_turn_left() {  
    hamster_wheels(-30, 30);  
    wait(1000);  
}
```

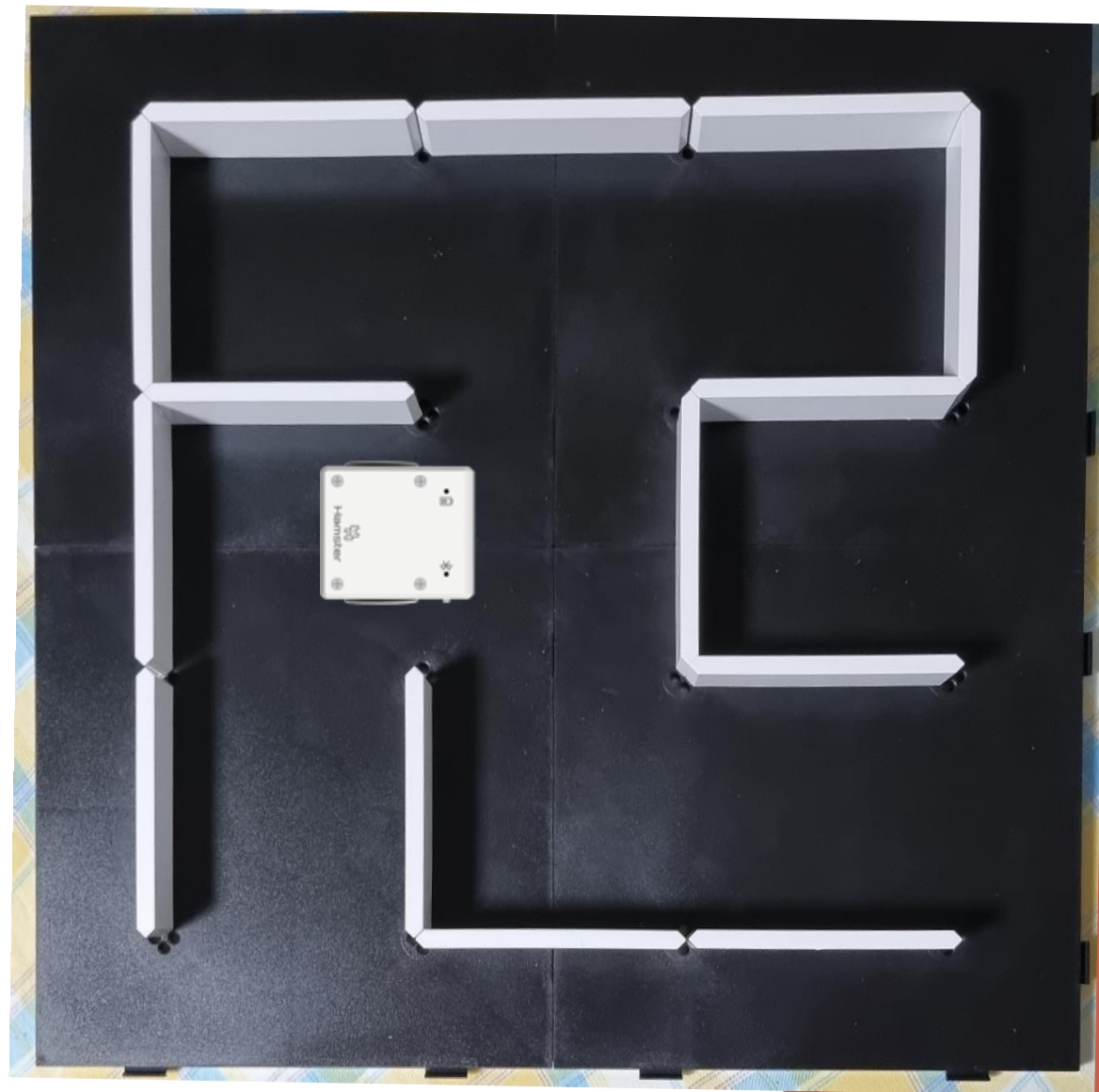
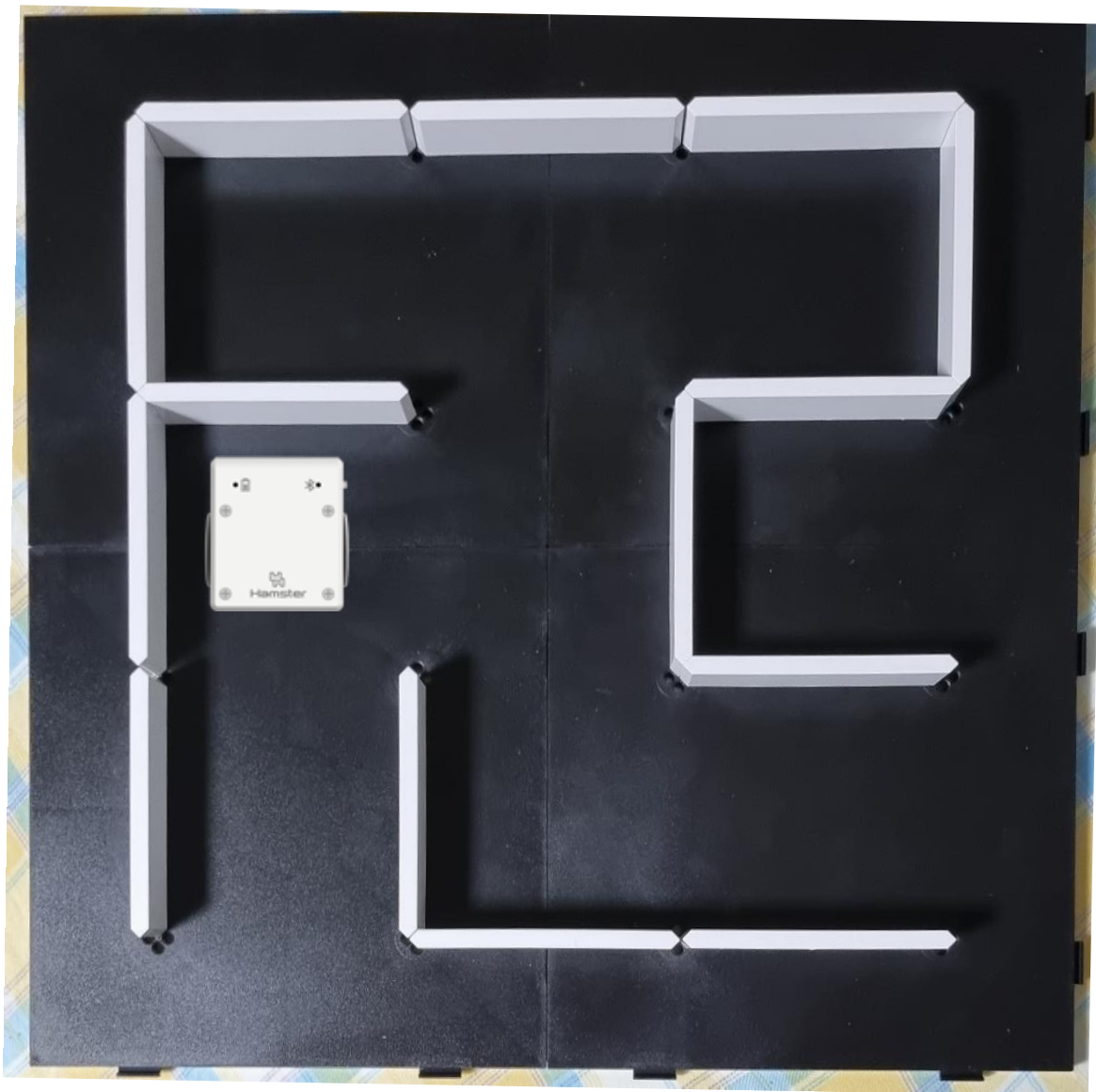
```
void pivot_turn_right() {  
    hamster_wheels(30, -30);  
    wait(1000);  
}
```

```
void wheels_ms(int lw, int rw, int ms) {  
    hamster_wheels(lw, rw);  
    wait(ms);  
}
```

```
int main(int argc, char *argv[]) {  
    hamster_create();
```

```
    while(1) {  
        // 왼쪽에 길이 있으면,  
        if(hamster_left_proximity() <= 20) {  
            wheels_ms(40, 40, 1500);  
            pivot_turn_left();        // 좌회전  
            wheels_ms(40, 40, 1100);  
        }  
        // 앞쪽에 벽이 있으면,  
        else if(hamster_right_proximity() >= 73) {  
            pivot_turn_right();    // 우회전  
        }  
        else { // 직진  
            hamster_wheels(30, 30);  
            wait(10);  
        }  
    }  
    return 0;  
}
```

미로 탈출



```
// 미로탈출 (좌수법) ver 1.1
```

```
#include "roboid.h"
```

```
void pivot_turn_left() {  
    hamster_wheels(-30, 30);  
    wait(1000);  
}
```

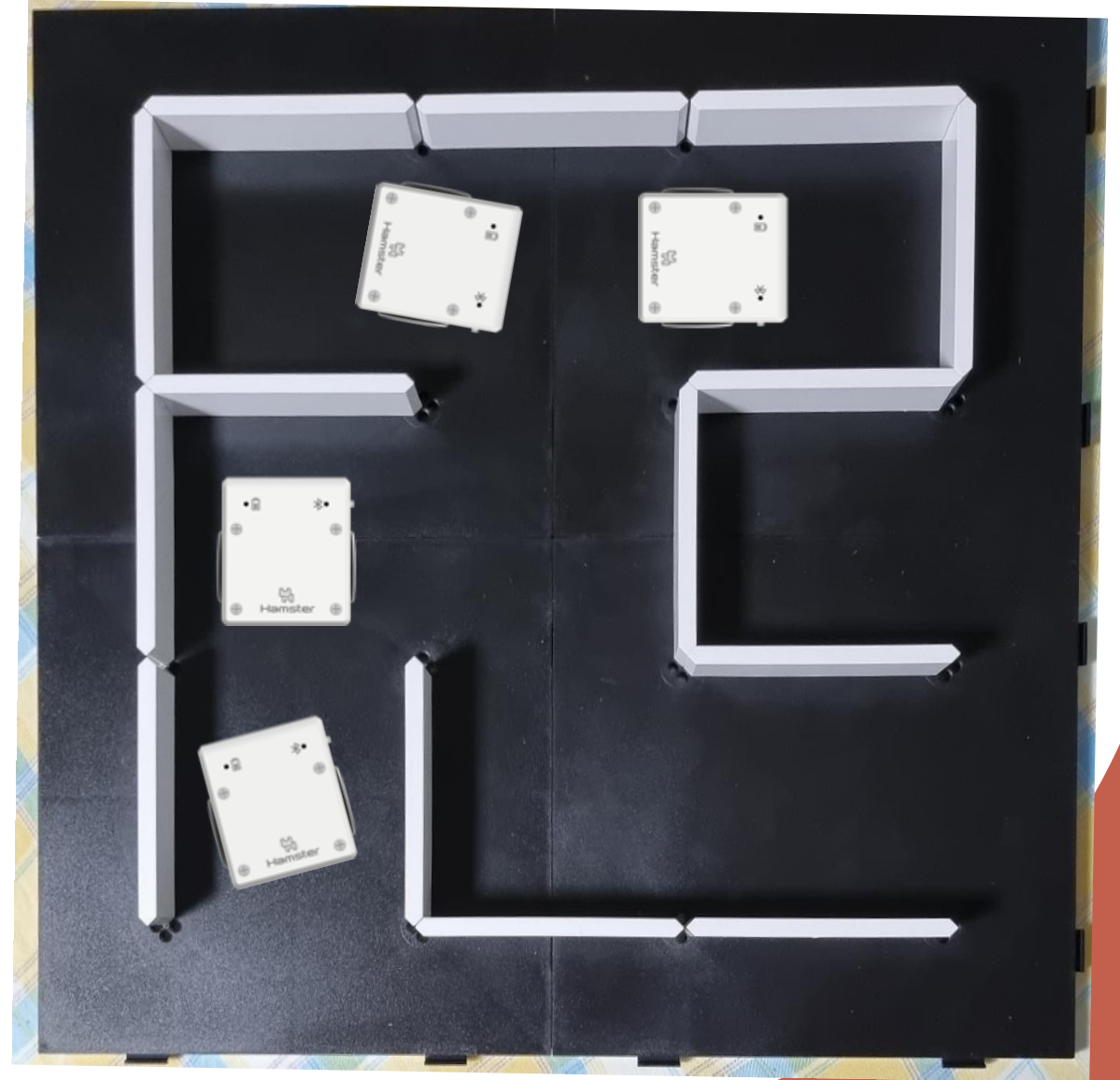
```
void pivot_turn_right() {  
    hamster_wheels(30, -30);  
    wait(1000);  
}
```

```
void wheels_ms(int lw, int rw, int ms) {  
    hamster_wheels(lw, rw);  
    wait(ms);  
}
```

```
int main(int argc, char *argv[]) {  
    hamster_create();
```

```
    int center_val = hamster_left_proximity(); // 시작하면서 센서값 저장  
    while(1) {  
        if(hamster_left_proximity() <= 20) { // 왼쪽에 길이 있으면  
            wheels_ms(40, 40, 1500);  
            pivot_turn_left(); // 좌회전  
            wheels_ms(40, 40, 1100);  
        }  
        else if(hamster_right_proximity() >= 73) { // 앞쪽에 벽이 있으면,  
            pivot_turn_right(); // 우회전  
        }  
        else { // 직진  
            // 너무 왼쪽으로 붙는 경우 살짝 오른쪽으로 보정  
            if(hamster_left_proximity() > center_val+10)  
                wheels_ms(40, 30, 10); // 찰금 우회전  
            // 너무 오른쪽으로 붙는 경우 살짝 오른쪽으로 보정  
            else if(hamster_left_proximity() < center_val-10)  
                wheels_ms(30, 40, 10); // 찰금 좌회전  
            else  
                wheels_ms(30, 30, 10);  
        }  
    }  
    return 0;  
}
```

미로 탈출



```
// 미로탈출 (우수법) ver1.0
```

```
#include "roboid.h"
```

```
void pivot_turn_left() {  
    hamster_wheels(-30, 30);  
    wait(1000);  
}
```

```
void pivot_turn_right() {  
    hamster_wheels(30, -30);  
    wait(1000);  
}
```

```
void wheels_ms(int lw, int rw, int ms) {  
    hamster_wheels(lw, rw);  
    wait(ms);  
}
```

```
int main(int argc, char *argv[]) {  
    hamster_create();
```

```
    while(1) {  
        // 오른쪽에 길이 있으면,  
        if(hamster_right_proximity() <= 20) {  
            wheels_ms(40, 40, 1500);  
            pivot_turn_right(); // 우회전  
            wheels_ms(40, 40, 1200);  
        }  
        // 앞쪽에 벽이 있으면,  
        else if(hamster_left_proximity() >= 73) {  
            pivot_turn_left(); // 좌회전  
        }  
        else {  
            hamster_wheels(30, 30);  
            wait(10);  
        }  
    }  
    return 0;
```

```
}
```