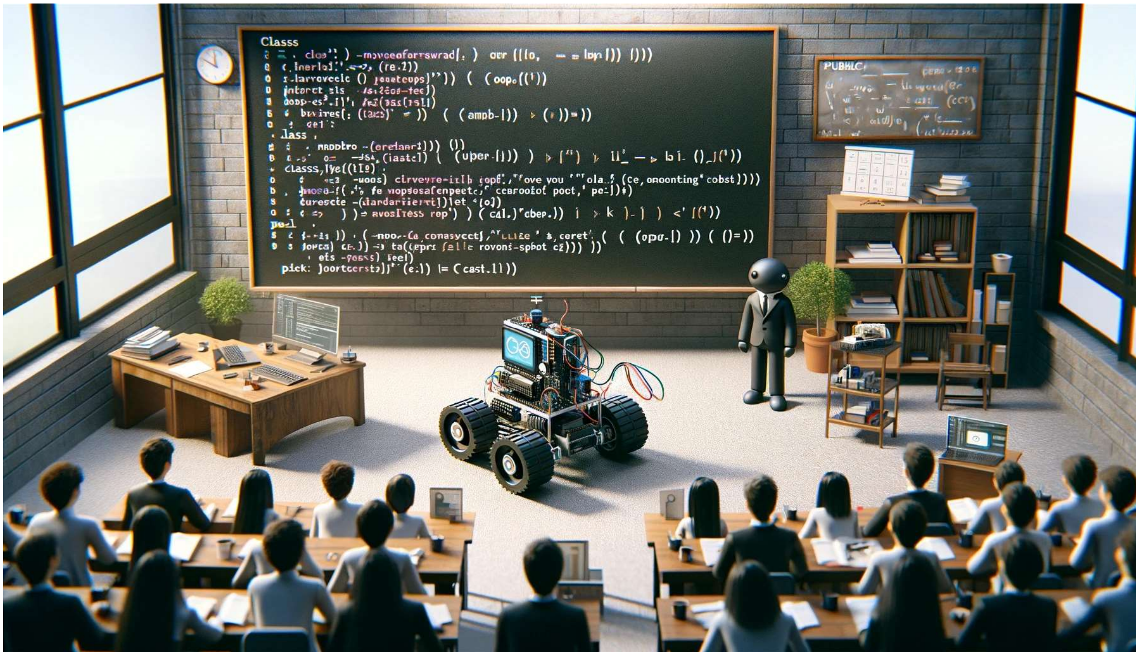
C 부터 C++ 언어까지
아두이노로봇 코딩 실습

"C++ Robotics with Arduino: A C to C++ Coding Adventure"



챗GPT가 생성한 달리 이미지

2024년 2월

㈜ 프라이봇 지음

머 릿 말

먼저 책을 새롭게 쓰게된 이유는

'neo-아두이노로봇카'로 C언어 코딩을 기초부터 배우는 많은 이용자들에게 새로운 배움의 기회를 제공하기 위함이다. 언제부터인가 C언어만으로는 코딩교육의 부족함을 느끼기 시작하던 차에 파이썬 언어 코딩이 대중에게 큰 인기를 얻고 있었다. 여기에 더 나아가서 C++ 언어를 습득할 필요가 매우 크다고 생각되어, 이미 사용중인 'neo-아두이노로봇카'로 C++ 중급 수준까지 코딩역량을 키울 수 있는 수단과 동기를 제공하는 것이다.

오래전부터 파이썬 언어를 사용할 수 있는 마이크로비트와 아두이노가 결합된 신제품 출시를 준비하면서, 기존 제품을 사용하는 고객들이 C언어 코딩교육만으로는 아쉬웠던 교육으로부터 한단계 수준을 상향시킬 '코딩교육 신무기'를 갖게 하려는 취지로 글을 쓰게 되었다.

예상 독자는

당연히 'neo-아두이노로봇카'를 이용하는 고객이 될 것이다. 그리고 당사에서 출시하는 신제품에 관심을 갖는 고객일 수도 있다. 그렇지만, 꼭 아두이노로봇 제품을 구입하지 않더라도, 무료로 배포되는 이 책을 읽으면서 C++ 코딩언어를 쉽게 이해하도록 작성하려고 노력했다.

이 책의 기원과 탄생배경

이 책의 기원은 'neo-아두이노로봇카'의 기원인 미국 Parallax사의 코딩로봇 온라인 배포판이 기초가 된다. 당사에서 오래전에 배포한 '아두이노로봇 가지고 놀기' 온라인 실습교재 1부와 2부 자료는 미국 Parallax사의 온라인 내용을 기초로 번역한 것이었다.

그리고 이 책이 탄생하게 되는 배경으로 덧붙일 말은 이 책의 다양한 요약과 설명 그리고 예제 코드개발이 최근 화두가 되고 있는 챗GPT의 도움을 받은 것이다. 참고로 이 책의 많은 삽화는 달리(Dall-E) 이미지를 사용했다.

이 책의 내용은 기존에 배포된 온라인 실습교재 내용과 가능한 중복되지 않으면서, 동시에 기존의 내용과 연관성을 유지하도록 창작하는 것이었다. 챗GPT는 의심의 여지없이 매우 훌륭한 협력자이고, 강력한 생산자이다. 이 책을 읽고 도움을 받을 많은 고객들 역시 챗GPT와 항상 소통하면서 도움을 받고 큰 성취를 이루기를 희망한다.

앞으로도 프라이봇의 신제품으로 여러분과 다시 만나기를 희망하고, 머지않아 그렇게 될 것으로 믿는다. 모두에게 신의 영광이 함께하기를 바란다.

지은이 정 욱진

******** 이 책으로 C언어와 C++언어를 가르치시는 분께 ********

이 책으로 학생들을 가르치는 선생님들께 아래 내용을 소개합니다. 그리고 함께 배포될 파워포인트자료를 참고하거나, 유튜브로 배포될 영상을 보조자료로 활용할 것을 추천드립니다.

이 책은 총 9장으로 구성되어 있습니다. 이 책에서 다루어질 내용은 C언어를 전혀 모르는 학생에게는 미흡할 수 있습니다. C언어의 기초문법이나 사용방법이 최대한 배제되었기 때문입니다. C언어를 전혀 경험하지 못한 학생을 대상으로 한다면, C언어로 'neo-아두이노로봇카' 다루는 방법과 기초문법을 간단히 익힌 다음 이 책을 사용하기를 추천드립니다.

C언어와 C++언어는 같은 듯하면서도, C++언어가 완전히 새로운 개념과 접근방식을 요구하기 때문에, C언어의 기초문법 정도만이라도 먼저 경험하고 이 책을 사용하기를 추천합니다. 이 책의 C++ 관련내용 핵심은 클래스의 개념을 로봇동작을 통해 배우는 것입니다. 따라서 C++스타일의 I/O스트림이나 네임스페이스 개념, 변수의 유형과 수명에 대한 개념 특히 수정자와 한정자 그리고 C++에서 다루는 향상된 for 반복문 등의 개념을 의도적으로 다루지 않습니다.

아래는 각 장별로 다루어질 내용을 요약한 것입니다.

제 1 장 SelfAbot 클래스 이해하기 : C++ 객체지향프로그래밍이 무엇인지 소개하고, 아두이노 예제를 통해서 클래스 개념을 익히도록 했습니다. 그리고 아두이노로봇에 적용할 라이브러리코드 'SelfAbot'클래스를 소개하고, 설명합니다.
이 장은 실습보다 개념을 설명하는데 주안점을 두었으므로, 학생들이 지루해한다면 맨 나중에 천천히 다루어도 됩니다. 'SelfAbot'클래스를 이후 장에서 계속 설명할 것입니다.

제 2 장 아두이노 함수 및 함수 오버로딩 : 아두이노 예제를 클래스 개념으로 사용하는 예제와 함수 오버로딩 사용법에 대해 설명하고 실습합니다.

제 3 장 로봇 초기화 및 기본동작 메소드 : 아두이노로봇의 동작을 클래스 개념을 이용해서 코딩하고 실습합니다. 오래전에 배포한 '아두이노로봇 가지고 놀기' 온라인 실습교재 1부에서 언급한 내용을 클래스 개념으로 동일하게 아두이노로봇의 기본동작을 습득하는 실습입니다.

제 4 장 아두이노로봇 움직임 제어하기 : 앞장에서 다룬 아두이노로봇의 동작에 새로운 주행동작을 실습합니다. 클래스 개념으로 로봇의 동작을 쉽게 확장하는 방법을 경험시켜 보세요. 더 섬세하고 더 다양하게 움직이는 아두이노로봇을 배우는 실습입니다.

제 5 장 포토트랜지스터 센서 광신호 메소드: rcTime()
이 장은 광센서를 다루는 클래스 메소드를 소개하고, 실습합니다. 동시에 동일한 광센서를 사용하면서 센서의 신호처리 방법을 다양하게 변형하고 다룰 수 있는 확장 개념을 익힐 차례입니다.

제 6 장 적외선 센서를 이용한 자율주행 로봇 : 적외선 센서로 주행하는 실습을 이미 많이 시도했을 것입니다. 이 장에서는 적외선 신호를 더 다양하게 또 다른 목적으로 사용하는 실습 경험을 제공합니다.

제 7 장 초음파센서를 장착한 아두이노로봇 : 적외선 송수신센서와 차별화된 초음파센서가 로봇에 어떻게 통합되고 이용되는지 실습합니다. 초음파센서로 로봇의 자율주행을 더 지능적으로 만들 수 있습니다.

제 8 장 클래스 상속으로 전문로봇 만들기 : C++ 클래스 개념에서 상속의 개념을 학생들에게 전달하기 위한 실습입니다. 'SelfAbot'기본 클래스(슈퍼클래스 또는 상위클래스)에는 포함되지 않은 메소드들(앞 장마다 소개한 실습 예제들)을 모두 모아서 하위 클래스로 만들고, 클래스 상속 개념을 실습합니다.

제 9 장 로봇의 고급 이동과 제어 : 앞 장까지 실습한 코드를 대상으로 블루투스/와이파이/셀룰러/IOT 통신등 무선방식으로 데이터를 송수신하는 사례를 소개하고, 어떤 장점과 혜택이 가능한지 실습합니다. 그리고 로봇이 동작하는동안 발생할 수 있는 다양한 상황을 예상하고 어떻게 예외처리 코드로 반영하는지에 대하여 소개합니다.

마지막으로 이 책에서 소개한 'SelfAbot'클래스 개념과 코드는 아두이노로봇을 동작시키기 위한 최종본이나 최종 결과물이 아닙니다. 이 코드는 단순히 학생들이 C++ 클래스 개념을 숙지하기 위한 한가지 사례에 불과함을 알립니다. 당연히 더 많고 다양한 코드의 목적이나 형태를 반영하지 못합니다.

이 점을 감안해서, 학생들이 자유롭게 자기만의 아두이노로봇 클래스를 만들고 테스트해보기를 권장해주시기 바랍니다.

******** 이 책으로 혼자서 C++언어를 독학하려는 분께 ********

이 책은 C언어를 전혀 경험하지 못한 분들에게 추천해도 됩니다. 코딩 초보자가 가장 쉽게 C++ 코딩언어를 이해할 수 있도록 하는 것이 이 책의 취지입니다. 다만, C++ 언어의 개념과 체계를 이해하는 것만으로도 매우 복잡하고 난해한데, C언어의 기초문법을 이 책에서 더 언급하게 되면 분량의 증가와 복잡함이 늘어날 것으로 생각되어 생략했습니다.

만약 C언어 기초문법을 전혀 모르는 분이라면, 오래 전에 배포한 "아두이노로봇 가지고 놀기" 온라인 실습교재 1부에 설명한 기초문법 부분을 먼저 읽어보기를 추천합니다. 만약 유튜브로 시청하기를 희망한다면, 기초문법을 다룬 '초등학생도 C언어 문법 기초 40분으로'라는 영상을 시청하면 됩니다. https://youtu.be/joUNuEGv3i0

'**코딩언어를 배운다**'라는 사실은 지금까지는 '코딩의 기초문법을 익히는 과정'이 필수로 여겨지던 시절이었습니다. 그렇지만, 챗GPT가 출현한 이후 지금의 관점과 시각을 모두 고쳐야 합니다. 코딩문법을 당장 몰라도, 단 한 줄의 코드를 작성할 수 없어도, 코드의 전개 과정과 코드의 실행 효과를 상상할 수 있다면 챗GPT를 사용해서 수준높은 코드를 작성할 수 있는 시대입니다. '컴퓨팅사고'가 가장 요구되는 시대에 도달했다고 말합니다. 단도직입적으로 설명하면, 내가 코드 문법에 기초해서 코드를 한줄 한줄 작성하는 것은 어렵지만, 내가 원하는 코드의 목적과 방식에 대하여 챗GPT에게 코드작성을 요청하고, 챗GPT가 답변할 때 내가 그 코드를 읽고, 판단하고 검증해서 변형해갈 수 있는 능력이 더 요구된다는 의미입니다.

그래서 **코드 작성에 필수적이었던 '코딩문법'은 이제 필수적인 학습 문턱이 아닙니다.** 코딩문법보다 코딩의 개념을 이해하는 것이 더 필수적인 세상이 되었습니다. 코딩문법과 코딩개념은 이제 동의어가 아닙니다. 그래서 처음 코딩을 배우려는 독자에게 이 책을 먼저 경험해보라고 추천합니다. 이 책은 C 언어보다 깊은 개념의 코딩언어 C++ 언어를 쉽게 이해할 계기를 제공합니다.

이 책에는 로봇동작을 위한 전자부품의 상세한 설명이나 사용법, 그리고 아두이노로봇을 처음 구매한 독자가 로봇을 조립하는 과정을 소개하는 내용은 다루어지지 않습니다. 더 상세한 자료를 찾고자 한다면, 위에 언급한 온라인 실습교재를 참고하면 됩니다. 그리고 센서부품에 대한 설명도 가능한 생략하고, 예전 자료에서 다루지 않았던 내용으로 더 새롭고 객체지향프로그래밍에 가깝도록 쓰였습니다.

마지막으로 전하고 싶은 말은 이 책으로 C++언어 코딩에 대해 처음 학습한다면, 이 책을 마치고난 다음 예전 자료를 간단히 숙독해보는 것도 좋겠다는 생각을 해봅니다. 기초는 언제나 중요하기 때문입니다.

저자와 연락하기

저자는 2012년부터 프라이봇(Fribot.com)을 통해 국내에서 '아두이노' 마이크로컨트롤러를 보급하고 책을 지어서 배포하는 등 국내 학생들의 코딩교육에 힘써 왔습니다. 동시에 미국 Parallax사의 제품 공식판매처로도 활동하고 있습니다. 국내 사용자들에게 아두이노로봇 제품을 출시하고 많은 교육자료를 배포하게된 배경도 미국 Parallax사와의 협력에 기인한 바가 큽니다.

프라이봇의 대표이자 이 책의 저자에게 연락하기 위한 수단은 프라이봇 홈페이지(fribot.com)에 공개된 주소/연락처 정보를 활용하면 됩니다.

연락수단으로는 전화/이메일/블로그 등 다양한 채널들을 사용할 수 있고,
카카오톡으로도 많은 질문을 받고 있으므로, 프라이봇 홈페이지 카카오톡 아이콘을 활용하면 됩니다.

다만 모든 작업을 수행할 인력이 충분하지 않아서, 간혹 회신이 늦어져 개인별로 소홀하게 여겨질 수도 있다는 점은 아쉽게 생각합니다. 당사의 기업 활동이 더 강력해지면 더 질 좋은 서비스를 할 수 있도록 노력하겠습니다.

감사합니다.

주요 자료 다운로드 출처

1) 'SelfAbot' 라이브러리 다운로드
   https://github.com/wookjin-chung/SelfAbot
2) 한글 자료 'C부터 C++언어까지 아두이노 로봇 코딩실습' 문서 및 ppt
   https://www.fribot.com/board/view?id=community_guide&seq=193
3) 유튜브 교육자료
   https://www.youtube.com/channel/UCJw4FMNvVXiektPpQPUriyQ
4) 프라이봇 블로그
   https://blog.naver.com/fribot/223365886844

목 차

제 1 장 SelfAbot 클래스 이해하기
1.1 C++ 객체지향프로그래밍
1.2 클래스 코딩: 아두이노 LED 깜박임
1.3 객체지향 프로그래밍: 캡슐화
1.4 'SelfAbot' 클래스 소개
1.5 클래스: 생성자, 멤버 변수, 상수
1.6 멤버 함수와 오버로딩
부록: 'SelfAbot.zip'라이브러리 헤더와 소스 파일

제 2 장 아두이노 함수 및 오버로딩
2.1 'SelfAbot'라이브러리 아두이노에 추가하고 예제 활용하기
2.2 아두이노 함수 실행하기
2.3 단순화를 위한 함수 오버로딩

제 3 장 로봇 초기화 및 기본동작 메소드
3.1 로봇 초기화를 위한 setup 메소드 구현하기
3.2 로봇 움직임 속성과 메소드
3.3 단순화를 위한 함수 오버로딩
3.4 표준형 서보모터 기본동작
3.5 연속 회전형 서보모터 기본동작

제 4 장 아두이노로봇 움직임 제어하기
4.1 서보모터 중심맞추기
4.2 직선주행과 로봇 보정
4.3 좌회전 / 우회전 그리고 연속동작
4.4 점진적 가속과 감속
4.5 추가 메소드 : maneuver()

제 5 장 포토트랜지스터 광신호 : rcTime()
5.1 아날로그 입력 신호처리를 위한 실습
5.2 디지털 입력신호 방식: rcTime() 메소드
5.3 두 개의 광센서(phototransistor)로 빛의 방향인식
5.4 빛을 추적하는 아두이노로봇

제 6 장 적외선 센서를 이용한 자율주행 로봇
6.1 장애물 탐지를 위한 적외선 센서 신호처리
6.2 적외선 센서 신호를 로봇에 통합하기

6.3 적외선 센서로 irDistance 메소드 사용하기
6.4 적외선 센서를 이용한 주변물체 스캔하기
6.5 적외선 센서로 스캔해서 가까운 물체 찾기

제 7 장 초음파센서를 장착한 아두이노로봇
7.1 초음파센서 신호처리하기
7.2 전방물체 레이더 화면 표시하기
7.3 초음파센서 탐색 로봇으로 주행

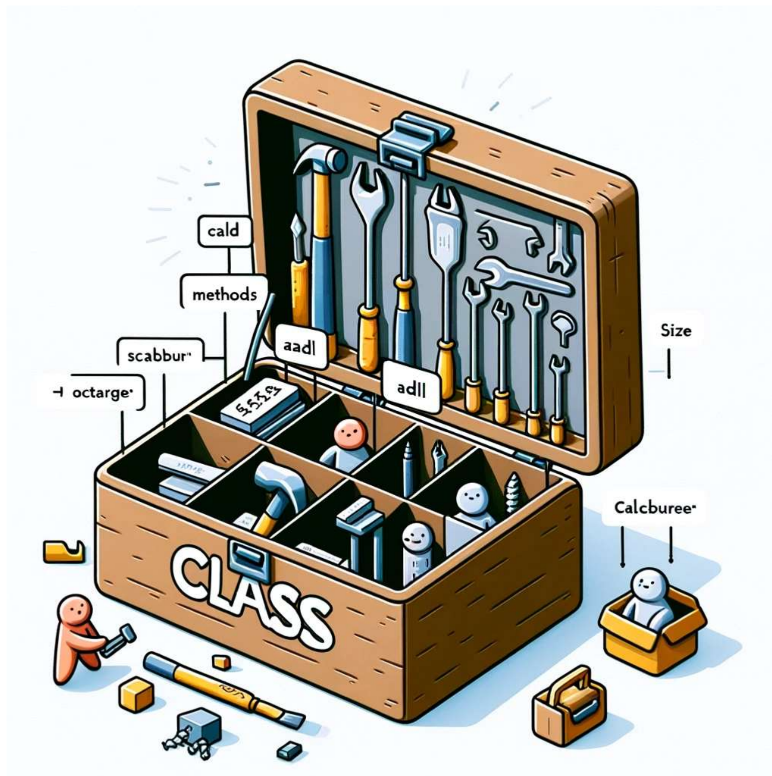
제 8 장 클래스 상속으로 전문로봇 만들기
8.1 'SelfAbot' 슈퍼클래스로 하는 하위 클래스 만들기
8.2 상속클래스로 .ino 파일 작성하는 방법
부록: 'EnhancedSelfAbot.zip'라이브러리 헤더와 소스 파일

제 9 장 로봇의 고급 이동과 제어
9.1 무선 로봇의 고급기능 구현하기
9.2 로봇 동작중 예상치 못한 상황관리

부 록 : 초음파 레이더영상을 위한 프로세싱 소스코드

제 1 장 SelfAbot 클래스 이해하기

1.1 C++ 객체지향프로그래밍
1.2 클래스 코딩: 아두이노 LED 깜박임
1.3 객체지향 프로그래밍: 캡슐화
1.4 'SelfAbot' 클래스 소개
1.5 클래스: 생성자, 멤버 변수, 상수
1.6 멤버 함수와 오버로딩



1.1 C++ 객체지향프로그래밍

C++에서 클래스란 "구조체(struct)의 확장된 형태"라고 생각할 수 있습니다. C 언어에서 구조체는 데이터를 묶어서 하나의 단위로 만드는 방법을 제공합니다. C++의 클래스도 이와 유사하게 데이터를 묶지만, 여기에 추가적으로 함수(메소드라고도 함)를 포함할 수 있습니다.

기본적인 개념은 다음과 같습니다:
(1) 데이터 캡슐화: 클래스는 데이터(변수)와 이 데이터를 처리하는 함수(메소드)를 하나의 단위로 묶습니다. 이렇게 하면 데이터와 그 데이터를 처리하는 로직이 서로 밀접하게 연결되어,

코드의 구조를 더 명확하고 관리하기 쉽게 만듭니다.

(2) 접근 제어: 클래스는 데이터와 함수의 접근성을 제어할 수 있습니다. public, private, protected와 같은 접근 지정자를 사용하여 클래스 외부에서 접근할 수 있는 멤버와 클래스 내부에서만 접근 가능한 멤버를 구분할 수 있습니다.

(3) 재사용 및 확장: 클래스를 사용하면 코드의 재사용성이 높아집니다. 클래스를 정의하면, 이를 기반으로 여러 객체를 생성할 수 있습니다. 또한 상속을 통해 기존 클래스의 기능을 확장하거나 변경할 수 있습니다.

간단한 예시로, 자동차 클래스를 생각해볼 수 있습니다. 이 클래스는 연료량(fuel), 속도(speed)와 같은 데이터를 가지며, accelerate(), brake()와 같은 함수(메소드)를 포함할 수 있습니다. 클래스를 사용하면 자동차의 상태와 행동을 한 곳에 정의하여 관리하기 편리해집니다.

```
class Car {
private:
    int fuel;
    int speed;

public:
    void accelerate() {
        // 연료를 사용하여 속도 증가
    }

    void brake() {
        // 속도 감소
    }
};
```

이렇게 클래스는 데이터와 함수를 하나의 논리적 단위로 묶어서 표현하고, 코드의 재사용성과 유지보수를 용이하게 합니다. C 언어 사용자는 구조체를 통해 이미 데이터를 그룹화하는 방법에 익숙할 것이므로, **클래스**를 '**기능을 포함하는 구조체**'로 이해하면 좋습니다.

1.2 클래스 코딩: 아두이노 LED 깜박임

클래스 개념을 사용해서, LED 깜박임을 위한 .ino 예제를 소개합니다. 클래스를 사용해서 아두이노 코드를 어떻게 작성할 수 있는지 살펴보세요. 아래 코드를 아두이노에 적용하고, 아두이노의 LED 핀번호를 5번핀에 연결하면 됩니다. LED와 저항을 아두이노에 연결하는 방법은 생략합니다.
아래 예제를 사용하는 방법은 제2장 1절에서 설명하는 'SelfAbot.zip' 라이브러리를 아두이노에 추가하고, 예제 메뉴를 열면 SelfAbot 폴더에서 사용할 수 있습니다.

Ex1.1_Cplus_LED_on_off.ino 파일을 업로드하세요.

```
#include "Arduino.h"

class SelfArduino {
public:
      SelfArduino() { }
      void setup() { }

      void digitalWrite(byte pin, byte value) {
            pinMode(pin, OUTPUT);
            ::digitalWrite(pin, value);
      }

private:

};

SelfArduino adu;
int pin = 5;
void setup() {
      adu.setup();
}

void loop() {
      adu.digitalWrite(pin, HIGH);
      delay(1000);
      adu.digitalWrite(pin, LOW);
      delay(1000);
}
```

위의 코드는 'SelfAbot'라이브러리를 사용하지 않고 클래스를 사용하는 예제이지만, 사용자의 편의를 위해 'SelfAbot'라이브러리에 포함시켜두었습니다. 여러분은 아두이노 IDE프로그램을 열어서 새스케치(new sketch)내에 위의 코드를 복사 붙여넣기 해도 됩니다.

위의 코드를 설명하면 다음과 같습니다.
아두이노의 디지털 핀과 상호 작용하기 위한 클래스 'SelfArduino'의 사용을 보여줍니다. 이 코드는 C++ 클래스와 아두이노 프로그래밍의 기본 원리를 따르며 간단합니다. 아래에 간략한 설명이 있습니다:

(1) 아두이노 라이브러리 포함:
#include "Arduino.h"
이 줄은 표준 아두이노 라이브러리를 포함하여 아두이노의 일반적인 함수와 타입에 대한 접근을 제공합니다.

(2) 클래스 정의 - SelfArduino:

```
class SelfArduino {
public:
      SelfArduino() { }
      void setup() { }
      void digitalWrite(byte pin, byte value) {
            pinMode(pin, OUTPUT);
```

```
            ::digitalWrite(pin, value);
        }
    private:
        // 여기에 비공개 멤버가 들어갈 것입니다
    };
```

‘SelfArduino’는 사용자 정의 클래스입니다. 사용자가 직접 이름을 만듭니다.
클래스를 초기화하는데 사용할 수 있는 생성자 SelfArduino()가 있습니다.

setup() 메소드는 비어 있지만 초기 설정 작업에 나중에 사용될 수 있습니다.

digitalWrite() 메소드는 디지털 핀의 모드를 OUTPUT으로 설정한 후 해당 핀에 값(HIGH 또는 LOW)을 작성하는데 사용됩니다.
::digitalWrite 앞의 :: 표시는 ‘SelfArduino’클래스에 속한 메소드가 아닌 아두이노 라이브러리에서 전역 함수를 호출한다는 것을 나타냅니다. 만약 ‘SelfArduino’클래스에서 작성한 digitalWrite(byte pin, byte value) 메소드의 이름을 다르게 짓는다면, ::digitalWrite(pin, value); 코드의 앞에 붙인 :: 이름 확장자(네임스페이스)를 사용하지 않아도 됩니다.

(3) ‘SelfArduino’클래스의 객체 생성:

```
    SelfArduino adu;
    int pin = 5;
```

클래스 ‘SelfArduino’의 객체(또는 인스턴스) adu가 생성됩니다.
pin은 5로 설정되어, 디지털 핀 5가 사용될 것임을 나타냅니다.

(4) 아두이노 setup() 함수:

```
    void setup() {
        adu.setup();
    }
```

이 함수는 아두이노가 시작할 때 한 번 실행됩니다.
adu의 setup() 메소드는 초기 설정을 위해 사용할 수 있으며, 현재는 구체적인 코드가 작성되지 않았으므로 어떤 동작도 실행하지 않습니다.

(5) 아두이노 loop() 함수:

```
    void loop() {
        adu.digitalWrite(pin, HIGH);
        delay(1000);
        adu.digitalWrite(pin, LOW);
        delay(1000);
    }
```

이 함수는 setup() 후에 반복적으로 실행됩니다.

디지털 핀 5를 HIGH(켜짐)로 설정한 후 1000 밀리초(1초) 동안 기다린 다음, LOW(꺼짐)로 설정하고 다시 1000 밀리초 동안 기다립니다. 이것은 핀 5에 연결된 LED나 장치에 깜박임 효과를 만듭니다. adu 인스턴스의 digitalWrite 메소드를 실행한 것입니다.

1.3 객체지향 프로그래밍: 캡슐화

C++ 객체지향 프로그래밍은 C언어 프로그래밍과 매우 다른 특징을 가집니다.
C언어 코딩개념을 상당부분 충족하지만, 더 다양하고 포괄적인 개념을 가집니다.

C언어 프로그래밍은 절차식 프로그래밍 방식에 기초합니다. 그래서 C언어 프로그래밍 방법은 프로그램이 진행되는 순서에 따라 나열하는 방식으로 코드를 먼저 작성하고, 그 다음 작업순서에서 입력과 출력방법에 대한 코드작성과 데이터 종류들을 나열하는 방법이 일반적 작성법입니다.

반면에 C++ 객체지향 프로그래밍은 데이터 '**캡슐화/접근 제어/재사용 및 확장**'의 원리적 접근 개념과는 별개로 다음 원리에 따라 코드를 작성하는 것이 일반적입니다.

(1) 먼저, 객체를 선언합니다.
객체의 구성에는 public 영역과 private 영역 그리고 protected 영역을 포함할 수 있습니다.
- public 영역: 객체가 생성된 클래스 외부 어디에서나 멤버에 액세스할 수 있습니다.
- private 영역: 멤버는 클래스 자체 내에서만 액세스할 수 있으며, 클래스 외부에서는 직접 액세스할 수 없습니다.
- protected 영역: 멤버는 클래스 내부 및 파생 클래스(하위 클래스)에서 액세스할 수 있지만 이러한 클래스 외부에서는 액세스할 수 없습니다.

(2) 데이터를 어떻게 다루는지에 대한 설계개념을 코딩합니다.
이를 위해 객체의 세부내용을 설계합니다.
데이터의 종류를 선언하고, 데이터를 처리하는 수단으로 메소드를 작성합니다.
세부내용에는 생성자 선언도 포함됩니다.

(3) 클래스를 사용해서 특정 '인스턴스'를 선언하고, 실제 용도에 적용하는 코드를 작성합니다.

(용어설명) ---

객체지향 프로그래밍에서 "instance"라는 용어는 종종 "인스턴스"로 번역되며, 이는 "객체"로도 이해될 수 있습니다. 프로그래밍에서 "인스턴스"는 구체적으로 어떤 객체나 구조의 구체적인 발생을 의미하는 반면, "객체(object)"는 객체에 대한 보다 일반적인 용어입니다. 클래스의 객체를 생성하면 본질적으로 해당 클래스의 "인스턴스"가 생성됩니다. 각 인스턴스에는 클래스에 의해 정의된 고유한 속성 및 메소드 세트가 있지만 이러한 속성의 특정 값과 동작 상태는 인스턴스마다 다를 수 있습니다.

캡슐화는 객체의 일부 멤버를 비공개로 만들고 이러한 비공개 멤버에 액세스하고 상호 작용할 수 있는 공개 메소드를 제공하는 것을 의미합니다. 데이터를 조작하는 코드와 데이터를 묶고 객체의 무결성을 보호하기 위해 객체의 일부 구성 요소에 대한 직접 액세스를 제한하는 것입니다. 무단 액세스 및 수정을 방지하기 위한 클래스 디자인의 핵심 개념으로, 모듈성을 촉진하고 외부 간섭과 객체 내부 상태의 오용을 방지합니다.

아래 작성된 아두이노 예제 코드는 캡슐화 기능을 추가해서, 개선된 아두이노 class 파일을 제시합니다. 이전과 비교해보세요. 캡슐화된 코드를 작성하기 위해, 우리는 일부 멤버를 비공개로 설정하고 이들과 상호 작용할 수 있는 공개 방법을 제공할 것입니다.

Ex1.2_CplusCapsule_LED_on_off.ino 파일을 참고하세요.

```
#include "Arduino.h"

class SelfArduino {
public:
    SelfArduino() { }

    // Public method to setup pins
    void setup(byte pin) {
        _pin = pin;
        pinMode(_pin, OUTPUT);
    }

    // Public method to write value to the pin
    void digitalWrite(byte value) {
        ::digitalWrite(_pin, value);
    }

private:
    byte _pin;  // Private member variable to store the pin number

    // You can add more private methods and variables here
};

SelfArduino adu;
int pin = 5;

void setup() {
    adu.setup(pin); // Setup the pin for abot
}

void loop() {
    adu.digitalWrite(HIGH); // Turn on the pin
```

```
    delay(1000);
    adu.digitalWrite(LOW);   // Turn off the pin
    delay(1000);
}
```

LED의 불빛을 켜고 끄는 캡슐화된 수정 버전 예제를 설명합니다.

(1) _pin은 핀 번호를 저장하는데 사용되는 'SelfAbot' 클래스의 전용 멤버 변수입니다. 이 변수는 클래스 외부에서 직접 액세스할 수 없으므로 클래스 내에 캡슐화됩니다.

(2) setup 메소드가 공개되어 핀을 초기화하는데 사용됩니다. 개인 _pin 변수를 설정하고 핀 모드를 초기화합니다.

(3) 'digitalWrite' 메소드에는 값('HIGH' 또는 'LOW')만 필요합니다. 개인 _pin 변수로 지정된 핀에 씁니다. 이 메소드는 비공개 _pin 멤버에 대한 제어된 액세스를 제공합니다.

캡슐화 예제는 객체의 내부 상태를 숨기고 외부 세계와 상호 작용하는데 필요한 것만 노출하는 방법을 보여줍니다. 이 접근 방식을 사용하면 더 나은 모듈식 설계가 가능해지며 코드를 더 쉽게 유지 관리하고 확장할 수 있습니다.

(참고) --

pin 변수: 비공개 변수명 앞에 밑줄('') 기호를 사용하는 이유

(1) 범위 명확성: 밑줄 접두사는 비공개 멤버(변수 또는 메서드)를 공개 또는 보호된 멤버와 구별하는데 도움이 됩니다.

(2) 이름 충돌 방지: 클래스 멤버의 밑줄 접두사는 이름 충돌을 피하는데 도움이 되며 클래스 멤버가 지역 변수 대신 사용될 때 이를 명확하게 해줍니다. 특히, Getter/Setter 메소드 사용의 일부 코딩 관행에서는 변수에 대한 getter 및 setter 메소드가 변수와 동일한 이름을 갖습니다. 멤버 변수에 밑줄을 사용하면 메서드와 변수를 구별하는데 도움이 됩니다(예: getter의 경우 'age', 전용 멤버의 경우 '_age').

(3) 규칙과 가독성: 비공개 멤버에 대한 밑줄 접두사는 코드베이스 전체의 일관성을 보장하기 위해 이러한 표준의 일부가 될 수 있습니다. 코드를 시각적으로 분할하여 클래스 내부에 어떤 변수가 있는지 한 눈에 더 쉽게 읽고 이해할 수 있습니다.

(4) 역사적 및 언어별 규칙: Python은 이름 변경에 이중 밑줄(__)을 사용합니다. 이는 변수가 비공개임을 나타내는 더 강력한 규칙이지만 단일 밑줄은 일반적으로 "소프트" 비공개 표시기로 사용됩니다. 다른 개발자에게 비공개 멤버로 처리되어야 한다고 말하는 것이 관례에 가깝습니다. C++ 및 기타 언어에서 단일 밑줄 접두사는 C++와 같은 언어에서도 사용되지만 이는 언어 요구 사항이라기보다는 관례에 가깝습니다.

아래 참고사항은 클래스 파일이 왜 헤더파일과 소스파일로 분리되어 있는지, 또는 어떻게 분리되어 만들어지는지를 설명하는 내용입니다.

(참고) ---

class 파일을 헤더파일(.h)과 소스파일(.cpp)로 분리하는 방법

C++에서 클래스 정의를 헤더(.h) 파일과 구현(.cpp) 파일로 분할하는 것은 다음과 같은 여러 가지 이유로 일반적인 관행입니다.

(1) 인터페이스와 구현의 분리: 이러한 분리를 통해 인터페이스(.h 파일에 정의됨)를 구현 세부 정보(.cpp 파일에 있음)와 분리할 수 있습니다. 헤더 파일은 클래스(또는 함수)가 "무엇을" 수행할지 선언하고, 구현 파일은 클래스(또는 함수)가 "어떻게" 수행할지 정의합니다. 이러한 추상화는 객체 지향 프로그래밍의 핵심 원칙인 캡슐화에 유용합니다.

(2) 컴파일 시간 단축: .cpp 파일의 구현을 변경하면 .h 파일을 포함하는 모든 파일이 아닌 해당 파일만 다시 컴파일하면 됩니다. 이렇게 하면 대규모 프로젝트에서 컴파일 시간을 크게 줄일 수 있습니다.

(3) 다중 정의 방지: C++에서 단일 정의 규칙(ODR)은 엔터티(예: 클래스, 함수 또는 변수)가 코드에서 정확히 하나의 정의를 가져야 함을 나타냅니다. .cpp 파일에 정의를 배치하고 .h 파일에 선언을 배치하면 여러 소스 파일에 동일한 헤더가 포함된 경우 발생할 수 있는 동일한 엔터티의 여러 정의를 방지할 수 있습니다.

(4) 향상된 가독성 및 구성: 인터페이스를 구현과 분리하여 유지하면 코드를 더 읽기 쉽고 탐색하기가 더 쉬워집니다. 개발자는 구현 세부 사항에 얽매이지 않고 헤더 파일을 살펴봄으로써 클래스의 인터페이스를 빠르게 이해할 수 있습니다.

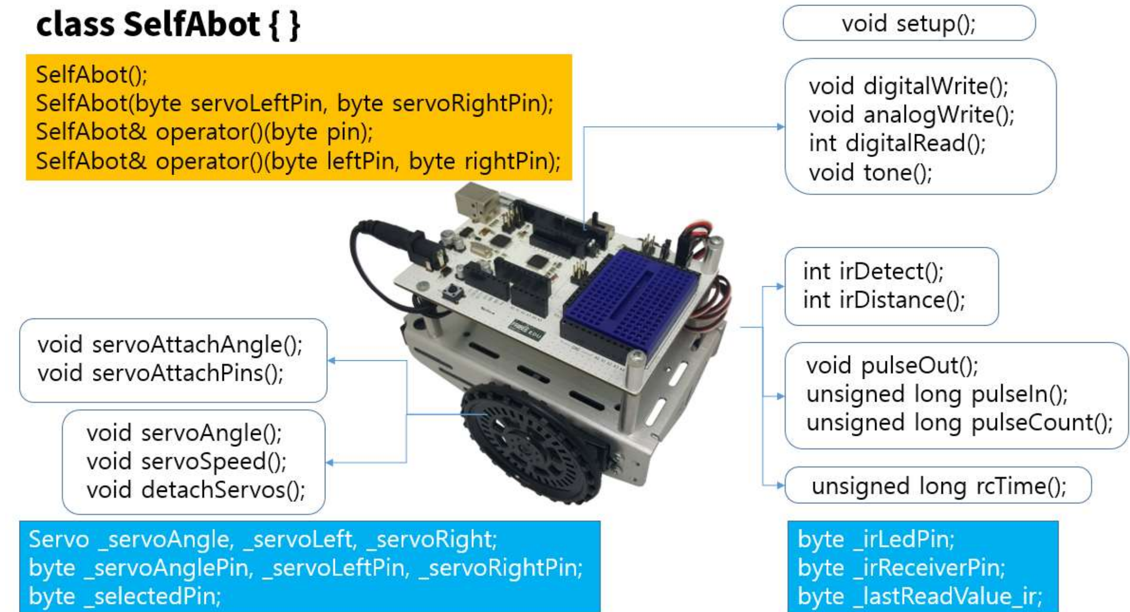
(5) 팀 개발에 더 좋음: 팀 환경에서는 서로 다른 팀 구성원이 코드의 서로 다른 부분에서 작업할 수 있습니다. 명확한 분리를 통해 많은 충돌 없이 병렬 개발이 가능합니다. 한 팀 구성원은 구현 작업을 수행하고 다른 구성원은 인터페이스 작업을 수행하거나 프로젝트의 다른 부분에서 인터페이스를 사용할 수 있습니다.

(6) 재사용성과 모듈성 장려: 구현을 분리함으로써 동일한 헤더 파일을 여러 프로젝트에서 공유할 수 있습니다. 이러한 모듈성 덕분에 코드를 더 쉽게 재사용할 수 있습니다.

(7) 전방 선언 및 종속성 감소: 헤더 파일에서 전방 선언을 사용하면 다른 헤더를 포함할 필요성이 줄어들어 코드베이스의 여러 부분 간의 결합 및 종속성이 줄어듭니다.

요약하면, C++에서 .h와 .cpp 파일을 분리하는 것은 단지 스타일적인 선택이 아니라 언어의 원칙을 지원하고 특히 더 크고 복잡한 프로젝트에서 개발 프로세스를 개선하는 근본적인 관행입니다.

1.4 ‘SelfAbot’ 클래스 소개



‘neo-아두이노로봇’을 동작시키는 클래스를 설계하고, 적용하기 위한 ‘SelfAbot’ 클래스를 소개합니다.

아래에서는 앞서 설명한 클래스 개념에 기초해서 ‘SelfAbot’ 클래스보다 일반화된 개념을 익힐 수 있도록 확장된 개념으로 설명합니다. 따라서 아래 설명이 실제 ‘SelfAbot’ 클래스 개념보다 넓은 의미의 소개라는 점을 알려드립니다.

(1) ‘SelfAbot’ 클래스:
‘neo-아두이노로봇’ 클래스의 청사진입니다. 로봇의 동작기능(메소드)과 특성(속성/변수)을 정의합니다. 그외에는 아무것도 하지 않습니다.

(2) ‘SelfAbot’ 객체 디자인:
객체 디자인은 .h 와 .cpp 파일로 이루어진 클래스 전체 코드를 설계하는 과정을 말합니다. 세부적인 요소들을 아래에 나열해두었으므로 살펴보기 바랍니다. 아래에 설명한 다양한 측면을 고려해서 클래스를 디자인해야 합니다.

<u>‘SelfAbot’ 객체 개념화:</u>
'SelfAbot' 클래스가 할 수 있거나 가질 수 있는 것(예: 앞으로 이동, 뒤로 이동 또는 서보 상태)에 대해 생각할 때 디자인 측면에서 'SelfAbot' 객체를 구체적으로 설명하는 것입니다.

<u>메소드 및 속성 정의:</u>

예를 들어 **메소드 코드**로는 앞으로 이동하거나, 정지하거나, 방향을 바꾸거나, 센서와 상호작용하는 등의 작업을 수행할 수 있습니다. 그리고 'SelfAbot' 로봇의 특성이나 상태를 묘사하는 **속성 변수**로는 서보에 연결된 핀을 나타내는 변수, 현재 속도, 센서 값 등이 포함될 수 있습니다.

추상화 및 캡슐화:
추상화: 'SelfAbot' 객체를 디자인할 때 복잡성을 추상화합니다. 서보가 전기 수준에서 작동하는 방식이나 마이크로 컨트롤러의 I/O 핀의 세부 사항에는 관심이 없습니다. 대신 더 높은 수준의 기능에 중점을 둡니다.
캡슐화: 이 설계 프로세스에는 외부에서 숨겨야 하는 내부 세부 정보(캡슐화)와 공개 인터페이스로 노출되어야 하는 세부 정보(공개 메소드 및 속성)를 결정하는 작업이 포함됩니다.

상호작용 및 관계:
다른 객체와의 상호 작용: 디자인의 일부는 'SelfAbot'이 다른 객체 또는 시스템과 상호 작용하는 방식을 이해하는 것입니다. 여기에는 명령을 수신하고, 센서 데이터를 처리하고, 외부 장치와 통신하는 방법이 포함될 수 있습니다.
관계 및 계층: 다양한 유형의 로봇이나 구성 요소가 있는 경우 상속이나 구성을 염두에 두고 'SelfAbot'을 설계할 수 있습니다. 예를 들어 'SelfAbot'은 보다 전문화된 로봇 유형의 기본 클래스가 될 수도 있고, 더 작고 집중된 클래스에서 기능을 구성할 수도 있습니다.

사용 사례 및 시나리오:
사용 사례 구상: 'SelfAbot'이 사용될 다양한 시나리오를 상상하는 것은 객체를 설계하는데 도움이 됩니다. 예를 들어 미로 탐색, 선 따라가기, 센서 입력에 반응하기 등이 있습니다. 이러한 사용 사례는 관련 방법 및 속성의 개발을 요구합니다.

유연성 및 확장성을 위한 계획:
적응성: 디자인에서는 'SelfAbot' 객체를 얼마나 쉽게 적용하거나 확장할 수 있는지 고려해야 합니다. 예를 들어 새로운 기능을 추가하거나 다양한 하드웨어 구성에 적응해야 합니다.
요약하자면, 디자인된 'SelfAbot' 객체는 로봇 시스템이 무엇을 할 수 있어야 하며 어떤 특성을 가지고 있는지에 대한 개념적 모델입니다. 추상화, 캡슐화, 적응성과 같은 원칙을 염두에 두면서 기능, 속성, 상호 작용 및 잠재적 사용 사례를 정의하는 것이 혼합되어 있습니다.

(3) 코드에서의 'SelfAbot' 인스턴스:
실제 적용할 코드에서 SelfAbot myRobot;과 같은 코드를 작성하면 myRobot은 'SelfAbot' 클래스의 인스턴스입니다. 프로그램에서 상호 작용할 수 있는 특정 로봇 객체입니다. 많은 인스턴스(예: myRobot, yourRobot 등)를 구분해서 만들 수 있으며, 각 인스턴스는 'SelfAbot' 클래스에서 정의한 대로 고유한 상태와 동작을 가진 실제 엔터티가 됩니다.

'SelfAbot' 클래스는 추상적인 청사진이고, 객체 디자인은 일반화하는 코드 개념이며, 인스턴스는 해당 클래스의 개념을 코드의 구체적인 형태로 구현한 것입니다. ‘SelfAbot’ 클래스의 헤더파일 (SelfAbot.h)과 소스파일 (SelfAbot.cpp)은 제1장 끝에서 소개합니다.

1.5 클래스: 생성자, 멤버 변수, 상수

(1) 생성자
기본 생성자:
기본 생성자는 인수 없이 객체가 생성될 때 호출되는 특수한 유형의 생성자 메소드입니다.
'SelfAbot'의 경우 기본 생성자는 서보의 핀 번호나 기본 상태와 같은 멤버 변수의 초기 값을 설정할 수 있습니다.
예: SelfAbot::SelfAbot() { _selectedPin = INVALID_PIN; }

매개변수화된 생성자:
이 유형의 생성자는 객체를 생성할 때 매개변수를 전달하여 특정 값으로 객체를 초기화할 수 있도록 해줍니다.
'SelfAbot'의 경우 매개변수화된 생성자는 왼쪽 및 오른쪽 서보의 핀 번호를 인수로 사용하여 사용자가 이러한 구성 요소에 사용할 핀을 지정할 수 있습니다.
예: SelfAbot::SelfAbot(byte ServoLeftPin, byte ServoRightPin) { ... }

(2) 멤버 변수와 상수
멤버 변수:
클래스 내부에 정의된 변수입니다. 객체의 상태를 유지합니다. 'SelfAbot'에서 멤버 변수에는 서보의 핀 번호, 현재 속도 또는 상태 플래그가 포함될 수 있습니다.
예: byte _servoLeftPin;

상수:
상수는 한번 설정하면 변하지 않는 값입니다. C++에서는 const 또는 #define을 사용하여 정의되는 경우가 많습니다. 'SelfAbot'의 경우 INVALID_PIN과 같은 상수를 사용하여 유효하지 않거나 정의되지 않은 핀 번호를 나타낼 수 있습니다.
예: const byte INVALID_PIN = 255;

1.6 멤버 함수와 오버로딩

(1) 멤버 함수 (또는 ‘메소드’ ‘메서드’라고도 함)
멤버 함수는 클래스에서 생성된 객체가 수행할 수 있는 작업(또는 동작)을 작성한 코드입니다. arduino 또는 C++ 기반 환경의 컨텍스트에서 클래스를 다룰 때 로봇의 청사진을 만드는 것입니다. 예를 들면, 아두이노 클래스에서 digitalWrite()는 코드가 지정한 핀에서 디지털 출력 5V와 0V를 만드는 동작을 합니다. SelfAbot 클래스에서도 다양한 멤버 함수를 정의해두었으며, 이후에서 더 자세히 설명할 예정입니다.

(2) 함수 오버로딩
함수 오버로딩은 두 개 이상의 함수(또는 메소드)가 이름은 같지만 매개변수가 다를 수 있는 객체 지향 프로그래밍의 기능입니다. 유사한 작업을 수행하지만 다른 입력이 필요할 수 있는

동일한 이름을 가진 여러 메소드를 만드는 방법입니다. 그래서, 함수 오버로딩을 사용하면 여러 메소드가 이름은 같지만 매개변수는 다를 수 있습니다.

이는 메소드를 사용하는 방법에 유연성을 제공합니다. 아래 예제 코드는 함수 오버로딩을 간단히 설명하지만, 실제 아두이노로봇의 'SelfAbot'클래스 코드를 사용해서 함수 오버로딩이 어떻게 사용되고, 어떤 효과가 있는지 살펴볼 기회가 있습니다.

클래스 메소드의 두 가지 사용 예제:

void SelfAbot::digitalWrite(byte value)와
void SelfAbot::digitalWrite(byte pin, byte value)는 모두 유효하며 다른 용도로 사용됩니다.

예를 들면, digitalWrite() 함수에서 매개변수 pin 과 value 값을 함께 매개변수로 전달하거나, 함수 오버로딩을 사용해서 인스턴스 값으로 pin 값을 전달하고 value 값만 매개변수로 전달하는 것이 가능합니다. abot 인스턴스 코드를 표현할 때, 다음 두가지 형태가 모두 가능합니다.

아두이노 .ino 코드에서 함수오버로딩을 사용하지 않는 경우와 사용하는 경우를 표현한 예입니다. 아래 두 가지 코드의 사용형태는 실제로 동일한 코드실행 결과를 보여줍니다. 단지 코드의 표현만 다르게 나타냅니다.

```
SelfAbot abot; // abot 인스턴스 생성

// 함수 오버로딩을 사용하지 않은 예제코드
abot.digitalWrite(pin, value);

// 함수 오버로딩을 사용하는 예제코드
abot(pin).digitalWrite(value);
```

부록: 'SelfAbot.zip'라이브러리 헤더와 소스 파일 (SelfAbot.h & SelfAbot.cpp)

```
//'arduino robot class library SelfAbot.h' @0.1.10
#ifndef SelfAbot_h
#define SelfAbot_h

#include "Arduino.h"
#include <Servo.h>

class SelfAbot {
public:
	SelfAbot();
	SelfAbot(byte servoLeftPin, byte servoRightPin);
	SelfAbot& operator()(byte pin);
	SelfAbot& operator()(byte leftPin, byte rightPin);

	void setup();

	void digitalWrite(byte pin, byte value);
	void digitalWrite(byte value);
	void analogWrite(byte pin, int value);
	void analogWrite(int value);
	int digitalRead(byte pin);
	int digitalRead();
	void tone(byte pin, unsigned int frequency, unsigned long duration);
	void tone(unsigned int frequency, unsigned long duration);

	void servoAttachAngle(byte servoAnglePin); //
	void servoAttachPins(byte servoLeftPin, byte servoRightPin); //

	void servoAngle(int angle);
	void servoSpeed(int speed);
	void servoSpeed(int leftSpeed, int rightSpeed);
	void detachServos();

	int irDetect(byte irLedPin, byte irReceiverPin, int frequency);
	int irDistance(byte irLedPin, byte irReceivePin);

	void pulseOut(byte pin, unsigned long duration);
	unsigned long pulseIn(byte pin, int state);
	unsigned long pulseCount(byte pin, unsigned long duration);

	unsigned long rcTime(byte pin);


private:

	byte _servoAnglePin, _servoLeftPin, _servoRightPin;
	Servo _servoAngle, _servoLeft, _servoRight;
	byte _selectedPin;

	byte _irLedPin;
	byte _irReceiverPin;
	byte _lastReadValue_ir;
};

#endif
```

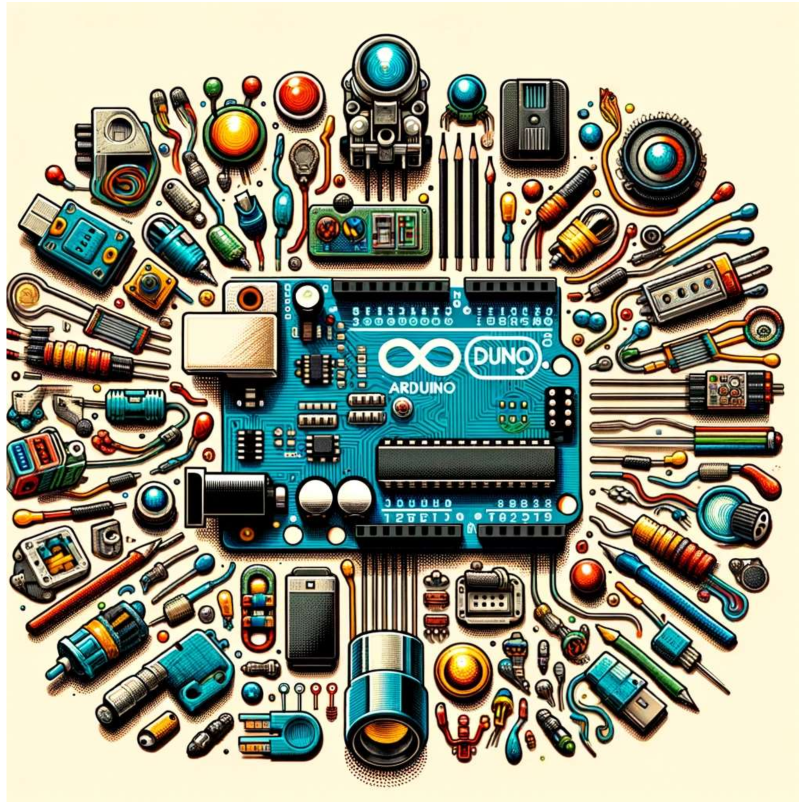
```
//'arduino robot class library SelfAbot.h' @0.1.10
#include "SelfAbot.h"
#define INVALID_PIN 255

int SelfAbot::_lastReadValue = 0;

SelfAbot::SelfAbot() {
	_selectedPin = INVALID_PIN;
}

SelfAbot::SelfAbot(byte servoLeftPin, byte servoRightPin)
	: _servoLeftPin(servoLeftPin), _servoRightPin(servoRightPin) {
	_selectedPin = INVALID_PIN;
}

SelfAbot& SelfAbot::operator()(byte pin) {
	_selectedPin = pin;
	return *this;
}

SelfAbot& SelfAbot::operator()(byte leftPin, byte rightPin) {
	_servoLeftPin = leftPin;
	_servoRightPin = rightPin;
	return *this;
}

void SelfAbot::setup() {
	if (_servoLeftPin != INVALID_PIN && _servoRightPin != INVALID_PIN) {
		_servoLeft.attach(_servoLeftPin);
		_servoRight.attach(_servoRightPin);
	}
}

// Original method
void SelfAbot::digitalWrite(byte pin, byte value) {
	pinMode(pin, OUTPUT);
	::digitalWrite(pin, value);
}
// Overloaded method
void SelfAbot::digitalWrite(byte value) {
	if (_selectedPin != INVALID_PIN) {
		pinMode(_selectedPin, OUTPUT);
		::digitalWrite(_selectedPin, value);
	}
}

// Original method
void SelfAbot::analogWrite(byte pin, int value) {
	::analogWrite(pin, value);
}
// Overloaded method
void SelfAbot::analogWrite(int value) {
	if (_selectedPin != INVALID_PIN) {
		::analogWrite(_selectedPin, value);
	}
}

// Original method
int SelfAbot::digitalRead(byte pin) {
	pinMode(pin, INPUT);
	return ::digitalRead(pin);
}
// Overloaded method
int SelfAbot::digitalRead() {
```

```
    if (_selectedPin != INVALID_PIN) {
        pinMode(_selectedPin, INPUT);
        return ::digitalRead(_selectedPin);
    }
}

// Original method
void SelfAbot::tone(byte pin, unsigned int frequency, unsigned long duration) {
    ::tone(pin, frequency, duration);
}
// Overloaded method
void SelfAbot::tone(unsigned int frequency, unsigned long duration) {
    if (_selectedPin != INVALID_PIN) {
        ::tone(_selectedPin, frequency, duration);
    }
}

// Original method
void SelfAbot::servoAttachPins(byte servoLeftPin, byte servoRightPin) {
    if (servoLeftPin != INVALID_PIN) {
        _servoLeftPin = servoLeftPin;
        _servoLeft.attach(_servoLeftPin);
    }
    if (servoRightPin != INVALID_PIN) {
        _servoRightPin = servoRightPin;
        _servoRight.attach(_servoRightPin);
    }
}

// Original method
void SelfAbot::servoAttachAngle(byte servoAnglePin) {
        if (servoAnglePin != INVALID_PIN) {
                _servoAnglePin = servoAnglePin;
        _servoAngle.attach(_servoAnglePin);
        }
}
// Overloaded method
void SelfAbot::servoAngle(int angle) {
    _servoAngle.write(angle);
}
// Overloaded method
void SelfAbot::servoSpeed(int speed) {
    if (_selectedPin == _servoLeftPin) {
        _servoLeft.writeMicroseconds(1500 + speed);
    } else if (_selectedPin == _servoRightPin) {
        _servoRight.writeMicroseconds(1500 + speed);
    }
}
// Overloaded method
void SelfAbot::servoSpeed(int leftSpeed, int rightSpeed) {
    _servoLeft.writeMicroseconds(1500 + leftSpeed);
    _servoRight.writeMicroseconds(1500 + rightSpeed);
}
// Overloaded method
void SelfAbot::detachServos() {
    _servoLeft.detach();
    _servoRight.detach();
}

// Original method
int SelfAbot::irDetect(byte irLedPin, byte irReceiverPin, int frequency) {
        _irLedPin = irLedPin;
        _irReceiverPin = irReceiverPin;
        pinMode(_irLedPin, OUTPUT);
```

```
    pinMode(_irReceiverPin, INPUT);
    tone(_irLedPin, frequency, 8);
    delay(1);
    int _lastReadValue_ir = ::digitalRead(_irReceiverPin);
    delay(1);
    return _lastReadValue_ir;
}

// Original method
int SelfAbot::irDistance(byte irLedPin, byte irReceivePin) {
        int distance = 0;
    for(long f = 38000; f <= 42000; f += 500) {
        distance += irDetect(irLedPin, irReceivePin, f);
    }
    return distance;
}

// Original method
void SelfAbot::pulseOut(byte pin, unsigned long duration) {
    pinMode(pin, OUTPUT);
    digitalWrite(pin, HIGH);
    delayMicroseconds(duration);
    digitalWrite(pin, LOW);
}

// Original method
unsigned long SelfAbot::pulseIn(byte pin, int state) {
    pinMode(pin, INPUT);
    return ::pulseIn(pin, HIGH);
}

// Original method
unsigned long SelfAbot::pulseCount(byte pin, unsigned long duration) {
    pinMode(pin, INPUT);
    unsigned long startTime = micros();
    unsigned long endTime = startTime + duration;
    unsigned long count = 0;

    while (micros() < endTime) {
        if (::digitalRead(pin) == HIGH) {
            count++;
            while (::digitalRead(pin) == HIGH);
        }
    }

    return count;
}

// Original method
unsigned long SelfAbot::rcTime(byte pin) {
    pinMode(pin, OUTPUT);
    digitalWrite(pin, HIGH);
    delay(1);
    pinMode(pin, INPUT);
    digitalWrite(pin, LOW);
    unsigned long startTime = micros();
    unsigned long elapsedTime = 0;
    while (digitalRead(pin) == HIGH) {
        elapsedTime = micros() - startTime;
    }
    return elapsedTime;
}
```

제 2 장 아두이노 함수 및 오버로딩

2.1 'SelfAbot'라이브러리 아두이노에 추가하기
2.2 아두이노 함수 실행하기
2.3 단순화를 위한 함수 오버로딩



이 책은 ‘아두이노 우노(arduino uno)’ 마이크로컨트롤러를 사용하는 교육용로봇으로 C++ 객체지향프로그래밍을 익히는 교재입니다. C++ 코딩을 배우도록 고안된 ‘SelfAbot’클래스에 아두이노 표준 입출력 함수들을 포함했습니다. 그래서 C++ 로봇 코딩으로 들어가기 이전에 아두이노 표준함수들이 클래스에서 어떻게 사용되는지 먼저 배우도록 안내합니다.

C++ 아두이노 로봇의 동작에 더 흥미를 가진다면 제 2 장을 건너뛰고 제 3 장을 먼저 들어가도 좋습니다.

2.1 ‘SelfAbot’ 라이브러리 아두이노에 추가하고 .ino 예제 활용하기

C++ 언어 클래스 개념을 아두이노로봇으로 쉽게 익힐 수 있게 고안된 ‘SelfAbot’라이브러리는 글머리에서도 소개했지만, 프라이봇 홈페이지(https://fribot.com) 공지사항 안내를 받거나, 깃허브(https://github.com/wookjin-chung/SelfAbot)에 공개한 파일을 다운로드하면 됩니다. 제 1 장의 클래스 개념설명이 지루했던 분이어도 지금부터는 직접 실습을 통해서 흥미를 느낄 수 있도록 안내합니다. **(참고: 깃허브 다운로드후 “SelfAbot.zip”으로 이름 변경해야 합니다.)**

아두이노에서 라이브러리를 추가하는 방법은 여러 가지입니다. 여기서는 직접 'SelfAbot' 라이브러리 추가 방법을 설명합니다. 아두이노 IDE통합프로그램을 다운로드받고 라이브러리를 설치하세요. 실습에서 사용하는 ‘아두이노로봇’은 아두이노 우노(arduino UNO)기반의 ‘프라이비(fribee EDU)’ 아두이노우노 호환보드입니다.

<u>아두이노 IDE통합 프로그램에 라이브러리 추가하기</u>

(1) SelfAbot.zip 파일을 다운로드받습니다. (※ gitHub 에서 다운로드하세요)
(2) 아두이노 IDE에서 라이브러리 추가를 선택합니다.

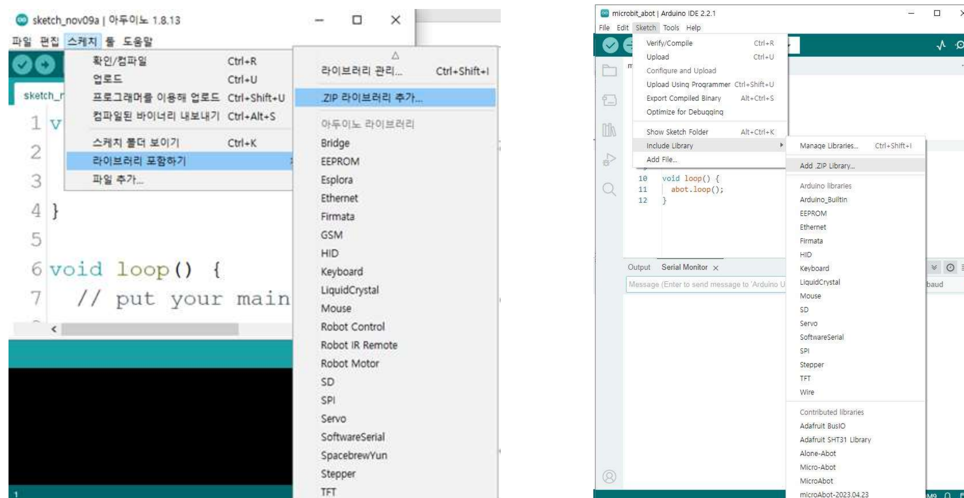


그림2.1(a) 아두이노 IDE프로그램 구버전　　　(b) 아두이노 IDE프로그램 신버전

당신의 컴퓨터에서 조금 전 다운로드 받은 ‘SelfAbot.zip’ 파일을 클릭 선택합니다.
(3) 이제 당신의 컴퓨터에서는 'SelfAbot'라이브러리를 사용할 준비가 되었습니다.

<u>아두이노 ‘SelfAbot’ 라이브러리 .ino 예제 활용하기</u>

아두이노 프로그램에 라이브러리를 추가하면, File->Examples->SelfAbot 폴더에 포함된 다양한 ino 예제를 활용할 수 있습니다. 예제들은 각 장별로 구분되어 있어서, 이 책의 각장에서 소개하는 예제를 쉽게 찾아서 사용할 수 있습니다.

아두이노의 ‘SelfAbot’라이브러리 폴더에는 SelfAbot 이외에 EnhancedSelfAbot 라이브러리도 함께 포함되어 있습니다.

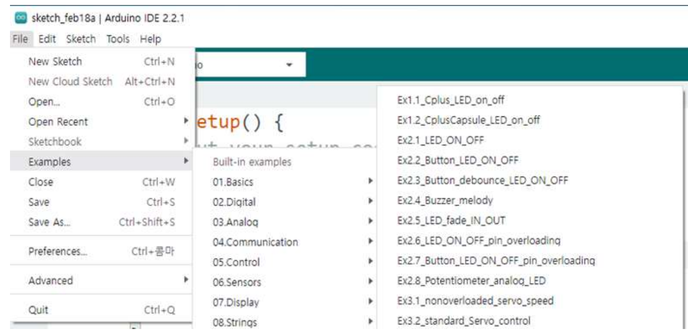


그림2.2 : 아두이노 SelfAbot 라이브러리 예제 활용하기

7장까지 내용이 'SelfAbot'라이브러리를 활용한 예제라면, 8장은 클래스 상속을 활용한 하위 라이브러리 'EnhancedSelfAbot'를 활용한 예제를 사용할 수 있습니다. 로봇을 더 섬세하고 간명하게 다루는 방법을 활용해보세요.

2.2 'SelfAbot'을 사용해서 아두이노 함수 실행하기

2.2.1 digitalWrite() 함수로 LED 켜고 끄기

아두이노 5번핀에 LED 양극(+)을 연결하고, 이어서 직렬로 220Ω 저항을 연결한 다음 접지(- : GND)에 연결하면 됩니다. LED연결방법을 나타낸 그림2.2를 참고하세요. 하드웨어 구성을 마쳤다면, 아래 코드를 아두이노에 업로드해서 LED 불빛이 1초 간격으로 켜지고 꺼지는지 관찰해보세요.

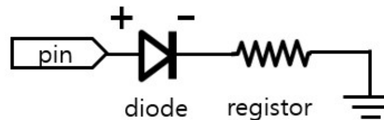


그림2.2 : 아두이노 핀에 LED회로를 연결하는 방법

아래 C++언어 형식의 Ex2.1_LED_ON_OFF.ino 코드(좌측 코드)는 C언어 형식으로 아두이노 LED를 켜고 끄는 코드(우측 코드)와 어떤 차이점이 있나요?

Ex2.1_LED_ON_OFF.ino 파일을 업로드하세요. 왼쪽 코드는 C++언어로 작성한 코드이고, 오른쪽 코드는 C언어로 작성한 코드입니다.

```
#include "SelfAbot.h"
SelfAbot abot;
int pin = 9;

void setup() {
   // abot.setup();
}
void loop() {
      abot.digitalWrite(pin, HIGH);
      delay(1000);
      abot.digitalWrite(pin, LOW);
      delay(1000);
}
```

```
int pin = 9;

void setup() {
      pinMode(pin, OUTPUT);
}
void loop() {
      digitalWrite(pin, HIGH);
      delay(1000);
      digitalWrite(pin, LOW);
      delay(1000);
}
```

#include "SelfAbot.h" 이 줄에는 'SelfAbot'클래스 정의가 포함됩니다. 이는 arduino IDE에 SelfAbot.h 파일에 정의된 코드를 사용하도록 지시합니다. 라이브러리 코드는 이 스케치와 동일한 폴더 또는 arduino 라이브러리 폴더에 있어야 합니다.

SelfAbot abot; 여기서 SelfAbot 클래스의 abot이라는 객체(또는 인스턴스)가 생성됩니다. 개인용 자동차가 자동차의 일반적 개념의 특정 인스턴스인 것과 마찬가지로 'abot'을 'SelfAbot' 클래스의 특정 인스턴스로 생각하세요.

int pin = 9; 이 줄은 pin이라는 정수 변수를 선언하고 값 9로 초기화합니다. 이 숫자는 이 스케치에 사용될 arduino 보드의 디지털 핀 번호를 나타냅니다.

//abot.setup(); 이 줄은 주석 처리되어 있습니다(//는 주석을 나타냄). 활성화된 경우 abot 인스턴스에 대한 setup 메소드를 호출합니다. 이 방법은 핀 모드 설정이나 구성 요소 초기화와 같은 'SelfAbot' 클래스의 초기 설정에 사용됩니다.

abot.digitalWrite(pin, HIGH); abot 인스턴스에 있는 'SelfAbot'클래스의 digitalWrite 메소드를 호출합니다. 디지털 핀(이 경우 핀 번호 '9')을 'HIGH'로 설정합니다. 이는 5V를 의미하며 해당 핀을 켭니다. 아래 메소드 내에 pinMode() 설정이 포함되어서, 별도로 코드실행하지 않아도 됩니다.

```
void SelfAbot::digitalWrite(byte pin, byte value) {
    pinMode(pin, OUTPUT);
    ::digitalWrite(pin, value);
}
```

delay(1000); 이 함수는 1000밀리초(또는 1초) 동안 프로그램을 일시 중지합니다. 여기서는 짧은 시간 동안 핀을 높게 유지하는데 사용됩니다.

abot.digitalWrite(pin, LOW); 이전 digitalWrite와 유사하지만 이번에는 핀을 0V인 LOW로 설정하여 해당 핀에 연결된 모든 것을 끕니다.

delay(1000); 이번에도 프로그램을 1초 더 일시 중지하고 해당 시간 동안 핀을 낮게 유지합니다.

2.2.2 digitalRead() 함수로 버튼 입력 출력하기입니다.

아래 코드는 조금 전 설명한 코드의 맥락과 같이 이해되지만, ‘SelfAbot’클래스의 새로운 메소드 digitalRead()를 사용하는 점이 다릅니다.

그림2.3은 디지털신호 버튼입력을 아두이노 핀으로 입력받고, 동시에 버튼입력 신호에 따라 디지털신호를 출력할 LED회로를 함께 연결하는 방법입니다. LED핀과 버튼 핀의 핀번호는 사용자의 스케치 번호와 일치시키면 됩니다.

abot.digitalRead(buttonPin) 코드는 아두이노의 digitalRead 함수를 사용하는 정의된 메소

드입니다. abot 인스턴스와 함께 사용되어, buttonState 변수에 값이 저장됩니다. 그리고 if 조건문을 사용해서 변수값이 '1'일때와 '0'일 때 digitalWrite 함수가 각각 다르게 동작하도록 작성되었습니다.

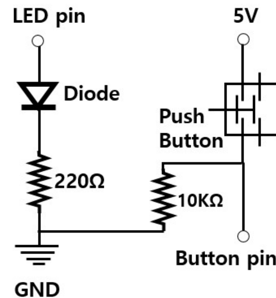


그림2.3 : 아두이노핀에 버튼을 연결하는 방법 (풀다운 방식)

Ex2.2_Button_LED_ON_OFF.ino 파일을 업로드하세요.

```
#include "SelfAbot.h"
SelfAbot abot;
const int buttonPin = 8;
const int ledPin =  9;
int buttonState = 0;

void setup() {
  //  abot.setup();
}
void loop() {
  buttonState = abot.digitalRead(buttonPin);
  if (buttonState == HIGH) {
    abot.digitalWrite(ledPin, HIGH);
  } else {
    abot.digitalWrite(ledPin, LOW);
  }
}
```

참고로 버튼회로를 구성하는 방법은 오래전에 배포한 '아두이노로봇 가지고 놀기' 온라인 실습교재를 참고하거나, 이 책 제2장 3절에서 소개한 회로도를 참고해서 더 많은 내용을 살펴볼 수 있습니다.

버튼회로에서 한가지 더 유의할 사항은 버튼스위치가 물리적/기계적 접촉으로 디지털 신호를 발생시키기 때문에 버튼을 누르면 한번만 ON/OFF 전기적 신호를 만드는 것이 아니라, 여러 번 반복될 수 있습니다. 이런 문제는 마이크로컨트롤러가 입력신호를 처리할 때 불안정한 이유가 됩니다.

아래 소개할 **디바운싱 코드**는 버튼 상태가 변경될 때 발생하는 물리적 바운싱에 관계없이 버튼을 누르거나 놓을 때 단일 켜기/끄기 신호만 등록되도록 하는 토글(toggle) 방식입니다. 실제로 저항과 커패시터같은 외부 회로를 사용해서 디바운싱 효과를 만들 수 있지만, 소프트웨어 방식이 더 쉽고 저렴하게 구현할 수 있어서 일반적으로 사용됩니다.
디바운싱의 타이밍은 아래 코드에서 debounceDelay는 디바운스 시간을 사용자의 편의에 최

대한 적합하게 조절하면 됩니다. 버튼회로의 신호발생 신뢰도를 높이고, 사용자의 반응에 민감한 조건으로 설정해서 사용하면 됩니다.

(용어) **디바운스(debounce) 란?** ---
사용자가 여러번의 요청을 보내는 경우에 일정시간을 두어, 맨 처음 혹은 마지막에 한 번만 실행하고, 나머지 일정시간 동안에는 이벤트들을 무시하는 기법입니다.

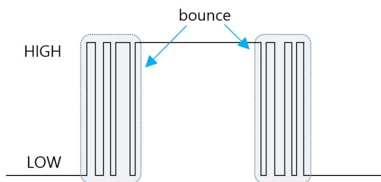


그림2.4 : 디바운스 개념도

Ex2.3_Button_debounce_LED_ON_OFF.ino 파일을 업로드하세요.

```
#include "SelfAbot.h"
SelfAbot abot;

const int buttonPin = 8;
const int ledPin =  9;

int ledState = HIGH;
int buttonState;
int lastButtonState = LOW;

unsigned long lastDebounceTime = 0;
unsigned long debounceDelay = 50;

void setup() {
  abot.setup();
  abot.digitalWrite(ledPin, ledState);
}

void loop() {
  int reading = abot.digitalRead(buttonPin);

  if (reading != lastButtonState) {
    lastDebounceTime = millis();
  }

  if ((millis() - lastDebounceTime) > debounceDelay) {
    if (reading != buttonState) {
      buttonState = reading;
      if (buttonState == HIGH) {
        ledState = !ledState;
      }
    }
  }

  abot.digitalWrite(ledPin, ledState);
  lastButtonState = reading;
}
```

2.2.3 tone() 함수로 부저 소리내기입니다.

arduino에 연결된 부저를 사용하여 멜로디를 연주하는 arduino 스케치입니다. tone() 함수를 사용해서 다양한 음계의 소리를 만들 수 있는데, 여기서는 'SelfAbot'클래스에서 작성한 tone 메소드를 사용해서 멜로디를 연주하는 실습을 소개합니다. 클래스에서 작성한 tone 메소드는 다음과 같습니다. 두 개의 메소드는 함수오버로딩을 사용하지 않는 버전과 사용하는 버전입니다.

```
void SelfAbot::tone(byte pin, unsigned int frequency, unsigned long duration) {
    ::tone(pin, frequency, duration);
}
void SelfAbot::tone(unsigned int frequency, unsigned long duration) {
    if (_selectedPin != INVALID_PIN) {
        ::tone(_selectedPin, frequency, duration);
    }
}
```

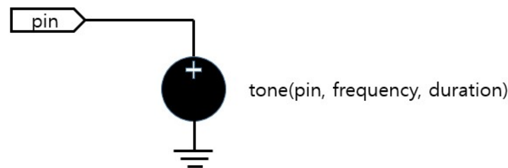


그림2.5 : 부저(피에조스피커) 아두이노 핀연결 개략도

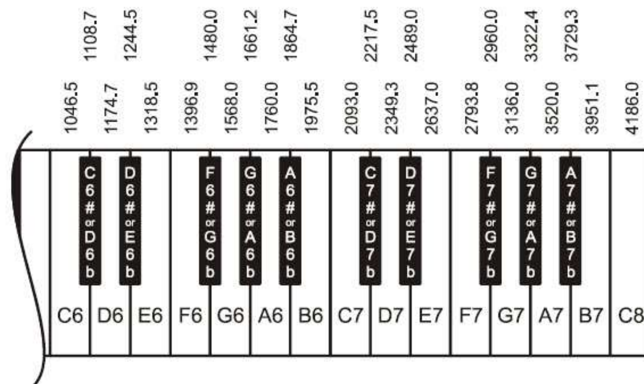


그림2.6 : 음계별 주파수 값 (그림출처: parallax.com)

Ex2.4_Buzzer_melody.ino 파일을 이용하세요.

```
#include "SelfAbot.h"
#include "pitches.h"

SelfAbot abot;

int melody[] = {
  NOTE_C4, NOTE_G3, NOTE_G3, NOTE_A3, NOTE_G3, 0, NOTE_B3, NOTE_C4
};
int noteDurations[] = {
  4, 8, 8, 4, 4, 4, 4, 4
};
```

```
void setup() {
//   abot.setup();
   for (int thisNote = 0; thisNote < 8; thisNote++) {
      int noteDuration = 1000 / noteDurations[thisNote];
      abot.tone(6, melody[thisNote], noteDuration);

      int pauseBetweenNotes = noteDuration * 1.30;
      delay(pauseBetweenNotes);

      noTone(6);
   }
}

void loop() {
}
```

위의 코드를 설명하면 다음과 같습니다.

(1) 라이브러리 포함:

```
#include "SelfAbot.h"
#include "pitches.h“
```

SelfAbot.h: 여기에는 'SelfAbot'클래스의 정의가 포함됩니다.

pitches.h: 이 파일에는 다양한 음을 연주하는데 사용되는 음표 주파수(예: NOTE_C4, NOTE_G3 등)에 대한 정의가 포함되어 있습니다.

(2) 객체 생성:

SelfAbot abot; 이 줄은 'SelfAbot' 클래스의 'abot' 객체(또는 인스턴스)를 생성합니다. 이 객체는 톤 재생과 같이 이 클래스에서 제공하는 메소드에 접근하는데 사용됩니다.

(3) 멜로디 및 지속 시간 정의:

int melody[] = { ... };

int noteDurations[] = { ... };

melody[]: 연주할 음표의 주파수를 포함하는 배열입니다.

noteDurations[]: 멜로디 각 음표 길이를 포함하는 배열입니다. 지속 시간은 전체 음표의 분수로 표시됩니다(예: 4분음표는 '4', 8분음표는 '8').

(4) setup 함수:

```
void setup() {
   for (int thisNote = 0; thisNote < 8; thisNote++) {
      …
   }
}
```

이 함수는 arduino가 시작될 때 한 번 호출됩니다. 여기에는 멜로디의 음표를 반복하는 'for' 루프가 포함되어 있습니다. 각 음표는 '6'번 핀에서 소리를 생성하는 'abot.tone()' 메소드를 사용하여 재생합니다.

(5) 멜로디 연주:

```
int noteDuration = 1000 / noteDurations[thisNote];
abot.tone(6, melody[thisNote], noteDuration);
```

noteDuration: 각 음표의 지속 시간을 밀리초 단위로 계산합니다.
abot.tone(6, melody[thisNote], noteDuration): 현재 음표를 재생합니다. tone 메소드는 부저가 연결된 핀 번호(6), 음표의 주파수(melody[thisNote]), 음표 재생 시간(noteDuration)의 세 가지 매개변수를 사용합니다.

(6) note 사이의 pause (pauseBetweenNotes 변수):

```
int pauseBetweenNotes = noteDuration * 1.30;
delay(pauseBetweenNotes);
noTone(6);
```

음과 음사이의 쉬는 시간이 멜로디를 표현하는데 매우 중요합니다. 소리음들 사이의 쉬는 시간을 표현하기 위한 변수입니다. 음표를 연주한 후 음표 지속 시간의 1.30배에 해당하는 짧은 일시 중지(pauseBetweenNotes)가 있으며, 그런 다음 noTone 기능이 핀 6의 tone 함수를 중지합니다.

(7) 반복 loop 함수:

```
void loop() { }
```

이 경우 'setup' 함수에서 멜로디가 한 번만 재생되므로 'loop' 함수는 비어 있습니다.

2.2.4 analogWrite() 함수로 LED 제어하기

아두이노에 LED 부품을 하드웨어로 연결한 다음, 아두이노에 업로드할 스케치 analogWrite() 함수로 LED 제어하는 코드를 아래에 소개합니다. 스케치는 arduino의 핀 9에 연결된 LED에 페이딩 효과를 만드는 것입니다.

아두이노 우노의 경우, 아날로그 출력(PWM)이 가능한 핀은 6개이고 핀번호는 핀 ~D3, ~D5, ~D6, ~D9, ~D10, ~D11 번입니다. 보드에서 물결모양으로 표시된 핀 번호들입니다.

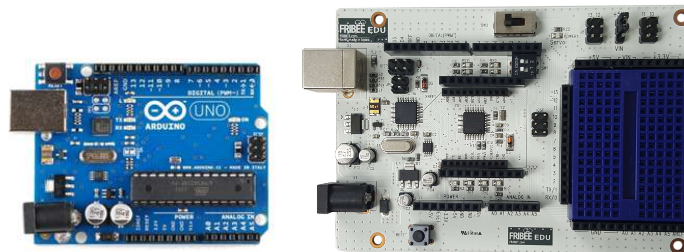


그림2.7 : 아두이노 우노 보드와 프라이비_에듀 우노 호환보드

아두이노 로봇 실습에 사용되는 ‘프라이비 보드(FRIBEE EDU)’는 ‘아두이노 우노(arduino UNO)’와 동일한 설정 방법을 사용합니다. 그리고 서보모터를 추가로 동작시킬 수 있도록 지원하는 3핀 포트와 회로기능이 포함되고, 동시에 엑스비/지그비 안테나, 블루투스 안테나, 와이파이(WiFi) 안테나를 장착할 소켓을 내장하고 있어서 쉽게 무선 확장이 가능합니다.

프라이비 보드의 무선기능 확장에 적합한 안테나 종류와 사용 방법은 제 9 장에서 다시 설명하겠습니다.

Ex2.5_LED_fade_IN_OUT.ino 파일을 업로드하세요.

```
#include "SelfAbot.h"

SelfAbot abot;
int pin = 9;

void setup() {
  //  abot.setup();
}
void loop() {
  for (int fadeValue = 0 ; fadeValue <= 255; fadeValue += 5) {
    abot.analogWrite(pin, fadeValue);
    delay(30);
  }
  for (int fadeValue = 255 ; fadeValue >= 0; fadeValue -= 5) {
    abot.analogWrite(pin, fadeValue);
    delay(30);
  }
}
```

(1) SelfAbot 라이브러리 포함:
#include "SelfAbot.h" 이 라인에는 로봇이나 전자 시스템의 다양한 구성 요소를 제어하기 위한 맞춤형 라이브러리인 'SelfAbot' 라이브러리를 포함합니다.

(2) SelfAbot 인스턴스 생성:
```
SelfAbot abot;
```

'SelfAbot'클래스의 abot이라는 인스턴스(객체)가 생성됩니다. 이 객체는 'SelfAbot'클래스가 제공하는 기능에 액세스하는데 사용됩니다.

(3) 핀 설정:
int pin = 9; 정수 변수 'pin'을 선언하고 LED(또는 기타 구성 요소)가 연결된 Arduino 보드의 핀 번호 '9' 값으로 초기화합니다.

(4) setup 함수:
```
void setup() {
  //abot.setup();
```

```
}
```

setup() 함수는 arduino의 전원을 켜거나 재설정할 때 한 번 실행됩니다. 이 코드에는 필요한 경우 설정을 초기화하는데 사용할 수 있는 주석 처리된 //abot.setup(); 줄이 포함되어 있습니다.

(5) loop 함수:

```
void loop() {
  // Code inside this function
}
```

loop() 함수는 setup()와 달리 반복 실행됩니다. 페이딩 효과를 구현하는 논리가 포함되어 있습니다. 아래에서 페이딩을 위한 논리를 설명합니다.

(6) 페이딩 논리:
코드는 페이딩 효과를 생성하기 위해 두 개의 for 루프를 사용합니다.
첫 번째 for 루프: 밝기가 점차 증가합니다.

```
for (int fadeValue = 0; fadeValue <= 255; fadeValue += 5) {
  abot.analogWrite(pin, fadeValue);
  delay(30);
}
```

이 루프는 fadeValue를 5의 크기로 0에서 255까지 증가시킵니다.
abot.analogWrite(pin, fadeValue): 이 줄은 fadeValue를 핀으로 보내 LED의 밝기를 아날로그 함수로 미세하게 표현합니다. 아날로그값 범위는 0(꺼짐)부터 255(최대 밝기)까지입니다.
delay(30): 각 밝기 변경 사이에 30밀리초 동안 루프를 일시 중지하여 눈에 띄는 페이딩 효과를 만듭니다.

두 번째 for 루프: 밝기가 점차 감소합니다.

```
for (int fadeValue = 255; fadeValue >= 0; fadeValue -= 5) {
  abot.analogWrite(pin, fadeValue);
  delay(30);
}
```

이 루프는 fadeValue를 255에서 다시 0으로 감소시킵니다.
루프의 나머지 부분은 첫 번째 루프와 유사하게 작동하지만 역으로 LED의 밝기를 단계적으로 어둡게 만듭니다.

2.3 단순화를 위한 함수 오버로딩 사용하기

2.3.1 digitalWrite() 함수로 LED 켜고 끄기입니다.

아래 코드는 abot 인스턴스 변수로 pin 값을 전달하는 방식의 아두이노 코딩 예제입니다. 이런 코드 사용을 함수 오버로딩이라고 부릅니다. 함수 오버로딩을 사용하면, 아두이노 핀 정보를 보다 명확하게 전달하는 효과가 있습니다.

앞서 2.2.1절에서 오버로딩을 사용하지 않고 digitalWrite() 함수로 LED 켜고 끄기하는 .ino 예제 Ex2.1_LED_ON_OFF.ino 를 실습했던 기억을 떠올려보세요. 그 다음 아래 예제를 다시 업로드하고 동일한 효과를 실습해보기 바랍니다.

Ex2.6_LED_ON_OFF_pin_overloading.ino 파일(overloaded 타입)을 업로드하세요.

overloaded	non-overloaded

#include "SelfAbot.h"
 	
SelfAbot abot;
int pin = 9;
 	
void setup() {
// abot.setup();
}
void loop() {
abot(pin).digitalWrite(HIGH);
delay(1000);
abot(pin).digitalWrite(LOW);
delay(1000);
}
	#include "SelfAbot.h"
	
	SelfAbot abot;
	int pin = 9;
	
	void setup() {
	// abot.setup();
	}
	void loop() {
	abot.digitalWrite(pin, HIGH);
	delay(1000);
	abot.digitalWrite(pin, LOW);
	delay(1000);
	}

지금부터 오버로드된 operator() 메소드와 함께 'SelfAbot'클래스를 사용하는 방법을 보여줍니다. 오버로드된 코드의 사용법을 이해하기 쉽게 설명하면 다음과 같습니다.

abot(pin).digitalWrite(HIGH); 이 줄이 함수 오버로드된 부분입니다. abot 인스턴스 객체는 'SelfAbot'클래스의 operator() 메소드를 호출하는 괄호 abot(pin)과 함께 사용됩니다. 이 메소드는 핀 번호를 승인하고 'SelfAbot'객체에 대한 참조를 반환하도록 설계되었습니다. 그런 다음 이 객체에서 digitalWrite(HIGH) 함수가 호출되어 지정된 핀(이 경우 핀 9)을 HIGH(일반적으로 5V)로 설정하고 해당 핀에 연결된 LED 또는 기타 장치를 켭니다.

abot(pin).digitalWrite(LOW); 이전 줄과 유사하지만 핀을 LOW(0V)로 설정하여 연결된 장치를 끕니다.

(참고) ---

함수 오버로드를 사용하는 이유는: 인스턴스 변수로 전달할 수 있는 값을 명시적으로 전달할 수 있습니다. 추가해서 설명하면, 함수의 이름을 동일하게 사용하면서 매개변수의 갯수가 다

른 유형 또는 매개변수의 데이터 타입이 다른 유형을 허용하는 여러 버전의 함수를 만들 수 있습니다. 이는 함수를 사용하는 방법에 더 많은 유연성을 제공하여 코드를 더 다양하게 만들고 더 읽기 쉽게 만듭니다.

2.3.2 버튼회로를 함수 오버로딩으로 실습하기

버튼회로는 아두이노와 같은 마이크로컨트롤러에 외부 스위치 신호를 전달하는 방법입니다. 특히 arduino와 같은 마이크로컨트롤러에서 버튼을 사용할 때 안정적인 버튼 상태(HIGH 또는 LOW)를 보장하기 위해 풀업 또는 풀다운 저항을 사용하는 것이 일반적입니다. 두 구성 모두 고유한 특성을 갖고 있으며, 버튼을 누르지 않을 때 디지털 입력 핀의 "불안정" 상태를 방지하기 위해 사용합니다.

풀업 방식과 풀다운 방식의 버튼회로를 아래 표로 비교해보세요. 풀업 및 풀다운 저항은 버튼회로의 디지털 입력 핀에 대해 안정적이고 예측 가능한 상태를 보장하여 플로팅 상태로 인해 발생하는 예측할 수 없는 판독값을 방지하는데 사용됩니다. 아래 회로도면에서 디지털핀 2번(D2)가 어떻게 연결되는지 기억하기 바랍니다.

표2.1 버튼회로를 구성하는 두 종류 전자회로

	풀업(pull-up) 회로	풀다운(pull-down) 회로
회로 도면	D9 5V D2 Photo Diode 10KΩ 220Ω Push Button GND	D9 5V Photo Diode Push Button 220Ω 10KΩ D2 GND
회로 특징	풀업 저항 구성에서 입력 핀은 일반적으로 HIGH 상태로 풀링됩니다 (Vcc에 연결). 버튼을 누르면 회로가 접지(GND)로 연결되어 입력 핀이 LOW 상태가 됩니다.	풀다운 저항 구성에서는 입력 핀이 일반적으로 LOW 상태(GND에 연결)로 풀링됩니다. 버튼을 누르면 회로가 양의 전압(Vcc)으로 연결되어 입력 핀이 HIGH 상태가 됩니다.

Ex2.7_Button_LED_ON_OFF_pin_overloading.ino 예제를 업로드해서 실습해보세요.

```
#include "SelfAbot.h“

SelfAbot abot;
const int buttonPin = 8;
const int ledPin =  9;
int buttonState = 0;

void setup() {
  //  abot.setup();
}
```

```
void loop() {
  buttonState = abot(buttonPin).digitalRead();
  if (buttonState == HIGH) {
    abot(ledPin).digitalWrite(HIGH);
  } else {
    abot(ledPin).digitalWrite(LOW);
  }
}
```

loop 함수내의 코드를 설명하면 다음과 같습니다.

buttonState = abot(buttonPin).digitalRead(); 버튼 핀의 상태 값을 읽는 digitalRead 함수를 실행합니다. 이 함수는 'SelfAbot'클래스의 메소드입니다. abot 인스턴스의 함수 오버로드를 사용해서 핀 정보 buttonPin 값을 전달합니다.

이어서 if 조건문을 실행할 때,

buttonState 값이 HIGH 상태이면 abot(ledPin).digitalWrite(HIGH);를 실행하고, LOW 상태이면 abot(ledPin).digitalWrite(LOW);를 실행합니다. 가령 abot(ledPin).digitalWrite(HIGH);는 함수 오버로드를 사용해서 ledPin 정보를 전달하고, LED를 켭니다.

2.3.3 아날로그 입력

또 다른 실습예제는 가변저항으로 입력전압을 변화시키면, 아날로그 입력값의 크기에 따라 아날로그 PWM출력을 LED 밝기 변화로 표현하는 코드입니다. 아날로그 입력신호를 처리하기 위한 'SelfAbot'클래스에서 정의한 메소드는 없습니다. 따라서 아두이노의 표준 코딩을 사용하면 됩니다.

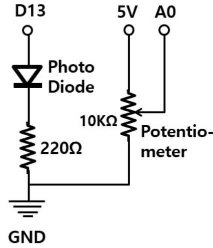


그림2.8 : 아날로그 입력회로

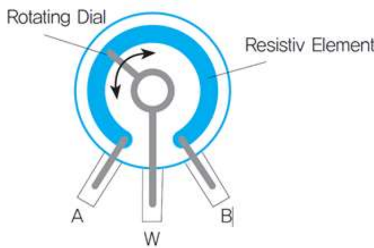


그림2.9 : 가변저항기

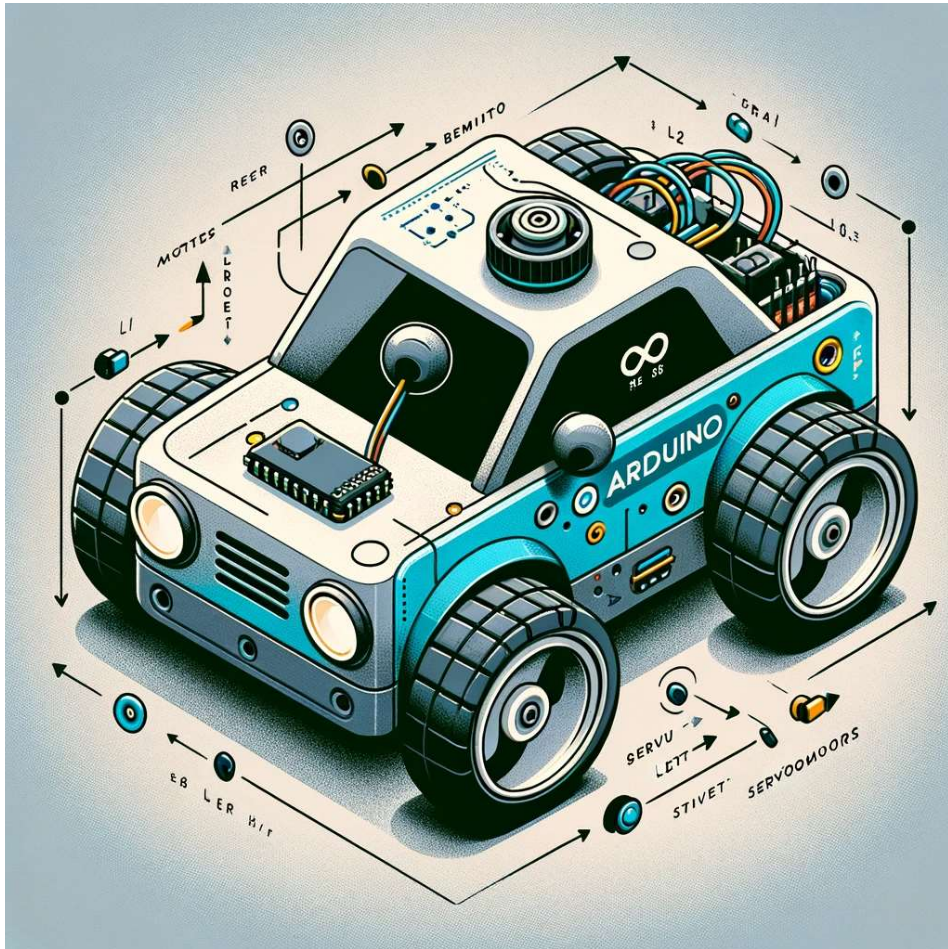
가변저항기의 A핀과 B핀은 전체저항을 연결하고, 가변저항기의 손잡이 놉을 돌리면 W핀과 연결된 저항 값 A-W 그리고 W-B 저항이 변화합니다.

Ex2.8_Potentiometer_analog_LED.ino 파일을 업로드해서 살펴보세요.

```
#include "SelfAbot.h"

SelfAbot abot;
int pin = 9;
int sensorPin = A0;
int sensorValue = 0;
void setup() {
  //  abot.setup();
}
void loop() {
    sensorValue = analogRead(sensorPin)/4;
    abot.analogWrite(pin, sensorValue);
}
```

제 3 장 로봇 초기화 및 기본동작 메소드

3.1 로봇 초기화를 위한 setup 메소드 구현하기
3.2 로봇 움직임 속성과 메소드
3.3 단순화를 위한 함수 오버로딩
3.4 표준형 서보모터 기본동작
3.5 연속 회전형 서보모터 기본동작



‘**neo-아두이노로봇카**’는 지금까지 C언어 코딩을 실습하는 용도로 사용됩니다. 많은 사용자들이 C언어 학습만으로는 부족함을 느끼고 있어서, C++언어를 사용해서 아두이노로봇을 활용할 수단을 찾으려고 합니다.
이 장에서는 ‘SelfAbot’클래스 매소드를 설명하고, 로봇동작에서 각각의 메소드를 구체적으로 어떻게 연결하여 사용하는지 살펴보겠습니다. 아래 코드에 사용된 ‘SelfAbot’클래스의 헤더 파일(.h)과 소스 파일(.cpp)은 제1장 4절에서 개략적으로 소개하였으니 참고 바랍니다.

3.1 로봇 초기화를 위한 setup 메소드 구현하기

아두이노 setup() 메소드는 로봇의 초기 설정을 담당합니다. 이 함수에서 서보 모터 핀이 유효한지 확인하고, 유효하다면 해당 핀에 서보 모터를 연결합니다. 로봇을 제대로 작동시키기 위해서는 초기화 과정이 필수적입니다. 초기화 과정에서 서보 모터, 핀 설정 등 로봇의 기본적인 설정을 수행합니다.

로봇을 초기화하기 위한 setup() 함수를 사용하는 방법은 다음과 같습니다:

(1) 서보 모터 핀 설정: 로봇의 서보 모터를 연결할 핀 번호를 지정합니다. 이는 보통 로봇 객체를 생성할 때 수행됩니다.
예를 들어, SelfAbot abot(LEFT_SERVO_PIN, RIGHT_SERVO_PIN); 같이 로봇 객체를 생성할 때 왼쪽 및 오른쪽 서보 모터에 대한 핀 번호를 전달할 수 있습니다. 이 방식은 매개변수가 있는 인스턴스 객체를 생성함으로써 서보모터의 핀번호를 전달하는 방법입니다.

(2) setup() 메소드 호출: setup() 메소드는 로봇의 서보 모터를 실제로 핀에 연결하고, 필요한 경우 추가적인 초기화 작업을 수행합니다. arduino의 메인 setup() 함수 내에서 로봇 객체의 setup() 메소드를 호출합니다. 그래서 생성자가 아닌 setup() 메소드에서 초기화설정이 필요한 코드를 추가함으로써, 로봇의 초기화 조건을 설정할 수 있습니다.

```
void setup() {
    abot.setup(); // 로봇의 setup 메소드 호출
}
```

(3) 필요한 추가 초기화: 로봇에 따라 추가적인 초기화 작업이 필요할 수 있습니다. 예를 들어, 센서를 초기화하거나 특정 핀을 입력 또는 출력으로 설정하는 등의 작업이 이에 해당됩니다.

setup() 함수는 arduino 프로그램이 시작될 때 한 번만 호출되며, 로봇의 기본적인 설정을 수행하는데 사용됩니다. 이 과정을 통해 로봇은 이후에 실행될 다양한 명령을 수행할 준비를 마치게 됩니다.

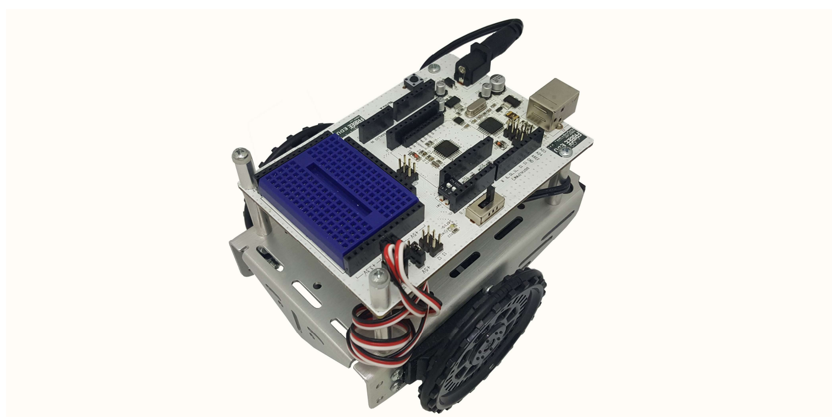


그림3.1 : neo-아두이노 로봇카

3.2 로봇 움직임 속성과 메소드

'SelfAbot'클래스에서 아두이노로봇의 서보모터를 움직이기 위한 멤버 변수 및 멤버 함수를 소개합니다. 각각의 역할을 살펴보겠습니다.

멤버 변수

Servo _servoAngle, _servoLeft, _servoRight;
arduino Servo 라이브러리의 'Servo' 클래스 객체입니다. 각 객체는 특정 서보 모터를 제어합니다. _servoAngle 변수는 표준형 서보모터를 움직이기 위한 멤버 변수이고, 나머지 두 개 _servoLeft, _servoRight는 로봇의 양쪽 바퀴에 연결된 연속 회전형 서보모터를 움직이기 위한 멤버 변수입니다.

byte _servoAnglePin, _servoLeftPin, _servoRightPin;
각 서보모터가 연결된 핀번호를 저장하는 변수입니다. 표준형 서보모터와 연속 회전형 서보모터에 연결된 핀번호를 저장합니다.

byte _selectedPin;
이 변수는 현재 제어되거나 상호 작용 중인 특정 서보 모터나 기타 구성 요소의 핀 번호를 저장하기 위한 것입니다.

오버로드된 함수에서의 사용:
오버로드된 멤버 함수에서 _selectedPin을 사용하면 _selectedPin에 저장된 핀 번호를 내부적으로 참조하여 서로 다른 구성 요소에 대한 작업에 동일한 함수 이름을 사용할 수 있습니다. 이렇게 하면 함수가 호출될 때마다 핀 번호를 명시적으로 전달할 필요가 없습니다.

단일 인스턴스 변수 작업:
_selectedPin을 원하는 구성요소의 핀 번호로 설정하면 해당 특정 구성요소에 대한 후속 작업을 수행할 수 있습니다. 이는 여러 구성 요소(예: 여러 서보 모터)를 관리하는 클래스에서 특히 유용합니다.

멤버 함수

아래 두 개의 멤버함수 servoAttachAngle(byte ServoAnglePin)과 servoAttachPins(byte ServoLeftPin, byte ServoRightPin) 는 오버로딩을 사용하지 않은 원래 함수형태를 그대로 사용하도록 작성하였습니다. 그리고 나머지 서보모터 동작과 관련된 멤머 함수(또는 메소드)들은 오버로딩을 사용해서 사용하는 코드로 작성합니다.

servoAttachAngle(byte ServoAnglePin);
각도 제어에 사용되는 서보를 지정된 핀(servoAnglePin)에 연결합니다. 표준형 서보모터의 핀 값을 명시적으로 전달하기 위한 함수입니다. 이 함수는 오버로드를 사용하지 않지만, 오버로드를 사용하려면 추가 코드가 필요합니다.

servoAttachPins(byte ServoLeftPin, byte ServoRightPin);

왼쪽 및 오른쪽 서보 모터를 해당 핀에 연결합니다. servoAttachPins() 함수에서 두 개의 매개변수 값으로 왼쪽 서보모터와 오른 쪽 서보모터의 핀 값을 명시적으로 전달합니다. 이 함수는 오버로드를 사용하지 않습니다. 오버로드를 사용하려면 별도의 메소드를 추가로 작성해야 합니다.

Ex3.1_nonoverloaded_servo_speed.ino 파일을 업로드해보세요.

```
#include "SelfAbot.h"

SelfAbot abot;

void setup() {
    abot.setup();
    abot.servoAttachPins(13, 12);

    abot.servoSpeedNonoverload(150, -150);
    delay(2000);
}

void loop() {

}
```

이 코드 예제에서는 함수 오버로드를 하지 않는 방식의 메소드 servoSpeedNonoverload()를 사용합니다.

먼저 라이브러리 코드로부터 동작하는 원리를 이해하기 쉽게 설명합니다.

(1) 메소드 : SelfAbot::setup()

```
void SelfAbot::setup() {
    if (_servoLeftPin != INVALID_PIN && _servoRightPin != INVALID_PIN) {
        _servoLeft.attach(_servoLeftPin);
        _servoRight.attach(_servoRightPin);
    }
}
```

로봇에 장착된 서보모터를 초기화하는 기능입니다.

왼쪽 및 오른쪽 서보의 핀(_servoLeftPin 및 _servoRightPin)이 유효한지 확인합니다 (잘못되었거나 사용되지 않은 핀을 나타내기 위해 이미 정의된 상수인 INVALID_PIN과 동일하지 않은지 조건 검사함).

핀이 유효한 경우 서보 객체(_servoLeft 및 _servoRight)를 해당 핀에 연결합니다. 여기에서 "연결"은 서보 모터가 연결된 핀을 서보 라이브러리에 전달하여 이를 제어할 수 있음을 의미합니다.

(2) 메소드 : SelfAbot::servoAttachPins(byte ServoLeftPin, byte ServoRightPin)

```
void SelfAbot::servoAttachPins(byte servoLeftPin, byte servoRightPin) {
```

```
        if (servoLeftPin != INVALID_PIN) {
            _servoLeftPin = servoLeftPin;
            _servoLeft.attach(_servoLeftPin);
        }
        if (servoRightPin != INVALID_PIN) {
            _servoRightPin = servoRightPin;
            _servoRight.attach(_servoRightPin);
        }
    }
```

이 코드는 오버로드 함수를 사용하지 않고, 왼쪽 서보의 핀 값과 오른 쪽 서보의 핀 값을 매개변수로 사용해서 두 개의 바퀴에 해당하는 서보모터 변수에 저장하고, 서보모터의 attach() 함수를 실행해서 서보모터의 준비상태를 만듭니다.

이 기능은 왼쪽과 오른쪽 서보 모터가 어떤 핀에 연결되어 있는지 지정하는데 사용됩니다.
각 핀을 개별적으로 검사합니다. 핀이 INVALID_PIN이 아닌 경우 새로운 핀 번호를 저장하고 왼쪽 또는 오른쪽 핀에 왼쪽 또는 오른쪽 서보를 연결합니다.

(3) 메소드 : SelfAbot::servoSpeedNonoverload(int leftSpeed, int rightSpeed)

```
    void SelfAbot::servoSpeedNonoverload(int leftSpeed, int rightSpeed) {
        _servoLeft.writeMicroseconds(1500 + leftSpeed);
        _servoRight.writeMicroseconds(1500 + rightSpeed);
    }
```

좌우 서보모터의 속도를 설정하는 기능입니다.

서보 모터를 정밀하게 제어하는 방법인 'writeMicroseconds' 방식을 사용합니다. '1500'은 서보의 중립점이며 속도 변수를 추가하면 서보가 시계 방향 또는 시계 반대 방향으로 회전할 수 있습니다. 서보모터가 동작하는 메소드 servoSpeedNonoverload()는 추가로 서보핀 정보를 사용하지 않습니다. 대신 서보핀정보는 servoAttachPins() 메소드를 통해 전달하고, 동일한 서보핀 정보를 servoSpeedNonoverload() 메소드가 사용함으로써, 서보모터 핀정보가 암시적으로 전달되어 적용됩니다.

위에서 소개한 아두이노 .ino 스케치 코드 라인별로 상세히 설명합니다.

(1) SelfAbot 라이브러리 포함하기

```
    #include "SelfAbot.h"
```

이 코드에는 SelfAbot 라이브러리가 포함되어 있어 SelfAbot 클래스와 해당 기능을 사용할 수 있다는 것을 선언합니다.

(2) 로봇 설정

```
    SelfAbot abot;
    void setup() {
        abot.setup();
        abot.servoAttachPins(13, 12);
```

```
        abot.servoSpeed(150, -150);
        delay(2000);
    }
```

SelfAbot abot; 코드라인은 abot이라는 이름의 SelfAbot 객체를 생성합니다.
'setup()' 함수는 arduino가 재설정되거나 전원이 켜질 때 한 번 실행됩니다.

abot.setup() 메소드는 서보 모터를 초기화합니다.
abot.servoAttachPins(13, 12)는 왼쪽과 오른쪽 서보의 핀을 각각 13과 12로 설정합니다.
abot.servoSpeed(150, 150)은 왼쪽 및 오른쪽 서보의 속도를 설정합니다.
delay(2000)는 2000밀리초(또는 2초) 동안 코드를 일시 중지합니다.
'loop()' 함수는 비어 있습니다. 지금은 arduino가 반복적인 코드실행 작업을 수행하지 않음을 의미합니다.

(참고) 'neo-아두이노로봇카' 서보모터를 동작시키기 위한 준비작업 -------------------------

(1) 아두이노 스케치를 IDE프로그램을 사용해서 보드에 업로드합니다.
(2) 아두이노보드에 서보모터 연결핀들을 모두 연결합니다. 흰색, 빨간색, 검은색을 구분해서 올바른 방향으로 연결해야 합니다. (흰색: 신호, 빨간색: 5V 전원, 검은색: 접지, 0V)
(3) USB 케이블로 전원을 공급하는 경우, 별도의 AA배터리는 준비되지 않아도 됩니다. 만약, USB 케이블을 제거하고 서보모터의 동작을 테스트하려는 경우 AA배터리 5개를 장착하고 전원잭을 보드에 연결해야 합니다.

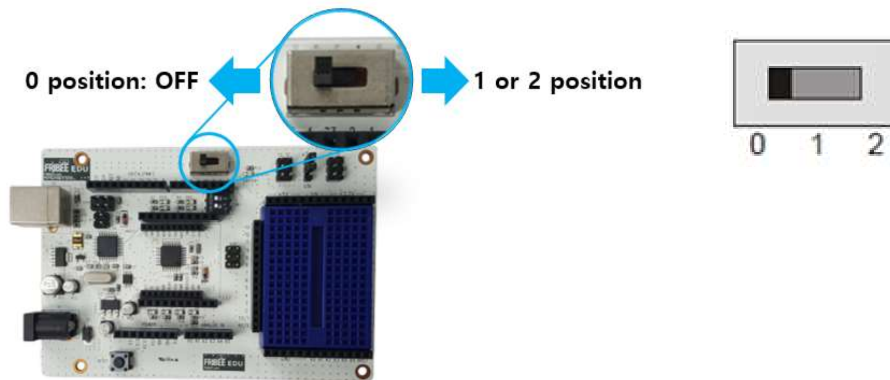


그림3.2 : 아두이노로봇 보드(프라이비-에듀 보드)의 3접점 스위치

(4) 아두이노보드 3접점 스위치를 아래 그림 화살표방향으로 전환해야 합니다. 3접점 스위치는 0,1,2로 구분되며 '0'위치는 AA배터리 전원을 차단하고, '1'위치는 아두이노프로그램만 동작하는 전원 공급되고, '2'위치에서 서보모터 동작까지 가능한 전원을 공급합니다.

(참고) 서보에 전원을 공급할 때 로봇이 매우 비정상 동작을 한다면, 제4장1절 '서보 중심맞추기'를 먼저 실행하고, 다시 예제 코드를 실행해야 할 수 있습니다.

3.3 단순화를 위한 함수 오버로딩

오버로딩된 메소드를 사용하여 로봇의 기능을 보다 간결하고 유연하게 사용할 수 있습니다. 아래 아두이노로봇 메소드들은 모두 함수 오버로딩을 사용하는 코드로 작성되었습니다.

servoAngle(int angle);

서보 각도를 설정하는 함수로써, 각도 제어에 사용되는 표준형 서보모터를 0 ~ 180도 범위에서 특정 각도로 설정합니다. 조건문의 조건이 참이면 _servoAngle.write(angle) 메소드를 실행합니다. write(angle) 메소드는 표준형 서보를 동작하기 위한 아두이노 함수입니다. 이때 _servoAngle를 나타내는 서보 인스턴스는 오버로딩 값으로 전달됩니다. 그래서 서보모터 메소드의 매개변수 값은 angle 값만 전달하면 됩니다.

```
void SelfAbot::servoAngle(int angle) {
    if (angle >= 0 && angle <= 180) {
        _servoAngle.write(angle);
    } else {
        Serial.println("Error: Invalid input value");
    }
}
```

servoSpeed(int speed);

현재 선택된 서보모터의 속도를 설정합니다. 이 메소드를 사용해서 서보를 특정 속도로 조정할 수 있습니다. servoSpeed(int speed) 메소드는 한 개의 연속회전형 서보모터를 동작시키기 위한 아두이노 servo 클래스의 writeMicroseconds() 함수를 사용합니다. 이 메소드는 인스턴스 변수를 사용해서 해당 핀 정보를 전달합니다. 그래서 해당되는 핀에 연결된 서보모터의 속도 값만 매개변수로 전달합니다.

한가지 고려할 사항은 사용중인 핀정보가 왼쪽 바퀴의 핀인지 또는 오른쪽 바퀴의 핀인지를 판단하고, 코드의 의도대로 서보모터가 동작하도록 하는 것입니다. 매개변수가 한 개인 경우의 오버로드된 인스턴스 변수는 _selectedPin을 사용해서 전달합니다.

그래서 _selectedPin값이 servoAttachPins(byte servoLeftPin, byte servoRightPin) 메소드 실행으로 사용자가 저장한 왼쪽 바퀴의 인스턴스 값인지 또는 오른쪽 바퀴의 인스턴스 값인지를 조건문으로 판단해서 사용자 코드의 의도와 동일하게 동작을 수행합니다. 아래 메소드를 사용해서 실제 코드에서 사용하는 예제는 3.5절에서 다시 설명합니다.

```
void SelfAbot::servoSpeed(int speed) {
    if (_selectedPin == _servoLeftPin) {
        _servoLeft.writeMicroseconds(1500 + speed);
    } else if (_selectedPin == _servoRightPin) {
        _servoRight.writeMicroseconds(1500 + speed);
    }
}
```

servoSpeed(int leftSpeed, int rightSpeed);

왼쪽과 오른쪽 서보 모터 속도를 독립적으로 설정합니다. 양쪽 바퀴동작을 위한 설정 함수를

통해 왼쪽 및 오른쪽 서보의 속도를 독립적으로 제어합니다.

servoSpeed(int leftSpeed, int rightSpeed) 메소드는 두 개의 서보모터를 동시에 동작시키기 위해 연속회전형 서보모터에 연결할 아두이노 servo 클래스의 writeMicroseconds() 함수를 사용합니다. 이 메소드는 leftSpeed 값과 rightSpeed 값을 각각 왼쪽 서보모터와 오른쪽 서보모터를 동작시키는 용도로 사용합니다.

왼쪽 바퀴와 오른쪽 바퀴를 동작하기 위한 핀 정보는 함수 오버로드 기능을 사용해서 인스턴스 값으로 전달합니다. 메소드를 실제 사용하는 코드예제는 3.5절에서 다시 설명합니다.

```
void SelfAbot::servoSpeed(int leftSpeed, int rightSpeed) {
    _servoLeft.writeMicroseconds(1500 + leftSpeed);
    _servoRight.writeMicroseconds(1500 + rightSpeed);
}
```

detachServos();

모든 서보 모터를 핀에서 분리하여 제어하지 못하게 하고 전력 소비도 줄이는 코드입니다. 메소드 detachServos()를 호출하면, 좌측과 우측 바퀴에 할당된 서보모터의 연결을 해제합니다. 함수 오버로드기능으로 특정 바퀴만 해제하려고 한다면, 추가 코드가 필요합니다.

```
void SelfAbot::detachServos() {
    _servoLeft.detach();
    _servoRight.detach();
}
```

이러한 메소드들을 사용해서 로봇의 서보 모터를 정밀하게 제어하고, 다양한 움직임과 동작을 수행합니다. 이 설계는 로봇의 요구 사항에 따라 각 모터를 독립적으로 제어하거나 다른 모터와 함께 제어할 수 있는 모듈식 접근 방식을 용이하게 합니다.

3.4 표준형 서보모터 기본동작

오버로딩된 메소드와 원래 메소드의 차이점을 살펴보고, 오버로딩된 메소드가 로봇동작에 어떻게 사용되는지를 살펴보기 바랍니다. 오버로딩된 메소드를 사용하면 로봇동작에 기여하는 서보모터의 핀 정보를 명시적으로 전달하는 효과가 있습니다.

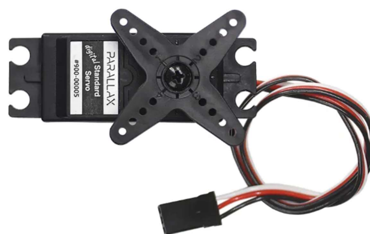


그림3.3 : 패럴렉스 표준형 서보모터

아래 아두이노 Ex3.2_standard_Servo_control.ino 파일을 업로드하고, 표준형 서보모터가 어떻게 동작하는지 살펴보세요. 오버로딩이 적용되지 않은 원시적인 메소드 형태와 함수 오버로딩이 적용된 메소드를 함께 소개하는 예제입니다.

서보모터를 처음 사용하려면, 서보모터를 전기적 준비상태로 설정해야 합니다. 아두이노에서는 이런 동작을 servo.attach() 라는 명령으로 실행합니다.
동일한 목적으로 아래 abot.servoAttachAngle(servoPin) 코드는 표준형 서보모터의 전기적 준비상태를 만들기 위한 코드입니다. 서보모터의 초기설정에서 핀번호를 매개변수로 명시적인 값을 전달하면, 사용자가 코드에서 작성한 서보모터 핀 정보를 인지하는 효과가 있습니다.

표준형 서보모터의 움직임을 만들기 위한 메소드 abot(servoPin).servoAngle(pos)는 abot 인스턴스 변수를 사용해서 핀정보 10를 전달하고, 실제로 움직임의 크기를 나타내는 angle 값만 매개변수를 사용해서 전달합니다.

오버로딩되지 않은 메소드:

abot.servoAttachAngle(servoPin);

오버로딩된 메소드:

abot(servoPin).servoAngle(pos);

Ex3.2_standard_Servo_control.ino 파일을 업로드하세요.

```
#include "SelfAbot.h"
const byte servoPin = 10;
SelfAbot abot;

void setup() {
    abot.setup();
    abot.servoAttachAngle(servoPin);
}

void loop() {
    for (int pos = 0; pos < 180; pos += 1) {
      abot(servoPin).servoAngle(pos);
      delay(15);
    }
    for (int pos = 180; pos >= 1; pos -= 1) {
      abot(servoPin).servoAngle(pos);
      delay(15);
    }
}
```

3.5 연속 회전형 서보모터 기본동작

로봇의 두 바퀴는 연속회전형 서보모터 움직임을 만들기 위한 메소드를 사용합니다. 한가지 메소드는 abot(13).servoSpeed(0)이고, 다른 메소드는 abot(13,12).servoSpeed(-150,150)입니다. 두 개의 메소드는 한 개의 매개변수를 사용하는 방식과 두 개의 매개변수를 사용하는 방식으로 서로 다른 유형입니다. 두 가지 종류의 메소드 모두 함수 오버로딩을 사용해서 메소드의 매개변수 이외에 서보모터의 핀 정보를 인스턴스 변수로 전달합니다.

그림3.4 : 패럴렉스 연속회전형 서보모터

표준형 서보모터를 사용하는 방식과 동일하게 연속회전형 서보모터 역시 초기설정이 필요합니다. 양쪽 바퀴에 연결된 연속회전형 서보모터의 설정을 위해 함수 오버로딩을 사용하지 않는 메소드 servoAttachPins(byte servoLeftPin, byte servoRightPin)를 먼저 사용합니다.
이 코드 실행으로 좌측 바퀴와 우측 바퀴에 대응하는 핀정보가 사용자에 의해 설정됩니다. 메소드에서 이 값들은 좌측바퀴에 해당하는 인스턴스 변수에 저장하고, 동시에 좌측바퀴의 서보모터 초기설정 servo.attach() 함수를 실행합니다. 우측 바퀴도 동일하게 실행합니다. 아래 예제코드를 살펴봅시다.

Ex3.3_continuous_Servo_speed.ino 파일을 업로드하고 테스트하세요.

```
#include "SelfAbot.h"

const byte servoLeftPin = 255;
const byte servoRightPin = 255;
SelfAbot abot(servoLeftPin, servoRightPin);

void setup() {
      abot.setup();
      abot.servoAttachPins(13, 12);
}
void loop() {
      abot(13).servoSpeed(0);
      abot(12).servoSpeed(0);
      delay(1000);

      abot(13).servoSpeed(150);
      abot(12).servoSpeed(-150);
      delay(1000);

      abot(13, 12).servoSpeed(-150, 150);
      delay(1000);
}
```

위 예제 코드의 상세한 설명입니다. 'SelfAbot'클래스를 사용하여 로봇을 제어하는 것을 목적으로 하며, 특히 서보 모터 동작에 초점을 맞추고 있습니다.

(1) 'SelfAbot'라이브러리 포함:

```
#include "SelfAbot.h“
```

여기서 SelfAbot 라이브러리를 포함시켜 'SelfAbot'클래스와 그 기능에 접근합니다.

(2) 서보 핀을 위한 상수 설정:

```
const byte servoLeftPin = 255;
const byte servoRightPin = 255;
```

여기서 servoLeftPin과 servoRightPin 두 개의 상수가 255라는 값으로 정의됩니다. 그리고 255는 아두이노의 특정한 핀 값이 아니면서, 초기 값이 지정되지 않는 상태를 방지하기 위해 임시 값으로 사용합니다.

(3) 'SelfAbot'의 인스턴스 생성:

```
SelfAbot abot(servoLeftPin, servoRightPin);
```

이 코드는 'SelfAbot'클래스의 객체 abot을 생성하고, 왼쪽 및 오른쪽 서보 핀으로 초기화합니다. 초기 값으로 255를 사용했기 때문에, 아래처럼 실제 핀 번호는 코드에서 나중에 설정합니다.

(4) setup() 함수:

```
void setup() {
    abot.setup();
    abot.servoAttachPins(13, 12);
}
```

setup()은 프로그램이 시작될 때 아두이노가 한 번만 호출하는 함수입니다. 설정을 초기화하는데 사용됩니다. abot.setup()은 abot 객체의 setup 메소드를 호출하는데, 이는 서보를 초기화하는 역할을 합니다.
abot.servoAttachPins(13, 12)은 아두이노 보드의 디지털 핀 13와 12에 서보를 연결합니다.

※ 예제 코드의 abot 인스턴스가 기본생성자를 사용하는지 또는 두 개의 매개변수 생성자를 사용하는지 궁금하지 않나요? 조금 더 상세히 살펴봅시다.

abot 인스턴스는 SelfAbot abot(servoLeftPin, servoRightPin); 코드를 사용해서 'SelfAbot' 클래스의 두 매개변수 생성자로 생성합니다. 이 생성자는 'servoLeftPin'과 'servoRightPin' 이라는 두 가지 인수를 사용합니다. 두 매개변수 모두에 값을 전달하므로(두 매개변수가 255로 초기화되었음에도 불구하고) 2바이트 값을 매개변수로 예상하는 생성자를 호출합니다.

이후 abot.servoAttachPins(13, 12); 코드는 이미 생성된 abot 인스턴스에서 SelfAbot 클래스의 메소드를 호출합니다. 새 인스턴스를 생성하는 것이 아니라 arduino 보드의 디지털 핀 13와 12에 서보를 연결하기 위하여 이미 생성된 인스턴스 'abot'을 사용합니다.

그 다음 abot(13).servoSpeed(0); 코드 역시 'SelfAbot'클래스의 새 인스턴스를 생성하지 않고, 이미 생성된 abot 인스턴스를 사용하고 해당 상태를 수정합니다.
abot(13) 구문은 'SelfAbot'클래스의 operator() 함수를 활용합니다. 이 연산자는 'SelfAbot' 클래스에 오버로드되어 _selectedPin 멤버 변수를 인수로 전달된 값으로 설정합니다. 이 경우 _selectedPin을 13으로 설정합니다.

이 연산자 호출 후에 servoSpeed(0)는 동일한 인스턴스 abot에서 호출되며, 이는 방금 13으로 설정된 _selectedPin 값을 기반으로 작동합니다. 이 패턴을 사용하면 읽기 쉽고 표현력이 풍부한 방식으로 동일한 객체에 대한 메소드 호출을 연결할 수 있는 유연한 인터페이스가 가능합니다.

명확하게 말하면 abot(13).servoSpeed(0);는 다음 단계를 수행합니다.
abot 인스턴스에서 operator()(13)를 호출하고 _selectedPin을 13으로 설정합니다.
그런 다음 abot 인스턴스에서 servoSpeed(0) 메소드를 호출합니다.
이 메소드는 _selectedPin 값(현재는 13)을 기반으로 작동합니다.

(5) loop() 함수:

```
void loop() {
    abot(13).servoSpeed(0);
    abot(12).servoSpeed(0);
    delay(1000);
    abot(13).servoSpeed(150);
    abot(12).servoSpeed(-150);
    delay(1000);
    abot(13, 12).servoSpeed(-150, 150);
    delay(1000);
}
```

loop() 함수내에서 실행되는 코드들은 다음과 같이 동작합니다.
abot(13).servoSpeed(0); 및 abot(12).servoSpeed(0);은 13번 및 12번 핀에 연결된 서보를 각각 정지시킵니다. 서보모터의 속도값이 0 이면 정지상태를 표현합니다.

delay(1000);은 프로그램을 1000 밀리초(1초) 동안 일시 중지합니다.

그 후 서보는 다른 속도 예를 들면 150 또는 -150 값으로 설정됩니다. 해당 숫자만큼 서보모터의 속도가 시계방향 또는 반시계방향으로 회전합니다.

abot(13, 12).servoSpeed(-150, 150);은 동시에 두 개의 서보에 다른 속도를 설정하여 로봇

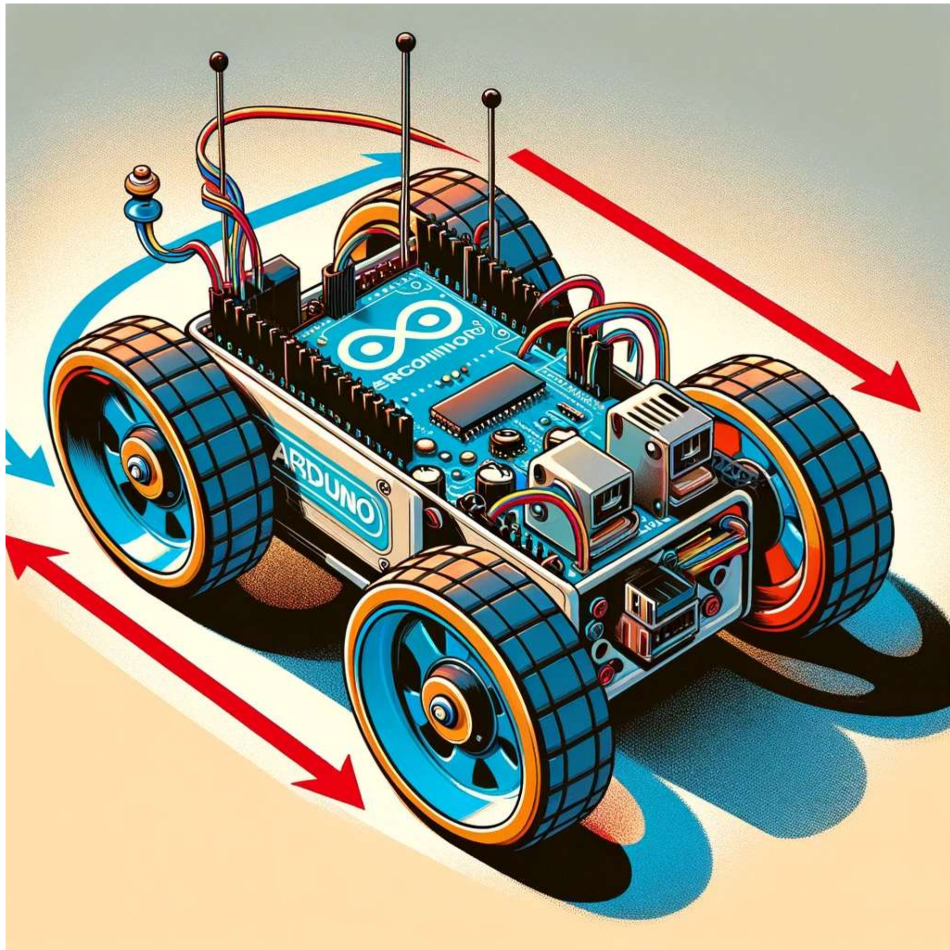
이 특정 패턴으로 회전하거나 움직이도록 할 수 있습니다. 첫 번째 매개변수 값은 왼쪽 서보모터에 연결된 바퀴 속도를 제어하고, 두 번째 매개변수 값은 오른쪽 서보모터에 연결된 바퀴의 속도를 제어합니다. 이 코드는 기본적으로 두 개의 매개변수가 동시에 제공되어 두 개의 서보 모터를 제어하고 정지, 시작 및 방향 전환에 필요한 두 개의 바퀴동작을 제어할 수 있습니다.

또 다른 1초의 지연 루프가 반복됩니다.

'SelfAbot'클래스는 함수 오버로딩과 쉽게 이해할 수 있는 명령어를 사용하여 로봇을 제어하는 조직적인 방법을 제공합니다.

제 4 장 아두이노로봇 움직임 제어하기

4.1 서보모터 중심맞추기
4.2 직선주행과 로봇 보정
4.3 좌회전 / 우회전 그리고 연속동작
4.4 점진적 가속과 감속
4.5 추가 메소드 : maneuver()



지금까지 'SelfAbot'클래스 내부 메소드를 설명하는데 초점을 맞추었다면, 제 4 장부터는 아두이노로봇의 'SelfAbot' 클래스를 어떻게 사용해서 동작시키는지에 초점을 맞출 것입니다.

C언어로 아두이노로봇을 다루어본 사용자라면, 더 친근하게 실습들을 비교할 수 있습니다. 본 교재를 처음 접하는 사용자라도 쉽게 이해할 수 있도록 최대한 쉽고 간결하게 설명합니다. 만약 설명이 부족하다고 느끼는 사용자라면 C언어로 작성된 '아두이노로봇 실습 안내서'를 읽어보거나 공개 유튜브 시청을 추천합니다.

4.1 서보모터 중심맞추기

연속회전형 서보모터를 프라이비 보드에 연결하는 방법은 아래 그림4.1을 참고해서 연결하세요. 서보핀을 프라이비 보드와 연결할 때, ***검정색 와이어가 브래드보드 방향으로 향하도록 꽂으면 됩니다. 13번, 12번 핀을 사용하거나, 11번 10번 핀을 사용할 수 있습니다.***

아두이노 보드에 처음 전원이 공급되면 부트로더(boot loader)가 동작하기 시작할 때, LED핀 13번에서 깜박이는 신호가 감지됩니다.(아두이노 보드의 기본 설정동작) 그래서 로봇에 reset 신호를 사용할 때마다, 로봇이 잠깐씩 움직이는 경향이 있습니다. 이런 문제를 회피하려면, 서보모터 핀으로 11번 10번을 사용하면 이런 문제가 해소됩니다.

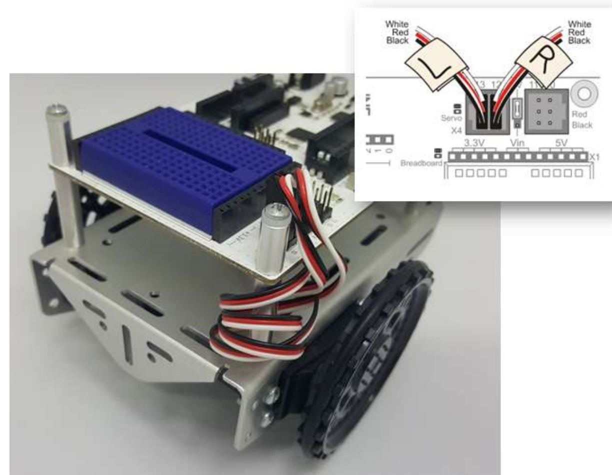


그림4.1 : 아두이노로봇 서보 핀 연결하는 방법

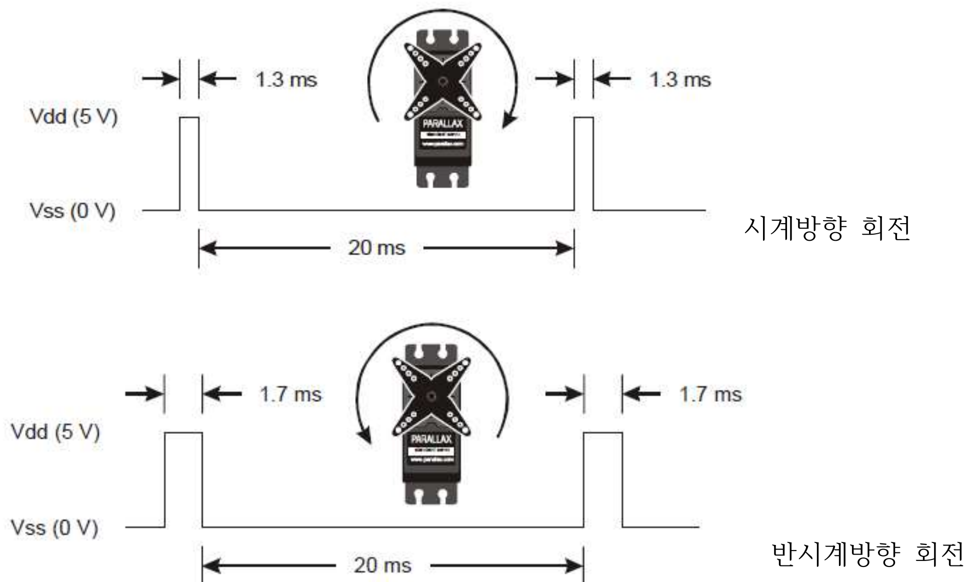


그림4.2 : parallax 연속회전형 서보모터의 PWM 변조제어 방법 (출처: parallax.com)

(참고) --

로봇주행시 서보모터 기본동작의 이해

전진동작을 위한 '아두이노로봇'의 서보 바퀴회전을 시도하려면, 진행방향에서 볼 때 우측바퀴는 시계방향으로 회전해야 하고, 반대쪽 좌측바퀴는 반시계방향으로 회전해야 합니다.

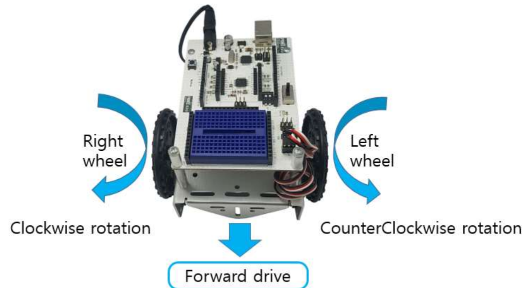


그림4.3 : 전진동작 서보모터 회전방향

서보모터 동작은 고정 주기내에서 펄스폭 변조(PWM) 제어를 사용합니다. 서보모터의 시계방향 회전 또는 반시계 방향 회전은 PWM 제어에서 펄스폭의 중앙 값인 '0'에서 변화합니다. 즉 서보 모터의 회전 방향과 크기는 PWM 신호의 활성 펄스 폭을 변경하여 제어합니다. 표준 범위 내의 펄스가 더 짧거나 길면 모터가 중립 위치에서 시계 방향 또는 시계 반대 방향으로 회전하는 양을 정밀하게 제어할 수 있습니다.

서보모터가 공장에서 출하되어 사용될 때, 특히 정밀 제어가 필요한 로봇공학에서는 서보의 움직임을 위해 물리적으로 보정하는 것이 필요합니다. 서보모터는 이런 목적의 물리적 보정을 위해 가변저항을 내장합니다. 사용자는 스크루를 조정하는 방법으로 물리적 중간점을 찾아서 정렬할 수 있습니다.

그림4.4 : 서보모터 중심맞추기를 위해 스크류를 조정하는 방법 (출처: parallax.com)

'**서보모터 중심맞추기**'목적으로 예제 코드가 소개됩니다. parallax사의 연속회전형 서보모터 값이 '1500'일 때 중간점에 해당하고, 'SelfAbot' 클래스의 서보 메소드는 속도값이 '0'일 때 중간점입니다. 아래 코드를 아두이노로봇에 업로드하면 바퀴는 정지상태이어야 합니다. 그런데 미세하게 움직이고 있다면, 스크류를 움직여서 미세한 움직임이 없도록 조정해야 합니다. 이런 작업을 '**<u>서보모터 중심 맞추기</u>**'라고 부릅니다.

아래 코드를 업로드해서 바퀴가 움직이지 않도록 서보모터의 중심맞추기 작업을 실시하세요. 실습조건의 코드에서는 서보모터의 핀이 13번과 12번에 연결되어 있습니다.

Ex4.1_servo_centering.ino 스케치를 업로드하세요.

```
#include "SelfAbot.h"
#define INVALID_PIN 255;

const byte servoLeftPin = INVALID_PIN;
const byte servoRightPin = INVALID_PIN;

SelfAbot abot(servoLeftPin, servoRightPin);

void setup() {
	abot.setup();
	abot.servoAttachPins(13, 12);
}

void loop() {
	abot(13).servoSpeed(0);
	abot(12).servoSpeed(0);
	delay(1000);

	abot(13, 12).servoSpeed(0, 0);
	delay(1000);
}
```

4.2 직선주행과 로봇 보정

패럴렉스 연속회전형 서보모터의 속도를 변화시키는 그래프를 먼저 소개합니다. 속도값의 범위는 1500을 기준으로 -200부터 +200까지 변화시킬 수 있습니다. 아래 표가 나타내는 것처럼 실효적인 속도 변화는 -100부터 +100까지에서 거의 변화가 이루어집니다. 그리고 -40 또는 +40 속도값에서 거의 중간속도를 나타냅니다.

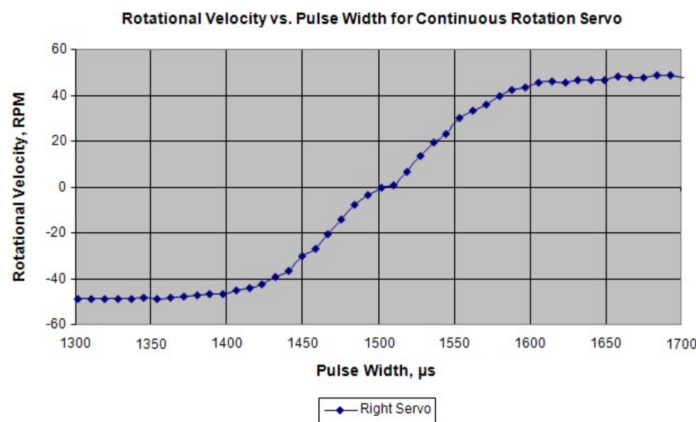


그림4.5 : 패럴렉스 연속회전형 서보모터의 속도에 대한 펄스폭 변화량 (출처: parallax.com)

로봇 공학에서 두 개의 서보모터를 사용해서 로봇이 직선주행할 때, 바퀴에 연결된 서보모터의 제조, 마모 또는 기계적 특성의 차이로 인하여 서보모터 메소드에 동일한 속도 값을 제공해도 실제 성능에서 불일치가 발생할 수 있습니다. 이런 문제를 해결하고 보다 정밀한 직선주행을 가능하게 하려면 각 모터의 속도를 보정하는 속도 조정방법을 구현할 수 있습니다.

기계적으로 동일하지 않은 서보모터를 고려해서, 두 개의 서보모터 회전속도를 미세하게 조정하는 코드 설계 예제를 소개합니다. 두 개 모터의 성능 차이를 확인하는 방법은 분당회전수를 측정하거나, 일정한 이동거리를 주행할 때 직선을 벗어나는 정도를 파악하는 것입니다. 이런 수정 작업은 배터리 소모 또는 부하 변동 등으로 모터 성능이 지속적으로 변할 수 있어서, 모니터링과 조정이 반복되어야 할 수 있습니다. 보정작업 동안은 USB 배터리 전원을 제거하고, 온전히 AA배터리 전원만으로 실시해야 합니다. 그래야만 로봇의 서보모터 보정작업이 정확하게 반영됩니다.

모터의 움직임을 실시간으로 정확하게 측정할 수 있는 수단은 엔코더 또는 유사한 센서를 사용하는 것입니다. 만약 엔코더가 장착되어 있지 않더라도, 표면질감과 같은 외부 주행환경 일관성이 유지되는 조건에서 교정을 수행하면 직선주행이 더 정확하게 개선될 수 있습니다.

abot 인스턴스를 사용해서, 정규화 및 편차 개념을 반영한 보정 코드를 아래에 소개합니다.

```
float deviationFactor = 0.0;
void driveStraight(int leftSpeed, int rightSpeed) {
        // Apply normalization to calculate correction coefficients
        float leftCoefficient = 1.0 + deviationFactor;  // Increase for left servo
        float rightCoefficient = 1.0 - deviationFactor; // Decrease for right servo

        int adjustedLeftSpeed = leftSpeed * leftCoefficient;
        int adjustedRightSpeed = rightSpeed * rightCoefficient;

        abot.servoSpeed(adjustedLeftSpeed, adjustedRightSpeed);
}
```

deviationFactor는 -1.0에서 1.0 사이의 단일 부동 소수점 숫자입니다. 값이 0이면 편차가 없음(두 서보가 동일한 속도로 실행됨)을 의미하고, 양수 값은 왼쪽 서보의 속도를 높이고 오른쪽 서보의 속도를 감소시키며, 음수 값은 그 반대를 의미합니다.

driveStraight 메소드에서는 정규화된 속도조정을 실시하기 위해 편차 계수가 각 서보의 속도를 수정합니다. 왼쪽 서보의 속도는 양의 편차로 증가하고 오른쪽 서보의 속도는 감소하며 그 반대도 마찬가지입니다.

지금 소개한 코드들을 직접 'SelfAbot'라이브러리에 추가거나, 아두이노 .ino 파일에 추가해서 테스트해보기 바랍니다. 로봇의 바퀴동작을 정확하게 구현하는 보정코드를 클래스 상속에서 다루는 방법도 나중에 다시 다루겠습니다.

먼저 아래 그림을 참고해서 직진주행 실습을 따라하면 됩니다. 실습 개념은 로봇의 한쪽 바퀴를 직선자에 맞춰 정렬하고, 두 바퀴 방향이 직선자와 수직방향이 되도록 배치합니다.
보정을 위해 아두이노로봇이 일정한 시간(아래 코드에서는 delay(2000))동안 전진 주행한 이후, 직선자에 정렬했던 바퀴가 이탈한 거리(deviationdistance)를 살펴보고, 이 거리가 최소가 되도록 보정 실습을 반복하면 됩니다. 반복 조건은 deviationfactor 값을 -0.15에서 +0.15사

이로 바꿔가면서 조절해보세요. float 변수이므로 소수점 값을 사용해야 합니다.

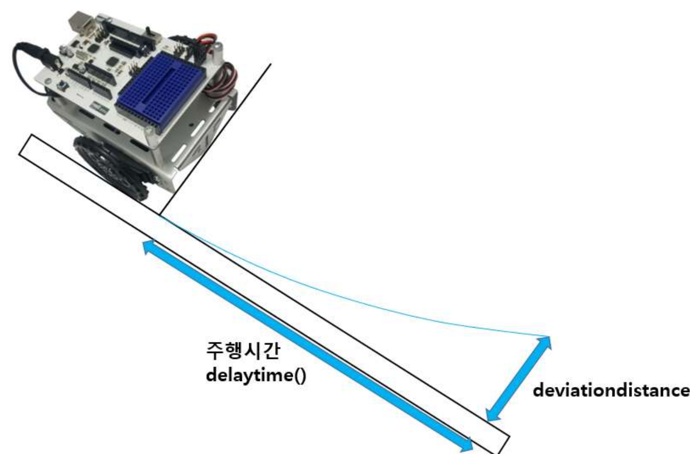


그림4.6 : 아두이노로봇 직진주행을 위한 보정 실습 개념도

Ex4_2_abot_driveStraight.ino 스케치를 업로드하세요.

```
#include "SelfAbot.h"
#define INVALID_PIN 255;

const byte servoLeftPin = INVALID_PIN;
const byte servoRightPin = INVALID_PIN;
float deviationFactor = 0.0; // min. val -0.15 max. val +0.15

SelfAbot abot(servoLeftPin, servoRightPin);

void setup() {
    abot.setup();
    abot.servoAttachPins(11, 10);
    driveStraight(40, -40);
    delay(8000);
    abot.servoSpeed(0, 0);
    delay(1000);
}

void loop() {

}

void driveStraight(int leftSpeed, int rightSpeed) {
    // Apply normalization to calculate correction coefficients
    float leftCoefficient = 1.0 + deviationFactor;  // Increase for left servo
    float rightCoefficient = 1.0 - deviationFactor; // Decrease for right servo

    int adjustedLeftSpeed = leftSpeed * leftCoefficient;
    int adjustedRightSpeed = rightSpeed * rightCoefficient;

    abot.servoSpeed(adjustedLeftSpeed, adjustedRightSpeed);
}
```

이 코드는 aduino 기반의 로봇(여기서는 ‘SelfAbot’클래스를 사용)을 제어하기 위한 예제입니다. 코드의 목적은 두 개의 서보모터 물리적 동작특성이 동일하지 않더라도, 미세한 보정을

적용해서 로봇에 연결된 두 개의 서보 모터가 연결된 로봇을 직진하게 하는 것입니다. 로봇이 정확하게 직진할 수 있도록 사용자가 'deviationFactor'값을 반복 교정해야 합니다.

두 개의 서보모터로 로봇이 전진할 때, 속도값이 100 이상이면 양쪽 바퀴속도 편차가 크게 느껴지지 않을 수 있습니다. 그렇지만, 최저속도 20 일 때 로봇은 전방 직선주행 라인을 기준으로 좌측 또는 우측으로 활처럼 많이 치우치면서 곡선 주행하게 됩니다. 이런 주행의 의미는 속도값이 20 ~ 100 사이의 다양한 속도값에 따라 보정계수가 다르게 적용되어야 한다는 의미입니다.

여기서는 보정의 편의성을 위하여, 주행속도가 거의 중간값에 해당하는 '40'을 사용해서 직선주행 근사값을 찾고 로봇의 다른 속도값에도 직선주행 오차가 최소화되는 조건으로 적용할 수 있습니다. 보정 값 deviationFactor를 찾는 원리는 로봇이 좌측으로 치우치면 0.0 ~ 0.15 범위의 양수 값을 적용하고, 우측으로 치우치면서 주행하면 -0.15 ~ 0.0 범위의 음수 값을 적용하면서 최적의 조건값을 찾기 바랍니다.

제시된 코드의 각 부분을 자세히 살펴보겠습니다.

(1) 라이브러리 및 변수, 객체 초기화:

```
#include "SelfAbot.h"
#define INVALID_PIN 255

const byte servoLeftPin = INVALID_PIN;
const byte servoRightPin = INVALID_PIN;
float deviationFactor = 0.0; // range -0.15 to +0.15
SelfAbot abot(servoLeftPin, servoRightPin);
```

SelfAbot.h 라이브러리를 포함하여 SelfAbot 클래스에 정의된 기능을 사용할 수 있습니다.
servoLeftPin과 servoRightPin는 서보 모터가 연결될 핀 번호를 나타냅니다. 여기서는 255로 초기화되어 있지만, 실제 핀 번호로 변경해야 할 수도 있습니다. deviationFactor는 서보 모터의 속도 교정 계수로 사용되며, 여기서는 0으로 설정되어 있습니다.

(2) setup() 함수에서 abot 객체를 생성하여 로봇을 제어합니다.

```
void setup() {
    abot.setup();
    abot.servoAttachPins(11, 10);
    driveStraight(40, -40);
    delay(2000);
    abot.servoSpeed(0, 0);
    delay(1000);
}
```

abot.setup(); 호출로 로봇을 초기화합니다.
abot.servoAttachPins(11, 10);로 왼쪽과 오른쪽 서보 모터가 연결된 핀을 설정합니다.
driveStraight(40, -40);로 로봇을 직진시키기 위한 함수를 호출합니다. 여기서 40, -40은 왼쪽과 오른쪽 서보의 속도를 나타냅니다.

delay(2000);로 2초간 대기한 후, abot.servoSpeed(0, 0);로 서보 모터를 정지시킵니다.

(3) loop() 함수:

```
void loop() {
    // Currently empty.
}
```

loop() 함수는 현재 비어 있으며, 반복해서 실행하고 싶은 코드를 여기에 추가할 수 있습니다.

(4) driveStraight() 함수:

```
void driveStraight(int leftSpeed, int rightSpeed) {
    // ... (코드 생략) ...
    abot.servoSpeed(adjustedLeftSpeed, adjustedRightSpeed);
}
```

이 함수는 로봇을 직진하게 만듭니다. leftCoefficient와 rightCoefficient는 각각 왼쪽과 오른쪽 서보 모터의 속도를 교정하기 위해 사용됩니다.

최종 교정된 servoSpeed(adjustedLeftSpeed, adjustedRightSpeed)로 abot.servoSpeed()를 호출하여 최대한 직선주행이 가능한 driveStraight()의 서보 모터 속도를 설정합니다. 이 코드는 로봇의 두 서보 모터를 제어하여 직진 동작을 수행합니다.

참고로 driveStraight()를 테스트하는 예제 Ex4_2_abot_driveStraight.ino 스케치를 사용해서 실제 테스해본 결과 로봇이 우측으로 치우치는 주행을 하였습니다. 'deviationFactor' 값을 -0.06 으로 보정했을 때 가장 직선에 근접해서 주행하는 최전 값을 찾았습니다. 여러분도 직접 다양한 주행 속도 값에서 테스트해보기를 추천합니다.

4.3 좌회전 / 우회전 그리고 연속동작

바퀴 두 개로 로봇의 동작을 만드는 경우, 여러 가지 형태의 회전에 대한 개념을 그림4.7 그림으로 묘사합니다.

제자리회전 : 먼저 양쪽 바퀴를 진행방향과 다르게 서로 반대방향으로 동일한 속도로 동작시키면, 로봇을 제자리에서 맴돌게 할 수 있습니다. 이런 동작으로 어떤 실습을 할 수 있는지에 대해서는 이후 내용에서 다루겠습니다.

피봇 회전 : 로봇의 한쪽 바퀴를 정지상태로 두고, 나머지 한쪽 바퀴를 회전시키면 로봇은 정지된 바퀴를 기준으로 회전합니다. 이런 동작을 피봇 회전(pivot rotation)이라고 부릅니다.

곡선 주행 : 로봇의 전진동작과 동일한 방향으로 두 개의 서보모터를 동작시키지만, 두 개의 서보모터 동작속도를 다르게 하면 로봇은 천천히 회전하는 바퀴의 방향으로 크게 동심원을 그리면서 회전합니다. 이런 동작을 곡선 주행(curved drive)이라고 부릅니다.

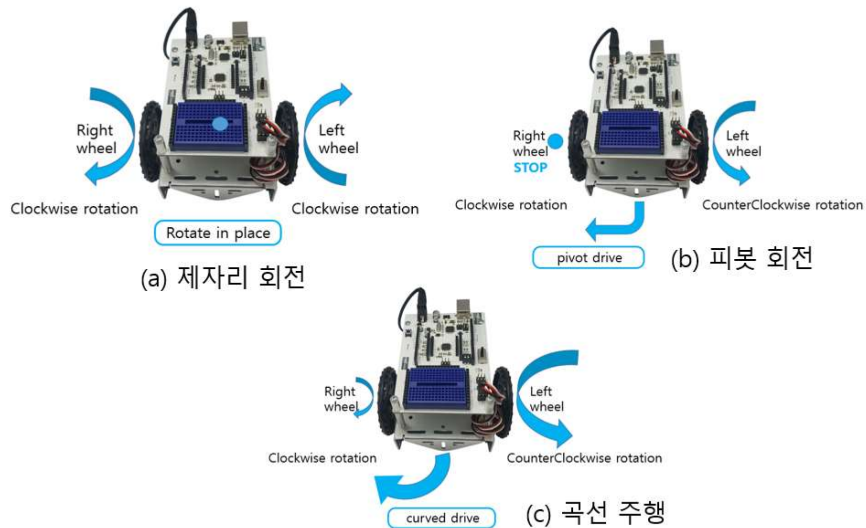


그림4.7 :로봇 회전형태 (a) 제자리회전 방법 (b) 피봇 회전방법 (c) 곡선 주행하는 방법

이제 아두이노로봇을 정지, 전진 그리고 피봇 좌회전과 피봇 우회전하고 후진하는 코드를 소개합니다. 아래 로봇동작 코드는 매개변수를 한 개 사용하는 경우와 두 개의 매개변수를 사용하는 경우 모두 스케치에서 표현했습니다. 함수 오버로드를 사용해서 어떤 경우이든 로봇의 바퀴동작을 제어할 수 있습니다.

Ex4.3_abot_basic_drive.ino

```
#include "SelfAbot.h"
#define INVALID_PIN 255;
const byte servoLeftPin = INVALID_PIN;
const byte servoRightPin = INVALID_PIN;

SelfAbot abot(servoLeftPin, servoRightPin);

void setup() {
	abot.setup();
	abot.servoAttachPins(11, 10);
}

void loop() {
	abot(11).servoSpeed(0);
	abot(10).servoSpeed(0);
	delay(1000);

	abot(11).servoSpeed(150);
	abot(10).servoSpeed(-150);
	delay(1000);

	abot(11, 10).servoSpeed(0, -150);
	delay(1000);
	abot(11, 10).servoSpeed(0, 150);
	delay(1000);
	abot(11, 10).servoSpeed(-150, 150);
	delay(1000);
}
```

위 실습 예제는 정지상태를 1초간 유지하고, 전진 동작 그리고 왼쪽 바퀴는 정지된 상태에서 우측 바퀴만 시계방향 반시계방향으로 각각 1초간 회전하고, 후진하는 동작입니다. 다양한 경우의 바퀴속도 변화를 설정해서 로봇의 동작을 만들어 보기 바랍니다.

4.4 점진적 가속과 감속

바퀴를 움직이는 서보모터의 속도를 점진적으로 선형 가속하는 코드를 사용하려면, 몇가지 고려해야할 사항들이 있습니다. 기본적인 서보모터 동작수단인 순간적 속도변경 방식과 비교해서 살펴보세요.

(1) 에너지 효율성
점진적 속도 변경: 경우에 따라 점진적인 가속 및 감속 방식이 에너지 효율적일 수 있습니다. 속도의 급격한 변화는 더 높은 전류 스파이크를 요구하며, 이는 배터리에 더 많은 부담을 줄 수 있습니다.
순간적인 속도 변경: 즉각적인 속도 변경은 높은 돌입 전류로 이어질 수 있으며, 이는 비효율적일 뿐만 아니라 배터리와 모터의 수명을 단축시킬 수도 있습니다.

(2) 기계적 스트레스
점진적 변화로 인한 스트레스 감소: 속도의 점진적 변화는 모터 및 연결된 모든 기계 구성 요소의 기계적 스트레스를 줄여줍니다. 이로 인해 해당 구성요소의 수명이 길어질 수 있습니다.
즉각적인 변경으로 인한 스트레스 증가: 갑작스러운 변경은 모터와 부하에 가해지는 갑작스러운 힘으로 인해 더 많은 마모를 일으킬 수 있습니다.

(3) 배터리 수명 및 성능
배터리 수명을 위한 점진적인 변화: 전력 수요의 원활한 전환은 배터리를 가열하고 손상시킬 수 있는 과도한 전류 소모를 방지하므로 시간이 지나도 배터리 상태를 유지하는데 도움이 될 수 있습니다.
순간적인 변경 및 잠재적인 배터리 변형: 갑작스러운 속도 변화로 인한 높은 피크 전류는 배터리에 부담을 주어 잠재적으로 전체 수명과 성능을 감소시킬 수 있습니다.

(4) 제어 및 정밀도
점진적 변경을 통한 더 나은 제어: 점진적인 변경을 통해 종종 모터에 대한 더 나은 제어가 가능하며 이는 정밀도가 요구되는 응용 분야에서 매우 중요할 수 있습니다.
즉각적인 변경을 통한 제어 과제: 즉각적인 변경은 구현이 더 간단하지만 특히 섬세하거나 정밀성이 요구되는 작업에서는 동일한 수준의 제어를 제공하지 못할 수 있습니다.

(5) 애플리케이션별 고려 사항
부하 특성: 각 방법 효율성은 모터에 의해 구동되는 부하 특성에 따라 달라질 수도 있습니다.
작동 환경: 온도, 사용 빈도, 필요한 토크 등의 작동 환경은 어떤 방법이 더 적합한지에 영향을 줄 수 있습니다.

아래 예제 gradualServoSpeed(int targetLeftSpeed, int targetRightSpeed)는 아두이노로봇의 바퀴를 점진적으로 움직이는 방식의 코드입니다. 순간적인 바퀴 속도변경 방식과 비교해서 어떤 차이가 있는지 살펴보세요. 아래 메소드는 목표속도를 즉시 실행하지 않고 점진적으로 변경 실행합니다. 그래서 목표속도에 도달할 때까지 현재속도를 단계적으로 증가 또는 감소시킵니다.

```
void SelfAbot::gradualServoSpeed(int targetLeftSpeed, int targetRightSpeed) {
    int currentLeftSpeed = 0; // Initialize current speed for left servo
    int currentRightSpeed = 0; // Initialize current speed for right servo
    int step = 5; // Determine the step size for speed change

    while (currentLeftSpeed != targetLeftSpeed || currentRightSpeed != targetRightSpeed) {
        unsigned long currentMillis = millis();

        // Accelerate or decelerate the left servo
        if (currentLeftSpeed < targetLeftSpeed) {
            currentLeftSpeed += step;
        } else if (currentLeftSpeed > targetRightSpeed) {
            currentLeftSpeed -= step;
        }

        // Accelerate or decelerate the right servo
        if (currentRightSpeed < targetLeftSpeed) {
            currentRightSpeed += step;
        } else if (currentRightSpeed > targetRightSpeed) {
            currentRightSpeed -= step;
        }

        // Update servo speeds
        _servoLeft.writeMicroseconds(1500 + currentLeftSpeed);
        _servoRight.writeMicroseconds(1500 + currentRightSpeed);

        // Wait for ms before the next change
        while (millis() - currentMillis < 10) {
            // Small delay to wait until ms has passed
        }
    }
}
```

위 코드의 동작방식을 아래에서 설명합니다.
'currentLeftSpeed' 및 'currentRightSpeed'는 각 서보 현재속도를 추적하는데 사용됩니다.
'while' 루프는 현재 속도가 두 서보의 목표 속도와 일치할 때까지 계속됩니다.
if 문은 속도를 높일지 줄일지 확인합니다.
'millis()' 함수는 타이밍에 사용됩니다. 이렇게 하면 속도가 10ms마다 변경됩니다. 시간간격을 10ms에서 20ms, 20-50ms, 50-100ms 사이의 값으로 바꾸면서 응답성과 정밀성을 충족하는 최적의 값을 찾으면 됩니다.

(참고) ---
차단 코드(blocking code)란? 바퀴와 연결된 서보모터 가속/감속 코드는 속도가 변화하는 시간이 존재하기 때문에, 특히 센서신호를 실시간으로 처리해야 하는 경우 고려해야할 점이 있습니다.

위의 예제 코드는 gradualServoSpeed(int targetLeftSpeed, int targetRightSpeed) 메소드가 실행되면, 메소드내 while 문이 완료될 때까지(즉 서보모터가 목표속도에 도달할 때까지) 프로그램이 다른 작업을 수행할 수 없습니다. 이런 코드 작용을 '차단코드(blocking code)' 라고 부릅니다. 아두이노와 같은 단일 스레드 환경에서는 코드 차단이 제한적 요소입니다. 차단 조건에서는 센서 입력에 응답하거나 다른 작업을 처리하거나 동시 작업을 수행할 수 없습니다. arduino 로봇이 실시간 입력(예: 장애물 감지 센서)에 반응해야 하는 경우 차단 기능을 사용하는 것이 문제가 될 수 있습니다. 함수가 실행되는 동안에는 이러한 입력을 효과적으로 처리할 수 없습니다.

이런 문제를 해결하기 위해 '상태머신 로직' 이나 '타이머 인터럽트 또는 콜백' 수단을 사용할 수 있습니다. 이와 같은 '비차단 코드(non-blocking code)'를 사용하려면 더 복잡한 설정이 필요합니다.

현재의 예제코드는 정지상태에서 임의의 속도로 움직이는 최초 바퀴동작에서만 유효합니다. 그렇지만, 바퀴의 속도가 계속 변화하는 경우 현재 속도값을 더 고려해야 합니다. 아래 코드는 현재 속도 값을 매개변수(int currentLeftSpeed, int currentRightSpeed)로 사용한 개념을 적용한 예제 코드입니다.

```
void SelfAbot::gradualServoSpeed(int currentLeftSpeed, int currentRightSpeed, int targetLeftSpeed, int targetRightSpeed) {
    int step = 5; // Determine the step size for speed change

    while (currentLeftSpeed != targetLeftSpeed || currentRightSpeed != targetRightSpeed) {
        unsigned long currentMillis = millis();

        // Accelerate or decelerate the left servo
        if (currentLeftSpeed < targetLeftSpeed) {
            currentLeftSpeed += step;
        } else if (currentLeftSpeed > targetLeftSpeed) {
            currentLeftSpeed -= step;
        }

        // Accelerate or decelerate the right servo
        if (currentRightSpeed < targetRightSpeed) {
            currentRightSpeed += step;
        } else if (currentRightSpeed > targetRightSpeed) {
            currentRightSpeed -= step;
        }
```

```
            // Update servo speeds
            _servoLeft.writeMicroseconds(1500 + currentLeftSpeed);
            _servoRight.writeMicroseconds(1500 + currentRightSpeed);

            // Wait for 5ms before the next change
            while (millis() - currentMillis < 5) {
                // Small delay to wait until 5ms has passed
            }
        }
    }
```

두 개 바퀴의 현재 속도와 목표 속도 값들을 모두 관리하는 것은 사용자에게 불편함을 만들 수 있습니다. 그래서 처음 서보모터를 움직이기 시작하면, 서보모터의 현재 속도 값을 멤버 변수에 자동으로 저장하는 수단을 사용할 수 있습니다.

수정된 코드에서는 앞서 소개한 매개변수를 클래스의 멤버 변수로 캡슐화해서 코드에 적용할 수 있습니다. 아래 예제코드를 'SelfAbot'클래스에 추가해서 테스트해볼 수 있습니다.

```
class SelfAbot {
private:
    int _currentLeftSpeed;    // Current speed for left servo as a class member
    int _currentRightSpeed;   // Current speed for right servo as a class member

public:
    SelfAbot() : _currentLeftSpeed(0), _currentRightSpeed(0) {
        // Default constructor logic, if any
    }
    // Constructor with servo pin numbers
    SelfAbot(byte servoLeftPin, byte servoRightPin): _currentLeftSpeed(0), _currentRightSpeed(0) {
        attachServos(servoLeftPin, servoRightPin);
    }
    void gradualServoSpeed(int targetLeftSpeed, int targetRightSpeed);
};

void SelfAbot::gradualServoSpeed(int targetLeftSpeed, int targetRightSpeed) {
    int step = 5; // Determine the step size for speed change
    while (_currentLeftSpeed != targetLeftSpeed || _currentRightSpeed != targetRightSpeed) {
        unsigned long currentMillis = millis();
        // Accelerate or decelerate the left servo
        if (_currentLeftSpeed < targetLeftSpeed) {
            _currentLeftSpeed += step;
        } else if (_currentLeftSpeed > targetLeftSpeed) {
            _currentLeftSpeed -= step;
        }
        // Accelerate or decelerate the right servo
        if (_currentRightSpeed < targetRightSpeed) {
            _currentRightSpeed += step;
        } else if (_currentRightSpeed > targetRightSpeed) {
            _currentRightSpeed -= step;
        }
```

```
            // Update servo speeds
            _servoLeft.writeMicroseconds(1500 + _currentLeftSpeed);
            _servoRight.writeMicroseconds(1500 + _currentRightSpeed);

            // Wait for 10ms before the next change
            while (millis() - currentMillis < 10) {
                // Small delay to wait until 10ms has passed
            }
        }
    }
```

아래 아두이노 스케치는 gradualServoSpeed() 함수로 사용하는 방법입니다. 아래 스케치를 업로드해서 점진적인 속도변화를 실습해보세요. 점진적인 속도변화를 만드는 지금의 메소드는 속도 0 ~ 100사이를 5씩 증가할 때마다 10ms 시간이 소요되므로, 목표속도까지 200ms 의 시간이 걸립니다. 이 점을 기억하고 로봇을 동작시켜야 합니다.
선형적으로 변화하는 점진적 서보 속도변화를 다르게 만들려고 한다면, 코드에서 step 간격을 조정하거나, delaytime 에 해당하는 while (millis() - currentMillis < 10) 코드에서 10의 값을 다르게 조정하면 됩니다.

gradualServoSpeed() 함수는 currentLeftSpeed, currentRightSpeed 전역변수를 사용해서 현재상태의 속도값을 저장하고, 사용해서 로봇의 점진적인 속도변화를 만들 수 있습니다.

지금의 메소드는 제8장에서 다루는 클래스 상속에서 서보모터의 점진적 속도변화를 다루는 실습예제로 다시 소개할 예정입니다.

Ex4.4_abot_gradual_speed.ino 스케치를 업로드하세요.

```
#include "SelfAbot.h"
#define INVALID_PIN 255;

const byte servoLeftPin = INVALID_PIN;
const byte servoRightPin = INVALID_PIN;

int currentLeftSpeed = 0;
int currentRightSpeed = 0;

SelfAbot abot(servoLeftPin, servoRightPin);

void setup() {
    abot.setup();
    abot.servoAttachPins(11, 10);
}
void loop() {
    gradualServoSpeed(100, -100);
    delay(2000);
    gradualServoSpeed(20, -20);
    delay(2000);
    gradualServoSpeed(-100, 100);
    delay(2000);
    gradualServoSpeed(0, 0);
    delay(2000);
}
```

```
void gradualServoSpeed(int targetLeftSpeed, int targetRightSpeed) {
    int step = 5; // Determine the step size for speed change
    while (currentLeftSpeed != targetLeftSpeed || currentRightSpeed != targetRightSpeed) {
        unsigned long currentMillis = millis();
        // Accelerate or decelerate the left servo
        if (currentLeftSpeed < targetLeftSpeed) {
            currentLeftSpeed += step;
        } else if (currentLeftSpeed > targetLeftSpeed) {
            currentLeftSpeed -= step;
        }
        // Accelerate or decelerate the right servo
        if (currentRightSpeed < targetRightSpeed) {
            currentRightSpeed += step;
        } else if (currentRightSpeed > targetRightSpeed) {
            currentRightSpeed -= step;
        }
        // Update servo speeds
        abot(11, 10).servoSpeed(currentLeftSpeed, currentRightSpeed);

        // Wait for 10ms before the next change
        while (millis() - currentMillis < 10) {
            // Small delay to wait until 10ms has passed
        }
    }
}
```

(1) 로봇의 하드웨어 설정:
로봇의 좌/우 서보를 아두이노 보드의 지정된 핀(11번, 10번)에 연결합니다.
아두이노 보드가 적절하게 전원 공급되고 있고 서보가 로봇에 올바르게 정렬되어 장착되어 있는지 확인하세요.

(2) 스케치 업로드:
아두이노를 컴퓨터에 USB로 연결합니다. 아두이노 IDE에서 Ex4.4_abot_gradual_speed.ino 스케치를 보드에 업로드합니다.

(3) 아두이노로봇 동작:
코드가 업로드되고, 3접점 스위치를 2의 위치로 전환하면, 서보가 움직이기 시작해야 합니다. 서보의 속도가 0에서 100으로 점진적으로 증가하고, 2초 동안 유지한 후 다시 점진적으로 0으로 감소하는 것을 관찰하세요. 서보 속도의 점진적 증가 및 감소는 로봇의 움직임을 더 부드럽게 만들어야 합니다.

(4) 다양한 속도와 지속 시간으로 실습:
loop 함수 안의 gradualServoSpeed 매개변수를 수정하여 다양한 속도로 실습해보세요.
delay 지속 시간을 조정하여 로봇이 더 많은 시간 동안 움직이거나 방향을 바꿀 수 있도록 하세요. 변경 사항이 로봇의 움직임과 속도 전환에 어떻게 영향을 미치는지 관찰하세요.

(5) 코드 이해하기:
gradualServoSpeed 함수를 살펴보며 점진적 속도 변경이 어떻게 구현되는지 이해하세요.

while 루프의 사용과 각 서보의 속도가 어떻게 작은 단계(step 변수)로 조정되어 목표 속도에 도달하는지 주목하세요.

이 실습 시나리오는 서보 모터의 속도를 부드럽고 제어된 방식으로 조절하는 방법을 이해하는 데 도움을 주며, 로봇 프로젝트에서 더 복잡하고 미묘한 움직임을 만드는 기초를 제공합니다.

4.5 추가 메소드 : maneuver()

maneuver 함수는 서보모터 움직임 속도 값을 매개변수로 전달하면서, 지속시간을 매개변수에 추가하는 메소드입니다. C언어 기초편에서 사용했던 maneuver() 함수를 메소드로 구현해 봅시다. 이전에 사용했던 maneuver() 함수는 두 개의 서보모터를 동시에 움직이면서, 서보모터의 지연시간을 매개변수에 추가한 형태입니다.

아래 maneuver 메소드는 C언어 기초편에서 사용했던 maneuver() 함수와 조금 다릅니다. 어떤 점이 다른지 살펴봅시다. 클래스에 추가해서 사용할수도 있고, .ino 파일에서 직접 사용할 수도 있습니다.

```
void SelfAbot::maneuver(int speedLeft, int speedRight, int msTime) {
    servoSpeed(speedLeft, speedRight);
    if (msTime == -1) {
        // Indefinite operation, servos remain engaged
    } else {
        delay(msTime);
        _servoLeft.detach();
        _servoRight.detach();
    }
}
```

속도 설정 방법은 먼저 왼쪽 및 오른쪽 서보의 속도를 설정합니다. 이는 각각 왼쪽 및 오른쪽 서보에 대해 두 가지 속도 값을 사용하는 'SelfAbot' 클래스의 'servoSpeed' 메서드를 사용하여 수행됩니다.

서보의 지연시간은 msTime 매개변수를 확인합니다.
(1) msTime이 -1이면 서보는 무기한으로 계속 실행됩니다. 이는 미리 정의된 정지 시간 없이 연속 작동하는 경우에 유용합니다.

(2) msTime이 다른 값인 경우 메서드는 해당 기간(delay(msTime) 사용)을 기다린 다음 서보를 분리하여 중지합니다. 이런 서보 동작은 오래전 공개한 “온라인 실습교재”에서 소개한 maneuver() 함수와 조금 다릅니다.

‘C언어 기초편’에서 사용했던 maneuver() 함수는 연속적으로 함수를 반복실행해서 사용할 수 있지만, 지금 코드는 maneuver() 함수를 한번만 실행하는데 적합합니다. 예정 시간후 자동으로 서보가 정지하도록 작성되었기 때문입니다. 제시된 코드는 서보를 멈추는 명령으로

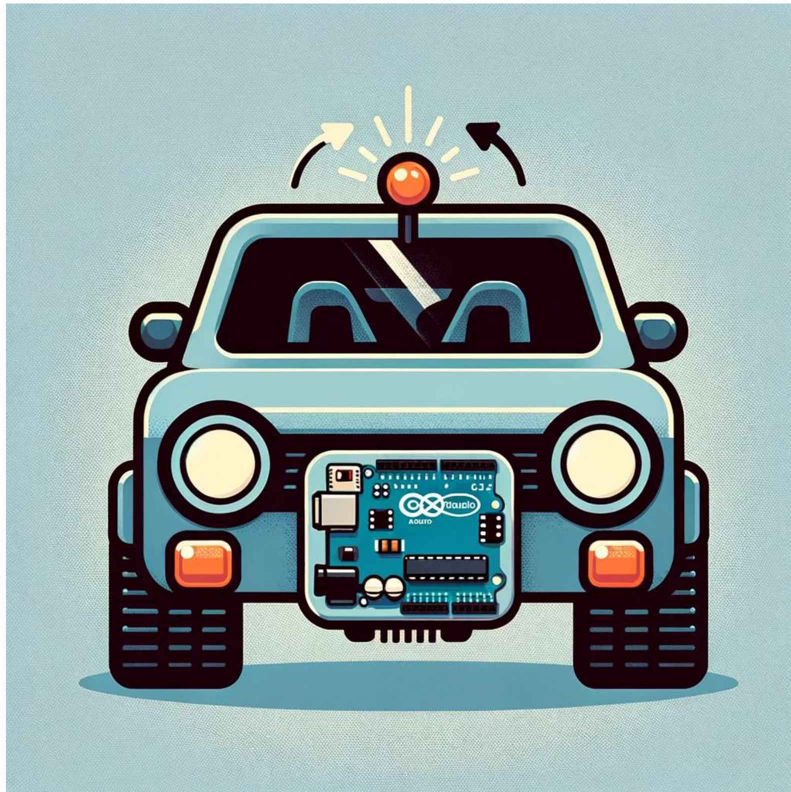
detach() 함수를 실행하는데, 예정된 시간 후에 서보를 중지시킵니다.
로봇의 계속적인 동작을 만드는데 maneuver() 함수를 사용한다면, 지금의 코드를 아래 예제처럼 수정해서 적용하면 됩니다.

Ex4.5_abot_maneuver_speed.ino 스케치를 업로드하세요.

```
#include "SelfAbot.h"
SelfAbot abot; // SelfAbot 클래스의 인스턴스 생성

void setup() {
      Serial.begin(9600); // 시리얼 통신 시작
      abot.servoAttachPins(11, 10); // 9번과 10번 핀에 서보를 연결
}
void loop() {
      maneuver(100, -100, 200);
      delay(2000); // 2초 동안 대기
}
void maneuver(int speedLeft, int speedRight, int msTime) {
      abot.servoSpeed(speedLeft, speedRight);
      if (msTime == -1) {
            // Indefinite operation, servos remain engaged
      } else {
            delay(msTime);
      }
}
```

제 5 장 포토트랜지스터 광신호 : rcTime()

5.1 아날로그 입력신호처리를 위한 실습
5.2 디지털 입력신호 방식: rcTime() 메소드
5.3 두 개의 광센서(phototransistor)로 빛의 방향인식
5.4 빛을 추적하는 아두이노로봇



적외선(Infrared) 센서의 파장대역을 나타내는 그림입니다. 적외선 빛의 영역은 눈으로 식별할 수 없는 파장대역입니다. 이 장의 실습용 포토트랜지스터는 850nm 파장에서 가장 민감하고, 제 6 장에서 사용할 적외선 센서는 980nm에서 가장 민감하게 반응합니다.

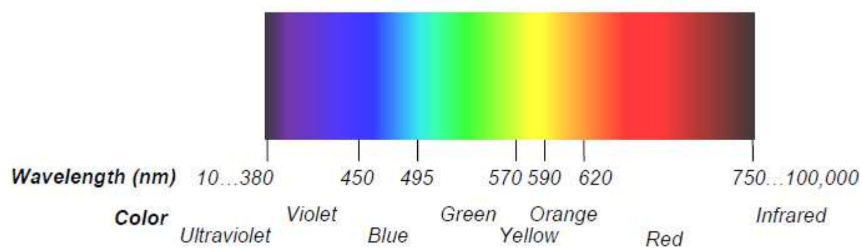


그림5.1 : 빛의 가시광선 색상과 비가시광선 파장대역 (출처: parallax.com)

아날로그 센서와 디지털 센서는 일반적으로 로봇 공학에 사용되며 각각 고유한 특성과 용도를 가지고 있습니다. 이 두 가지 유형의 센서 차이점을 이해하는 것은 센서를 로봇 시스템에 효과적으로 통합하는데 중요합니다.

아날로그 센서 특징은 다음과 같습니다.

아날로그 센서는 물리적 측정값을 나타내는 연속 신호를 생성합니다. 신호는 연속 범위에 걸쳐 달라질 수 있습니다.

아날로그 센서의 분해능은 측정되는 물리량의 매우 작은 변화를 측정하고 표현할 수 있으므로 이론적으로 무한합니다. 그러나 실제로 분해능은 잡음 수준과 처리를 위해 신호를 디지털화하는데 사용되는 아날로그-디지털 변환기(ADC)의 정밀도에 의해 제한됩니다.

아날로그 센서는 일반적으로 측정량에 비례하는 전압을 출력합니다. 이 전압의 범위는 최소값과 최대값 사이일 수 있습니다(예: 0~5V). 일반적으로 디지털 센서에 비해 간단하고 저렴합니다. 그러나 특히 높은 정밀도가 필요한 경우 관련 신호 처리의 비용과 복잡성이 더 높아질 수 있습니다.

아날로그 신호는 잡음과 간섭에 더 취약하여 특히 장거리 전송에서 신호 품질이 저하될 수 있습니다. 일반적인 예로는 서미스터(온도 센서), 포토다이오드(광 센서), 전위차계(위치 센서) 등이 있습니다.

디지털 센서 특징을 소개합니다.

디지털 센서는 일반적으로 이진 데이터 형식의 개별 신호를 생성합니다. 출력은 측정된 매개변수의 다양한 조건에 따라 높음(1 또는 HIGH) 또는 낮음(0 또는 LOW) 상태입니다.

디지털 센서의 해상도는 디지털 출력의 비트 수에 따라 제한됩니다. 예를 들어, 10비트 센서는 2^10(1024)개의 이산 상태를 나타낼 수 있습니다. 이러한 센서는 일반적으로 I2C, SPI 또는 UART와 같은 디지털 통신 프로토콜을 사용하여 통신합니다. 출력은 측정된 양의 디지털 표현입니다.

디지털 센서는 아날로그 센서보다 더 복잡하고 비용이 더 많이 들 수 있습니다. 일반적으로 신호 처리 기능이 내장되어 있어 보다 정확하고 안정적인 판독값을 제공할 수 있습니다.

디지털 신호는 아날로그 신호에 비해 노이즈에 덜 민감합니다. 이로 인해 장거리 전송 및 전기적 잡음이 심한 환경에서 디지털 센서의 신뢰성이 더욱 높아집니다. 예로는 디지털 온도계, 적외선 센서, 디지털 가속도계 등이 있습니다.

5.1 아날로그 입력신호처리를 위한 실습

C언어 아두이노로봇 실습안내서에서 소개했던 아날로그 입력신호를 화면출력하는 코드입니다. 아래 코드의 동작을 직접 실습해보세요. 아날로그 센서의 입력을 아두이노핀 A2 에 연결하면 됩니다.

아날로그 입력신호는 아래에 소개한 이유로 로봇 동작의 신호로 직접 사용하지 않는 경향이 있습니다. 'SelfAbot'라이브러리에서는 아날로그입력 신호를 처리할 메소드를 구현하지 않았습니다. 라이브러리와 상관없이 동작하는 코드를 실습해보세요.

어떤 이유로 로봇동작을 유도하는 외부 입력신호로 아날로그 신호를 덜 사용하는지 설명합니다. 디지털 입력신호 방식이 아날로그 입력신호 방식보다 더 선호되는 경향이 있습니다. 로봇작업의 특성과 정밀도, 단순성 및 소음에 대한 견고성 측면에서 디지털 신호의 장점이 결합되어 디지털 제어를 선호하는 경우가 많습니다.

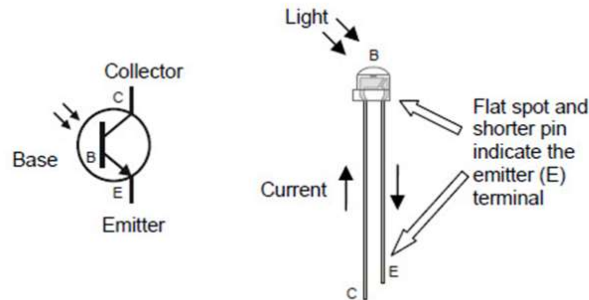
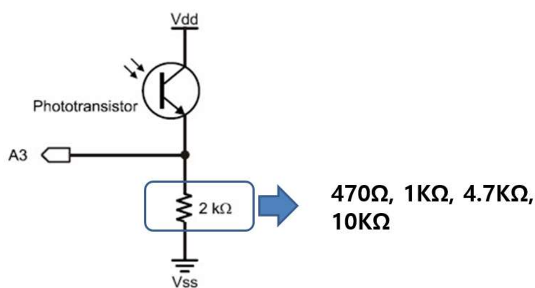


그림5.2 : 포토트랜지스터 심볼과 핀 개념도 (출처: parallax.com)



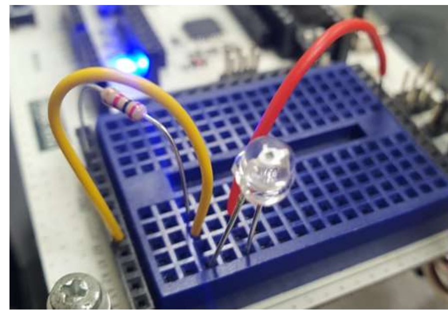


그림5.3 : 포토트랜지스터 광센서의 아날로그 신호처리 회로

포토트랜지스터의 아날로그 신호처리를 위해서는 그림5.3과 같이 연결하면 됩니다. 이때 광센서 신호의 감도를 조절하려면, 그림5.3처럼 저항 값 크기를 바꿔 출력해보세요.

Ex5.1_phototransistor_analogReadvolts.ino 스케치를 업로드하세요.

```
int sensorPin = A2;

void setup() {
  //  abot.setup();
      Serial.begin(9600);
}

void loop() {
      Serial.print("A2 = ");
      Serial.print(volts(A2));
      Serial.println(" volts");
      delay(1000);
}

float volts(int adpin)              // Measures volts at adPin
{                                    // Returns floating point voltage
      return float(analogRead(adpin)) * 5.0 / 1024.0;
}
```

(1) 정밀도 및 노이즈: 아날로그 신호는 노이즈 및 간섭에 더 취약할 수 있으며, 이로 인해 판독값이 덜 정확하거나 변동할 수 있습니다. 정밀도와 일관성이 핵심인 로봇공학에서 이는 심각한 단점이 될 수 있습니다. 반면에 디지털 신호는 잡음에 더 강합니다.

(2) 처리의 복잡성: 아날로그 신호는 처리를 위해 아날로그-디지털 변환기(ADC)를 사용하여 디지털 값으로 변환해야 합니다. 이 변환은 특히 고해상도 판독이 필요한 경우 복잡성을 추가하고 대기 시간을 유발할 수 있습니다.

(3) 이산 제어 요구: 모터 제어와 같은 많은 로봇 기능은 본질적으로 디지털 방식입니다(켜기/끄기, 앞으로/뒤로, 정지/이동). 이러한 컨트롤에 디지털 신호를 사용하는 것이 더 간단하고 효율적입니다.

(4) 제한된 아날로그 입력: arduino 보드에는 일반적으로 디지털 핀에 비해 제한된 수의 아날로그 입력 핀이 있습니다. 많은 센서와 액추에이터가 사용될 수 있는 로봇 공학에서는 디지털 핀의 가용성이 높아져 복잡한 설계에 더욱 실용적입니다.

(5) 표준화 및 호환성: 서보, 드라이버가 있는 DC 모터, 센서와 같은 많은 로봇 구성 요소는 디지털 제어 신호와 함께 작동하도록 설계되었습니다. 아날로그 신호와 함께 이러한 표준 구성 요소를 사용하려면 추가 회로 또는 변환 방법이 필요합니다.

(6) 전원 공급 장치 레벨에 따른 가변성: 아날로그 신호는 전원 공급 장치 전압의 변화에 따라 달라질 수 있으며 이로 인해 일관되지 않은 동작이 발생할 수 있습니다. 디지털 신호는 이러한 변동의 영향을 덜 받아 보다 안정적인 작동을 제공합니다.

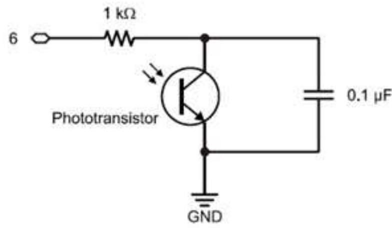
(7) 프로그래밍 및 디버깅의 용이성: 프로그래밍 및 디버깅 측면에서 디지털 신호를 사용하는 것이 더 쉬운 경우가 많습니다. 디지털 신호의 이진 특성(높음/낮음, 참/거짓)은 프로그래밍 논리와 잘 일치하므로 제어 알고리즘을 구현하고 문제를 해결하는 것이 더 간단합니다.

그렇지만, 연속적인 값 범위를 제공하는 판독 센서(예: 온도, 광도 또는 거리 센서) 또는 점진적인 변화와 미세한 제어가 필요한 상황(예: 전위차계를 사용하여 서보 모터의 위치를 제어하는 경우) 등에는 아날로그 입력이 더 적합할 수 있습니다.

5.2 디지털 입력신호 방식: rcTime() 메소드

포토트랜지스터 광센서를 사용해서 아날로그 신호 방식으로 처리하는 내용을 앞 부분에서 다루었습니다. 이제 동일한 포토트랜지스터 광센서를 사용해서 디지털 신호 방식으로 처리하는 내용을 소개할 차례입니다.

그림5.4는 광센서 감지신호를 RC회로와 결합해서, 디지털신호를 처리할 수 있습니다. 그림의 회로에서 디지털핀의 모드를 HIGH에서 LOW로 전환하면, RC회로가 만들어집니다.



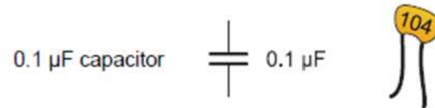
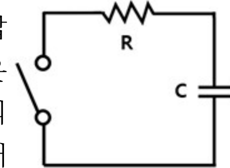


그림5.4 : 디지털신호 측정을 위한 회로 (출처: parallax.com)　　그림5.5 : 캐패시터 심볼과 표시방법

아래 Ex5.2 .ino 예제 코드는 'SelfAbot'클래스의 rcTime 메소드를 사용하여 arduino의 두 핀(8 및 9)에서 디지털 입력 신호를 읽습니다. 이 방법은 디지털 핀에서 아날로그와 유사한 값을 읽는데 사용되는 일반적인 기술인 RC time 즉 시정수라고 부르는 공학적 시간개념으로 디지털 입력신호를 처리하는 방식입니다.

RC회로 이해



RC 회로는 함께 사용되는 저항기와 커패시터로 구성된 기본적인 전기 회로입니다. 전압이 가해지면 커패시터는 저항을 통해 충전됩니다. 전압이 제거되면 커패시터가 방전됩니다. 충전 또는 방전에 걸리는 시간은 저항과 커패시터의 값에 따라 달라집니다. 그리스 문자 타우(τ)로 표시되는 RC 회로의 시정수(RCtime)는 충전시 커패시터 양단의 전압이 최대값의 약 63.2%까지 충전되거나, 방전시 방전되는데 걸리는 시간을 측정한 것인데, 이 순간을 1 타우라고 부릅니다. 방전시 최대값의 약 36.8%까지 감소합니다.

(참고) **시정수 rctime 이해하기** --

RC time 시정수는 커패시터가 저항기를 통해 얼마나 빨리 충전 또는 방전되는지를 나타내는 공학개념입니다. RC 회로는 전류의 흐름을 제한하거나 제어하는 저항부품(R)과 전기장에 전기 에너지를 저장하는 커패시터부품(C)로 구성된 간단한 전기회로입니다.

시간 상수 τ는 다음 공식을 사용하여 계산됩니다.

$\tau = R \times C$

τ(tau)는 초 단위의 시간 상수입니다.

R은 저항(Ω)입니다.

C는 패럿(F) 단위의 정전용량입니다.

이렇게 정의되는 시정수 또는 시상수를 이해해봅시다. 시정수 τ(tau)는 전체 100%에 대한 상대적 표현이므로 RC회로의 어떤 부품조합에서도 동일한 시정수(예를 들면 1 τ(tau)로 충전 63.2% 또는 방전 36.8%)로 충전과 방전의 상태를 나타낼 수 있습니다. 다만, 1 τ(tau)에 대한 환산 시간 값만 저항과 커패시터 부품의 조합 차이에 따라 다르게 표현됩니다.

(1) 커패시터 충전: 회로에 전원이 공급되면 커패시터는 저항기를 통해 충전을 시작합니다. 1

시정수(τ)는 커패시터 양단의 전압이 공급 전압의 약 63.2%에 도달하는데 걸리는 시간입니다.

(2) 커패시터 방전: 회로가 전원에서 분리되면 커패시터는 저항기를 통해 방전을 시작합니다. 여기서 1 τ는 커패시터의 전압이 초기값의 약 36.8%까지 떨어지는데 걸리는 시간입니다.

(3) 5 시상수: 일반적으로 커패시터는 약 5τ(5개의 시상수) 후에 거의 완전히 충전됩니다(99% 이상). 마찬가지로 5τ 후에는 거의 완전히 방전됩니다.

실제 예
RC 회로에 1000Ω(1kΩ)의 저항기와 1μF(1μF)의 커패시터가 있다고 상상해 보십시오. 이 회로의 시상수 τ는 다음과 같습니다.
$\tau = 1000\Omega \times 1\mu F = 1000 \times 10^{-6}$ seconds = 1 milisecond

이는 커패시터가 공급 전압의 63.2%까지 충전하거나 초기 전압의 36.8%까지 방전하는데 1 밀리초가 걸린다는 것을 의미합니다. 거의 완충되거나 완전 방전되는데 걸리는 시간은 약 5 밀리초입니다.

마이크로컨트롤러의 역할
arduino(또는 다른 마이크로컨트롤러)를 사용하여 충전 및 방전 프로세스를 제어하고 이러한 프로세스가 발생하는데 걸리는 시간을 측정합니다. 이 시간은 RC 회로의 특성을 나타냅니다.

코드에서 abot.rcTime(8)을 실행하면, 디지털 핀으로 읽는 값(digitalRead)이 High 상태에서 LOW 상태로 변화하는 순간까지의 지연시간을 읽어서 반환하는 값을 얻습니다. 그래서 아날로그 입력신호를 처리하는 것과 비교하면 디지털 입력신호 방식은 외부입력 신호가 변화하는 특정한 순간에 대한 처리방식이라고 할 수 있습니다.
아두이노 우노의 경우 마이크로컨트롤러가 HIGH 값을 인식하는 임계값은 0.6 * VCC(5V) 이고, LOW 값을 인식하는 임계값은 0.3 * VCC(5V) 이어야 합니다. RC회로에서 처음 5V 전압에서 전압이 점차 감소해서, 아두이노가 어느 특정 지점의 전압에서 HIGH 상태를 LOW 상태로 바꿔 인식하는 순간까지의 시간을 코드로 측정합니다.

외부 회로구성은 그림5.4를 참고하면 됩니다.

아래 rcTime() 메소드 구현에 있어서, 함수오버로딩은 사용하지 않았습니다. 함수오버로딩은 바퀴를 움직이는 핀정보만 적용하는 설계원칙에 근거한 것입니다. 만약 실습을 진행하는 구독자가 아래 rcTime() 메소드에도 함수 오버로딩을 적용하고자 한다면, 함수 오버로딩을 사용할 수 있는 메소드를 추가하면 됩니다.
이후에서 설명하는 모든 센서신호 처리 메소드들은 함수오버로딩을 사용하지 않습니다.

'SelfAbot'클래스의 rcTime() 메소드 구현내용을 소개합니다.

```
    unsigned long SelfAbot::rcTime(byte pin) {
```

```
        pinMode(pin, OUTPUT);
        digitalWrite(pin, HIGH);
        delay(1);
        pinMode(pin, INPUT);
        digitalWrite(pin, LOW);
        unsigned long startTime = micros();
        unsigned long elapsedTime = 0;
        while (digitalRead(pin) == HIGH) {
            elapsedTime = micros() - startTime;
        }
        return elapsedTime;
    }
```

'rcTime'이라는 'SelfAbot' 클래스 메소드는 디지털 입력 핀의 상태가 HIGH에서 LOW로 변경되는데 걸리는 시간을 측정합니다. 이 방법은 일반적으로 RC(저항기-커패시터) 회로에서 커패시터의 충전 또는 방전 시간을 측정하는데 사용됩니다. 이해하기 쉽게 분해하면 다음과 같습니다.

rcTime 메소드 단계별 이해하기:

(1) 핀 모드를 출력으로 설정: pinMode(pin, OUTPUT);
이 줄은 지정된 'pin'을 출력으로 설정합니다. RC 회로의 커패시터를 충전하려면 먼저 출력용 핀을 구성해야 합니다.

(2) 핀을 HIGH로 설정: digitalWrite(pin, HIGH);
이 라인은 HIGH 신호(대부분의 Arduino 보드에서는 5V)를 핀으로 보내 해당 핀에 연결된 커패시터를 충전하기 시작합니다.

(3) 짧은 지연: delay(1);
이로 인해 커패시터가 충전될 시간을 확보하기 위해 잠시 일시 중지됩니다. 이 지연 기간은 충전 시간에 영향을 미칠 수 있습니다.

(4) 핀을 입력으로 전환: pinMode(pin, INPUT);
핀이 입력으로 재구성되었습니다. 이는 커패시터의 방전 시간을 측정하기 위해 수행됩니다.

(5) 핀이 LOW인지 확인: digitalWrite(pin, LOW);
핀이 입력으로 설정되어 있더라도 이 라인은 내부 풀업 저항이 비활성화되었는지 확인하는데 자주 사용됩니다. 이는 핀이 arduino의 영향을 받지 않고 외부 회로의 영향만 받도록 보장합니다.

(6) 시작 타이밍: unsigned long startTime = micros();
이 줄은 arduino가 현재 프로그램을 실행하기 시작한 이후 마이크로초 시간을 제공하는 아두이노 내부함수 'micros()'를 사용하여 측정 시작 시간을 기록합니다.

(7) 핀이 LOW가 될 때까지의 시간 측정:
while 루프는 while(digitalRead(pin) == HIGH)는 핀이 HIGH로 유지되는 한 실행됩니다.
루프 내에서 elapsedTime = micros() - startTime;은 시작 시간 이후 경과한 시간을 계산합니다. 이는 핀이 LOW로 표시될 때까지 커패시터가 저항을 통해 방전하는데 걸리는 시간을 효과적으로 측정합니다.

(8) 경과 시간 반환: return elapsedTime;
마지막으로 이 메서드는 경과 시간을 마이크로초 단위로 반환합니다. 이 값은 핀에 연결된 특정 RC 회로의 특성인 RC 시간의 특정한 값을 나타냅니다.

아래 코드는 rcTime 메소드를 사용해서 외부 신호를 감지하는 용도로 사용할 수 있는 예제입니다.

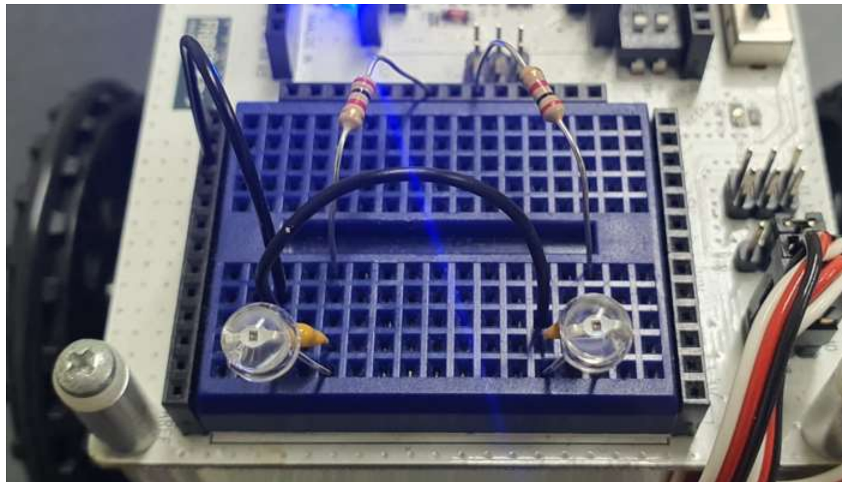


그림5.6 : 포토트랜지스터 광센서를 로봇에 장착한 모습

Ex5.2_phototransistor_rcTime.ino 스케치를 업로드하세요.

```
#include "SelfAbot.h"
SelfAbot abot;

void setup() {
  //  abot.setup();
    Serial.begin(9600);
}
void loop() {
    unsigned long rctimeValLeft = abot.rcTime(8);
    Serial.print("rcTime Left = ");
    Serial.print(rctimeValLeft);
    Serial.println(" us  ||  ");

    unsigned long rctimeValRight = abot.rcTime(6);
    Serial.print("rcTime Right = ");
    Serial.print(rctimeValRight);
    Serial.println(" us");
    delay(500);
}
```

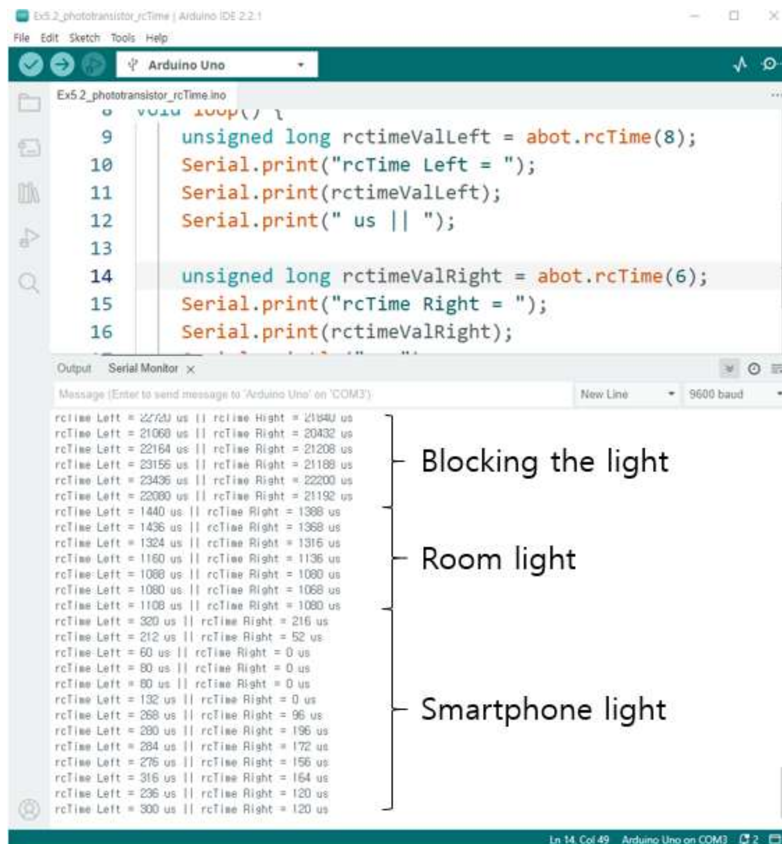


그림5.7 : 외부 불빛을 비추지 않을 때와 비출 때의 센서 출력 값 변화

그림5.7은 두 개의 광센서를 향해서 스마트폰 불빛을 비출 때와 비추지 않고 방안의 형광등 불빛 그리고 손으로 광센서를 감싸고 있을 때의 외부 빛 차단 출력 값 변화를 표시합니다. 아날로그 입력회로 방식과 비교해서 외부 빛의 변화에 따라 변화하는 데이터의 폭이 매우 큰 것을 살펴볼 수 있습니다.

디지털 입력신호 처리방식에 대한 장단점을 소개합니다.

장점:

(1) 단순성: 이 방법은 아날로그 핀에 비해 arduino 보드에 더 풍부한 디지털 I/O 핀을 사용합니다. 이는 더 넓은 범위의 애플리케이션에 대한 접근 방식을 더욱 다양하게 만듭니다.

(2) ADC 불필요: 이 방법은 디지털 핀에서 작동하므로 아날로그-디지털 변환(ADC)이 필요하지 않아 회로 설계와 코드가 단순화됩니다.

(3) 비용 효율적: 아날로그와 유사한 판독을 위해 디지털 핀을 활용하면 외부 ADC와 같은 추가 하드웨어가 필요하지 않으므로 비용 효율적일 수 있습니다.

(4) 사용자 정의 가능한 해상도: 타이밍과 코드 로직을 조작하여 특정 애플리케이션 요구 사항에 따라 측정 해상도를 어느 정도 사용자 정의할 수 있습니다.

단점:

(1) 낮은 정밀도: 특히 센서 출력의 범위가 넓거나 높은 분해능이 필요한 경우 디지털 판독값은 실제 아날로그 판독값에 비해 정확도가 떨어집니다.

(2) 노이즈에 민감함: 디지털 신호 처리는 전기적 노이즈에 더 취약할 수 있으며, 이는 타이밍 기반 측정의 정확도에 영향을 미칠 수 있습니다.

(3) 프로세서 시간: rcTime과 같은 방법은 핀이 상태를 변경하는 데 걸리는 시간을 측정하는 루프에 의존하는 경우가 많기 때문에 프로세서 집약적일 수 있습니다. 이는 다른 작업을 동시에 처리하는 프로세서의 능력을 제한할 수 있습니다.

(4) 보정 및 일관성 문제: 타이밍 기반 측정의 정확도는 온도, 전압 변화 또는 여러 보드 간의 변화와 같은 요인에 의해 영향을 받을 수 있습니다. 일관되고 안정적인 교정이 어려울 수 있습니다.

(5) 제한된 적용 범위: 이 접근 방식은 모든 유형의 센서, 특히 매우 정확하고 안정적인 아날로그 판독값이 필요한 센서에는 적합하지 않습니다.

5.3 두 개의 광센서(phototransistor)를 이용한 빛의 밝기 인식

두 개의 광센서를 사용해서 좌측과 우측의 빛의 세기를 비교할 수 있습니다. 이런 방식으로 좌측과 우측 중에서 어두운 쪽과 밝은 쪽을 구분할 수 있습니다. 이와 같은 센서인식을 로봇의 움직임과 연결 통합하면, 로봇이 밝은 빛을 따라가도록 하거나, 또는 밝은 빛을 피해서 어두운 곳으로 향하도록 코드를 작성할 수 있습니다.

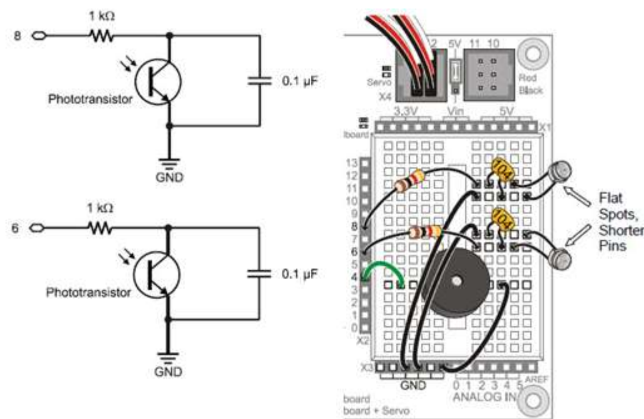


그림5.8 : 포토트랜지스터를 연결한 아두이노로봇 (출처: parallax.com)

C언어로 소개했던 코딩예제를 'SelfAbot' 클래스를 사용해서 C++코드로 작성하는 방법은 다음과 같습니다. 아래 코드는 왼쪽 광센서 신호를 abot.rcTime(8) 메소드를 사용해서 읽고, 오른쪽 광센서 신호를 abot.rcTime(6) 메소드로 읽은 다음 정규화하는 방식입니다. rctimeValLeft / (rctimeValLeft + rctimeValRight) - 0.5;처럼 센서 데이터를 정규화하면, 데이터를 쉽게 시각화할 수 있고 추가 데이터 처리가 단순해집니다.

Ex5.3_normalize_two_phototransistors.ino 스케치를 업로드하세요.

```
#include "SelfAbot.h"
SelfAbot abot;

void setup() {
  //  abot.setup();
    Serial.begin(9600);
}
void loop() {
    float rctimeValLeft = float(abot.rcTime(8));
    float rctimeValRight = float(abot.rcTime(6));

    float ndShade;
    ndShade = rctimeValLeft / (rctimeValLeft + rctimeValRight) - 0.5;
    for(int i = 0;i < (ndShade * 40) + 20; i++) {
      Serial.print(' ');
    }
    Serial.println('*');
    delay(100);
}
```

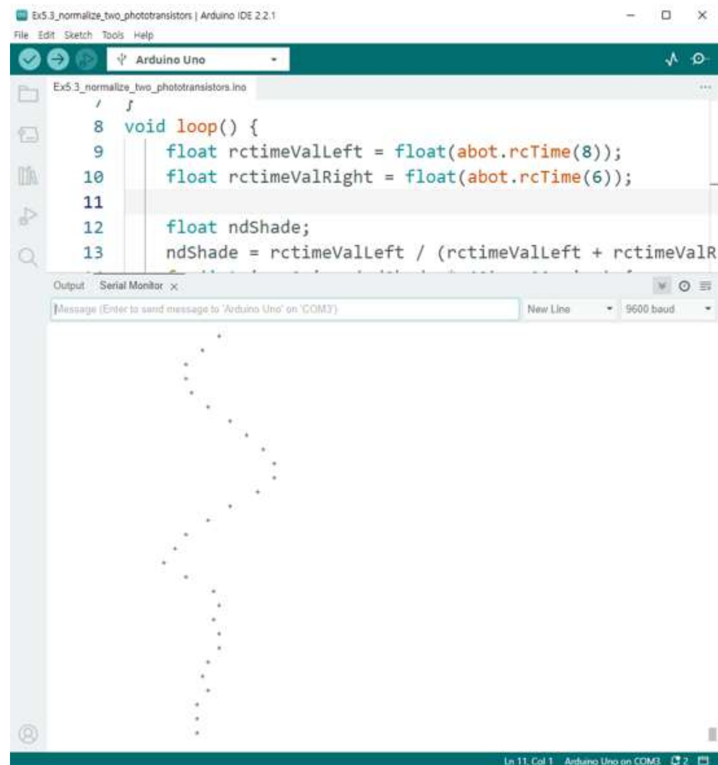


그림5.9 : 두 개의 광선서 빛의 차이를 시리얼출력 그래프로 표시

위에서 소개한 예제 코드는 다음 설명을 참고하기 바랍니다.
(1) 라이브러리 및 객체 초기화:

```
#include "SelfAbot.h"
SelfAbot abot;
```

SelfAbot.h 라이브러리를 포함하여 SelfAbot 클래스에 정의된 기능을 사용할 수 있습니다.
abot 객체를 생성하여 로봇을 제어합니다.

(2) setup() 함수:

```
void setup() {
  //abot.setup();
    Serial.begin(9600);
}
```

Serial.begin(9600);를 호출하여 시리얼 통신을 시작합니다. 이렇게 하면 시리얼 모니터를 통해 데이터를 보내고 받을 수 있습니다.

(3) loop() 함수:

```
void loop() {
    float rctimeValLeft = float(abot.rcTime(8));
    float rctimeValRight = float(abot.rcTime(6));
    …
}
```

loop() 함수에서는 rcTime 메소드를 사용하여 8번 핀과 6번 핀에 연결된 센서로부터 값을 읽어들입니다. rctimeValLeft와 rctimeValRight는 각각 왼쪽과 오른쪽 센서의 읽은 값을 저장합니다.

(4) 센서 값의 처리 및 시각화:

```
float ndShade;
ndShade = rctimeValLeft / (rctimeValLeft + rctimeValRight) - 0.5;
for(int i = 0;i < (ndShade * 40) + 20; i++) {
  Serial.print(' ');
}
Serial.println('*');
```

ndShade는 두 센서 값의 비율을 기반으로 계산된 값입니다. 이 값은 두 센서 값의 차이를 나타내며, -0.5를 빼서 값을 중앙에 맞춥니다.
for 루프와 Serial.print(' ');를 사용하여 ndShade 값에 비례하는 공백을 시리얼 모니터에 출력합니다. 이렇게 하여 센서 값의 차이를 시각적으로 표현합니다.
Serial.println('*');를 호출하여 별표를 출력합니다. 별표의 위치는 센서 값의 차이를 나타냅니다.

delay(100);는 루프의 각 반복 사이에 100밀리초의 지연을 추가합니다. 이렇게 하여 데이터의

읽기 및 출력이 너무 빠르게 반복되는 것을 방지합니다.

5.4 빛을 추적하는 아두이노로봇 실습

빛의 세기를 포토트랜지스터 센서를 사용해서 감지하고, 디지털 신호로 변환한 다음 서보모터를 동작하기 위한 변수의 값을 변경하는 과정을 보여주는 코드입니다. 앞서 설명한 것처럼, 센서데이터를 정규화한 값 ndShade는 -0.5에서 +0.5 사이를 변화합니다. ndShade 값을 기준으로 음수이면, 분자값이 상대적으로 작은 값을 나타내는 것이므로 rctimeValLeft값이 작은 것을 의미하고, 이것의 의미는 왼쪽이 밝다는 것을 의미합니다.

로봇이 전진하는 상태일 때, 양쪽 바퀴의 서보 속도 값을 조절함으로써 로봇이 빛을 따라가도록 할 수 있습니다. 만약, 왼쪽이 밝은 방향이면 로봇이 좌회전해야 하므로 왼쪽 바퀴의 속도값을 오른쪽 바퀴의 속도값보다 작게 조정하면 됩니다. 이런 조건을 충족하는 코드가 소개됩니다. 반대 쪽 오른쪽 바퀴의 서보모터 역시 동일한 원리가 적용됩니다.

아래 코드는 로봇의 서보모터를 직접 동작하기 이전에 외부 빛의 변화에 따라서 서보모터 동

Ex5.4_photoTr_rctime_serialprint.ino 스케치를 업로드하세요.

```
#include "SelfAbot.h"
SelfAbot abot;

void setup() {
  //  abot.setup();
    Serial.begin(9600);
}
void loop() {
    int leftSpeed, rightSpeed;
    float rctimeValLeft = float(abot.rcTime(8));
    float rctimeValRight = float(abot.rcTime(6));

    float ndShade;
    ndShade = rctimeValLeft / (rctimeValLeft + rctimeValRight) - 0.5;

    if (ndShade < 0.0) {
        leftSpeed = int(200.0 + (ndShade * 1000.0));
        leftSpeed = constrain(leftSpeed, -200, 200);
        rightSpeed = 200;
    } else {
        rightSpeed = int(200.0 - (ndShade * 1000.0));
        rightSpeed = constrain(rightSpeed, -200, 200);
        leftSpeed = 200;
    }

    Serial.print(leftSpeed, DEC);
    Serial.print("   ");
    Serial.print(ndShade, DEC);
    Serial.print("   ");
    Serial.println(rightSpeed, DEC);
    delay(1000);
}
```

작에 필요한 속도값이 어떻게 변화하는지 아두이노의 시리얼출력으로 점검하기 위한 목적입니다. 시리얼출력 화면으로 외부 빛의 변화에 값의 변화가 적절하다고 판단되면, 직접 서보모터를 동작시키는 코드 예제로 넘어가도 좋습니다.

```
19        rightSpeed = 200;
20      } else {
21        rightSpeed = int(200.0 - (ndShade * 1000.0));
22        rightSpeed = constrain(rightSpeed, -200, 200);
23        leftSpeed = 200;
24      }
25
26      Serial.print(leftSpeed, DEC);
27      Serial.print("  ");
28      Serial.print(ndShade, DEC);
29      Serial.print("  ");
30      Serial.println(rightSpeed, DEC);
31      delay(1000);
32  }
33
```

```
200  0.0081240539  191
196  -0.0038520693  200
137  -0.0624309492  200
159  -0.0403800582  200
200  0.0000000000  200
196  -0.0037257969  200
200  0.0151658654  184
200  0.0316934585  168
28  -0.1718987226  200
200  0.0329719781  167
200  0.0080000162  191
200  0.0185556411  181
200  0.1389910583  61
```

그림5.10 : 두 개의 광센서 감지값 차이를 로봇 속도값으로 변환하는 시리얼출력

앞선 예제에서 서보모터의 속도값이 정상적으로 만들어지고 있다고 판단했다면, 센서의 빛 방향에 따라 로봇이 동작하도록 코드를 수정하면 됩니다.

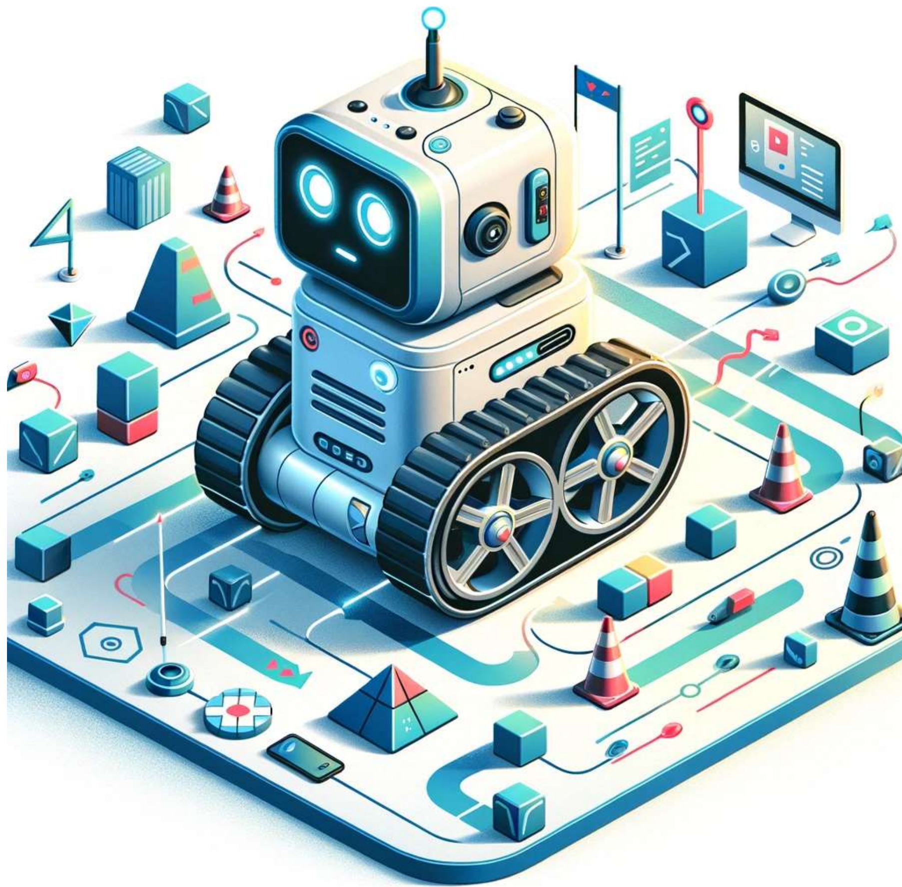
외부 빛의 방향에 반응하는 로봇의 주행동작을 만드는 원리는, 앞서 살펴본 leftSpeed와 rightSpeed 변수값을 로봇 주행메소드의 매개변수에 전달하면 됩니다. 라이브러리를 이용해서 로봇을 주행하게하는 메소드는 servoSpeed(leftSpeed, -rightSpeed)입니다. 인스턴스와 함께 코드를 사용하면 외부 빛에 반응하는 로봇의 동작을 만들 수 있는데, 아래에서 예제로 소개합니다.

Ex5.5_light_following_abot.ino 스케치를 업로드하세요.

```
#include "SelfAbot.h"
#define INVALID_PIN 255;
const byte servoLeftPin = INVALID_PIN;
const byte servoRightPin = INVALID_PIN;

SelfAbot abot(servoLeftPin,servoRightPin);

void setup() {
    abot.setup();
    abot.servoAttachPins(11, 10);
    Serial.begin(9600);
}

void loop() {
    int leftSpeed, rightSpeed;
    float rctimeValLeft = float(abot.rcTime(8));
    float rctimeValRight = float(abot.rcTime(6));

    float ndShade;
    ndShade = rctimeValLeft / (rctimeValLeft + rctimeValRight) - 0.5;

    if (ndShade < 0.0) {
        leftSpeed = int(200.0 + (ndShade * 1000.0));
        leftSpeed = constrain(leftSpeed, -200, 200);
        rightSpeed = 200;
    } else {
        rightSpeed = int(200.0 - (ndShade * 1000.0));
        rightSpeed = constrain(rightSpeed, -200, 200);
        leftSpeed = 200;
    }

    abot(11, 10).servoSpeed(leftSpeed, -rightSpeed);
}
```

제 6 장 적외선 센서를 이용한 자율주행 로봇

6.1 장애물 탐지를 위한 적외선 센서 신호처리
6.2 적외선 센서 신호를 로봇에 통합하기
6.3 적외선 센서로 irDistance 메소드 사용하기
6.4 적외선 센서를 이용한 주변물체 스캔하기
6.5 적외선 센서로 스캔해서 가까운 물체 찾기



적외선(Infrared 또는 IR) 센서는 학생들이 아두이노로봇에 센서를 연결하고 출력 데이터를 읽는 방법을 익히는 훌륭한 소재입니다. 특히 적외선 센서는 앞장에서 설명한대로 디지털 및 아날로그 신호를 모두 생성할 수 있어서, 다양한 실습경험을 가능하게 합니다.

이 장에서는 적외선 송신과 적외선 수신 방식으로 감지 신호를 다루는 방법을 소개합니다. 이 방식으로 로봇 전방의 물체를 감지하는 원리도 소개합니다. IR 센서를 사용한 프로젝트로 로봇 공학 및 자동화와 같은 분야의 핵심 개념인 '로봇이 환경과 상호 작용하는 방법'을 이해하고 배우기 바랍니다.

6.1 장애물 탐지를 위한 적외선 센서 신호처리

그림6.1에서 소개한 실습용 적외선센서는 송신용 LED와 수신용 receiver 부품으로 구성됩니다. 앞장에서 사용했던 포토트랜지스터 부품과는 다른 형상의 적외선 LED에 하우징을 사용해서 적외선 빛의 직진성을 더 좋게 만듭니다. 적외선 센서 송수신 부품을 아두이노 핀에 연결하는 방법은 아래 회로도를 사용해서 각각 연결하기 바랍니다.

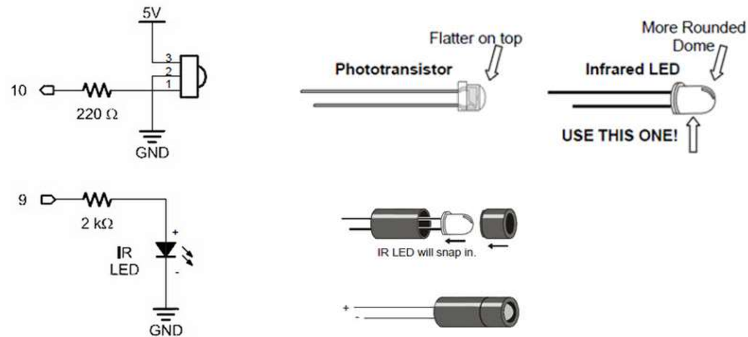


그림6.1 : 적외선 송수신센서 회로도와 부품 개략도 (출처: parallax.com)

적외선 센서 신호를 감지하기 위한 'SelfAbot'클래스의 메소드입니다. 적외선(IR) 센서를 읽기 위한 메소드 irDetect를 먼저 소개하고 설명합니다.

```
int SelfAbot::irDetect(byte irLedPin, byte irReceiverPin, int frequency) {
    _irLedPin = irLedPin;
    _irReceiverPin = irReceiverPin;

    pinMode(_irLedPin, OUTPUT);
    pinMode(_irReceiverPin, INPUT);

    tone(_irLedPin, frequency, 8);
    delay(1);
    int _lastReadValue_ir = ::digitalRead(_irReceiverPin);
    delay(1);

    return _lastReadValue_ir;
}
```

(1) 함수 선언

int SelfAbot::irDetect(byte irLedPin, byte irReceiverPin, int frequency) 이것은 SelfAbot 클래스의 멤버 함수 irDetect를 정의합니다. 함수는 세 개의 매개변수를 받습니다: irLedPin, irReceiverPin, 그리고 frequency.

(2) 변수 할당

_irLedPin = irLedPin;과 _irReceiverPin = irReceiverPin; 이 코드는 함수에 전달된 핀 번

호를 클래스의 내부 변수에 할당합니다.

(3) 핀 모드 설정
pinMode(_irLedPin, OUTPUT);과 pinMode(_irReceiverPin, INPUT); pinMode 함수는 핀을 입력 또는 출력 모드로 설정합니다. 여기서 _irLedPin은 출력(적외선 LED에 신호를 보내는데 사용)으로, _irReceiverPin은 입력(적외선 수신기로부터 신호를 읽는데 사용)으로 설정됩니다.

(4) 적외선 신호 발생
tone(_irLedPin, frequency, 8); tone 함수는 지정된 주파수로 _irLedPin에서 적외선 신호를 발생시킵니다. 이 신호는 일정 시간(여기서는 8) 동안 지속됩니다.

(5) 센서 읽기 및 반환
int _lastReadValue_ir = ::digitalRead(_irReceiverPin);
digitalRead 함수는 _irReceiverPin에서 디지털 신호를 읽어 _lastReadValue_ir 변수에 저장합니다. 이것은 IR 수신기가 감지한 값을 나타냅니다.
return _lastReadValue_ir; 반환을 위해 _lastReadValue_ir 변수의 값을 반환합니다.

irDetect() 메소드를 사용하려면 아래 그림을 참고해서 센서부품을 설치하기 바랍니다.

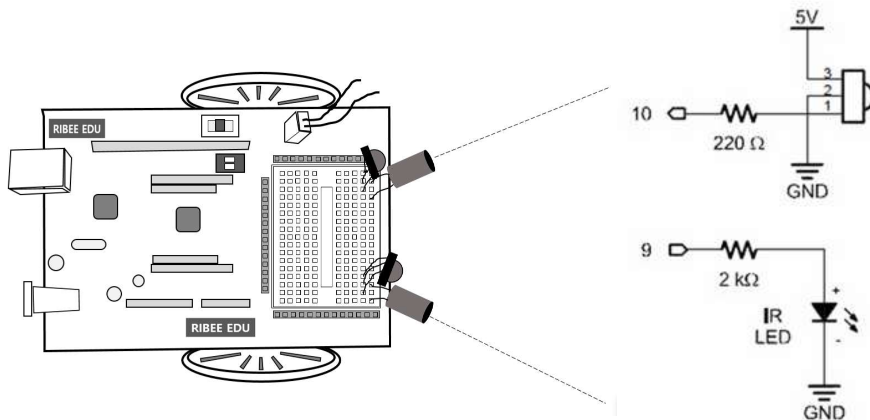


그림6.2 : 자율주행 아두이노로봇을 위한 센서 탐색방향과 설치 회로도

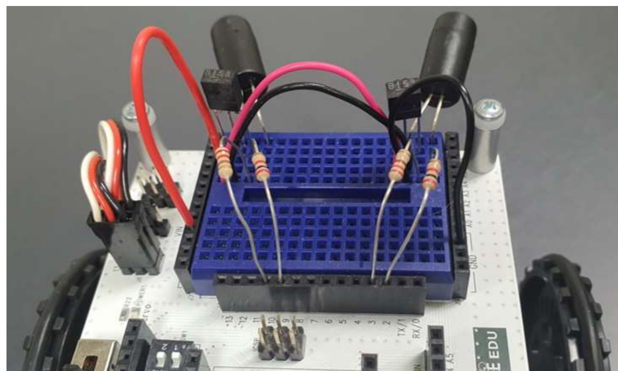


그림6.3 : 적외선 센서를 장착한 아두이노로봇

그림6.2의 아두이노로봇 적외선 센서 사용법은 좌측 전방과 우측 전방의 물체감지를 각각 독립적 신호로 감지하고 처리하는 방식입니다. 센서의 지향방향이 중심에서 더 좌측으로 그리고 더 우측으로 향하도록 설치하는 것이 특징입니다.

다음 예제 스케치는 'SelfAbot' 클래스의 적외선 감지용 메소드를 사용해서 로봇에 장착된 적외선 센서신호를 처리하는 코드입니다.

이런 센서 사용방식의 실습은 C언어 교육용으로 배포한 '아두이노로봇 가지고 놀기"온라인 실습교재'에서 설명한 내용과 동일합니다. C++언어로 작성된 'SelfAbot'을 사용해서 동일한 실습효과를 시도해보기 바랍니다.

Ex6.1_sensor_irdetect.ino 스케치를 업로드하세요.

```
#include "SelfAbot.h"
SelfAbot abot;

void setup() {
	Serial.begin(9600);
	abot.setup();
}

void loop() {
	int irLeft = abot.irDetect(9, 10, 38000); // 왼쪽 IR 센서 데이터 읽기
	int irRight = abot.irDetect(2, 3, 38000); // 오른쪽 IR 센서 데이터 읽기
	Serial.print("Left IR: ");
	Serial.print(irLeft);
	Serial.print(" || Right IR: ");
	Serial.println(irRight);
	delay(10);
}
```

이 코드를 실행하면, 왼쪽 적외선 센서와 오른쪽 적외선 센서의 값을 화면에 출력할 수 있습니다.

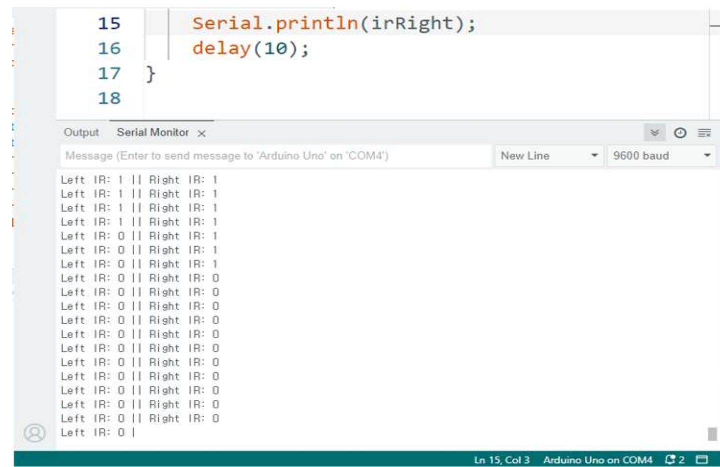


그림6.4 좌측과 우측 적외선 송수신센서의 디지털 출력 화면

센서 전방에 장애물이 있을 때와 없을 때 센서신호 출력이 어떻게 변화하는지 살펴보세요. 여기까지 코드가 잘 동작한다면, 적외선 센서신호와 연동해서 아두이노로봇의 양쪽 바퀴 동작에 연결해서 자율주행을 구현할 수 있습니다.

그림6.4 시리얼 출력화면은 전방에 물체가 감지되지 않으면 '1'을 출력하고, 물체를 감지하면 '0'을 출력합니다. 센서의 측정감도 이내에서는 물체를 인식하고, 그렇지 않으면 물체를 인식하지 못하는 단순한 물체감지 원리입니다. 센서가 전방의 물체를 감지하는 거리를 멀리까지 늘이거나, 또는 더 가깝게 줄이려면 그림6.2의 센서회로 저항 크기를 바꾸면 됩니다.

6.2 적외선 센서 신호를 로봇에 통합하기

적외선 센서로 전방의 장애물을 인식하고, 물체를 회피해서 주행하는 자율주행 코드의 예제입니다. 아래 제시된 코드는 양쪽 적외선 센서에서 물체가 감지되는 신호 '0' 이 감지되면, 로봇은 후진합니다. 그리고 왼쪽 적외선 센서에서만 물체가 감지되는 경우는 로봇이 우회전 하고, 오른 쪽 적외선 센서에서만 물체가 감지되면 로봇이 좌회전합니다. 물론 양쪽 적외선 센서 모두에서 물체가 감지되지 않으면 직진합니다. 이런 원리가 코드에 적용된 예제를 소개합니다.

적외선(IR) 센서 판독값을 기반으로 로봇의 움직임을 프로그래밍하는 두 가지 접근 방식인 'if-else' 문을 사용하는 것과 'switch' 문을 사용하는 방법을 비교해봅시다. 먼저 if-else 문을 사용하여 코드를 설명한 다음 switch 문을 사용하여 동일한 논리를 구현하는 방법을 보여 드리겠습니다.

```
int irLeft = abot.irDetect(9, 10, 38000); // 왼쪽 IR 센서 데이터 읽기
int irRight = abot.irDetect(2, 3, 38000); // 오른쪽 IR 센서 데이터 읽기

if ((irLeft == 0) && (irRight == 0)) {          // no obstacle
    abot(13, 12).servoSpeed(-100, -100);
    delay(20);
} else if ((irLeft == 0) && (irRight == 1)) {  // left detect
    abot(13, 12).servoSpeed(-100, 100);
    delay(1000);
    abot(13, 12).servoSpeed(100, 0);
    delay(400);
} else if ((irLeft == 1) && (irRight == 0)) {   // right detect
    abot(13, 12).servoSpeed(-100, 100);
    delay(1000);
    abot(13, 12).servoSpeed(0, -100);
    delay(400);
} else {                                        // both detect
    abot(13, 12).servoSpeed(-100, 100);
    delay(1000);
    abot(13, 12).servoSpeed(100, 0);
    delay(800);
}
```

if-else 구조 동작 방식은 각 조건을 순서대로 확인합니다. 로봇의 움직임은 'irLeft'와 'irRight' 센서 값의 조합에 의해 결정됩니다. 이 방법의 장점은 초보자가 이해하기 쉽고 간단하며, 각 센서 값의 조합논리를 명확하게 보여줍니다. 만약, 조건 수가 증가하면 'if-else' 문이 길어지고 관리하기 더 어려워질 수 있습니다.
동일한 로봇 동작 제어효과를 가지는 'switch' 문으로 변경해 보겠습니다.

if 조건문을 'switch' 문으로 바꾸려면 추가 작업이 필요합니다. 먼저 combinedState 변수를 사용해서 irLeft와 irRight의 2바이트 단일 값으로 결합합니다. 그런 다음 switch 문은 이 결합된 값을 기반으로 작업을 선택합니다.

Ex6.2_irdetect_navigation_abot_switch.ino 스케치를 업로드하세요.

```
#include "SelfAbot.h"
#define INVALID_PIN 255;

const byte servoLeftPin = INVALID_PIN;
const byte servoRightPin = INVALID_PIN;

SelfAbot abot(servoLeftPin, servoRightPin);

void setup() {
    //Serial.begin(9600);
    abot.setup();
    abot.servoAttachPins(13, 12);
}

void loop() {
    int irLeft = abot.irDetect(9, 10, 38000); // 왼쪽 IR 센서 데이터 읽기
    int irRight = abot.irDetect(2, 3, 38000); // 오른쪽 IR 센서 데이터 읽기
    int combinedState = (irLeft << 1) | irRight;

    switch (combinedState) {
        case 0: // Both sensors detect no obstacle
            abot(13, 12).servoSpeed(100, -100);
            break;
        case 1: // irLeft = 0, irRight = 1  left obstacle
            abot(13, 12).servoSpeed(-100, 100);
            delay(1000);
            abot(13, 12).servoSpeed(100, 0);
            delay(400);
            break;
        case 2: // irLeft = 1, irRight = 0  right obstacle
            abot(13, 12).servoSpeed(-100, 100);
            delay(1000);
            abot(13, 12).servoSpeed(0, -100);
            delay(400);
            break;
        case 3: // Both sensors detect an obstacle
            abot(13, 12).servoSpeed(-100, 100);
            delay(1000);
            abot(13, 12).servoSpeed(100, 0);
            delay(800);
            break;
    }
    delay(20);
}
```

이 코드에서 combinedState는 irLeft와 irRight의 상태를 모두 나타내는 값입니다. irLeft 값을 왼쪽으로 1비트 이동한 다음 irRight와 비트별 OR을 수행하여 생성됩니다. 이런 방식으로 'irLeft' 및 'irRight' 값(00, 01, 10, 11)의 가능한 각 조합은 고유한 'combinedState' 값(0, 1, 2, 3)으로 표시됩니다.

이 방식의 장점은 switch 문이 특히 조건이 많은 경우 더 간결하고 읽기 쉬울 수 있습니다. 또한 일반적으로 실행이 더 효율적입니다. 그렇지만, 초보자의 경우 combinedState 계산 방법을 이해하는 것이 다소 복잡할 수 있습니다.

초보자를 위한 설명

어떤 방법을 선택할 것인가? 두 가지 방법 모두 유효하지만 선택은 상황과 자신에게 편한 방식에 따라 달라집니다. 'if-else'는 더 간단한 반면, 'switch'는 여러 조건을 처리하는데 더 효율적이고 깔끔합니다.

논리적 이해를 추가하면, 방법에 관계없이 센서 판독값과 해당 작업 이면의 논리를 이해하는 것이 중요합니다. 두 가지 방법 모두 연습이 필요하고, 프로젝트에서 두 가지 방법을 모두 구현하여 다양한 상황에서 어떤 방법을 선호하는지 확인하는 것이 중요합니다. 코딩을 처음 시작할 때는 간단한 논리로 시작하고 구문과 개념에 익숙해짐에 따라 점차적으로 복잡성을 추가하세요. 프로그래밍을 배우는 가장 좋은 방법은 직접 해보는 것임을 기억하세요. 두 가지 방법을 모두 실험해보고 다양한 시나리오에서 어떤 방법이 가장 적합한지 확인하세요.

6.3 적외선 센서로 irDistance 메소드 사용하기

실습에서 사용할 적외선 센서의 주파수별 동작특성을 그림6.4로 살펴보면, 주파수 38000 Hz에서 가장 민감한 감도를 나타냅니다. 그리고 38000 Hz에서 더 높은 주파수 또는 더 낮은 주파수일 때 감도가 거의 선형적으로 감소하는 특성을 나타냅니다. 그림6.5의 주파수에 대한 감도 특성을 살펴보면, 38000 ~ 42000 Hz 범위에서 선형적 특성이 나타나고 이런 선형성을 이용해서 물체와의 거리를 측정하는 용도로 쉽게 변환할 수 있음을 나타냅니다.

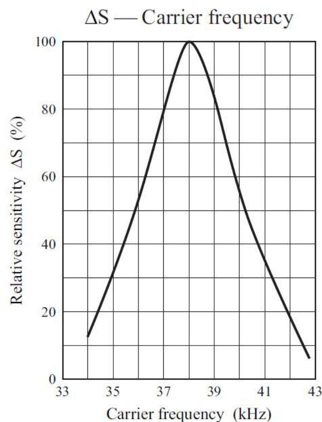


그림6.5 : 적외선 센서의 주파수별 특성 그래프 (파나소닉 PNA4602M)

주파수 스캔 방법으로 물체와의 거리를 측정하는 원리는 물체와의 거리가 멀수록 38000 Hz 조건에서만 물체가 감지되고, 물체와의 거리가 가까울수록 42000 Hz 주파수까지 물체를 감지할 수 있습니다. 아래 소개할 irDistance 메소드는 주파수를 스캔해서 각 주파수별 감지신호를 합산하는 방법으로 물체와의 상대적 거리를 차별화할 수 있습니다.

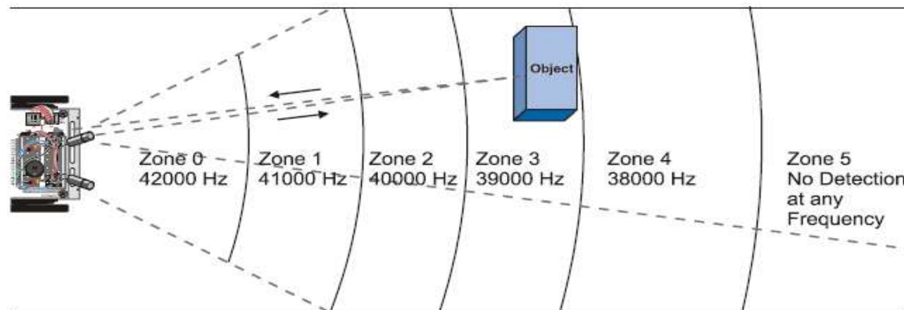


그림6.6 : 주파수별 거리측정을 위한 irDistance() 원리 (출처: parallax.com)

'SelfAbot'클래스의 irDistance 메소드는 물체와의 거리를 측정할 수 있습니다. 앞에서 소개한 irDetect 메소드를 사용해서 물체와의 거리를 측정하도록 하는 논리가 추가되었습니다.

```
int SelfAbot::irDistance(byte irLedPin, byte irReceivePin) {
    int distance = 0;
    for(long f = 38000; f <= 42000; f += 500) {
        distance += irDetect(irLedPin, irReceivePin, f);
    }
    return distance;
}
```

(1) 함수 선언
int SelfAbot::irDistance(byte irLedPin, byte irReceivePin): 'SelfAbot'클래스의 메소드입니다. 이 메서드는 'irLedPin'과 'irReceivePin'이라는 두 가지 매개변수를 사용합니다. IR LED와 수신기가 마이크로 컨트롤러에 연결된 핀 번호입니다. 이 메서드는 측정된 거리를 나타내는 정수 값을 반환합니다.

(2) 거리 변수 초기화
int distance = 0; 이 라인은 distance 변수를 ‘0’으로 초기화합니다. 이 변수는 다양한 주파수에서 측정된 거리 값을 누적하는데 사용됩니다.

(3) 거리 측정을 위한 주파수 스캔
for(long f = 38000; f <= 42000; f += 500) { ... } 이것은 주파수 범위를 반복하는 for 루프입니다. 38000Hz에서 시작하여 매번 500Hz씩 증가하고 42000Hz에서 멈춥니다. 이 주파수는 IR 신호를 전송하고 반사를 측정하는데 사용됩니다. 주파수의 스캔 범위는 사용중인 적외선

센서의 감도를 표시하는 부품 특성에 의존합니다.

(4) 거리 측정
distance += irDetect(irLedPin, irReceivePin, f); 반복문 내부에서 irDetect 함수가 현재 주파수 f로 호출됩니다. 'irDetect' 방법은 'irLedPin'을 사용하여 IR 신호를 보내고, 'irReceivePin'을 사용하여 이를 수신하고, 물체와의 거리를 측정하는데, 출력이 '1'일 때를 모두 합산합니다. 그런 다음 이 값은 distance 변수에 기록됩니다.

(5) 총 측정거리 반환
반복문이 완료된 후 누적된 총 거리가 반환됩니다. 이 값은 서로 다른 모든 주파수로부터 읽은 값들의 결과를 나타냅니다. 레이더의 기본 개념이라고 할 수도 있습니다.

적외선센서로 단순히 물체가 있는지 없는지를 판단하는 디지털출력 신호를 처리하는데 멈추지 않고, 전방 물체와의 거리를 측정하는 irDistance()메소드를 사용하는 스케치 예제를 소개합니다. 두 개의 센서를 이용해서 하나의 물체를 감지하려면, 아래 그림과 같이 가상의 물체에 초점을 맞추도록 센서의 지향방향을 맞추기 바랍니다.

물체를 감지하기 시작하는 가장 가까운 거리부터 가장 멀리 감지하는 거리 사이의 공간영역을 아래 스케치 실행결과를 살펴보면서 파악할 수 있습니다. 우리는 예상되는 감지영역내에 가상의 물체를 놓아두고, 로봇이 스캔해서 해당 물체를 잘 인지하는지 확인해볼 수 있습니다.

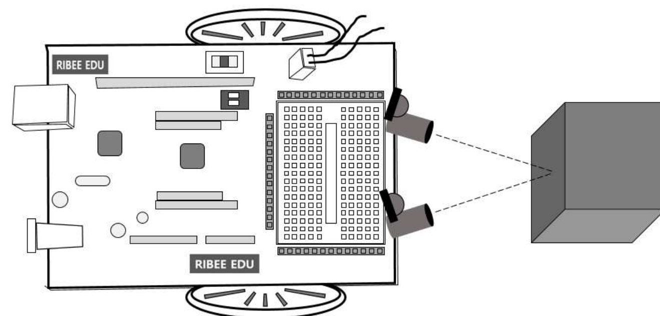


그림6.7 : 적외선센서로 전방 물체와의 거리를 스캔하기 위한 개략도

<u>초보자를 위한 개념 이해</u>
'irDistance' 기능은 IR 빛이 다양한 주파수에서 어떻게 반응하는지 관찰하여 거리를 측정할 수 있는 도구와 같습니다.

for 루프는 포괄적인 이해를 얻기 위해 매번 약간 다른 설정(다른 빈도)을 사용하여 일련의 실험을 진행하는 것과 같습니다. 이러한 측정값을 모두 합산하면(distance += irDetect(...)) 전체 거리 측정이 더욱 안정적이고 정확해집니다. 마지막으로 함수는 이 총 거리 값을 반환합니다. 이 방법은 측정 거리에 따라 서로 다른 주파수에서 IR 신호 응답의 변화를 사용하는 간

단하면서도 효과적인 방법으로, 로봇 공학에서 장애물을 감지하거나 탐색 목적으로 특히 유용합니다. 'SelfAbot'클래스의 irDistance 메소드로 물체와의 거리를 측정하는 아두이노 코드로 테스트해보세요.

Ex6.3_irdistance_serialprint.ino 스케치를 업로드하세요.

```
#include "SelfAbot.h"
SelfAbot abot;

void setup() {
	Serial.begin(9600);
	abot.setup();
}

void loop() {
	byte irLeftled = 9;
	byte irLeftreceiver = 10;
	byte irRightled = 2;
	byte irRightreceiver = 3;

	int irDistLeft = abot.irDistance(irLeftled, irLeftreceiver);
	int irDistRight = abot.irDistance(irRightled, irRightreceiver);

	Serial.print("Left IR: ");
	Serial.print(irDistLeft);
	Serial.print(" || Right IR: ");
	Serial.println(irDistRight);
	delay(50);
}
```

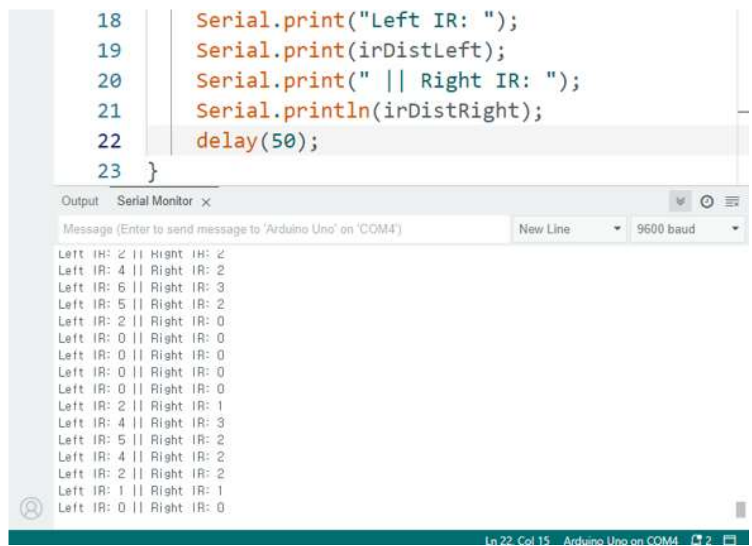


그림6.8 : 적외선센서로 거리측정 시리얼 출력

그림6.8에서 출력값이 ‘9’일 때는 가장 멀리 있는 물체를 감지할 수 있는 주파수 조건 38,000 Hz 에서도 감지되지 않는 경우이고, ‘0’ 이면 가장 측정감도가 낮은 주파수인 42,000 Hz 에서도 물체를 감지할 만큼 가까이 놓여있는 조건에서 감지한다는 것을 의미합니다.

6.4 적외선 센서를 이용한 주변물체 스캔하기

우리는 두 개의 적외선 센서를 사용해서 로봇 전방 물체와의 거리를 측정할 것입니다. 이것은 마치 적외선 센서를 사용해서 레이더를 구현하는 원리와 유사합니다. 그리고 복수의 센서를 사용하는 만큼 다양한 고려사항이 있습니다. 특히 센서의 종류가 다양하고 센서의 개수도 많은 경우 더욱 그렇습니다. 센서 데이터를 다룰 때 고려할 수 있는 몇가지를 아래에 먼저 소개합니다. 방법 선택은 주로 애플리케이션의 특정 요구 사항, 데이터 특성 및 환경 조건에 따라 달라집니다.

(참고) 센서 데이터를 통합하는 여러 가지 방법들 ------------------------------------

(1) 가중평균
설명: 신뢰성이나 정확성을 기준으로 각 센서의 판독값에 서로 다른 가중치를 할당합니다. 예를 들어, 하나의 센서가 특정 거리에서 더 정확한 것으로 알려진 경우 해당 센서의 판독값에 더 많은 가중치가 부여될 수 있습니다.
애플리케이션: 센서의 성능이 다양한 조건에 따라 달라질 때 유용합니다.

(2) 중앙값 필터링
설명: 평균을 구하는 대신 일련의 판독 값 중앙값을 취합니다. 이 방법은 크게 벗어난 값이나 산발적인 잘못된 판독값의 영향을 줄이는데 특히 효과적입니다.
애플리케이션: 센서 소음 수준이 높거나 갑작스럽고 잘못된 판독이 일반적인 환경에 이상적입니다.

(3) 모드 필터링
설명: 일련의 판독값 중에서 가장 자주 발생하는 값을 사용합니다. 이 방법은 안정적인 측정을 나타내는 공통 반복 값을 기대할 때 유용할 수 있습니다.
애플리케이션: 센서가 특정 거리 판독값을 반복적으로 반환해야 하는 시나리오에 유용합니다.

(4) 칼만 필터링
설명: 통계적 잡음과 기타 부정확성을 포함하여 시간이 지남에 따라 관찰된 일련의 측정값을 사용하고 알 수 없는 변수의 추정치를 생성하는 고급 방법입니다.
애플리케이션: 추적 및 내비게이션 시스템과 같이 실시간 데이터 처리가 필요한 애플리케이션에 널리 사용됩니다.

(5) 센서 융합 알고리즘
설명: 여러 센서의 데이터를 결합하여 정확도를 높이고 불확실성을 줄입니다. 이는 베이지안 네트워크, 기계 학습 모델 또는 간단한 논리 규칙을 포함한 다양한 알고리즘을 사용하여 수행할 수 있습니다.
애플리케이션: 여러 센서를 사용하여 환경의 다양한 측면을 측정하는 복잡한 시스템에 효과적입니다.

(6) 회귀 분석

설명: 회귀 모델을 센서 데이터에 맞춰 값을 예측합니다. 이 방법은 다양한 센서의 판독값 간의 관계를 식별하는데 사용할 수 있습니다.
애플리케이션: 센서 판독값이 특정 추세나 패턴을 따를 것으로 예상되는 상황에서 유용합니다.

(7) 데이터 평활화 기법
설명: 이동 평균, 지수 평활 또는 저역 통과 필터와 같은 평활화 기술을 적용하여 데이터의 노이즈 및 변동을 줄입니다.
애플리케이션: 시간이 지남에 따라 데이터가 점진적으로 변경될 것으로 예상되는 애플리케이션에 적합합니다.

(8) 의사결정 트리 또는 규칙 기반 시스템
설명: 센서 데이터를 활용하여 결정을 내리거나 분류하는 의사결정 트리 또는 규칙 기반 시스템을 구현합니다.
애플리케이션: 센서 판독값을 기반으로 결정을 내려야 하는 자동화 시스템에 유용합니다.

아두이노로봇에 설치된 2개의 적외선 센서를 함께 사용해서 전방 물체를 탐지하는 방법을 소개합니다. 두 개의 센서가 좌측과 우측 물체를 각각 감지하는 용도로 분리해서 사용되지 않고, 하나의 물체를 감지하는 용도로 사용할 것입니다. 두 개의 적외선 센서 데이터는 산술평균을 사용할 수 있습니다.

두 개의 센서가 하나의 센서처럼 동작하려면, 다음 몇가지 사항을 고려해야 합니다.
(1) 시야각: 두 센서의 시야각이 비슷하고 동일한 물체를 측정하는 경우 평균화는 개별 센서 판독값의 무작위 오류나 노이즈를 완화하는데 도움이 될 수 있습니다. 대상까지의 거리와 센서가 장착된 각도에 따라 효율성이 달라집니다. 센서는 판독 값이 원하는 지점에서 가장 정확하도록 배치되어야 합니다.

(2) 센서 정렬: 센서는 둘 다 동일한 영역이나 물체를 대상으로 정렬되어야 합니다. 정렬이 잘못되면 판독값이 부정확해질 수 있습니다. 효과적인 산술평균을 위해서는 센서의 시야가 관심 지점에 수렴되도록 센서를 정렬해야 합니다. 이를 위해서는 센서의 정확한 배치와 방향이 필요합니다.

(3) 일관된 환경 조건: IR 센서는 주변광, 표면 반사율, 온도와 같은 요인의 영향을 받을 수 있으므로 두 센서 모두 동일한 환경 조건에서 작동해야 합니다.

(4) 중복 감지 범위: 장애물 감지와 같은 애플리케이션에서 센서에 중복 적용범위가 있는 경우 평균화를 통해 보다 신뢰할 수 있는 중앙 거리 추정치를 제공할 수 있습니다.

로봇에 고정된 센서로 전방 물체를 탐색하려면, 로봇을 제자리 회전시켜야 합니다.

아두이노로봇이 제자리에서 회전하면서 적외선 센서를 사용해서 전방의 물체를 감지하는 레이더 기능을 구현하려면, 먼저 로봇을 제자리에서 회전하는 메소드를 만들어야 합니다. 아래 코드는 360도 회전하는 제자리회전 메소드입니다.

```
void rotateRobot(int angle) {
    float factor = 10.0;
    int duration = (int)(angle * factor);

    abot.servoSpeed(100, 100); // Adjust speed as needed
    delay(duration);

    abot.servoSpeed(0, 0);
}
```

제자리 회전할 때 로봇의 동작시간과 제자리 회전하는 각도를 일치시키려면, 실험적인 계수(여기서는 factor) 값을 찾아야 합니다. 각자의 로봇마다 조금씩 계수 값이 다를 수 있습니다. 계수 값을 찾는 요령은 다음과 같습니다.

(1) 테스트 설정: 로봇을 360도 회전시키고 걸리는 시간을 측정합니다. 로봇이 360도 회전하는 데 2000밀리초(2초)가 걸린다면 인자는 2000 / 360 ≒ 5.56이 됩니다. 따라서 1도당 로봇은 약 5.56밀리초 동안 회전해야 합니다. 코드에서 적용한 factor 값 '10.0'을 실험적으로 결정된 '5.56'으로 대체합니다.

또는 로봇이 360도만 회전하는 최적의 factor 값을 반복 테스트해보는 것입니다. 예를 들어 로봇의 servoSpeed 속도 값이 100 일 때 factor 가 5 정도의 값이라면, 40 일 때 factor 는 10 정도의 값을 가질 수 있습니다.

(2) 조정 및 측정: 로봇이 최대한 360도에 가깝게 회전할 때까지 회전 시간을 변경합니다. 이 회전에 걸린 시간을 기록해 두십시오.

(3) 계수 계산 및 적용: 밀리초 단위의 시간을 360으로 나누어 계수(회전 각도당 시간)를 구합니다.

로봇이 제자리회전에서 factor 값을 보정하는 스케치 코드를 소개합니다. 아래 예제를 업로드해서 최적의 보정을 먼저 실시하기 바랍니다. 이 코드의 결과는 로봇을 제자리 회전할 때, 회전하고자하는 각도를 매개변수로 사용하면 원하는 각도만큼 로봇이 제자리 회전하는 결과를 만듭니다.

로봇이 회전하는 속도는 일정하게 유지하면서 factor 값을 보정해야합니다. 회전속도가 변경하면, 변경하기 이전에 찾은 최적의 factor 값은 더 이상 유효하지 않습니다. AA배터리로 보정을 실시할 때, servoSpeed(40, 40)이면 factor 값은 10 정도 값을 갖습니다.

Ex6.4_rotateRobot_in_place.ino 스케치를 업로드하고 제자리회전하는 보정을 실시하세요.

```
#include "SelfAbot.h"
#define INVALID_PIN 255;
const byte servoLeftPin = INVALID_PIN;
const byte servoRightPin = INVALID_PIN;

SelfAbot abot(servoLeftPin, servoRightPin);
float factor = 10.5; // Factor to convert angle to duration

void setup() {
     abot.setup();
     abot.servoAttachPins(13, 12);
     rotateRobot(360);
}

void loop() {
  // put your main code here, to run repeatedly:
}

// The rotateRobot function
void rotateRobot(int angle) {
     int duration = (int)(angle * factor);
     abot.servoSpeed(40, 40); // Adjust speed as needed
     delay(duration);
     abot.servoSpeed(0, 0); // Stop the robot after rotating
}
```

로봇의 제자리회전 factor 값 보정이 완료되었다면, 아래 코드를 아두이노에 업로드해서 실제로 주변의 물체를 얼마나 정확하게 감지하는지 테스트해봅시다.

Ex6.5_irdistance_point_rotation.ino 스케치를 업로드하세요.

```
#include "SelfAbot.h"
#define INVALID_PIN 255;
const byte servoLeftPin = INVALID_PIN;
const byte servoRightPin = INVALID_PIN;

SelfAbot abot(servoLeftPin, servoRightPin);
float factor = 10.5; // Factor to convert angle to duration

void setup() {
     Serial.begin(9600);
     abot.setup();
     abot.servoAttachPins(13, 12);

     byte irLeftled = 9;
     byte irLeftreceiver = 10;
     byte irRightled = 2;
     byte irRightreceiver = 3;

     for (int angle = 0; angle < 180; angle += 10){
          rotateRobot(10);

          int irDistLeft = abot.irDistance(irLeftled, irLeftreceiver);
          int irDistRight = abot.irDistance(irRightled, irRightreceiver);
          int averageDistance = (irDistLeft + irDistRight) / 2;

          Serial.print("Angle: ");
          Serial.print(angle);
```

```
            Serial.print(" | Average Distance: ");
            Serial.println(averageDistance);
            delay(10);
      }
}

void loop() {

}

// The rotateRobot function

void rotateRobot(int angle) {
      int duration = (int)(angle * factor);
      abot.servoSpeed(40, 40); // Adjust speed as needed
      delay(duration);
      abot.servoSpeed(0, 0); // Stop the robot after rotating
}
```

소개한 코드는 정지상태에서 로봇의 주변을 살피는 탐색모드로 활용할 수 있습니다. 0도에서 180도까지 10도씩 제자리 회전하면서 물체를 탐지하고, loop회로에 의해 반복 실행됩니다. 데이터를 무선으로 전송하지 않기 때문에 USB케이블을 사용해야 하는 어려움이 있지만, 전방 물체를 탐색하는 원리를 익히는데 도움이 되었으면 합니다.

적외선 센서를 사용해서 주변을 둘러싸고 있는 물체들의 원근에 대한 판단을 어느 정도까지 구현할 수 있는지 시도해보고 스스로 판단해보세요. 주변에 큰 물체 작은물체 그리고 멀리와 가까이에 다양한 종류의 물체를 놓아두고 로봇이 식별하는 물체를 직접 느껴보세요. 이런 작업을 통해서 더 훌륭한 로봇의 동작을 만들 수 있습니다.

6.5 적외선 센서로 스캔해서 가까운 물체 찾기

지금까지 아두이노로봇의 동작은 정속주행 한가지만을 사용했습니다. 그리고 두 개의 적외선 센서를 레이더 탐색모드로 사용했던 것처럼 산술 평균 방식으로 데이터를 처리할 것입니다. 이런 조건에서 아두이노로봇이 전진하다가 전방에서 어떤 물체를 감지하게 된다면, 그 순간 로봇이 진행을 멈추고 레이더 탐색모드로 전환한 후 주변의 물체가 가장 멀리 있거나 없는 방향을 탐색해서 방향을 전환하고 주행하도록 만들 수 있습니다.

아래 소개하는 예제는 0 ~ 180도 각도사이에 놓여진 전방물체를 센서가 감지하고, 가장 가까운 물체방향을 로봇이 찾는 것입니다. 또는 로봇을 사용하여 주변 환경을 스캔하고, 가장 가까운 물체를 감지하는 방향으로 로봇을 회전시키는 프로그램입니다. 코드는 크게 로봇 설정, 환경 스캔, 로봇 회전 제어 등 몇 가지 주요 부분으로 구성되어 있습니다.

예제 코드에서 로봇 설정과 센서 초기화관련 설명은 생략하고, 센서로 물체를 찾는 원리와 로봇 회전에 대해 설명합니다.
(1) 물체를 감지하면 센서의 값은 작아지고 최소값은 ‘0’이어야 합니다. 그래서 distance 변수

를 최대값인 10으로 초기화하고, directionOfdistance 변수는 0으로 초기화합니다.

(2) 로봇을 제자리회전 방법으로 0˚에서 180˚까지 10˚씩 증가하며 각각의 방향에 대해 적외선 센서를 사용하여 거리를 측정합니다. 회전각도를 더 촘촘하게 하고 싶다면 반복문의 증가 값과 로봇을 회전시키는 함수 rotateRobot()매개변수를 더 작은 값으로 반영하면 됩니다.

이 코드는 각 방향에 대해 얻은 거리의 평균을 구하고, 이 평균 거리가 현재까지 감지된 최소 거리보다 작으면, 해당 거리와 방향을 변수에 저장합니다.

(3) 모든 스캔이 완료된 후에는 setTravelDirection 함수를 사용하여 로봇을 가장 가까운 물체가 감지된 방향으로 회전시킵니다.
로봇을 제자리회전시키는 함수는 rotateRobot() 함수이고, 양의 각도로 화전하거나 음의 각도로 회전시킬 수 있습니다. 그리고 setTravelDirection 함수는 로봇을 가장 가까운 물체가 감지된 방향으로 회전시키는 함수입니다.

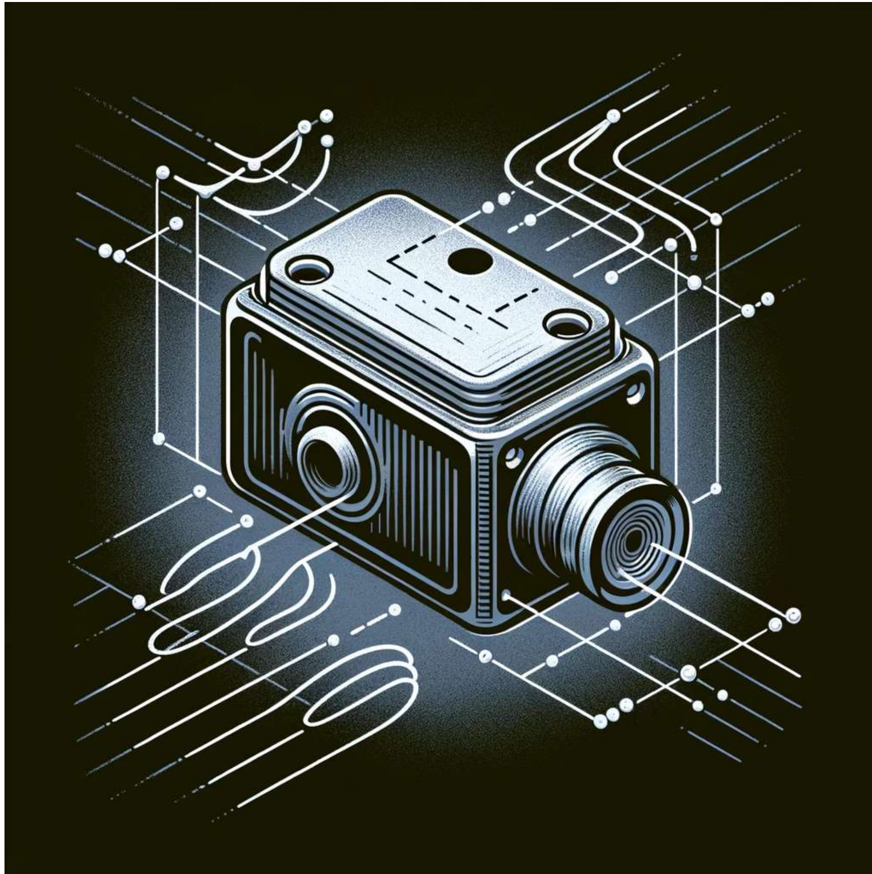
Ex6.6_irscan_to_findObject.ino 스케치를 업로드하세요.

```
#include "SelfAbot.h"
#define INVALID_PIN 255;
const byte servoLeftPin = INVALID_PIN;
const byte servoRightPin = INVALID_PIN;

SelfAbot abot(servoLeftPin, servoRightPin);
float factor = 10.5;

void setup() {
    abot.setup();
    abot.servoAttachPins(13, 12);

    byte irLeftled = 9;
    byte irLeftreceiver = 10;
    byte irRightled = 2;
    byte irRightreceiver = 3;

    int distance = 10;
    int directionOfdistance = 0;

    for (int angle = 0; angle <= 180; angle += 10) {
        rotateRobot(10);

        int irDistLeft = abot.irDistance(irLeftled, irLeftreceiver);
        int irDistRight = abot.irDistance(irRightled, irRightreceiver);
        int averageDistance = (irDistLeft + irDistRight) / 2;

        if (averageDistance < distance) {
            distance = averageDistance;
            directionOfdistance = angle;
        }

        delay(100);
    }
    // Set robot's direction to the direction of max distance
    setTravelDirection(directionOfdistance);
}
```

```
void loop() {
}

void rotateRobot(int angle) {
  if (angle >= 0) {
    int duration = (int)(angle * factor);
    abot.servoSpeed(40, 40);
    delay(duration);
    abot.servoSpeed(0, 0);
  } else {    // reverse rotate
    int duration = (int)(abs(angle) * factor);
    abot.servoSpeed(-40, -40);
    delay(duration);
    abot.servoSpeed(0, 0);
  }
}

void setTravelDirection(int angle) {
    // Rotate the robot to the desired direction
    rotateRobot(angle-180);
}
```

제 7 장 초음파센서를 장착한 아두이노로봇

7.1 초음파센서 신호처리하기
7.2 전방물체 레이더 화면 표시하기
7.3 초음파센서 탐색 로봇으로 주행



이 장에서는 아두이노의 표준함수인 pulseIn()을 사용하는 것이 목표입니다. ‘SelfAbot’ 클래스에서는 pulseIn() 메소드 이외에 pulseOut() 메소드를 추가로 정의하고, pulseCount() 메소드 역시 정의해두었습니다.

앞 장에서 배운 포토트랜지스터 광센서나 적외선 송수신센서를 사용하는 방식과 조금 다른 방법으로 초음파 센서를 동작시키는 방법을 설명합니다.

광센서와 초음파센서 모두 일반적으로 장애물을 감지하고, 물체와의 거리를 측정하기 위한 용도로 많이 사용합니다. ‘초음파’라는 음원을 이용하는 초음파 센서와 ‘적외선’이라는 빛의 파장을 이용하는 적외선 센서를 다음 표7.1과 표7.2에서 특징들을 비교합니다. 간단히 두 개의 센서 유형을 비교해보고 초음파센서 신호를 다뤄보겠습니다.

표7.1 초음파센서와 적외선센서의 여러 가지 특징 비교

	초음파 센서	적외선 센서
원리	초음파 음파(인간이 들을 수 있는 것보다 높은 주파수)를 방출하고, 물체에 부딪힌 후 에코가 되돌아오는 데 걸리는 시간을 측정합니다.	적외선을 사용하여 물체를 감지합니다. 일부는 반사된 빛의 강도를 측정하는 반면 다른 것(예: IR 거리 센서)은 반사된 빛의 각도나 시간을 측정하여 거리를 결정합니다.
구성요소	일반적으로 송신기(초음파 전송)와 수신기(에코 수신)로 구성됩니다.	IR 송신기(일반적으로 LED) 및 IR 수신기(예: 포토다이오드)를 포함합니다.
측정범위	일반적으로 IR 센서에 비해 더 먼 거리를 측정할 수 있습니다. 일반적인 범위는 수 센티미터에서 수 미터입니다. 음파는 IR광파보다 더 멀리 이동할 수 있습니다.	일반적으로 더 짧은 거리(종종 최대 몇 미터)로 제한되며 물체의 반사율 및 환경 조명 조건에 따라 크게 달라집니다. IR광은 음파보다 거리에 따라 더 빠르게 감소합니다.
정확도	매우 정확할 수 있지만 거리가 멀어질수록 정확도가 떨어집니다. 정확도는 물체의 표면 재질과 각도에도 영향을 받습니다.	일반적으로 짧은 거리에서는 정확하지만 물체의 색상과 반사율에 따라 영향을 받을 수 있습니다. 색상이 어두울수록 IR 광선을 더 많이 흡수하여 판독 정확도가 떨어집니다.
환경요인	조명 조건의 영향을 덜 받지만 온도와 바람의 영향을 받을 수 있습니다. 부드러운 재료나 각진 표면은 음파를 흡수하여 판독값이 부정확해질 수 있습니다.	주변광, 특히 햇빛의 영향을 받아 IR 신호를 방해할 수 있습니다. 먼지나 안개가 IR 빛을 산란시킬 수 있으므로 명확한 IR 경로가 필요합니다.
적용범위	장애물 회피를 위한 로봇, 탱크의 레벨 감지, 차량의 주차 센서 등 정확한 거리 측정이 필요한 응용 분야에 일반적으로 사용됩니다. 실내 및 실외 모두에 적합하지만 성능은 환경 조건에 따라 달라질 수 있습니다.	근접 감지, 라인 추적 로봇, 물체 감지/계수, TV 리모컨 및 자동 수도꼭지와 같은 장치에 널리 사용됩니다. 햇빛과 외부 광원에 민감하기 때문에 주로 실내 응용 분야나 통제된 환경에서 사용됩니다.

(참고) **초음파 신호의 항상 일정한 트리거 신호와 가변적인 반향 에코신호** --------------
초음파신호 측정의 핵심은 신호가 항상 동일하다는 것이 아니라 신호가 반환(에코)되는데 걸리는 시간이 반사되는 물체까지의 거리에 따라 다르다는 것입니다. 더 정확하게 설명하면, 트리거 핀에서 전송된 신호는 실제로 일정하며 일반적으로 빠른 HIGH 펄스입니다. 그렇지만, 펄스가 물체까지 이동하고 돌아오는데 걸리는 시간에 따라 에코 핀에서 수신되는 신호가 달라집니다. 물체가 멀수록 HIGH 펄스 지속 시간이 길어집니다.

한가지 간과하지 말아야 할 사실은 트리거 신호가 직접적인 초음파 송신 신호가 아니라, 해당 초음파부품이 트리거 신호를 생성하도록 유도하는 신호를 부품에게 보내는 것입니다. 따라서 해당 부품의 데이터시트에서 요구하는 신호를 만들어서 보내는 것이 필요합니다.

참고로 pulseOut() 메소드는 특정 핀에서 지정된 지속 시간의 디지털 펄스를 생성하도록 설계하였습니다. 초음파 센서처럼 정확한 타이밍 신호를 생성하고 특정 펄스 폭이 필요한 장치를 제어하기 위한 간단하면서도 강력한 도구입니다.

두가지 종류의 센서에 대해서, 센서 데이터의 선형성 특성과 시야각 특성을 살펴보겠습니다.

표7.2 초음파센서와 적외선센서의 선형성, 시야각 특성

	초음파 센서	적외선 센서
선형성	선형성: 일반적으로 초음파 센서는 유효 범위에 걸쳐 우수한 선형성을 갖습니다. 음파가 물체에 반사된 후 되돌아오는데 걸리는 시간은 거리에 정비례하므로 이상적인 조건에서는 선형 관계로 이어집니다. 선형성에 영향을 미치는 요인: 선형성은 대상 표면의 질감과 각도뿐만 아니라 온도, 습도와 같은 환경 요인에 의해 영향을 받을 수 있으며 이는 음속에 영향을 줄 수 있습니다.	선형성: IR 센서의 선형성은 IR 센서 유형에 따라 크게 달라질 수 있습니다. 일부 IR 거리 센서는 삼각 측량을 사용하여 작동 범위에 걸쳐 합리적인 선형 출력을 제공할 수 있습니다. 그러나 반사광의 강도 측정을 기반으로 하는 센서는 특히 근거리에서 비선형 반응을 보이는 경우가 많습니다. 선형성에 영향을 미치는 요소: 대상 물체의 반사율, 주변 조명 조건 및 입사각은 IR 센서 판독값의 선형성에 큰 영향을 미칠 수 있습니다.
시야각	확산: 초음파 센서는 일반적으로 IR 센서에 비해 빔 각도가 더 넓습니다. 즉, 더 넓은 영역을 포괄할 수 있지만 해당 영역 내 개체의 정확한 위치를 정확히 찾아내는데 정확도가 떨어집니다. 의미: 확산 범위가 넓어지면 센서가 바로 앞에 있지 않지만 더 넓은 빔 범위 내에 있는 물체를 감지할 수 있는 "오탐지"가 발생할 수 있습니다.	확산: IR 센서는 일반적으로 시야가 더 좁기 때문에 센서 바로 앞에 있는 물체를 보다 정확하게 타겟팅할 수 있습니다. 따라서 특정하고 좁은 영역에 있는 물체를 감지하는데 더 적합합니다. 의미: 좁은 확산은 축을 벗어난 물체를 감지할 가능성을 줄이지만 감지 영역도 제한하므로 전체 영역을 감지하려면 더 많은 센서 또는 센서 이동이 필요할 수 있습니다.

pulseCount() 메소드는 센서가 부착된 바퀴의 회전 수를 계산하거나 특정 시간 프레임 내에 센서를 통과하는 객체 수를 계산하는 등 지정된 시간 내의 이벤트를 계산하는데 특히 유용합니다. 바퀴의 회전수를 세기 위해 pulseCount()를 사용하는 예를 생각해 볼 수 있습니다.

(a) (b)

그림7.1 : 제조사별 초음파센서 핀 종류 (a) 3핀 (b) 4핀

7.1 초음파센서 신호처리하기

초음파센서 신호를 처리하기 위해서는 'SelfAbot'클래스의 pulseOut, pulseIn 메소드를 사용합니다. pulseOut, pulseIn 및 pulseCount 메소드를 사용하는 간단한 예제를 소개합니다. 이러한 방법은 일반적으로 초음파 센서 또는 인코더와 같은 다양한 센서 및 액추에이터에 사용됩니다. 초음파 센서는 일반적으로 펄스(pulseOut)를 방출한 다음 펄스가 반환되는데 걸리는 시간(pulseIn)을 측정하고, 측정 거리는 측정 시간 지연을 이용해서 계산합니다.

pulseIn() 메소드에 대해 더 자세히 살펴봅시다.
1) 트리거핀: 짧은 HIGH 펄스를 전송하여 측정을 시작합니다.
2) 에코핀: 물체에 부딪힌 후 HIGH 펄스를 수신합니다.

4핀 HR04 초음파센서

4핀 HR04 초음파센서를 사용해서 물체와의 거리를 측정하는 예제입니다.
초음파 센서의 트리거 핀을 arduino의 디지털 핀(예: 핀 7)에 연결합니다.
초음파 센서의 에코 핀을 다른 디지털 핀(예: 핀 8)에 연결합니다.

Ex7.1_HR04_ultrasonic_distance.ino 스케치를 아두이노에 업로드하세요.

```
#include "SelfAbot.h"

SelfAbot abot;
const byte triggerPin = 7; // Ultrasonic sensor trigger pin
const byte echoPin = 8;     // Ultrasonic sensor echo pin

void setup() {
    abot.setup();
    pinMode(triggerPin, OUTPUT); // Set the trigger pin as an output
    pinMode(echoPin, INPUT);      // Set the echo pin as an input
    Serial.begin(9600);
}

void loop() {
    abot.pulseOut(triggerPin, 10); // 10 microseconds pulse
    unsigned long duration = abot.pulseIn(echoPin, HIGH);

    // Calculate distance (in centimeters or inches)
    // Speed of sound = 34300 cm/s (in air), so time for there and back is duration
    // Distance = (speed of sound * time) / 2
    float distanceCm = duration * 0.0343 / 2;
    float distanceInch = duration * 0.0135 / 2;

    // Print the distance
    Serial.print("Distance: ");
    Serial.print(distanceCm);
    Serial.print(" cm, ");
    Serial.print(distanceInch);
    Serial.println(" inches");

    delay(1000); // Wait for a second before the next measurement
}
```

pulseOut(triggerPin, 10);은 초음파 센서를 트리거하기 위해 짧은 펄스(10마이크로초)를 보냅니다.
pulseIn(echoPin, HIGH);는 펄스가 반환되기를 기다리고 지속 시간을 측정합니다. 거리는 펄스 지속 시간을 기준으로 계산됩니다. 음파가 물체를 향해 갔다가 돌아오기 때문에 2로 나눕니다.

이 코드는 로봇이나 센서에서 펄스 관련 방법을 사용하는 기본 예를 제공합니다. 프로젝트의 특정 요구 사항에 따라 이 예제를 조정하고 확장할 수 있습니다.

3핀 Parallax 초음파센서

3핀 초음파센서를 사용해서 동일한 효과를 만들 수 있습니다. 패럴렉스사의 3핀 초음파센서는 4핀이 초음파 신호를 송신하는 핀(트리거핀)과 수신하는 핀(에코핀)을 하나의 핀으로 모두 신호 처리합니다. 그래서 4핀 방식의 코드에서 두 개의 핀 번호를 동일하게 적용하면 됩니다.

```
long duration, distanceCm;
pinMode(pingPin, OUTPUT);
digitalWrite(pingPin, LOW);
delayMicroseconds(2);
digitalWrite(pingPin, HIGH);
delayMicroseconds(5);
digitalWrite(pingPin, LOW);

pinMode(pingPin, INPUT);
duration = pulseIn(pingPin, HIGH);
distanceCm = duration * 0.0343 / 2;
```

위의 코드는 특히 Parallax 3핀 초음파 센서를 사용하여 거리를 측정하기 위한 것입니다. 아두이노 라이브러리의 표준 함수인 pulseIn() 함수를 사용하여 초음파 펄스가 물체까지 이동하고 돌아오는데 걸리는 시간(에코 시간)을 측정합니다. 'pulseIn()'의 코드와 역할을 살펴봅시다.

pinMode(pingPin, OUTPUT);: pingPin을 출력으로 설정합니다. 초음파 센서의 트리거 입력에 연결되는 핀입니다.

초음파 펄스 생성:
digitalWrite(pingPin, LOW); DelayMicroseconds(2); 깨끗한 LOW에서 HIGH 펄스를 보장합니다. LOW 상태를 사용해서 처음 라인이 낮은지 확인합니다. HIGH 펄스를 보내기 전에 핀을 LOW로 설정하면 신호가 안정화됩니다. 이는 핀의 잔류 전하 또는 노이즈로 인한 잘못된 트리거링을 방지하기 위한 디지털 신호 전송의 일반적인 관행입니다. LOW 상태는 핀을 알려진 상태로 "재설정"합니다. 'delayMicroseconds(2)'는 핀을 LOW로 설정한 후 짧은 버퍼 시간을 제공하여 HIGH 펄스가 전송될 때 명확하게 합니다. 이 짧은 지연은 핀의 이전 활동이나 노이즈와 펄스를 구별하는데 도움이 됩니다.
digitalWrite(pingPin, HIGH); DelayMicroseconds(5); 짧은 HIGH 펄스(5마이크로초)를 생성합니다. 이 펄스는 센서가 초음파를 보내도록 트리거하는 것입니다.
digitalWrite(pingPin, LOW); 핀을 LOW로 반환합니다. 초음파는 이제 공기를 통해 이동하고 있습니다. 이 시퀀스를 직접 대체하는 pulseOut 개념은 표준 arduino 라이브러리에 존재하지 않습니다.

에코 수신:
pinMode(pingPin, INPUT); 에코 신호를 수신하기 위해 pingPin 모드를 INPUT으로 변경합니다.

초음파의 비행 시간(travel time) 측정

duration = pulseIn(pingPin, HIGH); 이것이 중요한 부분입니다. 'pulseIn()'은 초음파 펄스 전송과 에코 수신 사이의 시간을 측정합니다. 핀이 HIGH(에코 시작)가 될 때까지 기다리고 타이밍을 시작한 다음 핀이 LOW(에코 끝)가 될 때까지 기다린 다음 타이밍을 중지합니다.

pulseIn() 함수는 지정된 핀의 펄스 길이(마이크로초 단위)를 측정합니다. 이 함수는 핀 번호와 측정할 펄스 유형(HIGH 또는 LOW)이라는 두 가지 인수를 사용합니다. 위 코드에서 pulseIn(pingPin, HIGH)는 에코 펄스의 지속 시간을 측정합니다. 펄스는 핀이 HIGH(에코 수신)가 될 때 시작되고 핀이 다시 LOW(에코 종료)가 되면 끝납니다.

'duration' 변수는 초음파가 전송되고 에코가 수신되는 사이의 시간을 마이크로초 단위로 기록합니다. 이 시간을 거리로 변환하려면 일반적으로 공기 중 소리의 속도(초당 약 343미터, 즉 0.0343cm/μs)를 사용하고 소리가 물체를 왕복해야 한다는 점을 고려합니다. 따라서 물체까지의 거리는 비행시간 * 0.0343 / 2 cm입니다.

'SelfAbot'클래스의 pulseOut 메소드를 사용해서 위의 코드를 다시 작성할 수 있습니다.

```
long duration, distanceCm;
abot.pulseOut(pingPin, 5);
duration = abot.pulseIn(pingPin, HIGH);
distanceCm = duration * 0.0343 / 2;
```

지금까지의 내용이 초음파센서의 동작원리를 이해하는데 도움이 되었기를 바랍니다.
Parallax사의 3핀 초음파센서로 물체와의 거리를 측정하는 .ino 코드를 아래에 소개합니다.

Ex7.2_parallax_3pin_ultrasonic_distance.ino 스케치를 아두이노에 업로드하세요.

```
#include "SelfAbot.h"

SelfAbot abot;
const byte pingPin = 7; // sensor trigger/echo pin

void setup() {
    abot.setup();
    Serial.begin(9600);
}
void loop() {
    abot.pulseOut(pingPin, 5); // 5 microseconds pulse
    unsigned long duration = abot.pulseIn(pingPin, HIGH);
    // Calculate distance (in centimeters or inches)
    // Speed of sound = 34300 cm/s (in air), so time for there and back is duration
    // Distance = (speed of sound * time) / 2
    float distanceCm = duration * 0.0343 / 2;
    float distanceInch = duration * 0.0135 / 2;

    // Print the distance
    Serial.print("Distance: ");
    Serial.print(distanceCm);
    Serial.print(" cm, ");
    Serial.print(distanceInch);
    Serial.println(" inches");
    delay(1000); // Wait for a second before the next measurement
}
```

7.2 전방물체 레이더 화면 표시하기

초음파센서를 180도 회전하는 동작과 초음파센서로부터 감지된 데이터를 화면에 출력하기 위해 processing 도구를 사용할 예정입니다. 초음파센서는 앞서 아두이노로봇을 제자리에서 피봇회전하는 방법을 사용하지 않고, 표준형 서보모터를 사용해서 초음파 센서만 회전하는 방식을 소개합니다.

Ex7.3_ultrasonic_radar_scan.ino 스케치를 아두이노에 업로드하세요.

```
#include "SelfAbot.h"

SelfAbot abot;
const byte servoPin = 10;
const byte pingPin = 7; // sensor trigger/echo pin
int distance;

void setup() {
	abot.setup();
	abot.servoAttachAngle(servoPin);
	Serial.begin(9600);
}

void loop() {
	for (int i = 0; i <= 180 ; i++) {
		abot.servoAngle(i);
		delay(15);
		distance = calculateDistance();

		Serial.print(i);
		Serial.print(",");
		Serial.print(distance);
		Serial.print(".");
	}

	for (int i = 180; i > 0 ; i--) {
		abot.servoAngle(i);
		delay(15);
		distance = calculateDistance();

		Serial.print(i);
		Serial.print(",");
		Serial.print(distance);
		Serial.print(".");
	}
}

int calculateDistance(){
	abot.pulseOut(pingPin, 5);
	unsigned long duration = abot.pulseIn(pingPin, HIGH);

	int distanceCm = (int)duration * 0.034/2;
	return distanceCm;
}
```

아두이노 시리얼 출력화면 데이터는 각도와 거리값만 출력합니다. 우리는 더욱 시각적으로 초음파센서가 감지한 전방의 물체를 화면에 표시하고 싶습니다. 이런 작업을 processing 프로

그래밍 도구를 사용하면 쉽게 거리/각도 데이터를 이미지 데이터로 바꾸어서 표현할 수 있습니다. 프로세싱 도구 사용법 내용을 여기서 모두 소개할 것은 아니기 때문에, 현재 초음파센서의 값을 화면에서 시각적으로 표현하는 방법만 소개하겠습니다.

프로세싱 프로그램 설치는 프로세싱 홈페이지(https://processing.org/download)에서 다운로드 받고 설치하면 됩니다. 프로세싱 프로그램을 설치한 이후, 아두이노 코드로 동작하는 서보모터와 초음파센서 그리고 아두이노에서 출력되는 센서 출력값들이 어떻게 프로세싱 프로그램과 연결되는지 살펴봅시다.

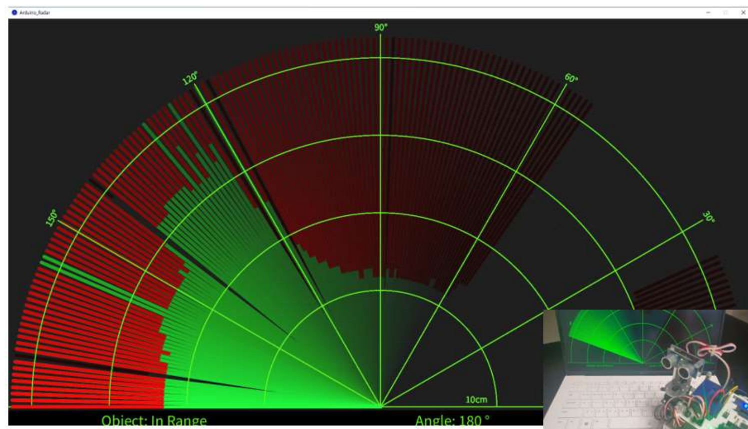


그림7.2 : 로봇에 장착한 초음파센서 레이더 화면

프로세싱과 아두이노 출력값이 연결되는 방식은 매우 간단합니다. 인터페이스를 살펴보면, 프로세싱의 serialevent() 함수를 사용하면 센서가 arduino 보드에서 프로세싱 IDE로 초음파센서가 측정한 각도와 거리에 대한 값을 받게 됩니다. 그러면 직렬 COM포트에서 데이터를 읽고 각도와 거리 값을 iAngle 및 iDistance 변수에 입력합니다. 마지막으로 이러한 변수는 레이더, 선, 감지된 개체 및 일부 텍스트를 그리는데 사용됩니다. 첨부되는 프로세싱코드를 실행할 때, 사용자는 아두이노와 컴퓨터가 연결된 올바른 COM포트만 프로세싱 코드에게 제공하면 됩니다. 프로세싱 소스코드는 아래에 첨부되는 링크를 사용하기 바랍니다.
https://www.superkitz.com/arduino-radar-using-ultrasonic-sensor-hc-sr04-diy-kit/
그리고 프로세싱 소스코드는 부록에도 남겨두겠습니다. 참고하기 바랍니다.

7.3 초음파센서 탐색 로봇으로 주행

초음파센서를 사용해서 로봇이 주행하는 전방에 장애물이 있는지를 감지하고 장애물이 감지되면 회피하는 가장 간단한 로봇동작을 구현하는 예제입니다. 코드 구현 방식은 정적 탐색과 동적 탐색 두가지 유형으로 소개합니다.

먼저 소개할 예제는 정적탐색(static exploration) 모드로 동작하는 예제코드입니다. 정적 탐색의 핵심은 정지한 상태에서 주변을 탐색하고, 미리 설정한 기준으로 안전한 각도구간을 결

정한 다음 이동하는 방식입니다. 코드가 동작하는 원리는 다음과 같습니다.
먼저 Setup 함수에서 먼저 시리얼 통신을 시작하고, SelfAbot 라이브러리를 설정합니다. 두 개의 바퀴에 연결된 서보모터와 초음파센서가 부착된 표준형 서보모터를 attach() 함수로 준비합니다.

Loop 함수는 계속해서 반복되는데, 코드의 주요 기능은 다음과 같습니다.

(1) calculateDistance 함수는 초음파 센서를 사용하여 물체와의 거리를 측정합니다.
(2) isSafeAreaDetected 함수는 스캔 각도에서 감지된 거리가 안전 거리보다 멀고, 연속적으로 장애물이 감지되지 않는 각도범위가 40도 이상인지 확인합니다.
(3) getSafeAreaCenterAngle 함수는 여러 거리 측정 값 중에서 안전 거리보다 멀고, 가장 장애물이 없는 넓은 각도영역을 찾아 해당 영역의 중간 각도를 반환합니다.
(4) setTravelDirection 함수는 로봇을 원하는 방향으로 회전시키는 역할을 합니다. 현재 각도를 고려하여 목표 각도까지의 회전 각도를 계산하고, 이를 이용하여 로봇을 제자리회전시킵니다.
(5) alignSensorToFront 함수는 로봇의 초음파센서가 앞쪽을 향하도록 정렬합니다.

이렇게 초음파 센서를 사용하여 물체와의 거리를 측정하고, 안전한 영역을 찾아 로봇을 회전시키는 프로세스가 진행됩니다.

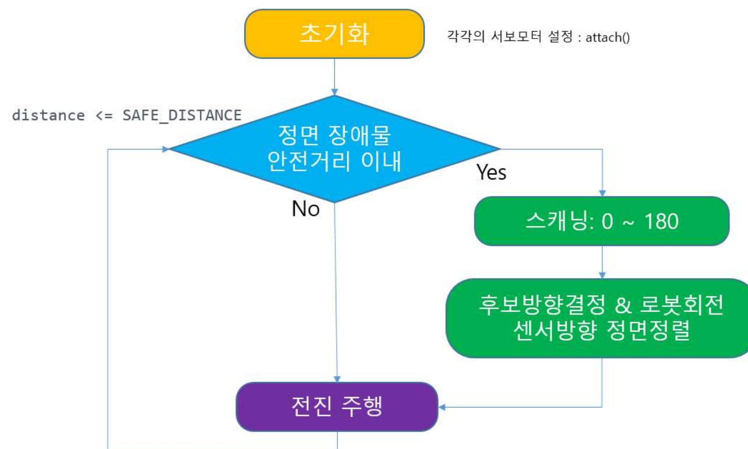


그림7.3 : 초음파센서 정적 탐색 논리순서도

아래 Ex7.4코드는 초음파센서로 정적탐색하는 로봇동작 예제를 소개한 것입니다. 이 방식은 주로 환경이 정적이거나 변화가 적을 때 사용됩니다. 초음파 센서를 사용하여 주변 환경을 탐색하고, 각도를 조절하여 안전한 영역을 파악하는 정적 탐색의 한 예입니다.

예제 코드에는 제3장의 표준형 서보모터 동작에 대한 예제 Ex3.2 와 제6장의 로봇 제자리회전에 대한 예제 Ex6.4 또는 Ex6.5를 참고할 수 있습니다.
아래 코드 예제에서 사용하는 isSafeAreaDetected 함수와 getSafeAreaCenterAngle 함수의

SafeArea는 아래 그림7.4에서처럼 로봇의 전방영역에서 안전거리 이내에 장애물이 존재하지 않는 범위를 사용합니다.

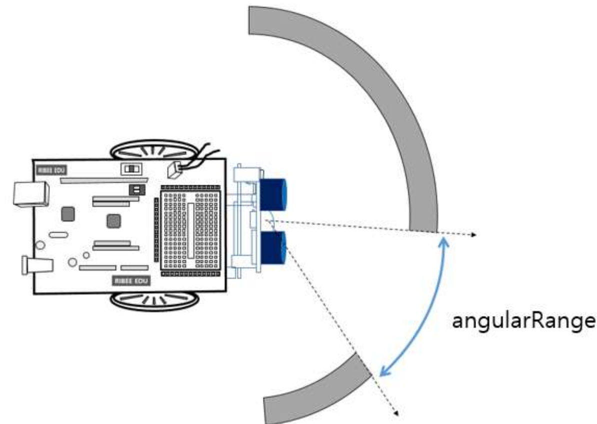


그림7.4 : 초음파센서가 안전영역으로 판단하는 각도 범위(angular range)

이 범위 값을 적정 값으로 사용하는 것은 로봇이 장애물을 통과할 수 있는지 없는지를 결정할 수 있습니다. 사용자의 로봇 동작 환경에 맞게 설정해야할 필요가 있습니다.

Ex7.4_Ultrasonic_sensor_search_driving.ino 예제를 업로드하세요.

```
#include "SelfAbot.h"

#define TRIGGER_PIN   7
#define ECHO_PIN      7
#define MAX_DISTANCE 200 // max distance (cm)
#define SAFE_DISTANCE 20 // safe distance (cm)
#define MAX_DETECTIONS 40

#define INVALID_PIN 255
const byte servoLeftPin = INVALID_PIN;
const byte servoRightPin = INVALID_PIN;

SelfAbot abot(servoLeftPin, servoRightPin);
float factor = 10.5;

struct Detection {
  int angle;
  unsigned int distance;
};

Detection detections[MAX_DETECTIONS];
int detectionCount = 0;
int notdetectedAngularrange = 40; // 장애물이 존재하지 않는 최소한의 각도범위

void setup() {
  Serial.begin(9600);
  abot.setup();
  abot.servoAttachPins(13, 12);
  abot.servoAttachAngle(10);
}
```

```
void loop() {
  int distance = calculateDistance();

  if(distance > 0 && distance <= SAFE_DISTANCE) {
    abot.servoSpeed(0, 0);

    detectionCount = 0;
    for(int angle = 0; angle <= 180; angle += 5) {
      abot.servoAngle(angle);
      delay(500);
      distance = calculateDistance();

      if (detectionCount < MAX_DETECTIONS) {

                    == 이하 코드 생략 ==
     나머지 코드는 아두이노 SelfAbot 라이브러리 예제 참조하세요!
```

위에서 소개한 예제와 다르게, (1)스캔해서 각도별 거리 데이터를 저장하는 동작과 (2)저장된 데이터를 분석하여 장애물이 없는 방향으로 로봇이 정렬하는 동작을 하나로 결합하는 방법이 있습니다. 이 방법은 데이터를 스캔하는 동안에도 분석을 동시에 수행하고, 이를 토대로 실시간으로 로봇의 이동 방향을 결정할 수 있습니다. 이런 방법을 동적 탐색(dynamic exploration)이라고 합니다.

앞서 소개한 정적 탐색과 비교해서 동적 탐색은 다음과 같은 차이점이 있습니다.
(1) 실시간 데이터 분석: 이전의 코드는 스캔이 완료된 후에 데이터를 분석하여 안전한 이동 방향을 결정했습니다. 그에 반해 동적 탐색 방법에서는 각도를 이동하면서 실시간으로 데이터를 분석하고, 안전영역 조건에 부합하면 로봇이 즉시 이동 방향을 결정하고 회전합니다.

(2) 스캔 및 분석의 통합: 동적 탐색 방법에서는 로봇이 각도를 이동하면서 스캔과 분석을 동시에 수행합니다. 이것은 실시간으로 환경을 탐색하고 즉각적으로 반응할 수 있도록 해줍니다.

(3) 효율성: 동적 탐색 방법은 스캔이 진행되는 동안에도 데이터를 분석하므로, 이동 방향을 결정하는 데 더 적은 시간이 걸릴 수 있습니다. 또한 스캔이 완료된 후에 이동 방향을 결정하는 과정을 생략함으로써 시간을 절약할 수 있습니다.

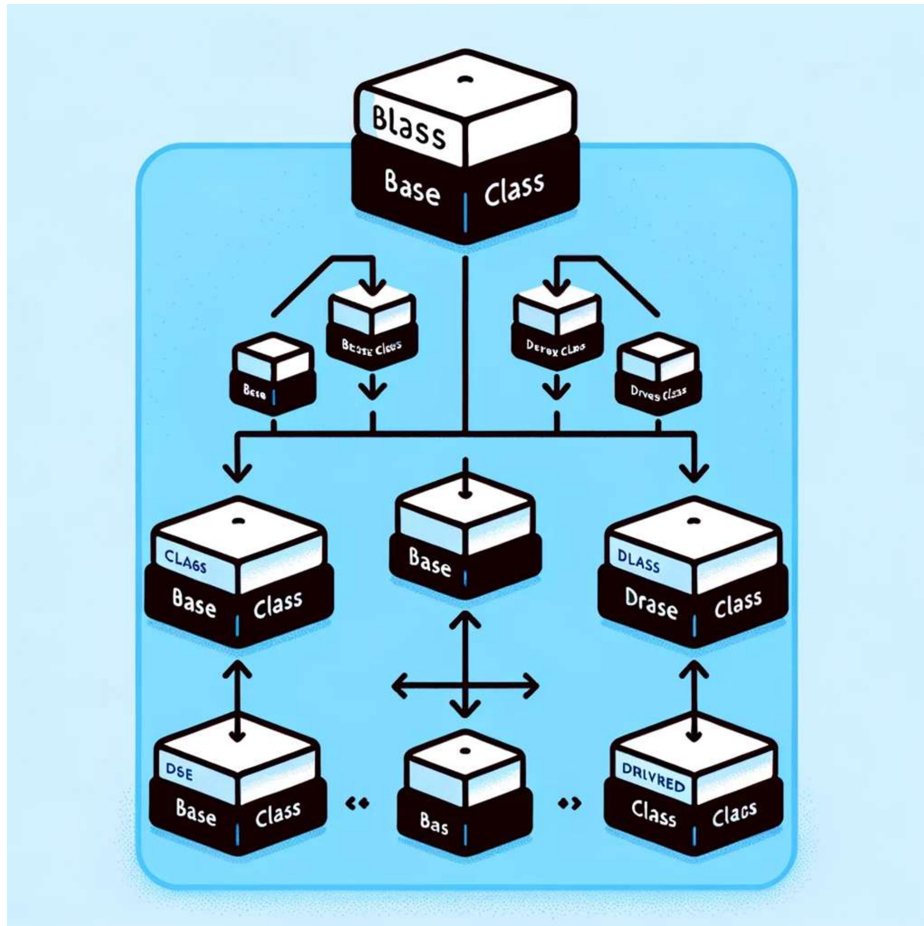
다음 소개할 예제는 조금 전 Ex7.4예제를 변형한 형태입니다. 이전 예제가 장애물이 없는 구간을 하나이상 찾더라도, 0 ~ 180도 각도구간 전체를 스캔하고, 다음 순서로 로봇 동작을 실행하는 방식으로 분리 처리했다면, 아래 소개하는 예제는 스캔 작업과 로봇의 회전 동작을 통합해서 처리하는 방식입니다. 이렇게 하려면, 데이터를 수집하면 즉시 데이터 분석을 실행하고 조건에 부합하는 장애물이 없는 구간을 찾는 즉시 로봇을 회전시키는 동작을 실행합니다. 이렇게 데이터를 처리함으로써 로봇은 조금 더 유연하게 동작할 수 있습니다.

아래 스케치 예제 Ex7.5_Ultrasonic_sensor_Integrated_scanning_rotation를 업로드해서 실행해보세요. 이 코드는 조금 전 코드보다 더 효율적이고 간명하게 작성된 예제입니다.

```
#include "SelfAbot.h"

#define TRIGGER_PIN   7
#define ECHO_PIN       7
#define MAX_DISTANCE 200 // max distance (cm)
#define SAFE_DISTANCE 20 // safe distance (cm)

#define MAX_DETECTIONS 40

#define INVALID_PIN 255
const byte servoLeftPin = INVALID_PIN;
const byte servoRightPin = INVALID_PIN;

SelfAbot abot(servoLeftPin, servoRightPin);
float factor = 10.5;

struct Detection {
  int angle;
  unsigned int distance;
};

Detection detections[MAX_DETECTIONS];
int detectionCount = 0;
int notdetectedAngularrange = 40; // 장애물이 존재하지 않는 최소한의 각도범위

void setup() {
  Serial.begin(9600);
  abot.setup();

                == 이하 코드 생략 ==
     나머지 코드는 아두이노 SelfAbot 라이브러리 예제 참조하세요!
```

제 8 장 클래스 상속으로 전문로봇 만들기

8.1 'SelfAbot' 슈퍼클래스로 하는 하위 클래스 만들기
8.2 상속클래스로 .ino 파일 작성하는 방법



클래스 상속이란 클래스(자식 클래스 또는 하위 클래스)가 다른 클래스(상위 클래스 또는 슈퍼클래스)로부터 속성과 동작(메서드 및 변수)을 상속할 수 있는 개념으로 객체 지향 프로그래밍(OOP)의 기본 개념입니다. 이 메커니즘을 사용하면 기존 클래스를 기반으로 새 클래스를 생성할 수 있어 코드 재사용성이 향상됩니다. 결론적으로 클래스 상속은 코드를 계층적 구조로 구성하여 재사용, 확장 및 관리가 용이하도록 만드는 객체지향프로그래밍(OOP)의 강력한 도구입니다.

클래스 상속이 수직적 관계("is-a")에서 클래스 접근을 다룬다면 프랜드 선언은 수평관계의 클래스 접근에 대하여 다루고, 구성 또는 포함 관계("has-a")는 하나의 클래스가 다른 클래스의 객체를 멤버로 포함하지만, 여기서는 소개하지 않겠습니다.

클래스 상속의 특징을 살펴보겠습니다.

(1) 코드 재사용: 상속을 통해 새 클래스는 기존 클래스의 속성과 메소드를 상속받을 수 있습니다. 이는 이미 작성된 코드를 다시 작성할 필요가 없음을 의미합니다. 하위 클래스에서 확장하거나 수정할 수 있습니다.

(2) 계층적 관계: 상속은 클래스 간에 계층적 관계를 생성합니다. 상위 클래스(기본 또는 슈퍼클래스라고도 함)는 여러 하위 클래스(파생 클래스 또는 하위 클래스라고도 함)에서 공유할 수 있는 일반 정의를 제공합니다.

(3) 메서드 재정의: 하위 클래스는 상위 클래스의 메서드를 재정의하여 상위 클래스와 다른 특정 구현을 제공할 수 있습니다.

(4) 확장성: 상위 클래스나 상위 클래스에서 상속하는 다른 클래스에 영향을 주지 않고 하위 클래스에 새 기능을 추가할 수 있습니다. 이렇게 하면 코드의 확장성이 향상됩니다.

(5) 다형성: 상속은 하위 클래스가 상위 클래스의 인스턴스로 처리될 수 있는 다형성 개념을 지원합니다. 이는 다양한 특정 인스턴스에 대해 일반화된 유형을 사용하려는 시나리오에서 특히 유용합니다.

클래스 상속을 설명하는 간단한 예제입니다.

```
class Vehicle {      // 상위 클래스
public:
    void startEngine() {
        // Code to start the engine
    }
};
class Car : public Vehicle {     // 하위 클래스
public:
    void openTrunk() {
        // Code to open the trunk
    }
};
```

이 C++ 예제에서 Car는 Vehicle 슈퍼클래스에서 상속된 하위 클래스입니다. 이는 Car 클래스의 객체가 startEngine 메소드(Vehicle에서 상속됨)와 openTrunk 메소드(Car에 특정)를 모두 사용할 수 있음을 의미합니다.

요약하자면, 클래스 상속은 적절히 사용될 때 우아하고 효율적인 코드 구조를 만들 수 있는 객체지향프로그래밍(OOP)의 강력한 도구입니다. 그러나 이를 신중하게 사용하는 것이 중요합니다. 초보자에게는 간단한 상속 사례로 시작하여 개념을 잡은 후 더 복잡한 시나리오로 넘어가는 것이 좋습니다.

8.1 'SelfAbot' 슈퍼클래스로 하는 하위 클래스 만들기

이 장에서는 'EnhancedSelfAbot' 이라는 하위 클래스를 만들어 보겠습니다. 이 클래스는 'SelfAbot'을 상위 클래스로 설정하고 상속받는 하위 클래스입니다. 한가지 적용사례를 아래에 소개합니다. 지금까지 설명했던 코드들중 하나이지만, 'SelfAbot' 기본클래스에는 포함되지 않았던 메소드들을 지금의 하위 클래스에서 적용할 수 있습니다. 'EnhancedSelfAbot'하위 클래스에 포함된 다양한 메소드중 '4.4절 점진적 가속과 감속'에서 소개했던 내용을 하위클래스로 표현한 내용을 소개합니다.

```
------------ EnhancedSelfAbot.h ----------------------------------------------
    #include "SelfAbot.h"

    class EnhancedSelfAbot : public SelfAbot {

    public:
        EnhancedSelfAbot() : SelfAbot(), _currentLeftSpeed(0), _currentRightSpeed(0) {}
        EnhancedSelfAbot(byte servoLeftPin, byte servoRightPin) : SelfAbot(servoLeftPin, servoRightPin), _currentLeftSpeed(0), _currentRightSpeed(0) {}
        void gradualServoSpeed(int targetLeftSpeed, int targetRightSpeed);

    private:
        int _currentLeftSpeed = 0;  // 현재 왼쪽 서보의 속도
        int _currentRightSpeed = 0; // 현재 오른쪽 서보의 속도
    };
------------ EnhancedSelfAbot.cpp --------------------------------------------
    void EnhancedSelfAbot::gradualServoSpeed(int targetLeftSpeed, int targetRightSpeed) {
        int step = 5; // 속도 변경 단계 설정

      while (_currentLeftSpeed != targetLeftSpeed || _currentRightSpeed != targetRightSpeed) {
            unsigned long currentMillis = millis();

            if (_currentLeftSpeed < targetLeftSpeed) {
                _currentLeftSpeed += step;
            } else if (_currentLeftSpeed > targetLeftSpeed) {
                _currentLeftSpeed -= step;
            }
            if (_currentRightSpeed < targetRightSpeed) {
                _currentRightSpeed += step;
            } else if (_currentRightSpeed > targetRightSpeed) {
                _currentRightSpeed -= step;
            }
            servoSpeed(_currentLeftSpeed, _currentRightSpeed);
            while (millis() - currentMillis < 10) {
                // 10ms가 지날 때까지 작은 지연
            }
        }
    }
```

위의 코드가 잠재적으로 가지고 있는 문제점을 개선한 코드를 하위 클래스에서 반영합니다. 'EnhancedSelfAbot' 파일에서 직접 살펴보기 바랍니다.

8.2 상속클래스로 .ino 파일 작성하는 방법

‘EnhancedSelfAbot’하위 클래스를 사용하는 방법은 ‘SelfAbot.zip’ 라이브러리와 동일하게 ‘EnhancedSelfAbot.zip’ 라이브러리를 아두이노 IDE프로그램에 추가하고, 별도의 라이브러리로 사용할 수 있습니다.
여기 실습에서는 ‘SelfAbot.zip’ 라이브러리를 설치할 때, 함께 ‘EnhancedSelfAbot’ 하위 클래스가 이미 아두이노 라이브러리 폴더에 설치되어서 사용할 준비가 되어있습니다.

```
EnhancedSelfAbot abot;
void setup() {
    abot.servoAttachPins(LEFT_SERVO_PIN, RIGHT_SERVO_PIN); // 서보 핀 설정
    abot.gradualServoSpeed(100, 100); // 점진적 속도 조절 시작
}
void loop() {
    // 메인 루프 코드...
}
```

아두이노 .ino 파일에서 ‘EnhancedSelfAbot’하위 클래스를 사용해서 아래 예시 코드처럼 작성하고 실습할 수 있습니다.

8.2.1 직진주행 보정을 위한 실습

제4장에서는 로봇의 직진 주행을 위해 deviationFactor 값을 .ino 코드에서 직접 수정하는 방법을 사용했습니다. 여기서는 로봇이 약속된 시간동안 주행할 때, 직선주행으로부터 편향되는 거리를 측정해서 매개변수로 보정하는 방법으로 deviationFactor를 보정하는 코드를 먼저 실행합니다. 이런 목적의 하위클래스 메소드 calibratedeviationFactor(int deviatedistance) 입니다.

```
void EnhancedSelfAbot::calibratedeviationFactor(int deviatedistance) {
      if (devidistance != 0) {
              float K = 0.005;
              _deviationFactor += K * (float)deviatedistance;
      } else {
              // blank
      }
}
```

위 메소드로 측정하기 위해서 몇가지 사용자의 고려사항이 있습니다.
(1) 이 메소드의 매개변수 deviatedistance는 로봇이 직진 주행했을 때 직선에서 수직으로 벗어난 거리(단위 밀리미터)값을 입력합니다. 로봇의 속도에 따라 직선에서 벗어나는 정도가 다르기 때문에 실습에서 중점적으로 적용할 속도를 보정하는 것이 좋습니다. 실습을 위해서 중간 속도에 가까운 40을 추천합니다.

(2) 측정 자를 사용해서 일정한 거리를 주행한 다음, 로봇이 직선에서 얼마나 벗어났는지 측정하고 위 메소드의 매개변수 값에 입력하고 다시 실행합니다. 만약 직선에서 벗어나는 순간이

나 벗어난 거리 등을 센서 데이터로 입력받을 수 있다면, 이 작업을 자동화할 수 있습니다. 아래 보정 실습 예제는 _deviationFactor 값이 항상 초기화되므로, 이전 실습에서 얻은 값이 반영되지 않습니다.

(3) 보정 작업을 반복할수록 로봇은 직선주행에 가까워질수 있습니다. 만약 예상된 결과와 달리 더 크게 직선으로부터 벗어난다면, 코드에서 사용하는 상수 값(예를 들면 'K'값)을 수정해야 할 수 있습니다. *(지금의 상수값 'K'는 정확한 실험적 결과가 아닙니다.)*

(4) 보정 루틴을 반복해서 로봇이 직선주행을 하게 된다면, 매회마다 주행 조건에 해당하는 보정계수 '_deviationFactor'값이 새롭게 업데이트되어 저장되고, '_deviationFactor'값이 클래스변수에 저장되어 최종 보정값으로 사용될 수 있습니다.
(참고로 driveStraight(leftSpeed, rightSpeed)메소드는 servoSpeed(leftSpeed, rightSpeed)와 동등한 목적으로 대체되어 사용될 수 있습니다.)

아래 소개한 예제코드로 로봇이 직선주행을 하기 위한 보정 루틴을 실행하는 경우, USB케이블을 분리하지 않고 실습을 완료해야 하는 제한사항이 있습니다. 만약 보정을 위한 주행거리를 더 멀리 설정하고 싶다면, 블루투스 또는 와이파이 무선통신 수단으로 PC의 시리얼 입력 데이터를 로봇으로 송신하고 보정작업 출력결과들을 수신 처리할 수 있습니다.

예제 Ex8.1의 실습순서를 간략히 소개하면 다음과 같습니다.
(1) 아두이노로봇 좌/우 바퀴 서보모터 핀을 13, 12번으로 연결합니다.

(2) 아래 예제 스케치를 업로드하고, 실습할 공간에 로봇을 정렬시킵니다.

(3) 스케치를 업로드하고, IDE프로그램의 시리얼모니터 창을 열고 's'를 입력하면 로봇이 예정된 시간동안 앞으로 전진합니다.

(4) 로봇이 일전한 시간동안 전진한 다음, 정지상태가 되면 시리얼모니터에 "Please measure the deviation distance and enter it (mm): " 출력문이 표시될 때, 로봇을 처음 출발한 위치이에 옮겨놓고 시리얼모니터창에 직선으로부터 일탈한 거리를 mm 단위로 입력하고, 엔터키를 누릅니다. (일련의 사용자 동작은 30초 이내에 이루어져야 합니다. 그렇지 않으면 맨처음부터 다시 시작해야 합니다.)

(※ 이탈거리(deviation distance)는 mm 단위로 입력하며, 로봇을 좌측으로 보정하려면 이탈거리에 '-' 부호를 붙여서 사용하고, 로봇을 우측으로 치우치도록 보정하려면 이탈거리에 '+'의 부호를 사용하면 됩니다.)

(5) 일회의 보정작업이 완료되면, 보정에 사용된 _deviationFactor 값이 출력됩니다. 이 값은 보정작업이 다시 반복되지 않는 동안 클래스의 멤버변수에 그대로 유지됩니다.

Ex8.1_straightDrive_calibration_practice.ino

```
#include "EnhancedSelfAbot.h"

const byte servoLeftPin = 255;
const byte servoRightPin = 255;
EnhancedSelfAbot abot(servoLeftPin, servoRightPin);

void setup() {
    Serial.begin(9600);
    abot.setup();
    abot.servoAttachPins(13, 12); // Replace with actual servo pins
    Serial.println("Calibration Practice for SelfAbot");
    Serial.println("Place the robot and press 's' to start moving.");
}

void loop() {
    if (Serial.available() > 0) {
        char command = Serial.read();
        if (command == 's') {
            performCalibration();
        }
    }
}

void performCalibration() {
    int servospeed = 40;
    int delaytime = 5000; // Time to make the robot go straight (5 seconds)
    int measuredDevidistance; // Save the measured deviation distance (unit : mm)
    unsigned long startTime = millis(); // Start time for timeout
    unsigned long timeout = 30000; // 30 second timeout

    abot.resetDeviationFactor();

    Serial.println("Going straight. Please wait...");
    abot.driveStraight(servospeed, -servospeed); // Straight
    delay(delaytime); // Go straight for the specified delay time
    abot.servoSpeed(0, 0); // Stop the robot

    Serial.println("Please measure the deviation distance and enter it (mm): ");
    delay(500); // Short delay to ensure the user sees the message

    // Clear the serial buffer to remove any unwanted or leftover data
    while (Serial.available() > 0) {
        Serial.read();
    }

    // Wait for new input
    while (Serial.available() == 0) {
        // Check if timeout has been reached
        if (millis() - startTime > timeout) {
            Serial.println("Input waiting time has expired. Please restart calibration.");
            return; // exit the function when timeout is reached
        }
        delay(100); // Small delay to prevent the loop from running too fast
    }

    measuredDevidistance = Serial.parseInt(); // Read the entered value...

    // Check for '0' input to exit without performing corrections
    if (measuredDevidistance == 0) {
        Serial.println("Exiting calibration without correction as the input is '0'.");
        return; // exit the function as '0' was entered
    }
```

```
    Serial.print("Measured deviation distance: ");
    Serial.print(measuredDevidistance);
    Serial.println("mm");

    // Convert mm to cm for calibratedeviationFactor if necessary
    float measuredDevidistanceCm = measuredDevidistance / 10.0;
    abot.calibratedeviationFactor(measuredDevidistanceCm);

    abot.driveStraight(servospeed, -servospeed); // Straight
    delay(delaytime); // Go straight for the specified delay time
    abot.servoSpeed(0, 0); // Stop the robot

    // Display the final corrected _deviationFactor value
    float finalDeviationFactor = abot.getDeviationFactor();
    Serial.print("Final corrected _deviationFactor: ");
    Serial.println(finalDeviationFactor);

    Serial.println("Calibration is complete. Test the robot's movements again.");
}
```

보정을 수행하는 performCalibration() 함수의 동작원리와 순서는 다음과 같습니다.

- 각 변수들의 초기값들(함수의 servospeed, delaytime 등)은 변경가능하므로 필요에 맞는 값으로 설정하고, 스케치를 업로드한 다음 시리얼 모니터 창을 열면 아래와 같은 문구("**Place the robot and press 's' to start moving.**")가 출력됩니다. 로봇의 3접점 스위치를 2의 위치에 놓고, 시리얼 모니터에서 's' 문자를 입력하고 엔터를 누릅니다.

- 로봇은 abot.resetDeviationFactor();를 호출해서 _deviationFactor 값을 0으로 초기화하고, delaytime 시간동안 주행한 다음 멈춥니다. 다시 시리얼 모니터에서 로봇이 직선주행에서 벗어난 거리를 mm 단위로 입력하라는 메시지가 출력됩니다.

  Going straight. Please wait...
  Please measure the deviation distance and enter it (mm):

- 로봇이 직선으로부터 이탈한 거리를 직접 측정하고, 개략적인 이탈 거리를 mm 단위로 시리얼 모니터에 입력하고, 엔터키를 누릅니다. 엔터키를 누르면 _deviationFactor 값을 보정해서 abot.driveStraight(servospeed, -servospeed); 함수를 실행합니다. 이때 로봇이 다시 주행을 시작하기 때문에 로봇을 맨처음 시작위치로 돌려 놓아야 합니다.
  당신이 입력한 값이 아래와 같이 화면에 표시될 것입니다.

  Measured deviation distance: ** mm
  Final corrected _deviationFactor: ***
  Calibration is complete. Test the robot's movements again.

  이제 직선주행을 위한 교정작업이 완료됩니다. 이 보정작업에서 찾은 **_deviationFactor: *** 값은 하위클래스의 driveStraight() 메소드를 사용하는데 필수 멤버변수입니다.**

  이후의 계속되는 하위클래스 사용을 위해 내가 사용중인 로봇의 최종 계산된 교정 인자(_deviationFactor)를 잘 기억해두기 바랍니다. 또는 별도로 메모해두기 바랍니다.
  이외에도 실습도중에 이탈거리를 30초이내에 입력하지 못하거나, 이탈거리를 '0'으로 입력하는 경우에 대한 예외처리들을 경험할 수 있습니다. 처음부터 다시 시도하면 됩니다.

이후 또다른 실습에서 abot.driveStraight(servospeed, -servospeed); 함수를 실행하는 경우, 앞서 저장된 _deviationFactor 값은 모두 지워집니다.

그래서 새로운 스케치 예제를 시작하는 경우, 앞서 calibration 함수 실행으로 찾아 두었던 _deviationFactor 값을 setter() 메소드를 사용해서 스케치의 setup 함수에서 설정하고 사용할 수 있습니다.

그림8.1 : 로봇의 직진주행을 위한 보정(calibration) 예제

(참고) --

getter() 메소드와 setter() 메소드를 사용하는 이유는?

'EnhancedSelfAbot' 라이브러리의 getter() 메소드와 setter() 메소드는 다음과 같습니다.

```
float EnhancedSelfAbot::getDeviationFactor() const {
        return _deviationFactor;
    }
void EnhancedSelfAbot::setDeviationFactor(float deviationFactor) {
        _deviationFactor = deviationFactor;
    }
```

_deviationFactor 멤버변수는 클래스의 private 영역에 존재하는 변수이므로, 객체 지향 프로그래밍에서의 캡슐화와 정보 은닉 원칙을 유지하기 위함입니다.

8.2.2 점진적 속도제어 실습

gradualServoSpeed 메소드를 사용하여 '아두이노 로봇의 점진적 속도변화 주행테스트'를 실습하는 .ino 파일 예제 코드입니다. servoSpeed() 메소드와 함께 gradualServoSpeed 메소드 역시 속도 값이 100 이하로 낮은 값일수록 로봇의 좌측 바퀴와 우측 바퀴 속도 편차가 두드러지게 표현되어, 직전주행임에도 불구하고 좌측이나 우측으로 현저히 기울면서 주행할 수 있습니다. 이런 문제를 개선하기 위하여, gradualServoSpeed 메소드는 servoSpeed() 메소드가 아니라 driveStraight 메소드를 사용합니다.

gradualServoSpeed 메소드는 이전 보정작업에서 찾은 값을 아래 setup 함수에서 setter() 메소드를 사용해서 코드에 반영합니다. abot.setDeviationFactor(-0.05);를 추가하고, 스케치를 업로드하면 됩니다. ***(참고: 예시에 표시된 -0.05 값은 사용자마다 다를 수 있으므로 직접 찾아서 사용해야 합니다.)***

Ex8_2_enhanced_gradual_servospeed.ino

```
#include "EnhancedSelfAbot.h"

const byte servoLeftPin = 255;
const byte servoRightPin = 255;
EnhancedSelfAbot abot(servoLeftPin, servoRightPin);

void setup() {
      Serial.begin(9600); // Start serial communication for debugging
      abot.setup();
      abot.servoAttachPins(13, 12); // Replace with actual servo pins

      abot.setDeviationFactor(-0.05);
}

void loop() {
      abot.gradualServoSpeed(100, -100);
      delay(2000); // Drive for 2000 milliseconds (2 seconds)

      abot.gradualServoSpeed(40, -40);
      delay(3000);
}
```

위의 스케치를 아두이노에 업로드한 다음, 로봇의 전원을 켜고 이동을 관찰하세요. 로봇은 점진적으로 변화하면서 타겟 속도에 도달해야 합니다. 로봇이 원하는 방향으로 정확히 이동하는지, 속도 변화가 부드러운지 관찰하세요. 로봇의 이동 경로나 속도 변경이 예상과 다르면 step 값을 조정해볼 수 있습니다.

그리고 _deviationFactor 값으로 입력된 -0.05가 잘 보정된 값이라면, 로봇의 속도값이 100으로 크거나 40으로 작더라도 직진주행에서 크게 벗어나지 않아야 합니다.

8.2.3 불빛을 따라서 주행하는 아두이노로봇

'EnhancedSelfAbot'클래스의 lightFollowing 메소드를 활용하는 아두이노 .ino 예제입니다. 이 예제는 포토트랜지스터를 사용해서 로봇이 자율적으로 판단하고 동작하는 사례입니다.
앞에서 살펴본 5장 2절 내용과 그림5.6을 참고해서 동일하게 회로를 구성하면 됩니다. 실습예제 Ex5.5와 동일한 내용을 구현하지만, 하위클래스에 코드의 내용을 상당부분 담고 있어서 지금의 .ino 코드 구성이 많이 간결합니다.

조금 다른 점은 앞선 예제 Ex5.5는 servoSpeed() 메소드를 사용하지만, 아래 코드는 로봇의 직선주행을 보정한 driveStraight() 메소드를 사용하는 점입니다. 그래서 로봇의 주행동작이

더 섬세하고 정확해질 수 있습니다.

Ex8_3_enhanced_lightfollowing_abot.ino

```
#include "EnhancedSelfAbot.h"

const byte servoLeftPin = 255;
const byte servoRightPin = 255;
EnhancedSelfAbot abot(servoLeftPin, servoRightPin);

void setup() {
    Serial.begin(9600); // Start serial communication for debugging
    abot.setup();
    abot.servoAttachPins(13, 12); // Replace with actual servo pins
}

void loop() {
    unsigned long leftrcTime = abot.rcTime(8);
    unsigned long rightrcTime = abot.rcTime(6);

    abot.lightFollowing(leftrcTime, rightrcTime);
    delay(50);
}
```

로봇이 움직이기 시작할 때, 랜턴이나 스마트폰 불빛을 켜서 로봇의 동작을 유도해보세요. 로봇이 빛을 비추는 방향으로 움직이기 시작하면 정상적인 동작입니다.

로봇의 미세한 동작을 수정해야할 필요가 있다면, 라이브러리 코드의 값들을 조정해야 할 수도 있습니다. 지금까지 학습으로 코드의 이해도가 높아졌다고 생각한다면, 시도해보기 바랍니다.

8.2.4 적외선 센서로 탐색주행하는 로봇 실습

실습을 위해 아두이노로봇에 적외선 LED와 적외선 수신기(receiver)를 좌측과 우측에 배치하고 회로연결을 완성하기 바랍니다. 상세한 회로구성과 동작원리는 6.1절과 동일합니다.

적외선 불빛이 사람의 눈으로는 식별되지 않기 때문에, 적외선 센서가 물체를 감지하는지 감지하지 않는지를 불빛으로 표시하려면, 좌측과 우측에 LED회로를 추가하고, LED코드를 추가하면 됩니다. 하드웨어 연결이 완성되었다면, 아래 코드를 참고해서 추가하세요.

```
pinMode(11, OUTPUT);
pinMode(4, OUTPUT);

if (irLeft == 0) {
    digitalWrite(11, HIGH);
} else {
    digitalWrite(11, LOW);
}
if (irRight == 0) {
```

```
        digitalWrite(4, HIGH);
    } else {
        digitalWrite(4, LOW);
    }
```

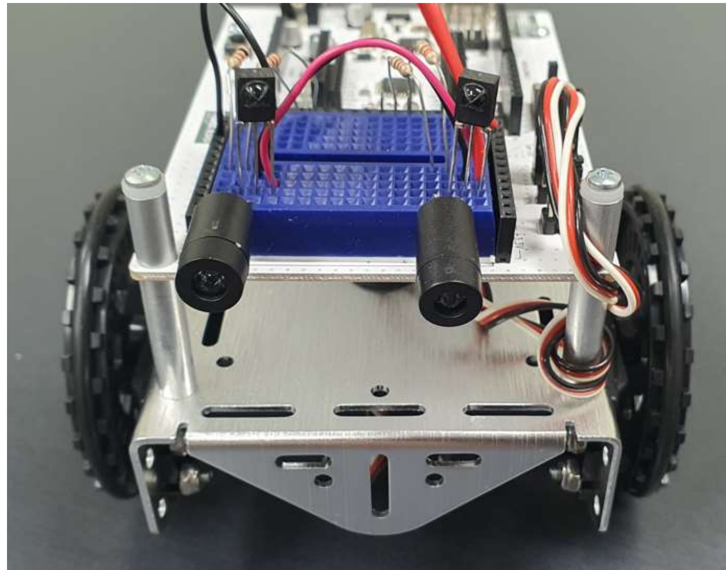


그림8.2 : 적외선 송수신센서 탐색주행 모습

Ex8_4_enhanced_irNavigationdriving_abot.ino

```
#include "EnhancedSelfAbot.h"

const byte servoLeftPin = 255;
const byte servoRightPin = 255;
EnhancedSelfAbot abot(servoLeftPin, servoRightPin);

void setup() {
    Serial.begin(9600); // Start serial communication for debugging
    abot.setup();
    abot.servoAttachPins(13, 12); // Replace with actual servo pins

    abot.setDeviationFactor(-0.05);
}

void loop() {
    int irLeft = abot.irDetect(9, 10, 38000);
    int irRight = abot.irDetect(2, 3, 38000);

    abot.irNavigationdriving(irLeft, irRight);
    delay(50);
}
```

좌측과 우측에 설치한 적외선 센서의 감도가 너무 민감하거나, 둔하게 느껴진다면 (물체와의 거리가 멀어도 인식하거나, 너무 가까워도 인식하지 못하는 경우) 적외선 회로의 저항의 크기를 바꾸면서 적정한 조건을 찾아야 합니다.

한가지 더 고려할 사항은 적외선 센서가 빛을 감지하는데, 주변환경 빛의 조건이 간섭을 일으

킬 수 있습니다. 가능하면 동일한 환경조건에서 모든 실습조건들을 맞추고, 동일한 환경에서 로봇의 동작을 실습하는 것이 좋습니다. 가령 교실내에서도 태양 빛이 드는 밝은 지역과 빛이 들지 않는 어두운 지역을 오가면서 실습하는 것은 추천하지 않습니다.

8.2.5 로봇 제자리회전 회전각도로 표현하기

아두이노로봇 제자리회전 목적의 동작을 쉽게 만들기 위하여, calibrateRotation() 메소드와 rotateAbot() 메소드를 사용합니다. 먼저 calibrateRotation() 메소드는 로봇이 360도 한바퀴만 제자리 회전하는데 소요되는 시간(전역변수: duration)을 사용해서, 로봇의 회전각도에 대응시킬 수 있는 보정계수 _setanglefactorCW, _setanglefactorCCW 를 계산할 목적으로 실행합니다.

로봇이 제자리회전할 때, 시계방향 회전과 반시계방향 회전 특성이 다를 수 있습니다. 이런 특성 차이가 생기는 이유는 서보모터의 미세한 차이부터 로봇의 무게중심 비대칭, 두 개 바퀴의 표면마찰 차이 등이 요인일 수 있습니다.

그래서 아래 제자리회전 보정을 위한 스케치 예제를 실행합니다.

Ex8.5_enhanced_rotateinplace_calibration.ino

```
#include "EnhancedSelfAbot.h"

const byte servoLeftPin = 255;
const byte servoRightPin = 255;
EnhancedSelfAbot abot(servoLeftPin, servoRightPin);

int duration = 4400;

void setup() {
    Serial.begin(9600); // Start serial communication for debugging
    abot.setup();
    abot.servoAttachPins(13, 12); // Replace with actual servo pins

    bool isClockwise = false; // Set true for clockwise, false for counterclockwise
    abot.calibrateRotation(duration, isClockwise);

    // Get and display the appropriate _setanglefactor value based on the direction
    if (isClockwise) {
        float setanglefactorCW = abot.getAngleFactorCW();
        Serial.print("_setanglefactorCW clockwise value : ");
        Serial.println(setanglefactorCW);
    } else {
        float setanglefactorCCW = abot.getAngleFactorCCW();
        Serial.print("_setanglefactorCCW counterclockwise value : ");
        Serial.println(setanglefactorCCW);
    }
}

void loop() {

}
```

반복 실습으로 한바퀴 회전에 필요한 시계방향과 반시계방향 매개변수 설정이 완료되면, 회전하고 싶은 각도 값을 rotateAbot() 메소드의 매개변수에 적용해서 로봇을 쉽게 회전시킬 수 있습니다. 센서동작이나, 로봇 주행방향을 선택할 때 rotateAbot() 메소드를 유용하게 사용해 보세요.

시계방향 회전의 경우, 보정계수(**_setanglefactorCW**)를 찾기 위한 코드실행 방법은 아래 스케치 Ex8.5 내용 중에서, **bool isClockwise = false;** 코드의 false를 true로 바꾸어서 스케치를 업로드하면 됩니다. 반시계 방향 회전을 위해서는 false 값을 유지하면서 360회전에 걸리는 duration 값을 입력하고 스케치를 업로드하면 보정계수(**_setanglefactorCCW**)가 찾아집니다.

로봇의 제자리회전을 위한 보정계수 _setanglefactorCW, _setanglefactorCCW 찾는 방법을 다시 상세히 설명합니다.
시계방향 회전에서 bool 변수를 true로 입력하고, 전역변수 duration 값을 임의로 입력해서 360도 회전하는 최적의 시간 값을 찾습니다. 한번 스케치를 실행할 때마다 시리얼출력화면에 _setanglefactorCW 값을 표시합니다. 가장 최적의 360도 회전조건에서 출력된 값을 기록하고 사용하면 됩니다.

그림8.3 : 로봇의 시계방향 회전동안의 보정계수 출력

반시계방향 회전에서 bool 변수를 false로 입력하고, 전역변수 duration 값을 임의로 입력해서 360도 회전하는 최적의 시간 값을 찾습니다. 한번 스케치를 실행할 때마다 시리얼출력화면에 _setanglefactorCCW 값을 표시합니다. 가장 최적의 360도 회전조건에서 출력된 값을 기록하고 사용하면 됩니다.

그림8.4 : 로봇의 반시계방향 회전동안의 보정계수 출력

그림8.3와 그림8.4의 결과로 하위 클래스 calibrateRotation() 메소드 보정을 완료한 다음에는 로봇의 회전각도를 매개변수로 사용해서 원하는 각도만큼 제자리회전시킬 수 있습니다. 보정 실습이 정상적으로 이루어졌다면, 시계방향 회전과 반시계방향 회전에서 큰 차이를 보이지 않고 원하는 각도만큼 회전해야 합니다.

아래 예제 스케치는 로봇을 시계방향으로 360도 회전하고 2초 동안 기다린 후 반시계방향으로 360도 회전시키는 동작을 반복합니다. 로봇이 시계방향 회전하려면 ratateAbot() 메소드 매개변수를 양수로 입력하고, 반시계방향으로 회전하려면 음수를 입력하면 됩니다.

Ex8_6_enhanced_abot_rotateinplace.ino

```
#include "EnhancedSelfAbot.h"

const byte servoLeftPin = 255;
const byte servoRightPin = 255;
EnhancedSelfAbot abot(servoLeftPin, servoRightPin);

void setup() {
    Serial.begin(9600); // Start serial communication for debugging
    abot.setup();
    abot.servoAttachPins(13, 12); // Replace with actual servo pins

    abot.setAngleFactorCW(10.97);
    abot.setAngleFactorCCW(12.22);
}

void loop() {
    abot.rotateAbot(360); // Rotate robot by 90 degrees
    delay(2000); // Wait for 2 seconds

    abot.rotateAbot(-360); // Rotate robot by -90 degrees (opposite direction)
    delay(2000); // Wait for 2 seconds
}
```

위의 calibrateRotation() 메소드 기능은 추가 센서부품을 사용하지 않고 보정을 완료하기 위한 매우 단순화된 코드형태입니다.

아래 코드로 calibration 보정을 시작하려면, 자이로스코프나 나침반처럼 이와 동등한 기능을 수행하는 센서를 추가로 사용해야만 작업을 자동화할 수 있습니다. 아래 예시 코드를 참고해보세요. 각도 데이터를 제공하는 센서 신호를 처리할 getOrientation() 메소드를 추가로 정의해야 합니다.

```
void EnhancedSelfAbot::calibrateRotation() {
    unsigned long startTime = millis(); // Start time
    float startOrientation = this->getOrientation(); // Get starting orientation
    this->servoSpeed(100, -100); // Start rotation

    float currentOrientation;
    do {
        currentOrientation = this->getOrientation();
        delay(10); // Small delay to prevent too frequent checks
```

```
    } while (abs(currentOrientation - startOrientation) < 360.0);
    // Rotate until a full 360-degree turn is completed

    this->servoSpeed(0, 0); // Stop rotation
    unsigned long endTime = millis(); // End time

    // Calculate time taken for a 360 degree rotation
    unsigned long timeTaken = endTime - startTime;

    _setanglefactor = timeTaken / 360.0;
}
```

소개한 코드는 로봇이 스스로 회전하는 각도를 센서로 직접 측정하고, 360도를 회전하는 순간 정지하게 됩니다. 그리고 로봇이 정지한 순간까지의 시간을 각도로 표현할 수 있는 각도 변환 계수를 _setanglefactor 에 저장해서, 로봇의 rotateAbot() 메소드는 회전각도 매개변수를 사용해서 제자리 회전할 수 있습니다. 정상적인 calibration 보정을 위해서 코드를 한번 실행하기만 하면 된다는 의미입니다. 이런 작업은 주로 setup 함수에서 실행하게 됩니다.

8.2.6 적외선 센서로 전방 물체탐지하고 로봇방향 정렬하기

두 개의 적외선센서를 아두이노로봇에 설치하고 사용하는 경우, 알려진 방식은 로봇의 좌측과 우측 전방장애물을 각각 감지하고 ‘0’ 또는 ‘1’ 의 디지털신호 감지용으로 사용하는 것입니다. 또 다른 사용방식은 적외선센서를 사용해서 전방물체와의 거리를 측정하고 거리에 따라 로봇을 다양하게 제어하는 용도로 사용하는 것입니다.

실습에서 사용하는 적외선센서는 고급센서가 아니기 때문에 절대거리 측정 정확도를 활용하기보다는 상대거리를 추정하는 용도로 사용할 수 있습니다. 특히, 초음파 센서와 다르게 적외선 광은 물체의 색상(예를 들어 흰색과 검정색)에 따른 거리 오차, 물체의 재질에 따른 거리 오차, 주변 광의 환경조건에 따른 오차를 포함하기 때문에 이런 점들이 최소화되는 조건에서 실습을 진행할 필요가 있습니다.

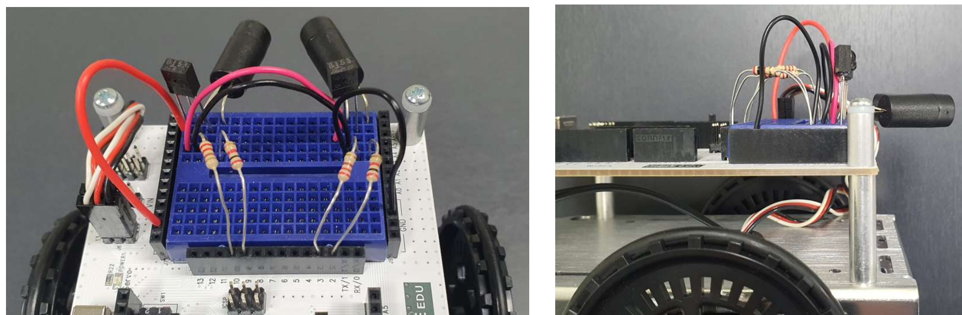


그림8.5 : 두 개의 적외선센서가 전방 물체에 초점을 맞추는 방법

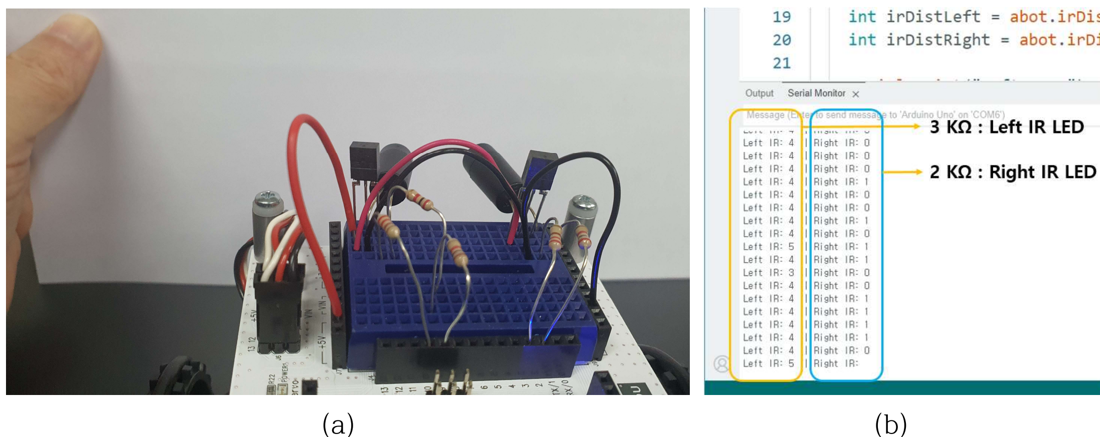
그리고 표7.2에서 설명한바와 같이 적외선 센서는 초음파 센서와 다르게 시야각이 더 좁기 때

문에 센서 바로 앞에 있는 물체를 보다 정확하게 타겟팅할 수 있습니다. 두 개의 적외선 지향 방향을 중심 안쪽으로 향하도록 만들어서 전방의 물체 감지에 유리하도록 만들 수 있습니다.

이제 실습의 제한요소를 충분히 반영하면서, 2개의 적외선 센서 배치를 그림8.5와 같이 로봇에 구성하고 전방물체와의 거리를 탐지해서, 로봇의 자율주행을 제어하는 실습을 진행합니다.

우리의 실습 논리구성은 **첫 번째** 로봇이 전진하다가 전방 물체와의 거리가 줄어들면 속도를 줄이고 한계거리에서 멈추는 동작입니다. **두 번째** 로봇이 멈춘 동작에서는 전방기준 -60도에서 +60도 사이를 제자리회전으로 스캔 탐색하고 가장 장애물이 없는 열려있는 방향으로 로봇 방향을 정렬하는 동작입니다.

두 번째 로봇동작은 제6장 6절에서 실습했던 'find object 프로젝트'와는 반대되는 개념을 코드에 적용하는 것입니다. 앞장의 실습이 정지상태에서 주변을 제자리회전하면서 가장 가까이 놓여있는 물체를 감지하고 그 물체의 방향으로 로봇의 방향을 정렬하는 것이었다면, 지금은 가장 열린 공간방향(물체가 가장 멀리 놓여있는 공간방향)으로 로봇의 방향을 정렬하는 것입니다.



(a) (b)

그림8.6 : 적외선 센서 로봇의 흰색A4용지 장애물 (a) 탐지 장면 (b) 시리얼출력

적외선센서의 흰색 장애물 감지 시리얼출력 값을 변화시키려면, 적외선 광을 출력하는 LED의 전류크기를 조절할 수 있습니다. 기존 실습에서는 디지털핀 2번과 9번에 2kΩ 저항이 사용되었습니다. 지금 실습에서는 더 가까운 거리에서 출력값의 크기 변화를 만들기 위하여 왼쪽 LED회로에 2kΩ 저항과 1kΩ 저항이 직렬이 되도록 전체 저항의 크기를 증가시켰습니다.

그림8.6(b)의 결과는 Left IR의 출력값이 Right IR의 출력값보다 더 큰 값으로 출력되어서, 지금보다 A4용지가 더 가까워지는 것도 식별할 수 있는 조건이 됩니다. 이제 우측의 LED회로도 3kΩ 저항 조건으로 변경합니다.

그리고 두 개의 적외선 센서 거리측정 정밀도보다는 상대적 거리변화를 잘 처리할 수 있기 위해서 평균값보다는 두 개의 출력 값을 합산해서 전방 물체와의 거리변수로 사용할 예정입니다. 아래 예제 스케치는 전방 물체와 가까워지면, 로봇이 정상주행으로 전진하다가 점차 속도

를 줄여서 정지상태에 이르는 목적의 코드입니다. 앞에서 살펴본 첫 번째 논리를 구현한 스케치입니다.

Ex8.7_enhanced_irdistance_speedcontrol_infrontofObstacle.ino

```
#include "EnhancedSelfAbot.h"

const byte servoLeftPin = 255;
const byte servoRightPin = 255;
EnhancedSelfAbot abot(servoLeftPin, servoRightPin);

void setup() {
    Serial.begin(9600);
    abot.setup();
    abot.servoAttachPins(13, 12);

    abot.setDeviationFactor(-0.05);
}

void loop() {
    byte irLeftled = 9;
    byte irLeftreceiver = 10;
    byte irRightled = 2;
    byte irRightreceiver = 3;

    int irDistLeft = abot.irDistance(irLeftled, irLeftreceiver);
    int irDistRight = abot.irDistance(irRightled, irRightreceiver);

    int totalIRValue = irDistLeft + irDistRight;

    Serial.print("Left IR: ");
    Serial.print(irDistLeft);
    Serial.print(" || Right IR: ");
    Serial.println(irDistRight);
    delay(50);

    if (totalIRValue > 9) {
        // If the total IR value is more than 9, run the robot forward normally
        abot.gradualServoSpeed(100, -100);
    } else if (totalIRValue >= 1 && totalIRValue <= 9) {
        // If the total IR value is between 5 and 9, reduce the speed gradually
        // Adjust the speed reduction factor as per your requirement
        int speed = map(totalIRValue, 1, 9, 0, 100);
        abot.gradualServoSpeed(speed, -speed);
    } else {
        // If the total IR value is less than 5, stop the robot
        abot.servoSpeed(0, 0);
    }

    delay(20);
}
```

이제 두 번째 논리를 추가할 차례입니다. 두 번째 논리의 핵심은 로봇이 정지상태일 때, -60도에서 +60도 각도까지 제자리회전으로 스캔한 후 가장 멀리까지 장애물이 존재하지 않거나 가장 멀리 장애물이 존재하는 방향으로 로봇을 정렬시키는 것입니다.

이런 목적의 로봇동작은 Ex8.7 예제의 내용중 loop() 함수내 else 조건문에서 추가될 것입니

다. 아래 코드는 로봇을 먼저 -70도 각도로 회전시킨 다음, 10도씩 회전시키면서 전방물체와의 거리를 측정하고 가장 거리가 먼 물체가 탐색되는 각도를 동시에 기록합니다.

Ex8.8_enhanced_irdistanceScan_stoppedPosition.ino

```
#include "EnhancedSelfAbot.h"

const byte servoLeftPin = 255;
const byte servoRightPin = 255;
EnhancedSelfAbot abot(servoLeftPin, servoRightPin);

void setup() {
     Serial.begin(9600);
     abot.setup();
     abot.servoAttachPins(13, 12); // Replace with actual servo pins

     abot.setDeviationFactor(-0.05);
     abot.setAngleFactorCW(10.97);
     abot.setAngleFactorCCW(12.22);
}

void loop() {
     byte irLeftled = 9;
     byte irLeftreceiver = 10;
     byte irRightled = 2;
     byte irRightreceiver = 3;

     int irDistLeft = abot.irDistance(irLeftled, irLeftreceiver);
     int irDistRight = abot.irDistance(irRightled, irRightreceiver);

     int totalIRValue = irDistLeft + irDistRight;

     Serial.print("Left IR: ");
     Serial.print(irDistLeft);
     Serial.print(" || Right IR: ");
     Serial.println(irDistRight);

     if (totalIRValue > 9) {
          // If the total IR value is more than 9, run the robot forward normally
          abot.gradualServoSpeed(100, -100);
     } else if (totalIRValue >= 1 && totalIRValue <= 9) {
          // If the total IR value is between 5 and 9, reduce the speed gradually
          // Adjust the speed reduction factor as per your requirement
          int speed = map(totalIRValue, 1, 9, 0, 100);
          abot.gradualServoSpeed(speed, -speed);
     } else {
          // If the total IR value is less than 5, stop the robot
          abot.servoSpeed(0, 0);
          delay(1000);

          int maxdistance = 0;
          int directionOfangle = 0;

          int currentAngle = -70;
          abot.rotateAbot(currentAngle);
          delay(200);

          for (int i = 0; i <= 13; i++) {
               if (i != 0) {
                    abot.rotateAbot(10);
                    currentAngle += 10;
                    delay(200);
               }
```

```
            int irDistLeft = abot.irDistance(irLeftled, irLeftreceiver);
            int irDistRight = abot.irDistance(irRightled, irRightreceiver);
            int totalIRValue = irDistLeft + irDistRight;

            if (totalIRValue > maxdistance) {
                maxdistance = totalIRValue;
                directionOfangle = currentAngle;
            }
        }
        abot.rotateAbot(directionOfangle - currentAngle);
    }
    delay(20);
}
```

전체 예정된 각도영역을 스캔 탐색한 다음, 가장 멀리 물체가 탐색된 각도로 로봇의 방향을 정렬하고 for 반복문을 종료합니다. 다시 loop 반복을 실행하면, 전방물체가 가까이에 존재하지 않기 때문에 로봇은 다시 정상적인 전진 동작을 시작할 수 있습니다. 완성된 스케치 예제를 아래 Ex8.8에 소개합니다. 스케치를 업로드해서 실습해보세요.

코드가 정상적으로 실행된다면, 전방의 물체를 탐색한 다음 가장 멀리 물체가 놓여있는 방향으로 로봇이 정렬합니다. 두 개의 적외선 센서신호를 합쳐서 사용하기 때문에 한 개의 센서가 조금 오작동하거나 두 개의 센서 사이에 값의 편차가 있어도 전방의 물체를 일관성있게 탐지하는데 도움이 됩니다. 복잡한 미로를 탐색하고 탈출하는 로봇을 만들어 보세요.

만약, 센서가 물체를 감지하는 허용범위를 벗어난 방향이 탐색영역내에서 하나이상 존재하는 경우, 맨 처음 방향으로 향할까요 아니면 맨 마지막 방향으로 향할까요? 예상되는 또 다른 문제가 있다면 어떤 것일까요? 당신의 상상이 로봇을 더 지능적이고 고급스럽게 만들 수 있습니다. 전방의 물체를 감시하는 로봇을 상상하면서 메소드를 더 세밀하게 구현해보세요!

8.2.7 초음파센서로 레이더 화면 출력하기

초음파센서를 7번 핀에 연결하고, 제7장 2절의 실습과 동일한 레이더화면을 프로세싱 프로그램과 연결해서 모니터에 표시할 수 있습니다. 아래 예제를 업로드해서 동일한 실습결과를 경험해보세요.

Ex8.9_enhanced_ultrasonic_radar_scan.ino 예제를 업로드하세요.

```
#include "EnhancedSelfAbot.h"

EnhancedSelfAbot abot;
void setup() {
    Serial.begin(9600);
    abot.setup();
    abot.servoAttachAngle(10);
    abot.setUltrasonicSensorPin(7, 7);
}
```

```
void loop() {
      abot.ultrasonicRadarData();
      delay(2000);
}
```

abot.setUltrasonicSensorPin(7, 7); 코드는 초음파센서의 triggerPin과 echoPin 번호를 설정하기 위한 setter 함수입니다. 지금 코드에서 사용중인 매개변수는 패럴렉스 사의 3핀 초음파센서를 사용한 예제이어서 동일한 핀번호를 사용합니다. 그리고 abot.ultrasonicRadarData() 코드가 초음파센서를 회전시키면서 거리를 탐지하고, 시리얼 포트로 데이터를 전송하는 코드를 구현합니다.

8.2.8 초음파센서로 탐색 로봇 주행하기

초음파센서로 로봇이 전방의 물체를 탐색하면서 주행하는 로봇예제 Ex7.5를 제7장에서 소개하였습니다. 여기에서는 7장 예제와 동일한 코드를 EnhancedSelfAbot 라이브러리를 사용해서 조금 더 섬세하게 동작하는 로봇으로 변경하여 실습합니다.

Ex8.10_enhanced_ultrasonic_navigating_driveRobot.ino 파일을 업로드하세요.

```
#include "EnhancedSelfAbot.h"

#define SAFE_DISTANCE 20 // safe distance (cm)
#define INVALID_PIN 255
const byte servoLeftPin = INVALID_PIN;
const byte servoRightPin = INVALID_PIN;

EnhancedSelfAbot abot(200, 20, 40);

void setup() {
      Serial.begin(9600);
      abot.setup();
      abot.servoAttachPins(13, 12); // Replace with actual servo pins
      abot.servoAttachAngle(10);

      abot.setUltrasonicSensorPin(7, 7);

      abot.setDeviationFactor(-0.05);
      abot.setAngleFactorCW(10.97);
      abot.setAngleFactorCCW(12.22);
}

void loop() {
   abot.gradualServoSpeed(80, -80); // 로봇 전진 시작

   scanInDirection(-30, 30); // -30도에서 30도까지 스캔
   delay(100); // 스캔 방향 변경 전 대기
   scanInDirection(30, -30); // 30도에서 -30도까지 스캔

   delay(100); // 루프의 반복 속도 조절을 위한 딜레이
}
```

```
void scanInDirection(int startOffset, int endOffset) {
  int step = startOffset < endOffset ? 15 : -15; // 스캔 방향 결정

  for(int offset = startOffset; step > 0 ? offset <= endOffset : offset >= endOffset;
offset += step) {
      abot.servoAngle(90 + offset); // 서보 모터를 해당 각도로 조정
      delay(30); // 서보 모터가 이동하고 안정화되는데 필요한 시간
      int distance = abot.calculateDistance(); // 현재 각도에서의 거리 측정
      if(distance <= SAFE_DISTANCE) {
        abot.gradualServoSpeed(20, -20); // 장애물이 감지되면 로봇 정지
        abot.detectAndProcessSafeArea(); // 상세 탐색 및 회피 동작 수행
        abot.gradualServoSpeed(80, -80); // 회피 동작 후 다시 전진
        break;
      }
  }
}
```

위의 실습예제는 전방의 물체가 나타나면 속도를 줄여서 탐색하고, 전방의 물체가 없는 방향을 찾으면 다시 정상속도로 주행하는 로봇으로 실습할 수 있습니다.

위 실습예제 Ex8.10을 성공적으로 수행하려면, 제8장의 앞부분 예제들을 모두 먼저 수행해야 합니다. 앞 예제들은 아두이노로봇의 기계적 움직임 편차들을 상쇄시키는 보정계수들을 찾는 예제들이 포함됩니다. 로봇이 속도에 무관하게 직선에 가깝게 주행하거나, 점진적인 로봇속도를 구현하거나, 제자리회전을 할 때 동일한 비율로 시계방향 회전 또는 반시계방향 회전을 가능하게 합니다.

예제 Ex8.10에서 setup() 함수에는 각자의 로봇이 갖는 고유한 보정계수들을 찾아서 적용해야 합니다.

```
abot.setDeviationFactor(-0.05);
abot.setAngleFactorCW(10.97);
abot.setAngleFactorCCW(12.22);
```

코드에서 적용된 값들은 실습자의 고유한 값이어야 합니다. 그리고 초음파센서를 3핀으로 사용하는 경우 7번핀을 사용하는 경우의 코드를 나타냅니다.

```
abot.setUltrasonicSensorPin(7, 7);
```

만약 여러분은 4핀 초음파센서를 사용한다면, 첫 번째 매개변수는 trigger핀으로 두 번째 매개변수는 echo핀으로 핀 번호를 적용하면 됩니다.

생성자에서 초음파센서를 탐색하기 위한 몇가지 변수를 지정할 수 있습니다. 아래 코드가 이 실습에서 사용할 고유한 변수들인데 자세히 설명합니다.

```
EnhancedSelfAbot abot(200, 20, 40);
```

생성자로 3개의 매개변수를 사용하는 객체를 선언합니다. 첫 번째 매개변수는 초음파센서가 가장 멀리 측정할 수 있는 거리로 200 cm 조건을 설정했습니다. 이 값은 개략적인 값이므로 실습자가 직접 테스트해서 더 실제에 가까운 값을 사용해도 됩니다.

두 번째 매개변수는 초음파센서 측정거리가 20 cm 이하의 조건으로 들어올 때 안전거리 이

하임을 판단하는 거리 값입니다. 실습자의 실습환경은 로봇의 주행속도와 센서의 탐색속도 그리고 주변 장애물의 복잡도를 고려해서 가장 최적의 값으로 변경할 수 있습니다.
※ 참고로 두 번째 매개변수 값은 #define SAFE_DISTANCE 20 코드와 동일한 의미입니다. .ino 코드에서도 안전거리 조건이 사용되기 때문에, 사용된 것입니다.

세 번째 매개변수 값은 초음파센서가 0도에서 180도까지 장애물을 탐색할 때, 안전거리 밖으로 인식되는 각도구간이 40도 이상인 조건을 의미합니다. 이 값의 의미는 안전거리 조건과 안전각도 구간을 조합하면 로봇이 충분히 선택된 방향으로 지나갈 수 있음을 의미합니다. 이 값 역시 사용자의 판단에 따라 다양하게 조정될 수 있습니다.

이상의 코드설계 개념을 참고해서 다양하게 실습해보기 바랍니다.

.ino 예제에서 사용된 scanInDirection() 함수를 살펴보겠습니다.

```
scanInDirection(-30, 30);
delay(100); // 스캔 방향 변경 전 대기
scanInDirection(30, -30);
delay(100);
```

코드에서 스캔범위는 -30에서 30까지 또는 30에서 -30까지 스캔합니다. 함수내에 구현된 코드는 step을 15로 하고, 삼원연산자를 사용해서 15를 더할지 뺄지를 판단해서 적용합니다.
for 반복문에서는 step 값(+15 또는 -15)을 조건에 맞게 계속 더하면서 반복을 실행합니다.

```
int step = startOffset < endOffset ? 15 : -15; // 스캔 방향 결정
for(int offset = startOffset; step > 0 ? offset <= endOffset : offset >= endOffset;
offset += step) {
    abot.servoAngle(90 + offset); // 서보 모터를 해당 각도로 조정
    delay(30); // 서보 모터가 이동하고 안정화되는데 필요한 시간
```

위의 코드 작용은 아래의 예시코드와 비교해서 더 유연하게 동작합니다.

```
int angles[] = {-30, -15, 0, 15, 30}; // 각도 오프셋을 배열로 정의
for(int i = 0; i < 5; i++) { // 배열의 각 요소에 대해 반복
  int angleOffset = angles[i];
  abot.servoAngle(90 + angleOffset); // 서보 모터를 해당 각도로 조정
  delay(100); // 서보 모터가 이동하고 안정화되는데 필요한 시간
```

초음파센서가 전방의 물체를 감지한 거리를 측정하고 기록합니다.

```
abot.calculateDistance()
```

if 조건문에서는 7장 예제의 조건과 다르게, 전방 안전거리 이내에 장애물이 감지되는 경우 즉시 멈추는 것이 아니라 20의 속도로 감속합니다. 그 다음 주변을 센서로 탐색하고 안전한 주행방향이 결정되면 다시 정상속도 80으로 주행합니다.

```
if(distance <= SAFE_DISTANCE) {
  abot.gradualServoSpeed(20, -20); // 장애물이 감지되면 로봇 정지
  abot.detectAndProcessSafeArea(); // 상세 탐색 및 회피 동작 수행
  abot.gradualServoSpeed(80, -80); // 회피 동작 후 다시 전진
```

```
        break; // 하나라도 조건을 만족하면 더 이상의 감지는 필요 없으므로 반복문 탈출
    }
```

라이브러리에서 사용된 메소드의 자세한 설명은 생략하겠습니다. 충분한 이해가 되지 않으면, 코드 작용원리를 챗GPT에게 질문하기 바랍니다. 당신이 인공지능과 대화하면서 새로운 코드 사용 가능성을 찾을 수도 있습니다.

여러분의 더욱 심오한 코드 이해가 지속되기를 기원합니다.

부 록: 'EnhancedSelfAbot.zip'라이브러리 헤더와 소스 파일 (EnhancedSelfAbot.cpp 파일은 다운로드해서 확인하기 바랍니다.)

```
#ifndef ENHANCEDSELFABOT_H
#define ENHANCEDSELFABOT_H

#include "SelfAbot.h" // Include the base class header
#include "Arduino.h"
#include <Servo.h>

struct Detection {
   int angle;
   unsigned int distance;
};

class EnhancedSelfAbot : public SelfAbot {
public:
      EnhancedSelfAbot();
      EnhancedSelfAbot(byte servoLeftPin, byte servoRightPin);
      EnhancedSelfAbot(unsigned int maxDistance, unsigned int safeDistance, int angularRange);

            static const int MAX_DETECTIONS = 40;

            void resetDeviationFactor();
            float getDeviationFactor() const;
            void setDeviationFactor(float deviationFactor);
            void calibratedeviationFactor(float deviatedistance);

            void driveStraight(int leftSpeed, int rightSpeed);
      void gradualServoSpeed(int targetLeftSpeed, int targetRightSpeed);
      void lightFollowing(unsigned long leftrcTime, unsigned long rightrcTime);
      void irNavigationdriving(int irLeft, int irRight);

      void rotateAbot(int angle);
      void calibrateRotation(int duration, bool clockwise);

      // Setters
      void setAngleFactorCW(float angleFactorCW);
      void setAngleFactorCCW(float angleFactorCCW);

      // Getters
      float getAngleFactorCW() const;
      float getAngleFactorCCW() const;

      void setUltrasonicSensorPin(byte triggerPin, byte echoPin);
      int calculateDistance();
      void ultrasonicRadarData();

      void detectAndProcessSafeArea();
      bool scanAndProcessAngle(int angle, bool isDescending);
      int findSafeAreaCenterAngle(bool isDescending);
      void setTravelDirection(int angle);

private:
            float _deviationFactor;
            float _setanglefactorCW, _setanglefactorCCW;
            int _currentLeftSpeed;    // Current speed for left servo as a class member
      int _currentRightSpeed;  // Current speed for right servo as a class member

      Detection _detections[MAX_DETECTIONS];
      int _detectionCount;
      int _notDetectedAngularRange;

      byte _triggerPin, _echoPin;
      unsigned int _maxDistance;
      unsigned int _safeDistance;
};

#endif // ENHANCEDSELFABOT_H
```

제 9 장 로봇의 고급 이동과 제어

9.1 무선 로봇의 고급기능 구현하기
9.2 로봇 동작중 예상치 못한 상황관리

아두이노로봇에 무선 기능을 추가하면 로봇의 기능과 애플리케이션이 크게 향상됩니다. 무선 기능이 왜 유익하고 어떤 이점이 있는지 살펴봅시다.

1) 이동성 및 작전 범위 증가:
무제한 움직임: 무선 통신은 로봇을 물리적 전선의 제약으로부터 해방시켜 무제한적인 움직임을 가능하게 합니다. 이는 다양한 환경에서의 탐색, 탐색 또는 상호 작용과 관련된 작업에 특히 중요합니다.
확장된 작동 범위: 로봇은 전선이 닿지 않는 거리에서도 작동할 수 있어 프로젝트와 응용 분야의 범위가 확장됩니다.

2) 원격 모니터링 및 제어:
실시간 데이터: 무선 통신을 통해 실시간 데이터 전송이 가능합니다. 이는 원격 위치에서 센서 판독값, 로봇 상태 또는 환경 조건을 모니터링할 수 있음을 의미합니다.

원거리 제어: 원격 워크스테이션, 스마트폰 또는 인터넷을 통해 로봇에 명령을 보낼 수 있으므로 필요할 때 원격 제어 및 개입이 가능합니다.

3) 확장성 및 네트워크 통합:
간편한 확장: 무선 시스템은 유선 시스템보다 더 쉽게 확장할 수 있습니다. 네트워크에 더 많은 로봇이나 센서를 추가하는데 복잡한 배선이 필요하지 않습니다. 기존 시스템과 무선으로 통신하도록 구성하기만 하면 됩니다.
대규모 시스템으로의 통합: 무선 로봇은 사물 인터넷(IoT)을 포함한 대규모 네트워크에 통합되어 다른 스마트 장치, 클라우드 기반 서비스 또는 데이터 분석 플랫폼과 상호 작용할 수 있습니다. 특히, 아두이노는 Wi-Fi, Bluetooth, Zigbee(XBee), LoRa 및 NFC와 같은 다양한 무선 통신 프로토콜을 지원하며, 각각은 다양한 애플리케이션에 적합한 고유한 기능을 제공합니다.

4) 협업을 통해 향상된 기능:
군집 로봇공학: 무선 통신을 통해 여러 로봇이 조화로운 그룹 또는 군집으로 함께 작업할 수 있습니다. 이러한 협업은 영역 매핑, 수색 및 구조, 물체의 집단 운송과 같은 작업에서 복잡한 그룹 행동과 효율성으로 이어질 수 있습니다.
데이터 공유: 로봇은 센서 데이터나 계산 결과를 서로 공유하여 더 많은 정보를 바탕으로 의사 결정을 내리고 적응적인 행동을 할 수 있습니다. 무선 시스템은 환경 모니터링, 스마트 홈 또는 산업 자동화와 같은 애플리케이션에 필수적인 실시간 데이터 수집 및 제어를 가능하게 합니다.

5) 유연성 및 적응성:
신속한 배포: 광범위한 인프라를 구축할 필요 없이 다양한 환경에 무선 로봇을 빠르게 배포할 수 있습니다.
환경에 대한 적응: 무선 로봇은 환경이나 중앙 제어 시스템의 무선 피드백을 기반으로 작동을 조정하여 동적이거나 예측할 수 없는 조건에 적합하게 만듭니다.

6) 향상된 사용자 상호 작용:
사용자 친화적인 인터페이스: 무선 통신을 통해 스마트폰, 태블릿, 컴퓨터와 같은 장치에서 사용자 친화적인 인터페이스를 사용하여 로봇을 제어할 수 있습니다.
교육 및 엔터테인먼트 가치: 무선 로봇은 교육 및 엔터테인먼트 분야에서 큰 잠재력을 갖고 있어 로봇 대회, 대화형 예술 설치 또는 교육 시연과 같은 대화형 및 참여형 활동을 허용합니다.

무선 통신은 다양한 가능성을 열어주지만 통신 범위, 배터리 수명, 무선 프로토콜 보안, 다른 무선 장치의 잠재적인 간섭 등의 요소를 고려하는 것이 중요합니다

9.1 무선 로봇의 고급기능 구현하기

이 책에서 실습용으로 사용중인 'neo-아두이노로봇카'제품은 무선통신을 위해 하드웨어 옵션이 설계되어 있습니다. 무선기능을 채택하려면, 셀룰러통신, 블루투스 통신, 엑스비/지그비 통신, 와이파이(Wi-Fi)통신을 위한 부품을 아두이노로봇의 슬롯에 쉽게 장착하고, 하드웨어 토글스위치를 사용하면 쉽게 무선통신 조건을 만들 수 있습니다. 'neo-아두이노로봇카'제품에 부합하는 무선 안테나 종류와 상세한 코드 작성방법 그리고 토글 스위치의 사용법은 프라이봇을 참고할 수 있습니다.

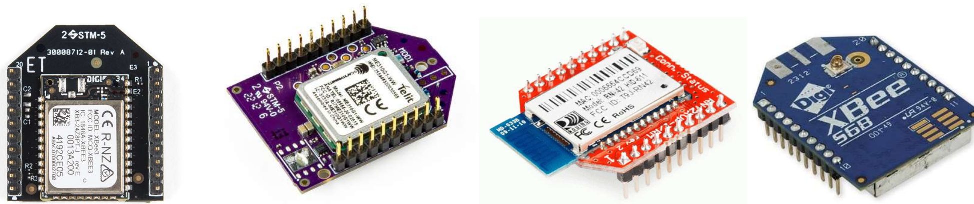


그림9.1 : 엑스비 핀 형태의 안테나 종류들

특히, 근거리 1:N 통신을 위한 통신수단으로 엑스비 통신이 많이 알려져 있는데 실습 예제에서는 엑스비 시리즈1 안테나를 사용하지만, 최근에는 엑스비 시리즈3이 출시되어 있습니다. 통신 성능이 더 많이 좋아졌다는 것을 의미합니다. 추후에 기회가 되면 근거리 무선통신 수단에 대해서도 다루도록 하겠습니다.

본 실습의 실습도구인 아두이노로봇 '프라이비(Fribee EDU)'보드에서 토글스위치를 사용하면, 직렬통신을 위한 Tx/Rx핀(0, 1번)을 USB통신과 무선통신을 위해 선택해서 사용할 수 있지만 동시에 모두 사용할 수는 없습니다.

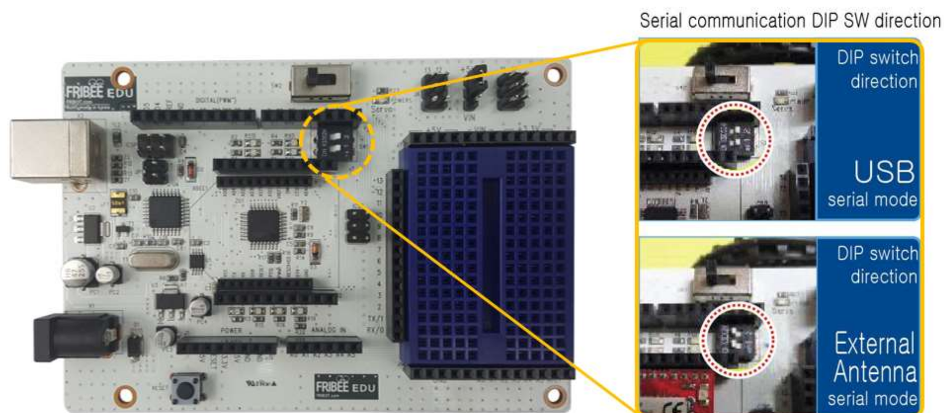


그림9.2 : 프라이비 보드를 이용한 H/W 시리얼통신용 토글(DIP)스위치

가장 쉬운 사용방법은 USB 통신으로 코드를 업로드하고, 화면으로 시리얼 데이터값들을 확인한 다음 토글스위치를 무선통신으로 전환하면 조금 전 USB 통신의 시리얼데이터 값들이 무선

으로 동일하게 전송되는 것을 보장합니다. *(물론 안테나를 위한 초기설정이나 조건설정은 사전에 준비되어 있어야 합니다.)*

데이터를 USB를 통해 통신할지 아니면 무선으로 통신할지 여부는 토글 스위치의 위치에 따라 선택되며, 그림9.2를 참고하면 됩니다.

아두이노(또는 아두이노호환 프라이비)보드로 무선통신을 위한 시리얼통신 수단은 두가지 방법이 가능합니다. 한가지는 조금 전 소개한 프라이비 하드웨어 시리얼을 사용하는 방법과 아두이노 소프트웨어 시리얼을 사용하는 방법입니다. 두가지 방법에 대한 차이점을 살펴보겠습니다.

하드웨어 시리얼(Tx/Rx 핀으로 0/1 핀 사용)

장점 : 신뢰성은 하드웨어 직렬 통신이 일반적으로 소프트웨어 직렬에 비해 더 안정적이며 더 높은 전송 속도를 처리할 수 있습니다. 이는 하드웨어가 직렬 통신용으로 특별히 설계되었기 때문입니다. 하드웨어가 직렬 통신을 처리하므로 CPU 오버헤드가 낮아져 프로세서가 다른 작업을 처리하는데 유리합니다. 그리고 인터럽트 처리에 있어서 하드웨어 직렬은 인터럽트를 더 잘 처리할 수 있으므로 보다 원활한 멀티태스킹이 가능합니다.

단점 : 하드웨어 직렬(arduino의 핀 0 및 1)을 사용하면 일반 I/O에 사용할 수 없으며 특히 코드 업로드 또는 직렬 모니터링의 경우 USB 통신으로 전환해야 작업이 진행됩니다. 그리고 하드웨어 직렬 통신 동안 CPU가 전송이 완료될 때까지 기다리며 제대로 관리하지 않음으로서 사이클을 낭비할 가능성이 있음을 의미합니다.

소프트웨어 시리얼(Tx/Rx 핀으로 0/1 이외의 핀 사용)

장점 : 하드웨어 직렬 핀을 다른 작업이나 USB 통신에 사용할 수 있도록 남겨두고 직렬 통신에 모든 디지털 핀을 사용할 수 있는 유연성이 있습니다. 특히, 하드웨어 직렬 포트가 하나만 있는 보드에 여러 직렬 연결을 생성할 수 있어서 다중 직렬 포트로 여러 직렬 장치와 통신하는 데 유용합니다.

단점 : 소프트웨어 직렬은 특히 덜 강력한 마이크로 컨트롤러에서 하드웨어 직렬만큼 높은 전송 속도를 안정적으로 처리할 수 없어서 제한된 전송속도를 구현하는 경우가 많습니다. 그리고 프로세서 오버헤드로 CPU를 사용하여 직렬 통신을 에뮬레이트하므로 중요한 타이밍이나 처리 작업이 중단될 수 있습니다. 또한, 하드웨어 직렬 포트와 같은 전용 하드웨어 버퍼링이 없기 때문에 버퍼 오버런에 더 취약합니다.

무선통신을 위한 고려사항들

(1) 간섭 및 잡음: 무선 통신은 간섭과 잡음에 취약하며, 이는 타이밍 중단에 대한 허용치가 낮기 때문에 소프트웨어 직렬에서 더 문제가 될 수 있습니다.

(2) 전력 소비: 무선 통신 방법에 따라 전력 소비가 중요한 요소일 수 있으며, 하드웨어와 소프트웨어 직렬 중 하나를 선택하면 이에 영향을 미칠 수 있습니다. 특히 소프트웨어 직렬 통

신 중에 CPU를 더 활성화해야 하는 경우에는 더욱 그렇습니다.

(3) 복잡성: 무선 통신을 구현하면 특히 잠재적으로 신뢰할 수 없는 매체를 통해 안정적인 데이터 전송을 보장한다는 측면에서 복잡성이 추가됩니다. 이는 오류 수정이나 버퍼 관리의 필요성과 같은 애플리케이션의 특정 요구 사항을 기반으로 하드웨어와 소프트웨어 직렬 간의 선택에 영향을 미칠 수 있습니다.

그리고 무선통신을 위해서 추가로 더 고려해야할 사항은 다음과 같습니다.
아두이노로봇의 통신기능 구현이외에 또 다른 상대방 기기(PC 또는 스마트폰, 또 다른 아두이노로봇 등)의 통신기능을 구현하는 것에 관한 것입니다. 안테나의 초기설정과 조건설정 그리고 스마트폰의 경우 통신을 위한 앱을 준비하는 것 등입니다.

특히 많은 학생들이 동시에 동일 공간내에서 무선통신 실습을 시작하는 경우, 학생 각각의 무선 안테나 아이디(ID: 식별자)를 다루는 작업이 필요할 수도 있습니다.

아두이노로봇의 무선 통신을 위한 '**하드웨어 시리얼**' 예제코드는 다음과 같습니다.
하드웨어 시리얼 통신 예제를 사용하면 USB케이블이 연결된 상태에서 키보드의 입력데이터를 아두이노에 전달하거나, 아두이노에서 전송하는 데이터를 컴퓨터 화면에서 출력할 수 있습니다.

```
void setup() {
    Serial.begin(9600);
}

void loop() {
    if (Serial.available() > 0) {
        // Read the incoming byte:
        char incomingByte = Serial.read();

        // Say what you got:
        Serial.print("I received: ");
        Serial.println(incomingByte, DEC);
    }
}
```

아두이노로봇의 무선 통신을 위한 '**소프트웨어 시리얼**' 예제코드는 다음과 같습니다.

```
#include <SoftwareSerial.h>
// RX and TX pins for the software serial
int rxPin = 10;
int txPin = 11;
// Initialize the software serial port
SoftwareSerial mySerial(rxPin, txPin);

void setup() {
  // Open the software serial port at 9600 bps
  mySerial.begin(9600);
```

```
    // Start the hardware serial port for debugging
    Serial.begin(9600);
  }
  void loop() {
    // Check if data is available to read
    if (mySerial.available()) {
      char inChar = (char)mySerial.read();
      Serial.print("Received: ");
      Serial.println(inChar);
    }
    // Send a message every second
    static unsigned long lastSendTime = millis();
    if (millis() - lastSendTime > 1000) {
      mySerial.println("Hello from Arduino!");
      lastSendTime = millis();
    }
  }
```

9.1.1 적외선 센서로 탐색한 물체를 레이더 출력화면으로 표시

본 실습화면은 앞장 “8.2.7 초음파센서로 레이더 화면 출력하기” 실습을 무선으로 재현한 것입니다. 예제 “Ex8.9_enhanced_ultrasonic_radar_scan.ino” 코드를 수정없이 사용하면서, 하드웨어 시리얼포트를 활용하기 위해 토글(DIP)스위치를 반대방향으로 전환하였습니다.

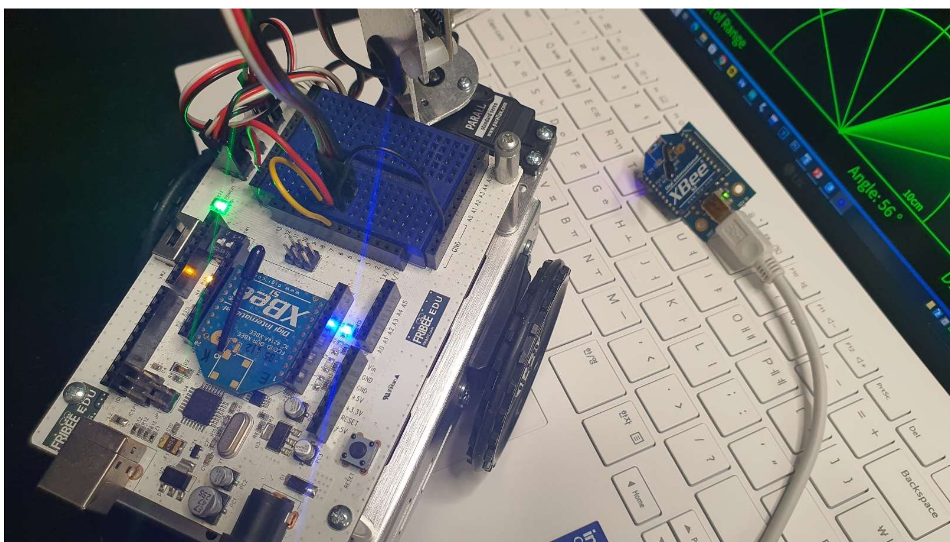


그림9.3: 초음파센서 데이터 레이더화면 무선데이터로 출력하기

우리는 아두이노를 포함해서 대부분의 마이크로컨트롤러 환경에서 유선 USB시리얼로 연결해서 코드를 작성하고, 화면출력으로 데이터를 점검하고, 무선 환경으로 변환하려면 소프트웨어 시리얼을 사용하기 위해 코드를 추가하거나 변경해야 합니다. 그런 다음 다시 변경된 코드를 테스트해야 할 필요를 종종 생각하게 됩니다.

하드웨어 시리얼을 사용하는 경우 USB시리얼 환경에서 출력화면을 점검한 결과가 최종결과가 된다는 의미입니다.

아두이노의 소프트웨어 시리얼을 구성하려면, 무선데이터를 전송할 Tx/Rx 핀을 점퍼선으로 기존의 하드웨어 시리얼 번호인 0번과 1번을 제외한 나머지 번호에 연결해서 소프트웨어 시리얼을 구성합니다.
아두이노의 하드웨어 시리얼을 구성하려면, 특별히 프리이비 보드처럼 토글(DIP)스위치를 갖추지 않더라도, 당신이 점퍼선으로 무선통신을 위한 핀(Tx/Rx)번호를 선택할 때, 0번과 1번을 사용하면 하드웨어 시리얼을 사용하는 상태가 된다는 의미입니다.

하드웨어 시리얼을 사용하면, 지금 실습에서 소개하는 것처럼 기존의 코드를 거의 수정하지 않거나 전혀 수정하지 않고 간단하게 무선통신 환경을 구축하는 의미가 있습니다.
그리고 그림9.3에서 사용된 무선통신용 안테나는 기존의 엑스비 시리즈1 안테나입니다. 지금은 시리즈1 안테나는 단종되었으며, 시리즈3 제품이 출시되고 있습니다. DIGI사의 엑스비 시리즈3은 더 우수한 성능을 가진 무선안테나로 IOT를 구축할 수 있는 도구가 될 수 있습니다.

여기서는 더 깊은 내용을 소개하지 않지만, 무선통신의 경우 여러개의 안테나 조건들을 설정해야 하는 경우는 로봇에 안테나를 부착하는 작업만으로 완료되지 않습니다. 특히, 메시 네트워크를 설정하는 경우처럼 엔드노드와 라우터와 코디네이터 등의 안테나 설정들을 모두 구성해야 하는 경우 또 다른 영역의 작업이 필요하다는 것을 의미합니다.

무선통신으로 로봇을 제어하는 더 많은 예제들은 추후에 다시 소개하겠습니다.

9.2 로봇 동작중 예상치 못한 상황관리

아두이노 로봇 운영 중 예기치 않은 상황을 관리하는 것은 특히 초보자에게 중요합니다. 로봇은 종종 동적인 환경과 상호 작용하며 예측할 수 없게 행동할 수 있습니다. 초보자가 알아야 할 예기치 않은 상황 관리에 대하여 8가지로 요약한 내용을 살펴보겠습니다.

(1) 로봇과 환경 이해
로봇 이해하기: 정상적인 조건에서 로봇이 어떻게 행동해야 하는지 이해합니다. 로봇의 능력과 한계를 잘 알아두세요.
환경 인식: 로봇이 작동하는 환경을 잘 알고 있어야 합니다. 다양한 바닥, 장애물, 조명 조건, 심지어 날씨도 성능에 영향을 줄 수 있습니다.

(2) 안전 및 오류 처리를 위한 프로그래밍
안전한 기본값: 예기치 않은 상황을 만났을 때 로봇이 안전한 상태로 멈추거나 대기 상태로 들어가도록 프로그래밍합니다.
오류 처리: 코드에 오류 처리를 구현합니다. 예를 들어, 센서가 읽기 실패할 경우 로봇이 안전하게 반응할 수 있도록 합니다.
타임아웃: 코드에 타임아웃을 사용하여 예상되는 이벤트가 지정된 시간 내에 발생하지 않으면 로봇이 기본 동작을 수행할 수 있도록 합니다.

(3) 센서 데이터 검증
센서 읽기 확인: 센서 읽기를 지속적으로 모니터링하여 이상 징후를 감지합니다. 예를 들어, 거리 센서가 갑자기 '0'이나 최대값을 읽으면 이를 잠재적 오류로 처리합니다.
중복 센서: 중요한 측정에는 중복 센서를 사용합니다. 이렇게 하면 한 센서가 실패하더라도 다른 센서에 의존할 수 있습니다.

(4) 전원 관리
배터리 수준 모니터링: 로봇이 배터리 부족을 감지하고 적절하게 반응할 수 있도록 합니다. 예를 들어, 홈 베이스로 돌아가거나 필수적이지 않은 기능을 종료합니다.
전압 조절: 전원 공급이 안정적인지 확인합니다. 전압 변동은 예측할 수 없는 행동을 유발할 수 있습니다.

(5) 기계적 안전 장치
물리적 제한: 범퍼와 같은 물리적 제한을 구현하여 충돌 시 로봇이 손상되는 것을 방지합니다.
긴급 정지: 필요한 경우 수동으로 로봇을 즉시 정지시켜서 로봇의 동작을 멈추도록 합니다.

(6) 테스트 및 디버깅
철저한 테스트: 제어된 환경에서 먼저 로봇을 테스트하고, 점차적으로 작업의 복잡성을 높여 갑니다.
디버깅: 아두이노 코드를 디버깅하는데 도움이 되는 도구를 사용하는 방법을 배웁니다. 시리

얼 모니터는 로봇에서 실시간 피드백을 얻는데 좋은 도구입니다.

(7) 외부 간섭 처리
간섭 관리: 전자기 간섭(EMI) 또는 다른 무선 장치로부터의 간섭 가능성을 인식합니다. 차폐나 주파수 변경으로 문제를 해결할 수 있습니다.

(8) 학습 및 반복 개발
지속적인 학습: 예기치 않은 사건은 학습 기회입니다. 이러한 경험을 활용하여 로봇의 설계와 코드를 개선하세요.
반복적 개발: 로봇을 반복적으로 개발합니다. 간단한 작업부터 시작하여 경험과 자신감이 쌓이면 복잡성을 추가하세요.

<u>아두이노로봇의 예외처리에 대한 제안들</u>
예외 처리는 프로그램 실행 중에 발생하는 오류 및 비정상적인 조건을 관리하는데 사용되는 프로그래밍 개념입니다. 그러나 arduino 및 기타 여러 임베디드 시스템의 맥락에서는 더 높은 수준의 프로그래밍 언어(예: C++ 또는 Java의 try-catch 블록)에서 볼 수 있는 전통적인 예외 처리를 사용할 수 없거나 사용하지 않는 경우가 많다는 점에 유의하는 것이 중요합니다. 이는 주로 마이크로컨트롤러의 메모리와 처리 리소스가 제한되어 있기 때문입니다.

아두이노로봇에 사용되는 플랫폼 arduino에는 일반적으로 단순화된 C++ 버전의 프로그래밍이 포함됩니다. 표준 C++는 예외 처리를 지원하지만 arduino 프로그래밍에서는 일반적으로 이 기능을 사용하지 않습니다. 대신 arduino 프로그래머는 다른 방법을 사용하여 오류와 예상치 못한 상황을 처리합니다. 다음은 arduino 기반 로봇에서 예외나 오류를 처리하는 방법에 대한 초보자 친화적인 설명입니다.

(1) 오류 조건 확인
예외에 의존하는 대신 잠재적인 오류 조건을 명시적으로 확인할 수 있습니다. 예를 들어, 센서가 실패하거나 잘못된 판독값을 제공하는 경우 센서 값이 합리적인 범위 내에 있는지 확인하는 코드를 작성합니다.

```
int sensorValue = analogRead(sensorPin);
if (sensorValue < MIN_VALUE || sensorValue > MAX_VALUE) {
    // Handle error: maybe set an error flag, try to read sensor again, etc.
}
```

(2) 반환 값 및 오류 코드 사용
함수는 문제가 발생했음을 나타내기 위해 특수 값이나 오류 코드를 반환할 수 있습니다. 기본 프로그램에서 이러한 반환 값을 확인하고 수행할 작업을 결정해야 합니다.

```
int result = performTask();
if (result == ERROR_CODE) {
    // Handle the error
}
```

(3) 오류 방지 상태
문제가 발생하면 로봇이 '안전한' 상태로 들어가도록 설계하세요. 예를 들어, 모터 드라이버에 오류가 발생하면 로봇은 손상이나 불안전한 동작을 방지하기 위해 모든 움직임을 멈춰야 합니다.

(4) 시간 초과 메커니즘
특히 통신 프로토콜을 처리하거나 이벤트를 기다리는 경우 코드에서 시간 초과 메커니즘을 구현합니다. 예상한 이벤트가 특정 시간 내에 발생하지 않으면 오류가 발생했다고 가정하고 그에 따라 처리할 수 있습니다.

```
unsigned long startTime = millis();
while (!eventOccurred()) {
    if (millis() - startTime > TIMEOUT_DURATION) {
        // Handle timeout
        break;
    }
}
```

(5) 워치독 타이머
arduino 보드를 포함하여 일부 마이크로컨트롤러에는 감시 타이머라는 기능이 있습니다. 프로그램이 중단되거나 무한 루프에 들어갈 경우 시스템을 재설정하는 하드웨어 타이머입니다.

(6) 디버깅 출력
직렬 통신을 사용하여 arduino에서 컴퓨터로 디버깅 정보를 보냅니다. 이를 통해 프로그램에서 무슨 일이 일어나고 있는지 이해하고 문제가 언제 어디서 발생하는지 식별하는데 도움이 됩니다.

```
Serial.println("Starting task...");
// Perform some task
Serial.println("Task completed.");
```

요약하면, Try-Catch 블록을 사용한 기존 예외 처리는 일반적으로 arduino 프로그래밍에 사용되지 않지만 오류 및 예상치 못한 상황을 처리하기 위해 사용할 수 있는 몇 가지 다른 전략을 추가할 수 있습니다. 여기에는 오류 조건 확인, 반환 값 및 오류 코드 사용, 오류 방지 상태 설계, 시간 초과 구현, 감시 타이머 사용 및 디버깅 출력 의존이 포함됩니다.

부 록

초음파 레이더영상을 위한 프로세싱 소스코드

```
/*
Radar Screen Visualisation for HC-SR04
Maps out an area of what the HC-SR04 sees from a top down view.
Takes and displays 2 readings, one left to right and one right to left.
Displays an average of the 2 readings
Displays motion alert if there is a large difference between the 2 values.
*/
import processing.serial.*;      // import serial library
Serial arduinoport;                    // declare a serial port
float x, y;                          // variable to store x and y co-ordinates for vertices
int radius = 350;                   // set the radius of objects
int w = 300;                        // set an arbitary width value
int degree = 0;                     // servo position in degrees
int value = 0;                      // value from sensor
int motion = 0;                     // value to store which way the servo is panning
int[] newValue = new int[181];  // create an array to store each new sensor value for each servo position
int[] oldValue = new int[181];  // create an array to store the previous values.
PFont myFont;                        // setup fonts in Processing
int radarDist = 0;                  // set value to configure Radar distance labels
int firstRun = 0;                   // value to ignore triggering motion on the first 2 servo sweeps
/* create background and serial buffer */
void setup(){
// setup the background size, colour and font.
size(1204, 650);
background (0); // 0 = black
myFont = createFont("verdana", 12);
textFont(myFont);
// setup the serial port and buffer
arduinoport = new Serial(this, "COM4", 9600);
}

/* draw the screen */
void draw(){
fill(0);                                // set the following shapes to be black
noStroke();                              // set the following shapes to have no outline
ellipse(radius, radius, 750, 750);      // draw a circle with a width/ height = 750 with its center position (x and y) set by the radius
rectMode(CENTER);                         // set the following rectangle to be drawn around its center
rect(350,402,800,100);                  // draw rectangle (x, y, width, height)
if (degree &gt;= 179) {                      // if at the far right then set motion = 1/ true we're about to go right to left
motion = 1;                             // this changes the animation to run right to left
}
if (degree &lt;= 1) {                         // if servo at 0 degrees then we&#039;re about to go left to right
motion = 0;                             // this sets the animation to run left to right
}
/* setup the radar sweep */
/*
We use trigonmetry to create points around a circle.
So the radius plus the cosine of the servo position converted to radians
Since radians 0 start at 90 degrees we add 180 to make it start from the left
Adding +1 (i) each time through the loops to move 1 degree matching the one degree of servo movement
cos is for the x left to right value and sin calculates the y value
since its a circle we plot our lines and vertices around the start point for everything will always be the center.
*/
strokeWeight(7);                         // set the thickness of the lines
if (motion == 0) {                       // if going left to right
for (int i = 0; i = 0; i--) {       // draw 20 lines with fading colour
stroke(0,200-(10*i), 0);            // using standard RGB values, each between 0 and 255
line(radius, radius, radius + cos(radians(degree+(180+i)))*w, radius + sin(radians(degree+(180+i)))*w);
}
}
```

```
/* Setup the shapes made from the sensor values */
noStroke();                              // no outline
/* first sweep */
fill(0,50,0);                              // set the fill colour of the shape (Red, Green, Blue)
beginShape();                            // start drawing shape
for (int i = 0; i &lt; 180; i++) {      // for each degree in the array
x = radius + cos(radians((180+i)))*((oldValue[i])); // create x coordinate
y = radius + sin(radians((180+i)))*((oldValue[i])); // create y coordinate
vertex(x, y);                          // plot vertices
}
endShape();                              // end shape
/* second sweep */
fill(0,110,0);
beginShape();
for (int i = 0; i &lt; 180; i++) {
x = radius + cos(radians((180+i)))*(newValue[i]);
y = radius + sin(radians((180+i)))*(newValue[i]);
vertex(x, y);
}
endShape();
/* average */
fill(0,170,0);
beginShape();
for (int i = 0; i = 360) {
stroke(150,0,0);
strokeWeight(1);
noFill();
for (int i = 0; i   35 || newValue[i] - oldValue[i] &gt; 35) {
x = radius + cos(radians((180+i)))*(newValue[i]);
y = radius + sin(radians((180+i)))*(newValue[i]);
ellipse(x, y, 10, 10);
}
}
}
/* set the radar distance rings and out put their values, 50, 100, 150 etc.. */
for (int i = 0; i &lt;=6; i++){
noFill();
strokeWeight(1);
stroke(0, 255-(30*i), 0);
ellipse(radius, radius, (100*i), (100*i));
fill(0, 100, 0);
noStroke();
text(Integer.toString(radarDist+50), 380, (305-radarDist), 50, 50);
radarDist+=50;
}
radarDist = 0;
/* draw the grid lines on the radar every 30 degrees and write their values 180, 210, 240 etc.. */
for (int i = 0; i = 300) {
text(Integer.toString(180+(30*i)), (radius+10) + cos(radians(180+(30*i)))*(w+10), (radius+10) + sin(radians(180+(30*i)))*(w+10), 25,50);
} else {
text(Integer.toString(180+(30*i)), radius + cos(radians(180+(30*i)))*w, radius + sin(radians(180+(30*i)))*w, 60,40);
}
}
```

```
/* Write information text and values. */
noStroke();
fill(0);
rect(350,402,800,100);
fill(0, 100, 0);
text("Degrees: "+Integer.toString(degree), 100, 380, 100, 50);            // use
Integet.toString to convert numeric to string as text() only outputs strings
text("Distance: "+Integer.toString(value), 100, 400, 100, 50);         // text(string,
x, y, width, height)
text("Radar screen code ", 540, 380, 250, 50);
fill(0);
rect(70,60,150,100);
fill(0, 100, 0);
text("Screen Key:", 100, 50, 150, 50);
fill(0,50,0);
rect(30,53,10,10);
text("First sweep", 115, 70, 150, 50);
fill(0,110,0);
rect(30,73,10,10);
text("Second sweep", 115, 90, 150, 50);
fill(0,170,0);
rect(30,93,10,10);
text("Average", 115, 110, 150, 50);
noFill();
stroke(150,0,0);
strokeWeight(1);
ellipse(29, 113, 10, 10);
fill(150,0,0);
text("Motion", 115, 130, 150, 50);
fill(0,450,500);
text("Developed by: SUPERKITZ ", 350, 400, 150, 50);
text("www.superkitz.com ", 400, 430, 150, 50);
fill(0,110,0);
}
/* get values from serial port */
void serialEvent (Serial arduinoport) {
String xString = arduinoport.readStringUntil('\n');  // read the serial port until
a new line
if (xString != null) {  // if theres data in between the new lines
xString = trim(xString); // get rid of any whitespace just in case
String getX = xString.substring(1, xString.indexOf("V")); // get the value of the
servo position
String getV = xString.substring(xString.indexOf("V")+1, xString.length()); // get
the value of the sensor reading
degree = Integer.parseInt(getX); // set the values to variables
value = Integer.parseInt(getV);
oldValue[degree] = newValue[degree]; // store the values in the arrays.
newValue[degree] = value;
/* sets a counter to allow for the first 2 sweeps of the servo */
firstRun++;
if (firstRun &gt; 360) {
firstRun = 360; // keep the value at 360
}
}
}
```

저작권 및 소유권 고지 및 사용 허가

© 2024, 프라이봇 정 욱진. 모든 권리 보유.

본 책과 그 내용은 저작권법에 의해 보호됩니다. 본 책은 교육적 목적으로 온라인에서 무료로 제공되며, 프라이봇은 교육 기관, 교사, 학생 및 자기계발을 위해 개인이 사용하는 경우에 한해 비상업적 목적으로 본 책의 내용을 사용할 수 있도록 허가합니다.

단, 본 책의 내용, 포함된 그림, 코드 및 기타 자료는 명시적인 서면 허가 없이 복제, 배포, 전송, 수정, 전시, 출판할 수 없습니다. 본 책에 포함된 달리(Dall-E) 이미지는 OpenAI의 정책에 따라 사용되었으며, 해당 이미지의 저작권은 OpenAI에 있습니다. 이 이미지들은 본 책의 내용을 보조하기 위한 목적으로만 사용되었습니다.

본 책에서 제공하는 코드 예제들은 교육적 목적으로 사용할 수 있으며, 해당 코드를 사용함에 있어서 발생하는 어떠한 결과에 대해서도 저자나 출판사는 책임을 지지 않습니다.

본 책의 내용은 저자의 지식과 경험을 바탕으로 작성되었으며, 최대한 정확한 정보를 제공하기 위해 노력하였습니다. 그러나 저자와 출판사는 본 책의 내용 사용으로 인해 발생할 수 있는 직접적, 간접적, 부수적, 특수적 또는 결과적 손해에 대해 책임을 지지 않습니다.

이 책은 발행 시점을 기준으로 제공되는 정보를 포함하고 있으며, 저자나 출판사는 책에 포함된 정보의 변경이나 업데이트에 대한 책임을 지지 않습니다.

교육 목적으로 본 책을 사용하시는 모든 사용자는 본 고지 사항을 준수해야 합니다. 상업적 사용, 번역, 수정 작업 등 본 책의 내용을 기타 방식으로 이용하고자 할 경우에는 반드시 저자 또는 출판사의 명시적인 서면 허가를 받아야 합니다.