2021-1

시스템 프로그래밍 및 실습

최종 결과 보고서

나만의 독서실

<Me Before You>

201421024 김　휘

201621028 황지훈

201621046 용상호

내용

1. 프로젝트 개요

본 프로젝트 개발의 목표는 다양한 I/O 용 센서 디바이스 제어가 필요한 응용 서비스를 제안하고, 3 대 이상의 RPi 환경에서 Linux 의 functions 를 활용하여 Sensor-Actuator 를 관리하는 프로그램을 개발하는 것이다.

우리 조는 다양한 I/O 용 센서 디바이스가 필요한 서비스로 '나만의 독서실'을 선정하였다. 최근 코로나 바이러스로 인해 외부 활동이 제한되고, 다중 이용 시설에 대한 주의가 각별해지고 있다. 평소 다양한 이유로 집 외의 학습 환경(카페, 스터디 카페, 독서실 등) 을 이용하던 대학생들은 이런 상황에 대처하여 새로운 학습 환경을 모색해야 할 처지가 되었다. 또한, 위의 이유로 인해 집 안에서 학습하는 시간이 증가하며, 집 안에서 학습을 진행할 때 최대한 집중력을 향상할 수 있는 환경을 만드는 것이 중요해졌다.

팀원 및 주변 지인들을 통해 평소 학생들이 집 외의 다른 학습 환경을 찾는 이유는 다음과 같았다.

1) 익숙한 환경에 계속 쉬게 된다.
2) 눈치 볼 사람이 없어 계속해서 자리를 이탈한다.
3) 휴식과 공부 공간의 분리가 없다.
4) 가족 구성원의 방해
5) 동기 부여 부족 등

위 원인들을 통해 집에서 효율적으로 공부를 하기 위해서는 1) 평소 휴식 공간으로 사용하던 곳을 학습에 적합한 환경으로 변경해야 하고, 2) 자리를 이탈하지 않도록 동기부여를 해줄 수 있는 방안이 필요하며 3) 동시에 가족 구성원들에게 학습 중임을 인지시켜 방해를 사전 차단하는 등의 조치가 필요함을 알 수 있었다. '나만의 독서실'은 이 모든 과정을 다양한 센서들의 상호 작용과 적절한 제어 및 모니터링을 통해 자동화하여, 사용자가 코로나 바이러스로부터 안전하면서도 효율적인 학습 공간을 구성할 수 있도록 한다.

1.1. 서비스 개요

집에서 학습 환경을 구성하고, 외부 학습 장점을 분석하여 적용하는 것이 프로젝트의 목표이므로, 집 안에서 학습을 진행하는 경우의 문제점을 파악한다.

1. 온/습도가 적절하지 않은 경우
2. 가족 구성원이 방해하는 경우
3. 장시간 공부로 인한 실내 산소 농도의 저하로 집중력에 영향을 미치는 경우
4. 학습 동기 부여 요소가 부족한 경우

학습을 진행하는 데 있어, 온도가 너무 춥거나 따듯한 경우 신체 에너지 소모로 인해 집중력에 방해가 될 수 있다. 또한 너무 건조하거나 습한 경우 개인에 따라 불편한 상황(비염 등)이 발생할 수 있다.

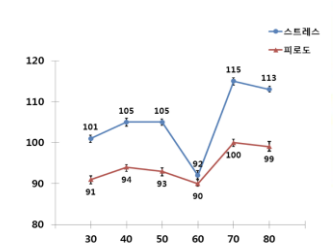
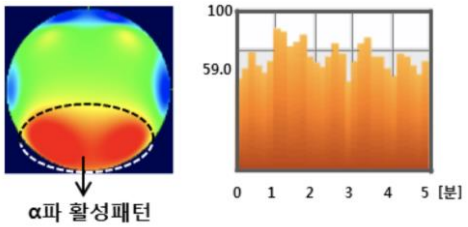


그림 9 습도 변화에 따른 스트레스 지수와 피로도의 변화



(b) 60%에서의 브레인맵핑과 집중도 패턴

출처 - 습도 변화에 따른 뇌파 기반 생체신호 변화에 관한 연구

A Study on the Variation of Physiology Signals based on EEG with Humidity

김 명 호†.김 정 민* (Myung-Ho Kim.Jung-Min Kim)

습도 변화에 따른 뇌파 기반 생체신호 변화에 관한 연구에 따르면 50%~60%[RH%]의 상대 습도로 유지될 때 피험자들의 스트레스와 피로도 가장 낮았으며, 습도가 적절하지 않은 경우 스트레스와 피로도가 증가하는 것을 알 수 있다.

표2. 온도별 FAIR 평가 결과 평균 비교

항목	M 표시치수			P 선택주의력			Q 자기통제력			C 지속성능력		
	원*	P*	S*	원	P	S	원	P	S	원	P	S
23℃	0.996	62	6	478	73	6	0.967	64	6	462.3	74	7
20℃	0.991	72	7	475	66	6	0.956	61	6	443	62.3	5

* 원 : 원점수, P : PR값(백분위), S : STN값(9간 규준점수)

출처:실내 온도가 재실자의 주의집중력에 미치는 영향에 관한 연구

The effect of indoor temperature on occupants' attention abilities

최유림* 전정윤** Choi, Yoo Rim Chun, Chung Yoon

실내 온도가 재실자의 주의집중력에 미치는 영향에 관한 연구에 의하면 선택주의력, 자기통제력 지속성 능력은 23°C 에서 가장 높았던 것을 확인할 수 있다. 하지만 주의 자기 통제 기능, 주의 자원 집중력은 20°C 에서 더 잘 발휘된다고 볼 수 있다. 따라서 집중력에 적절한 온도는 20-23°C 임을 알 수 있다.

따라서 센서를 이용해 온도/습도 변화를 인지하고 항상 학습에 가장 적절한 설정을 유지하도록 한다. 집 안에서 공부를 하는 경우 가족 구성원들과의 마찰 및 방해가 발생할 수 있기 때문에, 학습을 위해 방문을 닫고 의자에 앉은 경우 가족 구성원들이 학습 중인 상태를 인지하여 최대한 배려할 수 있도록 하였다. 또한 밀폐된 공간에서 장시간 학습할 경우 실내 이산화탄소 농도가 증가하는 경우를 가정하여 자동으로 창문을 개방하여 환기할 수 있도록 하였다. 마지막으로 학습에 동기 부여가 필요한 경우이다. 다중 이용 시설(카페, 독서실, 도서관) 등에서 학습을 진행할 경우 집중을 하지 않거나 자주 자리를 이탈하는 경우 옆사람, 혹은 주변 사람에게 눈치가 보여 그러지 않게 되는 경우가 존재한다. 따라서 사용자가 의자에 앉아있는 시간을 측정하고 시각화하여 스스로, 혹은 가족 구성원에 의해 동기 부여가 되도록 한다.

1.2. 시스템 개요

본 시스템에서 제공하는 기능은 다음과 같다.

a. **적정 온습도 유지**

온습도 센서를 활용하여 공부에 집중할 수 있는 최적화된 온도를 자동으로 유지한다.

b. **적정 산소 농도 유지**

방 안의 산소 농도를 계속해서 측정하고, 특정 농도 이하일 경우 산소 발생기 혹은 창문 개방을 통해 자동으로 공부에 최적화된 산소 농도를 유지한다.

c. **공부 시간 표시**

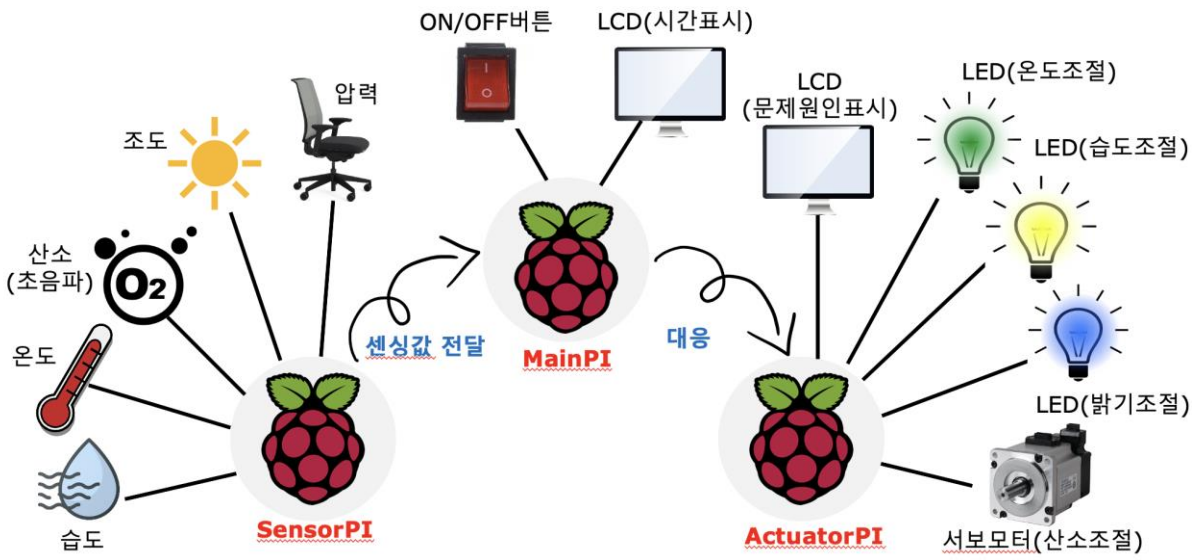
압력센서를 활용하여 사용자가 의자에 앉은 경우를 공부 중인 것으로 판단하고, 자동으로 공부 시간을 측정하고 표시한다.

d. **가족 구성원에게 '공부 중' 표시**

압력 센서가 눌린 경우 공부 중이라는 상태를 가족 구성원들에게 표시하여 방해를 줄일 수 있도록 한다.

e. **제어 시스템 제공**

스위치를 통해 '나만의 독서실' 서비스를 On/Off 할 수 있으며, 버튼을 통해 측정된 공부 시간을 초기화 할 수 있다.



위의 기능들을 제공하기 위해 구성한 시스템의 개요이다. 총 3 대의 라즈베리파이를 이용해 시스템을 구축하였고, 각각 센서 Pi, 메인 Pi, 엑추에이터 Pi 로 구성되어 있다. 각 파이가 맡은 역할은 다음과 같다.

< 센서 Pi – 관리자 영역 >

주된 목적이 센싱이기 때문에, 센서들과 가장 가까운 곳에 위치시키고, 센싱 알고리즘의 변경에 의한 센싱오류 등의 문제를 방지하기 위해 관리자 영역으로 설정한다.

1. 온습도 센서를 활용하여 온도, 습도가 미리 설정한 기준을 벗어날 경우 메인 파이로 신호를 전달한다.
2. 압력 센서를 통해 사용자의 착석 여부를 확인하여 신호를 전송한다.
3. 산소 센서를 이용하여 실내 산소 농도를 측정하고, 일정 기준 이하일 경우 메인 파이로 신호를 보낸다.

< 메인 Pi - 사용자 영역 >

메인 Pi 의 주된 목적은 센싱값에 따른 엑츄레이터의 동작여부를 결정 하는 것이다.

만약 사용자가 필요하다면, 제한된 기능에 내에서 UI 를 통해 수정을 허용한다.

(임계값,엑츄레이터 동작 단계 등) 하지만 그 부분을 제외한 기능은 관리자영역으로 설정하여 메인 알고리즘을 보호 한다. 시스템의 일부이긴 하지만, 사용자의 접근을 허용하기 때문에 사용자영역으로 구분한다.

1. 센서 Pi 로부터 전달 받은 신호를 수신하고 분석한다.
2. 수신한 신호가 적정 범위에 있는지를 확인한다.
3. 신호가 적정 범위를 벗어난 경우 이를 제어하기 위해 Actuator Pi 에 신호를 전송한다.

< Actuator Pi - 관리자 영역>

- 주된 목적이 엑츄레이터 작동이기 때문에, 엑츄레이터들과 가장 가까운 곳에 위치시키고, 동작 알고리즘의 변경에 의한 동작 오류 등의 문제를 방지하기 위해 관리자 영역으로 설정한다.

1. 메인 Pi 로부터 제어 신호를 수신한다.
2. 수신한 신호에 맞는 센서에 적정 신호를 송출하여 LCD, LED, 서보모터를 비롯한 액추에이터를 작동한다.

각각의 파이가 자신이 맡은 역할을 충실히 수행한다는 가정 하에, 시스템의 관점에서 동작 시나리오는 다음과 같다.

System Main Scenario

1. 스위치를 통해 시스템을 ON 한다. (공부를 시작하기 위함)
2. 각종 센서들이 동작하기 시작하며 문제 발생 여부를 확인한다.
3. 사용자가 의자에 앉으면 압력센서 인식하여 방 문에 붙여진 LED 가 ON 되고 앉은 시간을 계산하여 LCD 에 나타낸다.
4. 센서들이 문제 발생 여부를 확인한다.
   4-1. 적절한 온습도의 범위를 벗어난다면 적절한 온습도를 유지하기 위해 온도조절기와 습도조절기가 작동한다.
   4-2. 산소 센서를 통해 적정 산소 농도 범위를 벗어나면 적정 산소 범위가 될 때까지 창문을 개방한다.
   4-3. 조도 센서를 통해 적정 조도를 유지하기 위해 조명을 작동시킨다.
5. 시스템이 더 이상 필요하지 않을 때 스위치를 통해 OFF 한다.

2. 시스템 분석

2.1. 센서 개요

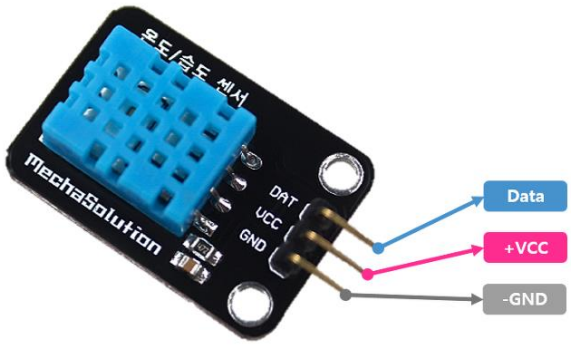
'나만의 독서실' 서비스를 개발하기 위해 사용한 센서를 기능별로 정리하면 다음과 같다.

기 능	이 름	수 량	인터페이스
온 / 습도 감지기능	온/습도 센서	1 EA	GPIO/WiringPi
조도 감지기능	조도 센서	1 EA	SPI
움직임 감지기능	압력 센서	1 EA	SPI
산소농도 측정 기능	초음파 센서(대체)	1 EA	GPIO
표시기능	LCD with adapter	2 EA	I2C
	LED	3 EA	GPIO
제어 기능	버튼	1 EA	GPIO
대응 기능	서보모터	1 EA	PWM
디지털 입력 변환	ADC(MCP3008)	1 EA	

각 센서는 대여를 통해 사용하였고, 이 중 ADC, 온습도 센서의 고장으로 센서를 교환하였다. ADC 의 경우에는, 조도 센서와 압력 센서를 이용하는 경우에 필요했기 때문에 0, 7 번 채널 총 2 개의 채널을 이용하여 사용하였다.

각 센서 및 액추에이터 들의 간략한 개요는 다음과 같다.

1. 온습도 센서



<라즈베리파이의 온습도 센서(DHT11)>

온습도 센서는 실내 혹은 특정 공간의 온도/습도를 측정하기 위해 사용된다. '나만의 독서실' 프로젝트에서는 방 내의 적정 온도/습도를 측정하기 위해 사용 예정이다. 온습도센서는 GPIO 인터페이스를 사용하고, Sensor Rpi 에 의해 제어된다. DHT11 센서는 통상적으로 20% ~80% 사이의 습도를 5%의 오차 범위 내로 측정하고, 0~50 도 사이의 온도를 + - 2 도 사이의 오차 범위 내로 측정한다. 일반적인 가정집의 환경을 고려하였을 때, 필요한 범위를 측정할 수 있는 센서이다.

2. 압력 센서

<라즈베리파이의 압력센서 FSR402>

압력센서는 센서에 가해지는 압력을 측정함으로써 특정 물체 혹은 생물이 센서 위에 존재하는 지의 여부를 측정하거나, 압력의 정도를 측정하는 등의 용도로 사용할 수 있다. '나만의 독서실' 프로젝트에서는 사용자의 의자 착석 여부와, 압력의 변화 양상을 압력센서를 통해 수신하여 사용자가 자리를 이탈하였는지의 여부를 파악할 수 있다. 압력센서는 SPI 인터페이스와 ADC(Analog Digital Converter)를 이용한다.

3. 조도 센서

<라즈베리파이의 조도 센서>

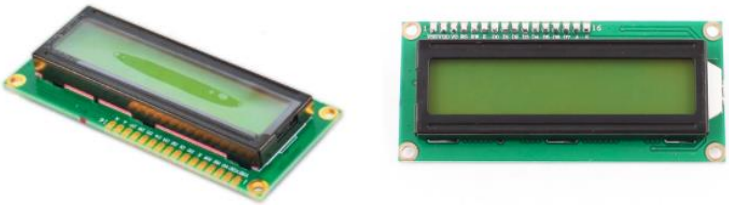
조도 센서는 빛의 밝기를 가변저항을 이용하여 아날로그 값으로 출력한다. 따라서 아날로그 값을 변환하기 위해 ADC(Analog to Digital Converter)를 이용해야 한다. 조도 센서의 인터페이스는 SPI 를 사용한다. '나만의 독서실'에서는 사용자가 책상에 앉아있는 동안 조도센서를 이용하여 주변 밝기를 인지하고, 학습에 필요한 밝기를 맞출 수 있도록 조명을 조절한다.

4. 초음파 센서

<라즈베리파이 초음파 센서>

초음파센서는 발신부와 수신부를 가진다. 발신부에서는 초음파를 발생시키고, 이 초음파가 물체에 맞고 수신부로 돌아오는 시간을 계산하여 거리를 측정하는 센서이다. 센서는 약 40kHz 의 초음파를 발생시키며 최대 4m 의 거리를 측정할 수 있다. '나만의 독서실' 프로젝트에서는 산소 농도 측정 센서의 역할을 대체하기 위해 사용하였으며, 거리가 가까운 경우 산소가 부족한 것으로 시뮬레이션 하였다.

5. LCD



<라즈베리파이의 1602 LCD>

1602 LCD 는 I2C 변환 모듈을 이용해 16 핀의 제어핀을 4 개의 핀으로 제어가 가능하도록 만들어진 모델이다. 이름이 1602 인 이유는 16 * 2 의 해상도를 가지기 때문이고, 동작 전압이 5V 이다. 실습시간에 공지된 바에 따르면 5V 를 사용하는 센서를 사용할 때에는 라즈베리파이의 전원을 해제하고 연결해야 고장을 방지할 수 있다고 한다. '나만의 독서실' 프로젝트에서는 사용자가 의자에 앉기 시작한 순간부터 일어나기까지 압력센서로 측정한 신호를 Main Rpi 가 전달받아 시간으로 계산하여 메인 파이에서 사용자에게 출력하고, Control Pi 에 어떤 신호가 들어온 것인지 전달한 것을 1602 LCD 를 통해 출력하는데 사용된다.

6. LED

<라즈베리파이의 LED>

LED 는 3.3V 전압을 이용하여 빛을 냄으로써 경고 혹은 알림을 보낼 수 있는 장치이다. 라즈베리파이에서는 GND 에 접지를 연결하고, VCC 에 전원을 연결한 뒤 IN 에 GPIO 인터페이스를 이용한 입력 신호를 통해 LED 를 제어할 수 있다. '나만의 독서실'에서는 사용자가 의자에 착석한 경우를 가족 구성원에게 알리는 용도로 사용하거나, 특정 기능의 동작 여부를 사용자에게 알리는 경우에 사용할 수 있다.

7. 버튼

<라즈베리파이의 버튼 BG-US-91570>

버튼은 GPIO 를 이용해 사용자로부터 입력을 받을 수 있게 해주는 장치이다. 버튼 핀의 한 쪽은 GND, 한 쪽은 상시 High(1)의 값을 주고 사용자 입력(버튼 on/off)에 따라 그 값을 다르게 할 수 있다. 사용자가 버튼을 누르는 경우 접지가 연결되어 상시 high 값이 접지로 빠져나가기 때문에 버튼의 입력 값이 LOW 가 된다. GND 대신에 3.3V(high)에 연결한뒤 상시 LOW 를 이용하게 되면 사용자가 버튼을 클릭하였을 때 HIGH 의 값으로 이용할 수 있다. '나만의 독서실'에서는 사용자가 학습 누적 시간을 초기화 하는데 사용할 수 있다.

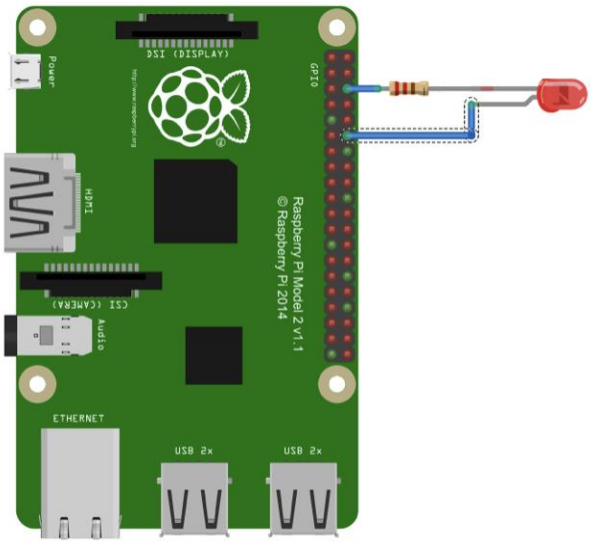
8. 서보 모터

<라즈베리파이에 이용된 서보 모터>

서보 모터는 0 도에서 180 도 사이의 각도로 모터를 동작하는 방식을 이용한다. PWM 인터페이스를 사용하며, 이를 통해 원하는 방식으로 액추에이터를 작동하거나, 그와 유사한 상황을 가정한 시뮬레이션을 진행할 수 있다. '나만의 독서실'에서는 서보모터를 창문 개방 및 온습도 조절 장치, 조명 조절 장치의 시뮬레이션에 사용한다.

2.2. 센서 인터페이스 설명

1. GPIO



<라즈베리파이>

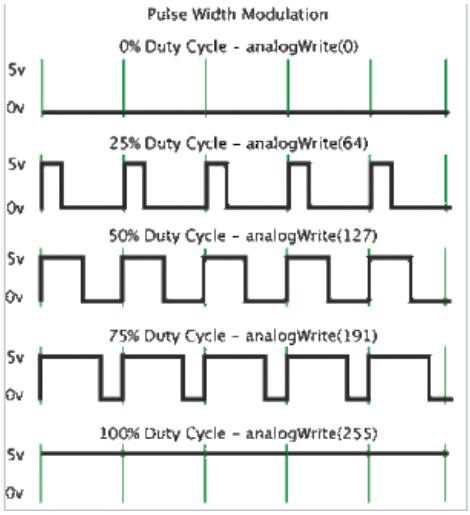
GPIO(general-purpose input/output)는 연결되어있는 PI 의 입력이나 출력동작이 사용자에 의해 제어가 가능하게 해주는 입출력 포트로 범용 입출력 인터페이스이다. 0 또는 1 의 값으로만 제어가 가능하기 때문에, 복잡한 구현에는 적절하지 않지만, 간단한 기능과 GPIO 들의 복합적인 기능을 통한 구현에는 적절하다. 각 핀들에 의해 입출력 제어가

가능하며, 라즈베리파이의 경우 초기모델은 26 개의 핀을 지원하였고, 현재 발매 중인 라즈베리파이의 GPIO 핀은 40 개 를 지원하고 있다. LED 가 연결되어 있는 핀의 값을 GPIO 를 통해 변경해줌으로 LED 의 on/off 를 구현할 수 있다.

export 를 하여 OS 로 부터 GPIO 사용 권한을 받을 수 있고 unexport 로 다시 반환할 수 있다. direction 을 통해 해당 GPIO 핀 번호에 값을 write 할 지, read 할 지 결정 할 수 있다.

프로젝트를 진행에서 LED,초음파센서,버튼을 구현하는데 GPIO 를 사용하였다. 0 과 1 을 사용하여 제어가 가능하였고, 0 과 1 을 입력하는 패턴이나 속도를 통해서도 의미있는 기능을 구현할 수 있었다.

2. PWM

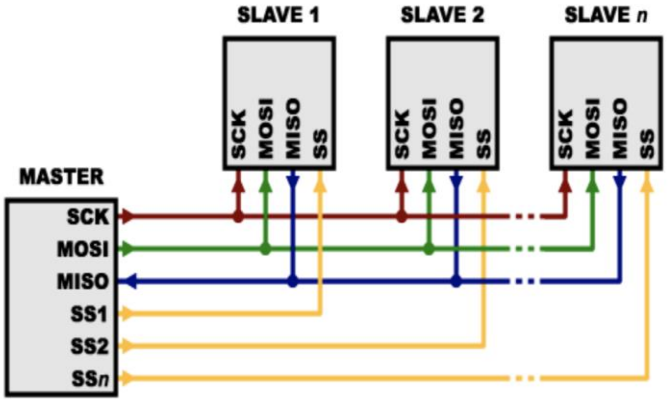


<PWM 펄스>

PWM(Pulse Width Modulation)은 펄스 폭 변조를 뜻한다. 0 과 1 로 이루어져 각진 디지털신호를 부드러운 아날로그 신호로 변조하는 것이다. PWM 은 period 와 duty-cycle 을 통해 조절이 가능하다. period 와 duty-

cycle 의 비율에 따라 입력되는 전압의 양이 차이가 생기기 때문에, 0 과 1 로 이루어진 신호지만 아날로그 형태처럼 동작하는 것이 가능하다.
프로젝트에선 1/10 의 비율로 period 와 duty-cycle 를 설정하여 안정적인 회전을 확인할 수 있었다.

3. SPI

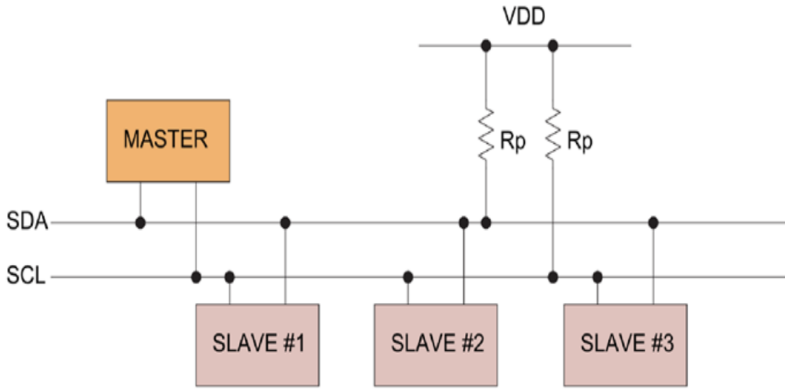


<SPI 연결>

SPI (Serial Peripheral Interface)는 기기와 기기간에 데이터를 주고받기 위한 통신 방식중에 하나이며, 직렬통신방식이다. 1 : N 의 동기식 통신방식이며, 동시에 송수신이 가능하지만, 각 기기별로 케이블이 필요하다는 단점이 있다. 기기별로 케이블이 플요하기 때문에, 연결된 기기의 숫자가 적고 전송 속도가 빨라야 하는 상황에서 주로 사용이된다. 1 : N 중 1 이 마스터, N 이 슬레이브 기기로 이루어진 마스터-슬레이브 방식을 사용한다.

SPI 통신을 위해서는 MOSI, MISO, SCK, SS 핀이 필요한데, MOSI 는 19 번 pin, MISO 는 21 번 pin, SCK 는 22 번 pin 에 해당한다.

4. I2C

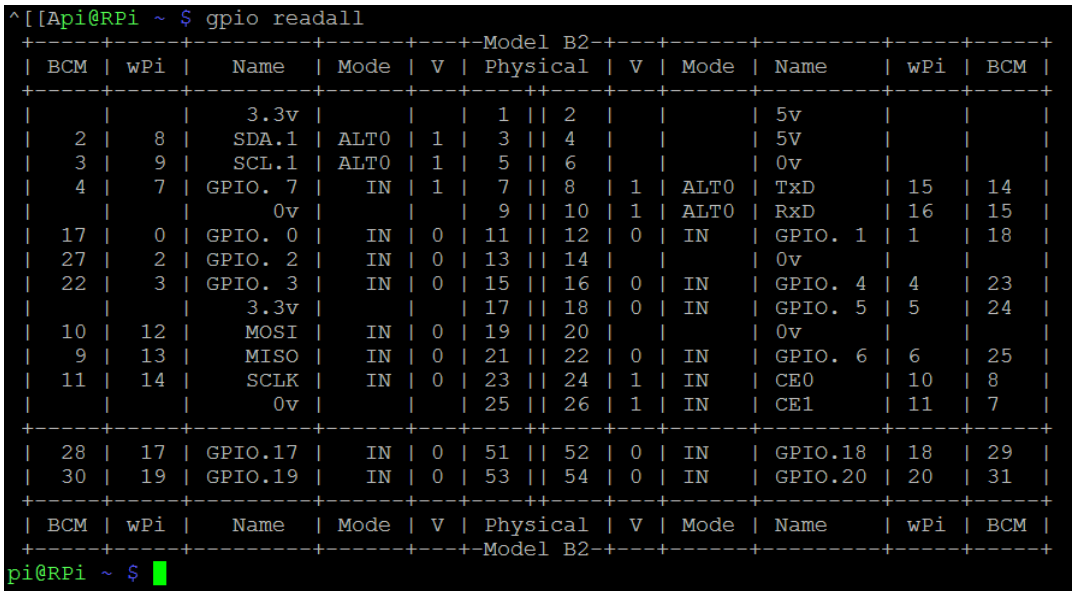


<I2C 연결>

I2C(Inter-Integrated Circuit)은 기기사이에 통신 링크를 제공하는 양방향 직렬버스이다. SPI 와 마찬가지로 1 : N 의 통신방식을 가지지만, 케이블대신 버스가 존재한다는 점에서 차이가 있다. 하나의 버스를 통해 버스에 연결되어 있는 기기는 모든 데이터를 공유 할 수 있다. 연결된 기기의 숫자가 많은 상황에서 주로 사용된다. SPI 와 마찬가지로 1 : N 중 1 이 마스터, N 이 슬레이브 기기로 이루어진 마스터-슬레이브 방식을 사용한다.

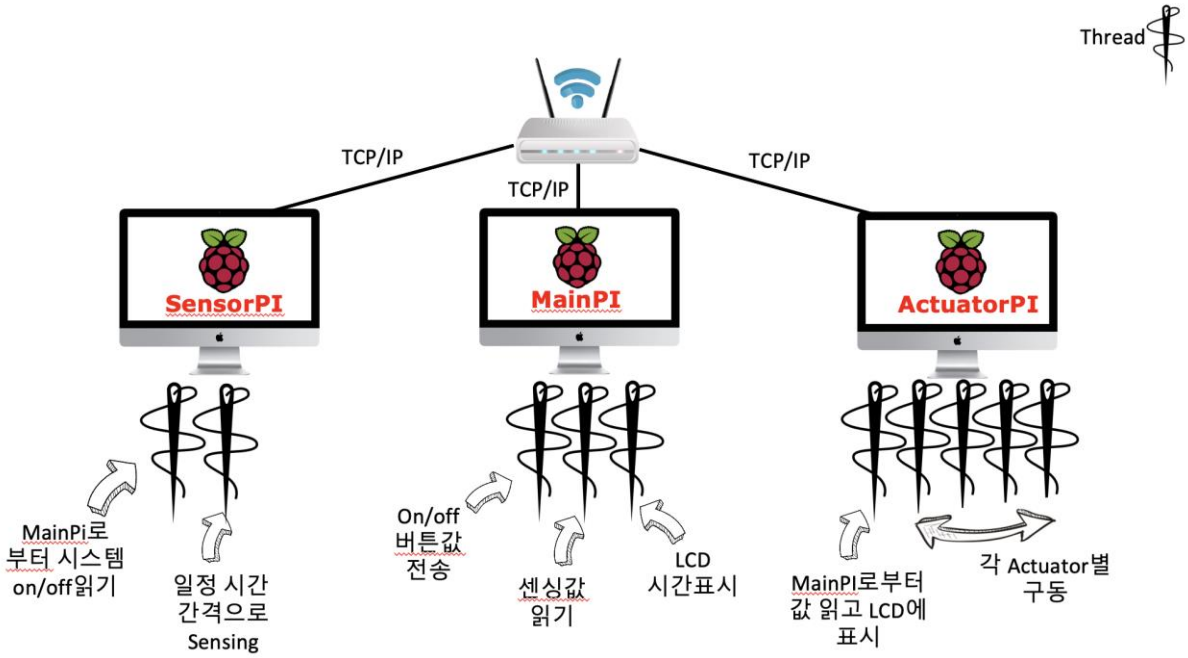
5. WiringPi

WiringPi 는 아날로그 읽기 및 쓰기를 지원하고 8 개의 범용 디지털 I / O 핀이 있다. wiringPi 라이브러리를 사용하면 ADC 없이, 긴 데이터를 읽어 올 수 있다는 장점이 있다. DHT11 온습도 센서의 경우 패리티 비트를 포함하여 총 40bits(5byte)의 값을 보낸다. 따라서 온/습도 센서를 구현할 때에는 긴 데이터를 받을 수 있는 wiringPi 라이브러리를 사용하는 것이 적절하다. “gpio readall” 명령어를 통해 wiringPi 가 사용하는 GPIO 핀 번호를 확인했다.



위 사진은 wiringPI 가 사용할 수 있는 GPIO 핀 번호를 보여준다. wiringPI 라이브러리의 wiringPiSetupGpio() 또는 wiringPiSetup() 함수를 통해 GPIO 핀 번호를 기준으로 할 지, WiringPI 번호를 기준으로 할 지 선택 할 수 있다.

2.3. PI 간 interaction 방식



<나만의 독서실 구성>

파이 간의 통신은 Wifi 를 이용한 무선 통신 방식으로 구현하였으며, 공유기의 설정에서 각 파이에 대한 ip 주소를 등록하여 놓았다. 실습 시간에 사용한 유선 방식이 아닌 무선 방식을 이용한 이유는 '나만의 독서실' 에서는 사용자가 공부하는 책상에 메인 파이가 위치하고, 나머지 센서 및 액추에이터들은 각자 자신이 맡은 역할을 수행할 수 있도록 방 내에 주어진 위치에 존재해야 하기 때문이다.

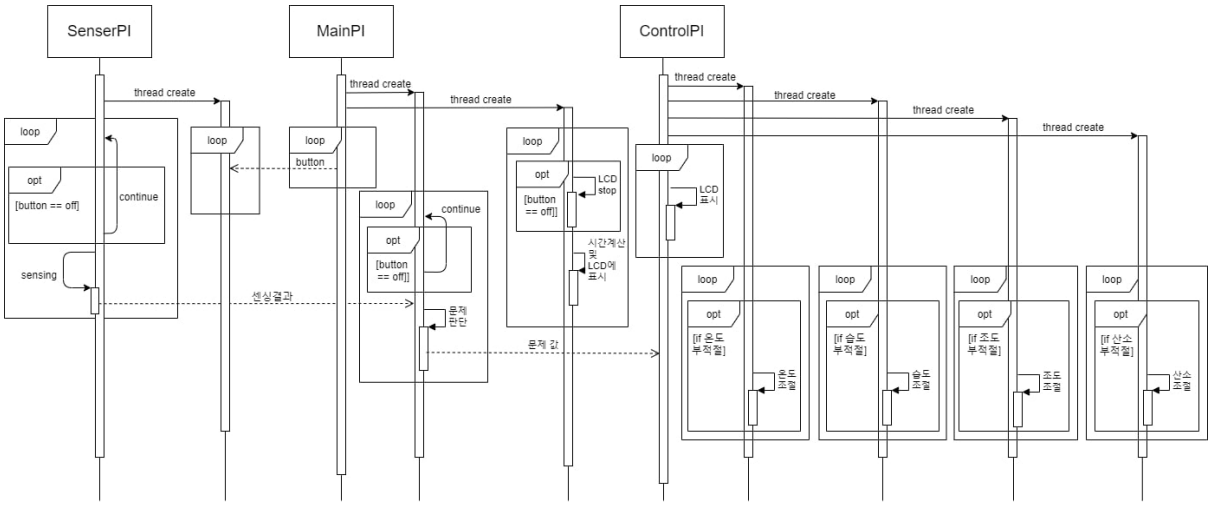
파이 간에는 소켓 통신을 이용해 서로 소통하도록 하였으며, Server - Client 구조를 사용하도록 하였다. 신호의 이동 방향이 Sensor Pi 에서 Main Pi 로, Main Pi 에서 Actuator Pi 로 이동하는 구조이기 때문에 Sensor Pi 는 client 의 역할, Main Pi 는 Sensor Pi 에 대한 Server 역할과 동시에 Actuator Pi 의 client 역할, Actuator Pi 는 Main Pi 에 대한 Server 역할을 하도록 하였다.

이렇게 구조를 잡으면서 가장 많이 고민했던 것은, 각 파이의 역할이었다. 처음에는 센서파이가 값을 계속 읽어들이며 특정 조건을 만족했을 때만 메인 파이로 값을 전달하고, 메인 파이가 이를 액추에이터 파이에 전달하면 액추에이터 파이가 값에 따라 동작하는 것으로 구현하였다. 하지만 이 경우 메인 파이의 존재 이유가 사라질 뿐만 아니라 각 파이의 기능이 명확히 구분되지 않는다는 단점이 존재하였다. 따라서 Sensor 파이는 오로지 값을 읽어들이는 기능만, Actuator Pi 는 오로지 actuator 를 동작시키는 기능만 수행하도록 하며 모든 처리 로직은 Main pi 에서 담당하도록 하였다.

각 파이는 멀티 스레드를 통해 구현하였는데, 이는 각 센서가 동시 다발적으로 (concurrently) 하게 수행해야 하는 기능이 정해져있기 때문이었다. Sensor Pi 는 Main Pi 로부터 시스템을 on/off 신호를 읽으며, 일정 간격을 두고 지속적으로 센싱을 해야 한다. 메인 파이로부터 신호를 읽는 기능과 센서로부터 값을 읽는 기능은 concurrent 하게 수행되어야 하기 때문에 두 개의 스레드를 이용하여 하나는 계속해서 메인 신호로부터 들어오는 값을 읽고, 하나는 계속해서 센서로부터 값을 읽도록 구현하였다. 메인 파이에서 각 센서에 대한 처리 로직별로 스레드를 구현하지 않은 이유는, Sensor 파이에서 값을 일괄로 묶어서 보낼 뿐만 아니라 하나의 조건문 분기에서 각 센서들에 대한 로직을 계산하기 때문에 굳이 멀티 스레드를 사용할 필요가 없었기 때문이다.

Main Pi 도 마찬가지로 Sensor Pi 에 On/Off 신호를 보내는 기능과 Actuator Pi 로 신호를 보내는 기능, 그리고 LCD 를 통해 현재까지의 공부 시간을 실시간으로 표시하는 기능을 모두 concurrent 하게 처리해야 한다. 특히 LCD 에 시간을 표시하는 기능을 구현하면서 sleep(1) 을 이용해 1 초에 숫자가 1 씩 증가되도록 구현하였는데, 이 기능이 멀티스레드로 구현되지 않으면 전체 시스템이 1 초마다 한 번씩 동작을 정지하기 때문에 반드시 멀티 스레드로 구현이 되어야 했다.

Actuator Pi 는 가장 많은 스레드를 이용한 파이이다. 메인 파이로부터 신호를 받는 스레드와, 각 액추에이터를 동작시키는 기능이 모두 개별 스레드로 구현이 되었기 때문이다. Actuator Pi 에서 멀티 스레드를 사용하지 않을 경우 메인 파이로부터 신호를 받는 동안에는 액추에이터가 동작하지 못하거나, 반대로 액추에이터가 동작하는 경우 메인 파이로부터 신호를 읽을 수 없다는 단점이 존재한다. 또한, 각 액추에이터마다 멀티 스레드를 사용한 이유는 동시에 여러개의 센서 값이 들어올 수 있기 때문이다. 하나의 액추에이터가 동작하는 동안 다른 액추에이터가 동작할 수 없거나 기다려야 한다면 우리가 의도한대로 동작하지 않는 것이기 때문에 멀티스레드를 이용하여 Actuator Pi 의 모든 기능이 독립적으로 구현되도록 하였다.



위 사진은 각 PI 간 인터렉션을 보여주는 다이어그램이다. PI 간 어떤 쓰레드를 만들고 어떤 데이터를 주고 받는지를 보여준다.

- sensorPI : 두 개의 쓰레드로 동작한다. 하나의 쓰레드는 mainPI 로부터 on/off 버튼 값을 입력 받아 센싱을 시작할 지 말 지 결정한다. 나머지 하나의 쓰레드는 실제로 센싱을 하는 쓰레드이다. 압력, 산소, 온/습도, 조도 값을 센싱하고 결과를 저장한다. 이 두 쓰레드는 전역변수 값을 공유하며 서로 상호작용 한다. mainPI 로 부터 on/off

버튼 값을 읽는 쓰레드가 전역변수에 system on/off 에 대한 값을 초기화하면, 또 다른 쓰레드에서는 이 값을 읽고 센싱을 시작하거나 중지한다.

- mainPI : mainPI 는 on/off 버튼 값을 읽는 쓰레드, sensorPI 로 부터 값을 받아 문제의 원인을 파악하는 쓰레드, 앉은 시간을 LCD 에 표시하는 쓰레드로 3 개의 쓰레드로 동작한다.

- controlPI(actuatorPI) : controlPI 는 5 개의 쓰레드로 구성된다. 크게 2 종류의 성격을 갖는 쓰레드인데, 하나의 쓰레드는 mainPI 로 부터 문제의 값을 읽고 LCD 에 표시하며 나머지 네 개의 쓰레드는 각각의 액츄레이터를 담당한다. 액츄레이터가 동시에 동작해야 할 상황이 발생하기 때문에 액츄레이터 마다 쓰레드를 구성했다. 쓰레드 들은 전역변수를 읽어 자신이 담당하는 액츄레이터를 돌릴 때인지 아닌지 판단한다.

3. 시스템 구현

3.1. 센서구현

- **온습도 센서**

  DHT11 온습도 센서를 사용하기 위해 이 센서에 대한 이해가 필요하다. 이 센서는 온도, 습도, 페리티비트 등 많은 값을 넘겨주기 때문에 파이와 의사결정 후에 데이터를 전송한다.

파이가 01 을 보내고 센서기 101 을 보낸 뒤, 실질적인 데이터를 전송한다. 온습도 센서를 사용하기 위해 WiringPi 라이브러리를 사용했다. WiringPi 에 piHiPri() 함수를 통해 파이에서 현재 실행 중인 프로세스(필자가 작성한 코드)가 preemption 되는 것을 방지한다. 즉, piHiPri()를 사용해서 OS 에게 현재 프로세스의 priority 값을 설정할 수 있고, 필자는 이 프로세스의 priority 를 최대 값으로 설정함으로써 센싱 프로세스가 죽지 않도록 했다.

이러한 이유는 DHT11 센서의 특징 때문이다. DHT11 센서는 시간 단위로 데이터 값을 전달해 주기 때문에, 센서 파이는 제때 데이터를 받을 수 있어야 한다. 따라서 프로세스가 preemption 되지 않게 하기 위해 priority 를 ceiling 으로 늘렸다.

```
// actual 데이터 받기
for (int dataIdx = 0; dataIdx < 5; dataIdx++) {
        for (int bit = 0; bit < 8; bit++) {  // 8bit씩
                int count = 0;
                while (digitalRead(DHT_GPIO) == laststate) { // 같은게 8bit 나오는지 검사함
                        count++;

                        delayMicroseconds(1);

                        if (count == 255) break; // 모두 1111 1111 이거나 0000 0000 인 경우
                }

                laststate = digitalRead(DHT_GPIO);

                if (count == 255) break;

                // 1이 26번 이상 반복되었다면 bit 값을 1로, 그렇지 않다면 0으로 판단함(펄스의 폭과 연관됨)
                // 그리고 읽은 bit 값을 shifting하여 MSB부터 LSB순으로 순서대로 채워가기
                data[dataIdx] = data[dataIdx] | ((count > LOWHIGH_THRESHOLD) << (7 - bit));
        }
}
```

위 코드는 실질적인 온습도 데이터를 받아오는 과정이다. DHT11 센서는 1byte 씩 총 5byte 의 데이터를 반환해 준다. 각각 온도 1byte, 온도 소숫점 아래 값 1byte, 습도 1byte, 습도 소숫점 아래값 1byte, 패리티비트 1byte 로 구성된다.

따라서 2 중 반복문을 사용해서 입력을 받도록 구현했다. data 배열의 한 칸은 8bits 이므로 shifting 연산을 통해 MSB 부터 시작해서 LSB 순으로 데이터를 채워갔다.

```
// 유효성 검사
unsigned char sum = 0;
for(int i = 0; i < 4; i++){
        // 마지막 1byte를 제외한 4byte의 합을 이용해 유효성 검사를 하기 위한 코드
        // 마지막 1byte는 패리티 비트이다
        sum += data[i];
}

if(sum == data[4]){   // 데이터가 유효한지 판단 조건 (data[4] : 패리티 비트)
        temp_result = data[2];    // 소수점 무시 (온도 소수점 윗자리)
        humid_result = data[0];   // 소수점 무시 (습도 소수점 윗자리)
}else{
        sleep(2);
        printf("checksum fail!\n");
}
```

위 코드는 패리티 비트를 의미하는 마지막 1byte 값을 통해 유효성 검사를 하는 과정이다. 온습도 값(4byte)을 모두 더한 값이 마지막 패리티 비트와 같다면 유효성을 통과한 것으로 간주한다.

유효성 검사를 통과했을 경우 전역변수인 temp_result, humid_result 를 초기화하고, 해당 값과 다른 센싱 값들이 mainPI 로 전달된다.

- 압력 및 조도 센서

압력 센서와 조도 센서는 SPI 라이브러리와 ADC 를 이용하여 구현했다. SPI 를 사용하면 여러개의 slave 를 물릴 수 있다. 압력 및 조도 센서는 아날로그 값을 센싱하기 때문에 디지털 값으로 변환하는 과정이 필요했고 MCP3008 ADC 를 사용했다.

```
void sense_press_and_light() {
        int fd = open(DEVICE, O_RDWR);

        if (fd <= 0) {
                printf("Device %s not found\n", DEVICE);
                return;
        }

        if (prepareSpi(fd) == -1) {
                return;
        }
        // 아날로그 값 -> 디지털 값
        // ADC에 물린 2개의 slave로 부터 데이터를 읽는다.
        press_result = readadc(fd, 0);   // 채널 0번으로 부터 읽음
        light_result = readadc(fd, 6);   // 채널 6번으로 부터 읽음

        close(fd);
}
```

위 코드는 압력과 조도를 센싱하는 함수이다. readadc 함수에 adc 의 파일 디스크립터와 adc 의 채널 번호를 설정해주면 adc 가 변환한 디지털 값을 읽을 수 있다. MCP 3008 ADC 에는 8 개의 채널이 있고 이 중 2 개의 채널(0 번과 6 번)에 센서를 물려 값을 읽었다. 위 코드를 보면 압력센서는 0 번 채널에, 조도 센서는 6 번 채널에 물렸음을 알 수 있다.

이 때 사용하는 두 개의 채널을 0 과 6 으로 사용한 이유는, 두 채널이 가까울 경우 서로에게 영향을 주어 값이 튀는 현상이 발생했기 때문이었다. 이 문제를 해결하기 위해서 저항을 이용해 값을 조금 낮춰주려고 노력했지만, 오히려 중간에 ADC 가 고장이 나기도 하여 0 번과 6 번을 사용하는 것으로 해결하였다.

- **초음파센서(산소센서 대체)**

초음파 센서는 순수히 GPIO 만을 사용하여 센싱이 가능하다. 두개의 구멍(하나는 IN 하나는 OUT)으로 통과하는 시간을 측정하고, 거리 = 시간 * 속력 공식을 이용해 거리를 측정한다.

```
GPIOWrite(POUT_ULTR, 0);

while(GPIORead(PIN_ULTR) == 0){
        start_t = clock();
}
while(GPIORead(PIN_ULTR) == 1){
        end_t = clock();
}

time = (double)(end_t - start_t)/CLOCKS_PER_SEC; // ms
oxygen_result = time/2 * 34000;   // time은 왔다 갔다 시간이므로 /2를 해줘야 함
```

초음파 센서의 두 구멍을 이용해 시간을 측정하고 이 시간 값을 통해 거리를 계산한다. 이때 왕복에 걸린 시간은 벽을 찍는데 걸리는 시간의 2 배이므로 공식에 대입할 때에는 시간의 0.5 배를 적용해야 한다.

3.2. 액추에이터 구현

actuatorPI 는 총 5 개의 thread 를 기반으로 하며 하나의 쓰레드는 mainPI 로 부터 문제의 값을 읽고 나머지 4 개의 쓰레드는 각각의 액추에이터를 담당한다. 각 액츄레이터는 parallel 하게 동작해야 하므로 멀티쓰레드를 이용해 구현했다.

센서에 직접적으로 반응하는 엑츄레이터는 정보표시를 위한 LCD 를 제외하고 4 개이다. 각 스레드를 create 함으로 스레드를 동작시키지만, 전역벽수가 -1 로 초기화되어있기 때문에 작동하지 않는다.

.

```
int motor_second = -1;//thread를 위한 전역변수 선언
int led1_sleep = -1;
int led2_sleep = -1;
int led3_sleep = -1;
pthread_t p_thread[4];
```

```
pthread_create(&p_thread[0], NULL, LED1start, NULL);//엑츄레이터 스레드 생성
pthread_create(&p_thread[1], NULL, LED2start, NULL);
pthread_create(&p_thread[2], NULL, LED3start, NULL);
pthread_create(&p_thread[3], NULL, motorstart, NULL);
```

위 사진은 actuatorPI 의 전역변수에 해당한다. mainPI 로 부터 문제의 값을 읽는 쓰레드(main thread)가 해당되는 전역변수에 -1 이 아닌 값을 초기화하면, 각 쓰레드는 자신에게 해당되는 전역변수의 값을 읽고 변화가 생겼을 경우 해당하는 액츄레이터를 동작시킨다.

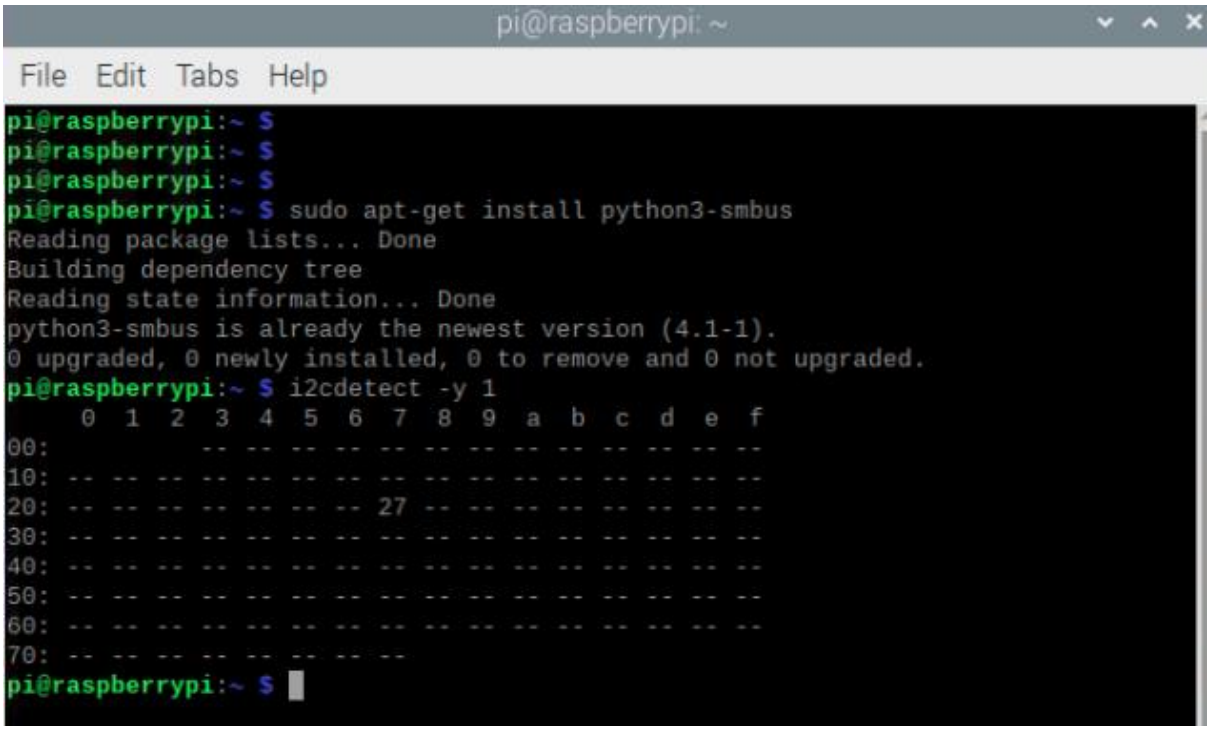
- LCD

우리가 사용한 1602 LCD 는 I2C 를 이용해 파이와 통신한다. 대부분 인터넷에서 찾아볼 수 있는 참조 문서들은 파이썬과 adafruit 을 이용한 방식이었고, I2C 에 대한 내용은 찾아보기 힘들었다. 고생 끝에 참조하여 작성한 코드는 크게 두 가지 부분으로 나눠진다.

첫 번째는 bus 를 open 하는 부분이고, 두 번째는 프로그램 분기에 따라 출력할 값은 open 한 bus 를 통해 출력하는 부분이다. Bus 를 Open 하는 부분은 여기를 참조하여 작성하였고, 출력 내용을 작성하는 부분은 다음의 python code 를 참조하여 작성하였다.

i2c 를 사용하기에 앞서, 라즈베리파이와 연결된 LCD 의 주소 값을 알아내야 사용을 할 수 있다. 따라서 라즈베리파이의 터미널에서 'i2cdetect

-y 1’을 입력하여 주소값을 알아낸다. 우리 조의 경우에는 0x27 이었기 때문에 코드에서는 상수로 선언하여 사용하였다.

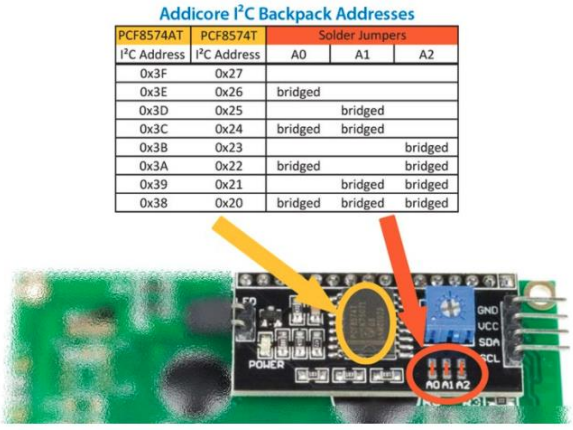


<lcd 의 주소를 확인한 장면>

```
/*
    Reference Python Code for i2c: https://bitbucket.org/MattHawkinsUK/rpispy-misc/raw/master/python/lcd_i2c.py
    Reference C Code for i2c bus opening : https://raspberry-projects.com/pi/programming-in-c/i2c/using-the-i2c-interface
*/

#define I2C_ADDR 0x27
```

<코드에 표시된 reference 사이트와 LCD slave 의 주소를 표시한 것>



I2C 주소는 lcd 모듈 뒷 면에 연결된 점퍼에 따라서 달라지는 것을 알 수 있는데, 우리는 어떤 점퍼도 연결하지 않았기 때문에 주소가 0x27 인 것 또한 알 수 있었다.

리눅스에서 I/O device 와의 소통은 모두 file 을 이용한다. I2C 도 마찬가지이며, bus_open() 에서는 i2c 파일을 open 하여 file 에 대한 descriptor 를 얻고, 획득한 fd 를 ioctl 함수의 인자로 넘겨줌으로써 하드웨어에 대한 통제 권한을 획득한다. ioctl 함수의 첫 번째 인자로 들어가는 i2c_SLAVE 는 linux/i2c-dev.h 에 미리 정의되어 있는 값이고, 여기에 우리가 선언한 i2c_ADDR 의 값을 넣어줌으로써 해당 디바이스에 대한 실행 권한을 획득한다.

```
void bus_open(){
  //OPEN I2C BUS
    char *filename = (char*)"/dev/i2c-1";
    if((file_i2c = open(filename, O_RDWR)) < 0){
        //ERROR
        printf("Failed to open i2c bus");
        return;
    }

    // int addr = 0x5a;
    if(ioctl(file_i2c, I2C_SLAVE, I2C_ADDR) < 0)
    {
        printf("Failed to acquire bus access and / or talk to slave. \n");
        //ERROR HANDLING; you can check errno to see what went wrong
        return;
    }

}
```

이후 lcd_init() 함수를 프로그램 시작 부분에서 호출하고, lcd_byte() 함수를 사용하여 lcd 에 출력하고자 하는 내용을 내보낸다. 우리가 lcd 에 출력하고자 하는 내용은 string 인데, bus 에는 write_text 함수는 character 를 지원하기 때문에 우리가 실제로 사용할 때는 1 character 씩 잘라서 보내줘야 함을 알 수 있다.

```
void lcd_string(char *message, char line){
    // Send string to display
    lcd_byte(line,LCD_CMD);

    for(int i = 0 ; i < strlen(message); i ++){
        lcd_byte(message[i],LCD_CHR);
    }
    for(int i = strlen(message);i < LCD_WIDTH;i++){
        lcd_byte(' ',LCD_CHR);
    }

}
```

< 출력 string 을 한 character 씩 잘라서 lcd_byte 함수를 이용해 출력하는 부분>

character 를 출력할 때 lcd 는 4-bit 씩 나눠서 출력하도록 되어 있기 때문에 bit masking 과 bit shifting 을 이용해 하나의 character (1 byte)를 4bit 씩 반으로 나눠서 출력한 것을 볼 수 있다.

```
void lcd_byte(char bits, int mode){
    // send byte to data pins
    // bits = data
    // mode 1 for data, 0 for command
    // 데이터를 4 비트 씩 쪼개서 전송한다! 따라서 비트마스킹 필요함

    char bits_high = mode | (bits & 0xF0) | LCD_BACKLIGHT;
    char bits_low = mode | ((bits<<4) & 0xF0) | LCD_BACKLIGHT;

    // High bits
    bus_write_text(bits_high);
    lcd_toggle_enable(bits_high);

    bus_write_text(bits_low);
    lcd_toggle_enable(bits_low);
}
```

- LED

순수히 GPIO 를 사용해서 구현했다. LED 는 총 3 개를 사용했고 각각의 LED 는 온도 대응, 습도 대응, 조도 대응에 해당한다.

```
while (1)
{
    if (led1_sleep >= 0) //main에서 전역변수 Led1_sleep를 0보다 크거나 같게 설정하면 if문이 실행
    {
        do
        {
            if (-1 == GPIOWrite(POUT, repeat % 2))
                return (3);
            usleep(100 * 1000 * led1_sleep);
            if (led1_sleep == 3 && repeat % 3 == 0)
                printf("on Brightness System : %d\n", repeat / 3); //시스템 조절 출력 - 정도에 따라 출력 횟수의 변화
            if (led1_sleep == 1 && repeat % 2 == 0)
                printf("on Brightness System : %d\n", repeat / 2); //시스템 조절 출력 - 정도에 따라 출력 횟수의 변화
        } while (repeat--);
        led1_sleep = -1;
        repeat = 10;
    }
}
```

여기서는 led1_sleep 전역변수의 변화를 감지(led1_sleep >= 0)하고 해당하는 led 의 밝기 및 반복 주기를 설정한다. while 문의 반복횟수는 10 회로 정해져 있지만, 10 회 중 극한값이 들어왔을 때는 10 을 2 로 나누어주어 빠르게 깜빡이는 효과를 주었고, 임계값을 넘었지만, 극한값이 아닌 값에 대해서는 10 을 3 으로 나누어주어 느리게 깜빡이는 효과를 주었다.

- 서보모터

산소 센서에 대한 대응을 의미하는 서보모터이다.

```
while (1)
{

    if (motor_second > 0) //main에서 전역변수 motor_second를 0보다 크게 설정하면 if문이 실행
    {
        PWMWriteDutyCycle(0, 100000);
        printf("oxygen fan : %d\n", motor_second); //1이상의 값을 넘겨주면 팬이 돌기 시작함.
        sleep(1);
        motor_second--;
    }
    else
    {
        PWMWriteDutyCycle(0, 1000000); //0이되면 쿨링팬이 작동하지 않음.
        sleep(1);
    }
}
```

PWM 라이브러리를 사용했고 period 대비 duty cycle 값을 이용해 모터를 얼마나 동작시킬지 결정했다. 모터가 동작하지 않는 경우는 period 의 값과 dutycycle 값을 일치시킴으로 모터가 회전하지 않도록 구현하였으며, mainPI 로 부터 전역변수 motor_second 값의 변화가 생겼을 경우에 period 의 값과 dutycycle 의 비율을 1/10 으로 변경해주어 모터를 안정적으로 동작하도록 하였다. mainPi 에서 수신한 산소농도에 따라 motor_second 값에 차이를 설정하였다. 그리고 motor_second--;를 통해 산소 농도에 따른 모터가 동작하는 시간에 차이를 두었다.

3.3. PI 간 통신 구현

같은 컴퓨터 내에 존재하는 프로세스 간에는 shared memory 또는 message passing 방법을 이용해 IPC 를 하지만, 현재 프로젝트와 같이 다른 컴퓨터에 존재하는 프로세스 간에는 socket 통신(message passing 일종)을 이용해 IPC 하는 것이 적절하기 때문에 socket 통신을 이용하여 파이 간 통신을 진행 했다.

socket 통신에서는 client 와 server 의 역할로 나뉘는데, server 가 먼저 port 를 열어 두어야 client 와 통신할 수 있다. 소켓 통신에서는 상대방의 IP 와 포트 번호가 필수적으로 필요로 된다.

세 개의 PI 모두 같은 공유기에 무선으로 연결했기 때문에 라우팅 없이 소켓을 이용한 IPC 를 구현했다.

- ### sensorPI 와 mainPI 간 통신

  sensorPI 와 mainPI 간 통신에서는 sensorPI 가 client, mainPI 가 server 역할을 하도록 구현했으며 IPv4 와 TCP 프로토콜을 이용한 통신을 했다.

```c
static void usingSocket(int *sock, char* inputServerInfo[], struct sockaddr_in* serv_addr){
        *sock = socket(PF_INET, SOCK_STREAM, 0);
        if(*sock == -1)
                error_handling("socket() error");

        memset(serv_addr, 0, sizeof(struct sockaddr_in));
        serv_addr->sin_family = AF_INET;
        serv_addr->sin_addr.s_addr = inet_addr(inputServerInfo[1]);
        serv_addr->sin_port = htons(atoi(inputServerInfo[2]));

        // sensorPI는 client입장이므로 미리 listen하고 있는 서버가 있어야한다
        if(connect(*sock, (struct sockaddr*)serv_addr, sizeof(struct sockaddr)) == -1)
                error_handling("connect() error");

        printf("Success conntect to Main PI\n");
}
```

  위 코드는 sensorPI 에서 서버(mainPI)와 소켓 통신을 연결하는 과정이다. 입력받은 서버의 ip 및 port 번호를 이용해 구조체를 만들고 connect 시스템 콜을 통해 OS 에게 서버와 연결할 수 있도록 요청한다.

```
serv_sock = socket(PF_INET, SOCK_STREAM, 0);
if (serv_sock == -1)
    error_handling("socket() error");

memset(&serv_addr, 0, sizeof(serv_addr));
serv_addr.sin_family = AF_INET;
serv_addr.sin_addr.s_addr = htonl(INADDR_ANY);
serv_addr.sin_port = htons(atoi(argv[1]));

if (bind(serv_sock, (struct sockaddr *)&serv_addr, sizeof(serv_addr)) == -1)
    error_handling("bind() error");
if (listen(serv_sock, 5) == -1)
    error_handling("listen() error");
    printf("Connected to Sensor Pi...\n");
if (clnt_sock < 0)
{
    clnt_addr_size = sizeof(clnt_addr);
    clnt_sock = accept(serv_sock, (struct sockaddr *)&clnt_addr,
                       &clnt_addr_size);
    sensor_socket = clnt_sock;
    if (clnt_sock == -1)
        error_handling("accept() error");
}
printf("Successfully Connected to Sensor Pi...\n");
```

위 사진은 서버역할을 하는 mainPI 에서 sensorPI 와 소켓통신을 위한 코드이다. 서버 역할을 하기 때문에 클라이언트가 connect 요청을 하기 전에 미리 소켓을 열어 두어야 하며 listen 상태로 기다려야 한다.

- mainPI 와 actuatorPI 간 통신

mainPI 와 actuatorPI 간 통신에서는 mainPI 가 client, actuatorPI 가 server 역할을 하도록 구현했으며 IPv4 와 TCP 프로토콜을 이용한 통신을 했다.

```
sock = socket(PF_INET,SOCK_STREAM,0);
act_socket = sock;
//printf("%d\n",sock);
memset(&actu_addr, 0, sizeof(actu_addr));
actu_addr.sin_family = AF_INET;
// printf("hello\n");
actu_addr.sin_addr.s_addr = inet_addr(addr);
// printf("%d\n",port_num);
actu_addr.sin_port = htons(port_num);
// printf("1111\n");
printf("Connecting to Actuator Pi....\n");
if(connect(sock,(struct sockaddr *)&actu_addr,sizeof(actu_addr))==-1){
    error_handling("connect() error");
}
printf("Successfully Connected to Actuator Pi\n");
```

위 사진은 mainPI 와 actuatorPI 간 통신에서 client 역할을 하는 mainPI 의 코드이다. connect 시스템 콜을 통해 서버(actuatorPI)와의 연결을 요청하고 있다.

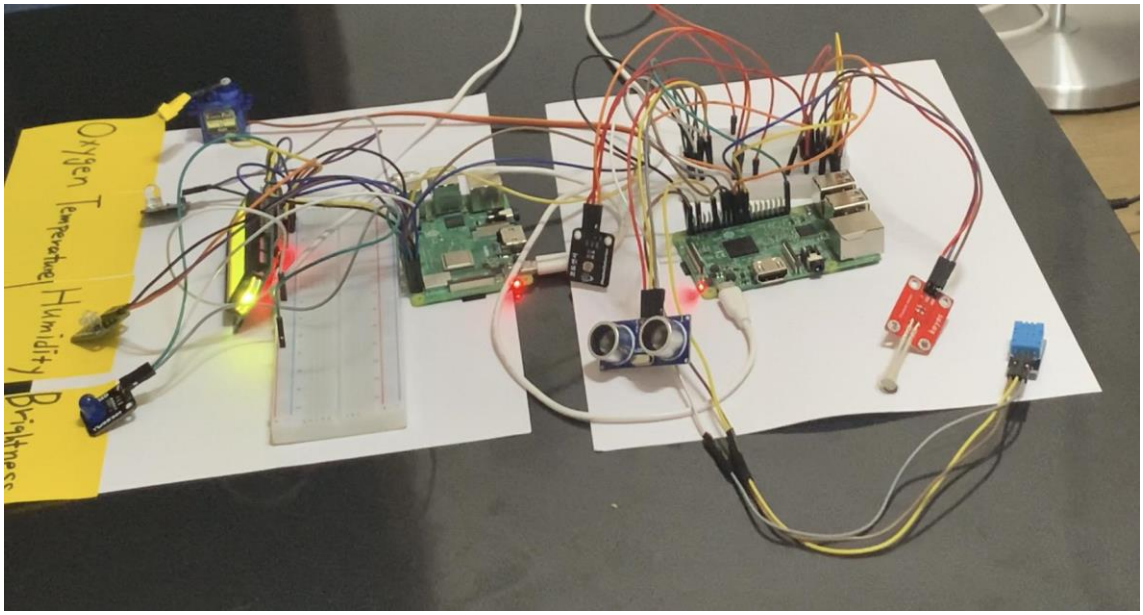
```
memset(&serv_addr, 0, sizeof(serv_addr));
serv_addr.sin_family = AF_INET;
serv_addr.sin_addr.s_addr = htonl(INADDR_ANY);
serv_addr.sin_port = htons(atoi(argv[1]));

if (bind(serv_sock, (struct sockaddr *)&serv_addr, sizeof(serv_addr)) == -1)
    error_handling("bind() error");
if (listen(serv_sock, 5) == -1)
    error_handling("listen() error");
if (clnt_sock < 0)
{
    clnt_addr_size = sizeof(clnt_addr);
    clnt_sock = accept(serv_sock, (struct sockaddr *)&clnt_addr,
                       &clnt_addr_size);
    if (clnt_sock == -1)
        error_handling("accept() error");
}
printf("Success connect to Main Pi\n");// if문 없이 출력이 된다면 연결 성공
```

위 코드는 서버 역할을 하는 actuatorPI 에서 client 와 소켓 통신을 위한 코드이다. 서버이기 때문에 client 가 connect 요청을 하기 전에 먼저 listen 하고 있어야 한다.

- htons() 함수는 숫자가 메모리에 네트워크 바이트 순서로 저장되도록한다. 이때 MSB 부터 시작한다. 따라서 숫자를 구성하는 바이트를 스왑하여 메모리에 바이트가 순서대로 저장되도록 한다. 즉, 빅 엔디안 컴퓨터에서 작업하는 경우 htons() 함수는 숫자가 메모리에 올바른 방식으로 이미 저장되어 있으므로 스와핑을 수행 할 필요가 없다. 라즈베리파이 아키텍쳐는 리틀 엔디안 기반이므로 htons() 시스템 콜이 사용되었다.

4. 결과



4.1. 추가 구현 사항

- 예상치 못한 종료에 대한 시그널 처리
  socket 통신 및 각종 라이브러리로 부터 export 해 둔 파일들을 안전하게 close 해주는 과정이 필요하다. 하지만 ctrl + c 등으로 인해 프로세스를 강제로 종료시킬 경우, close()가 호출되지 않고 프로세스가 강제종료 되기 때문에 ctrl + c 으로 종료되는 상황에 대한 시그널 핸들링이 필요했다.

```
void exit_handler(int sig){
   close(act_socket);
   printf(" main PI exit from act_socket\n");
   exit(1);
}

void exit_handler2(int sig){
   close(sensor_socket);
   printf(" main PI exit from sensor_socket\n");
   exit(1);
}
```

위 사진은 mainPI 에서 정의한 시그널 핸들러이다. ctrl + c 가 입력되었을 때 발생한 시그널(SIGINT)에 대해 대응하도록 정의했다. 열려있는 fd 에 대해 안전하게 close 를 한 뒤 프로세스를 종료시키는 과정을 포함하고 있다.

```
signal(SIGINT, exit_handler);
signal(SIGINT, exit_handler2);
```

위 코드는 위에서 정의한 시그널 핸들러를 등록하는 과정이다. 이 과정이 있어야 해당하는 시그널이 발생했을 경우 정의한 시그널 핸들러가 호출될 수 있다.

4.2. 개선사항

1. PWM 채널 활성화 라즈베리파이에는 PWM 채널이 두개가 존재하는데, 이중 두번째 채널인 PWM 을 생성하는 것 까지 성공하였으나, 해당 핀과 엑츄레이터를 연결하여 동작하는 부분에 대한 오류를 해결하지 못하였다. period 와 cycle 값을 설정 할 수 있었지만, 작동하지 않았고, PI 의 오류인지 프로그램의 오류인지 확인하지 못하였다. 수업시간에 학습한 내용이지만, PWM 로 LED 를 구현하여 조도센서의 값에 따라 자동으로 불의 밝기를 조절하는 기능을 구현하지 못한 점이 아쉬운 부분이다.

2. 산소농도센서 구매의 비용문제로 인해 산소센서를 초음파센서로 대체 하였기 때문에, 산소농도센서를 구현하지 못한 것이 아쉬운 부분이다. 센서를 통한 입력 값을 확인하고, 이에 맞는 알고리즘과 임계값 설정 등의 개발환경이 뒷받침 되었다면 조원들의 경험과 실력향상에 도움이 되었겠지만, 수업시간에 학습한 초음파센서로 대체하였기 때문에, 산소농도센서의 매커니즘에 대해 학습하지 못하였다.

3. 현재는 시스템구조를 mainPI, sensorPI, actuatorPI 로 구분지어 개발하였다. 하지만 사용자 PI 와 관리자 PI 로 구분짓고, 모듈화와 인캡슐레이션을 활용한다면, 제 3 자로 하여금 시스템구조를 더욱 쉽게

파악할 수 있지만, 조원들의 역할분담을 위해 사용자 PI, 관리자 PI 로 구현을 하지 못한점이 아쉽다.

4. 모터를 제외한 액츄에이터 를 시뮬레이션으로 구현을 한 점이 아쉽다. 복잡한 동작을 해야하는 엑츄에이터가 많았기 때문에, 1 번 PWM 채널을 활용한다고 하여도 시뮬레이션으로 대체하여야 하는 부분이 많았다. GPIO,PWM 방식의 한계이고, mainPI, sensorPI, actuatorPI 구조의 한계이기 때문에, 이를 극복하지 못한점이 아쉬운 부분이다.

5. 고찰

조원들은 모두 임베디드프로그래밍에 대해 이론적으로 기본적인 부분은 인지하고 있는 상태였지만, 실제 구현에 많은 어려움을 겪었다. 지금까지 해오던 프로그래밍과 다른 부분이 많았기 때문에 개념적인 부분에서 이해를 하는데 어려움이 있었기 때문이다. 하지만 실습수업과 자료조사를 통해 학습을 한 결과 개념을 이해하기 시작하였고, 개념을 이해하니 퍼즐이 맞춰진 듯 다른 과목에서 이수한 내용들과의 접점이 생기며 이해와 구현에 가속도가 붙었다. 또한 MultiThread 환경에서의 프로그래밍과 IPC 를 직접 구현해 보았는데, OS 과목에서 이론으로 학습한 내용을 직접 구현할 수 있는 기회를 가져 매우 좋았다. 실습을 통해 기본적인 thread 에 대해 학습하고, 이를 기반으로 더 심화된 thread 를 구현함으로 좋은 경험이 됐다. low-level 의 SW 의 역할에 대해 알아 볼 수 있었고 이들이 HW 와 상호작용을 통해 PC 가 구동될 수 있음을 배웠다. 매우 복잡한 프로그램을

구현하는 것은 아니지만, 조원들이 직접 구현을 하여 물리적인 작용을 하는 센서들과 엑츄레이터를 보며 학습에대한 성취감을 느꼈다.

구현을 하는데 있어 센서들이 망가진 경우가 가장 힘들었던것 같다. 한번은 ADC 와 온습도센서가 작동하지 않아 어떤 부품이 문제가 있을지 경우의 수를 따지고 코드를 점검하며 밤을 새웠지만, 알고보니 ADC 와 온습도센서가 모두 고장이 난 일이 있었다. 이렇듯 HW 적인 부분에서 대처하기 힘든 변수도 있기 때문에, 자신이 만든 프로그램에 확신이 없다면 정말 어려운 프로그래밍이 될수도 있다는 생각을 했다.

그리고 임계값을 설정하거나 변수값으로 동작의 정도를 설정하는 등의 값을 설정하는데 정말 많은 시행착오가 있었다. 많은 시도 끝에 적절히 작동하는 설정값들을 찾았고, 임베디드프로그래머의 고행을 알게되었다. 경험적으로 설정해야 하는 값들의 적정값을 찾는 것이 얼마나 힘든 일인지 느낄 수 있었고, 딥러닝 분야에서 하이퍼 파라미터들을 일일히 찾아가는 이유와 그 과정이 얼마나 힘든 것인지도 이해할 수 있었다.
또한 각자 구현한 내용을 병합하는 과정에서 2 박 3 일간 합숙을 했는데, 마치 해커톤을 진행하는 것과 같은 느낌을 받았으며 그 과정에서 조원들과 더 친해질 수 있었고, 좋은 인연을 만들게 된 것 같았다.

이번 프로젝트를 통해 각 조원들의 능력이 임베디드프로그래밍을 포함한 여러 영역에서 향상되었음을 느끼며, 적성을 찾는데 도움이되고 조원들의 미래에 도움이 되었다는 생각이 들었다.

6. Reference

- DHT11(온습도센서) 데이터시트:

https://www.mouser.com/datasheet/2/758/DHT11-Technical-Data-Sheet-Translated-Version-1143054.pdf

- Reference Python Code for i2c:

https://bitbucket.org/MattHawkinsUK/rpispy-misc/raw/master/python/lcd_i2c.py

- Reference C Code for i2c bus opening :

https://raspberry-projects.com/pi/programming-in-c/i2c/using-the-i2c-interface

- 프로젝트 소스코드 주소

https://git.ajou.ac.kr/YongSangHo/systemprogramming_mebeforeyou/-/tree/master