

스마트홈 관리 시스템

마이크로프로세서응용 기말 프로젝트 보고서

김성빈(22100113), 최준혁(21900764)

한동대학교 전산전자공학부

22100113@handong.ac.kr, 21900764@handong.ac.kr

요약

이 보고서는 스마트 홈 환경에서 실내 온도, 습도, 이산화탄소 농도, 소음 수준을 모니터링하는 시스템의 설계 및 구현에 대한 내용을 다룹니다. 이를 위해 다양한 센서와 주변 장치를 사용하여 데이터를 수집하고, 마이크로컨트롤러를 통해 실시간으로 처리 및 전송하는 방법을 설명합니다. 주요 센서로는 DHT22(온습도 측정), CO2 센서, 소음 센서가 사용되었으며, LED 매트릭스 디스플레이(HT16K33)를 통해 시각적 피드백을 제공합니다. 각 센서는 각각의 통신 표준(GPIO, UART, I2C, ADC)을 사용하여 nRF52840DK 보드와 통신합니다. 이 보고서는 또한 프로세스 및 인터럽트 관리, 인터럽트 설정(우선순위, ISR 등)에 대해 다루며, 시스템의 효율성과 안정성을 높이기 위한 방법을 제시합니다. 본 프로젝트를 통해 사용자에게 실내 환경의 실시간 모니터링 기능을 제공함으로써 쾌적하고 건강한 생활 환경을 유지할 수 있도록 하는 것을 목표로 합니다.

목차

프로젝트 개요.....	6
문제 정의.....	6
프로젝트의 중요성.....	6
사용자 인터페이스 정의.....	6
사용된 입출력 장치 목록.....	6
입력장치.....	6
출력장치.....	7
설계 및 구현.....	8
개발환경 설명.....	8
하드웨어 환경.....	8
탑재된 시스템 소프트웨어.....	8
개발에 사용된 언어 및 IDE.....	8
사용된 주변장치 상세 내역.....	8
빌드 환경(CMakeList.txt).....	8
프로그램의 구조.....	9
Flow Chart.....	9
모듈 설정 (prj.conf).....	10
주변장치.....	11
디바이스 트리 (devicetree overlay).....	11
디바이스 트리 노드 및 구성.....	12
1. DHT22 센서 노드.....	12
2. 사용자 IO 채널 노드.....	12
3. I2C0 버스 및 HT16K33 LED 매트릭스 노드.....	12
4. UART 인터페이스 노드.....	13
5. 핀 제어 설정.....	13
6. ADC 채널 노드.....	14
주변장치별 프로세서와 통신 표준 설명.....	14
DHT22 센서.....	14
통신 표준: GPIO.....	14
CO2 센서.....	15
통신 표준: UART.....	15

HT16K33 LED 매트릭스.....	15
통신 표준: I2C.....	15
소음 센서.....	15
통신 표준: ADC, GPIO.....	15
프로세스 및 인터럽트 관리.....	16
인터럽트 설정 (우선순위, ISR 등).....	16
버튼 ISR.....	16
타이머 ISR.....	16
구현 결과.....	17
사용자 입력 장치 사용방법 설명.....	17
동작시연 (데모 동영상과 추가 이미지를 사용하여 설명).....	17
데모 동영상.....	17
데모 이미지와 설명.....	17
자체 평가.....	22
난이도 평가 (평가 기준에 근거하여 제시).....	22
구현복잡도 및 완성도: 사용장치 수의 종류에 비례, 최종 데모의 성공 정도:.....	22
기술 난이도: 입출력 프로토콜의 종류, 인터럽트 복잡도, 어셈블리어 사용 정도.....	22
계획대비 완성도.....	24
조원의 개인 기여 내용.....	24
부록.....	25
작성된 파일 목록 및 설명 (프로젝트 README 파일 내용).....	26
코드 설명.....	26
온습도 센서 관련 코드.....	26
DHT22 장치 초기화.....	26
init_dht.....	27
get_temperature (get_humidity).....	27
CO2 센서 관련 코드.....	29
ckeck_uart_fsm.....	29
getChecksum.....	31
serial_callback.....	31
serial_write.....	33
사운드 센서 관련 코드.....	34
ADC 버퍼 및 시퀀스 구조체.....	34

ADC 채널 설정.....	34
ADC 데이터 읽기.....	34
사운드 값 처리.....	35
BLE 관련 코드.....	35
데이터 수신 처리 함수.....	35
데이터 송신 완료 처리 함수.....	36
CCCD 변경 처리 함수.....	36
GATT 서비스 및 특성 등록.....	37
Multi-thread 관련 코드 (I2C LED 출력).....	38
스레드 스택 선언.....	38
스레드 데이터 구조 선언.....	38
디스플레이 파라미터 구조체.....	38
디스플레이 스레드 함수.....	38
값을 디스플레이하기 위한 스레드 생성.....	39
개발과정에서 만났던 에러가 있는 경우, 에러메시지와 그 대처 방안 설명.....	40
개인 소감.....	40
개발과정에서 참조 했던 리소스.....	41
(1) 블로그 등 URL, 하드웨어 업체 제공 공식 문헌.....	41
(2) Youtube, StackOverflow, 개발 Community, Chat-GPT 등의 활용 내역.....	41

프로젝트 개요

문제 정의

현대 사회에서 스마트 홈 기술은 가정의 편의성과 안전성을 크게 향상시키고 있습니다. 이 프로젝트의 궁극적인 목표는 스마트 홈 환경에서 온습도, CO₂, 그리고 소음 데이터를 실시간으로 측정하고 이를 사용자에게 제공하는 통합 모니터링 시스템을 설계하는 것입니다. 이를 통해 사용자는 공기질과 소음 수준을 실시간으로 모니터링하여 쾌적하고 건강한 생활 환경을 유지할 수 있습니다.

프로젝트의 중요성

실내 공기질과 소음 수준은 건강과 생활의 질에 큰 영향을 미칩니다. 높은 CO₂ 농도는 집중력 저하와 피로감을 유발할 수 있으며, 온도와 습도 역시 쾌적한 실내 환경 유지에 중요한 요소입니다. 또한, 지속적인 소음 노출은 스트레스와 수면 장애를 초래할 수 있습니다. 따라서, 이러한 요소들을 실시간으로 모니터링하고 적절한 조치를 취할 수 있는 시스템의 필요성이 대두되고 있습니다.

사용자 인터페이스 정의

사용자가 쉽게 접근하고 사용할 수 있는 인터페이스를 제공합니다. 이를 통해 사용자는 언제 어디서나 실내 환경을 모니터링하고 필요 시 조치를 취할 수 있습니다. 사용자 인터페이스는 센서 데이터를 HT16K33 LED Matrix를 통해 확인할 수 있도록 설계되었습니다. 추가로 BLE Low-Energy(BLE) 통신을 통해 임베디드 환경에서 중요한 저전력 시스템을 설계하였습니다.

사용된 입출력 장치 목록

입력장치

- DHT22 온습도 센서
- CO₂ 센서
- 마이크 센서
- nRF52840DK board BTN 1, 2, 3, 4

출력장치

- HT16K33 LED Matrix
- nRF52840DK board LED 1, 2, 3, 4

설계 및 구현

개발환경 설명

하드웨어 환경

- nRF52840 SoC를 탑재한 nRF52840 DK 보드
 - nRF52840 SoC: <https://www.nordicsemi.com/Products/nRF52840>
 - nRF52840 DK: <https://www.nordicsemi.com/Products/Development-hardware/nRF52840-DK>

탑재된 시스템 소프트웨어

- Zephyr RTOS
 - Zephyr Official Website: <https://www.zephyrproject.org/>

개발에 사용된 언어 및 IDE

- 프로그래밍 언어: C, Assembly
- 개발 IDE: VSCode
 - VSCode Extension: nRF Connect
 - (<https://www.nordicsemi.com/Products/Development-tools/nRF-Connect-for-VS-Code>)

사용된 주변장치 상세 내역

- DHT22 센서, CO2 센서, 마이크 센서, HT16K33 LED 드라이버

빌드 환경 (**CMakeList.txt**)

```
cmake_minimum_required(VERSION 3.20.0)
find_package(Zephyr REQUIRED HINTS $ENV{ZEPHYR_BASE})

project(ble_peripheral_example)

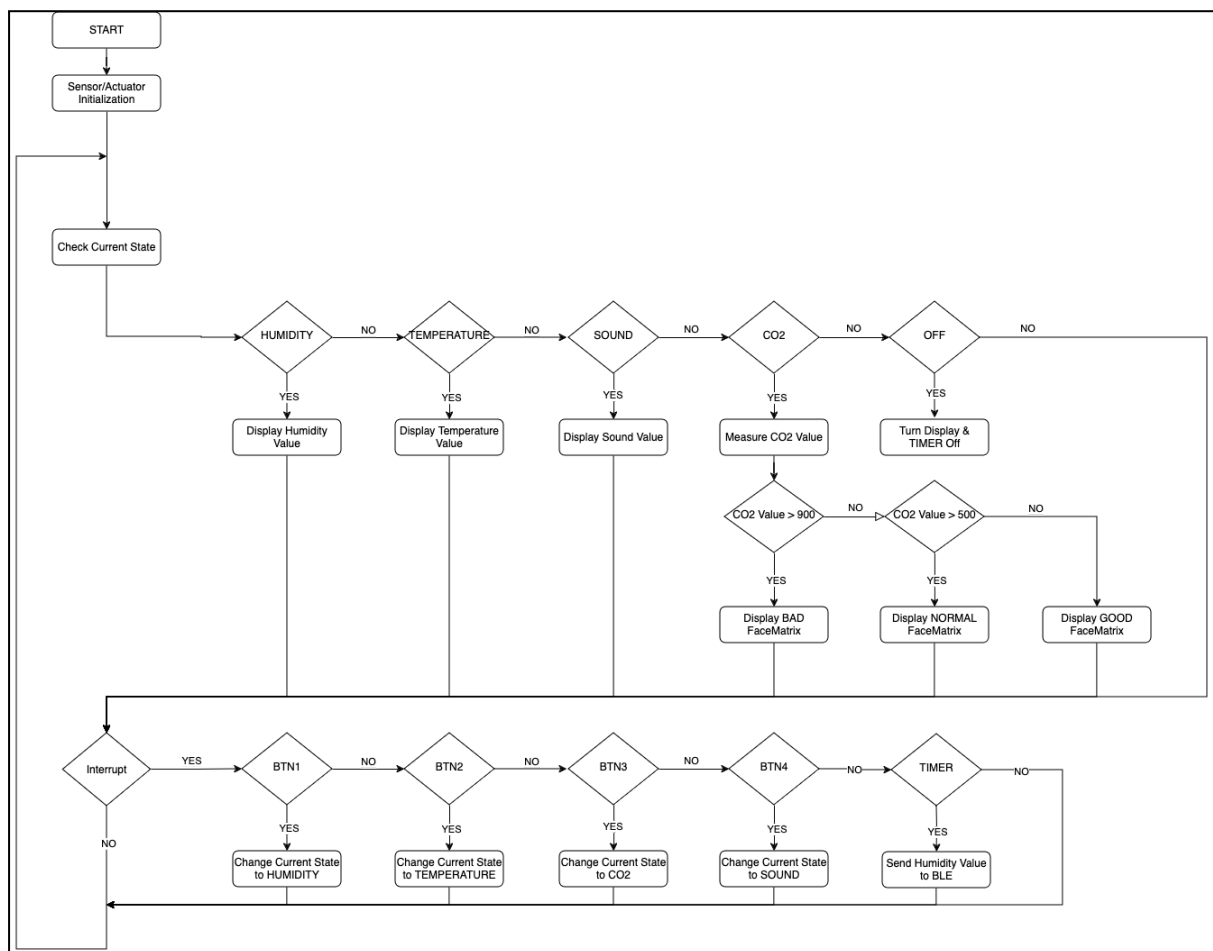
target_sources(app PRIVATE main.c src/my_service.c src/buttons.c
src/ht16k33_led.c src/dht22_sensor.c)
```

```
target_include_directories(app PRIVATE include)
```

해당 *target_sources* 값에 사용되는 소스 코드 파일들: *main.c*, *my_service.c*, *buttons.c*, *ht16k33_led.c*, *dht22_sensor.c*을 경로에 맞게 추가하여 VSCode extension인 nRF-Connect와 사용가능하도록 CMakeList.txt를 설정하였다.

프로그램의 구조

Flow Chart



Flow chart는 프로그램의 흐름과 논리를 시각적으로 표현하는 도구입니다. 프로그램의 구조, 데이터 흐름, 의사 결정 경로 등을 도식화한다.

본 프로그램의 flow chart를 나타낸 위 그림은 다음 단계들을 지닌다: 초기 단계는 각 센서를 초기화하고 값을 읽어들이기 준비를 거친다. 해당 프로그램은 Finite State Machine(FSM)를 모방하여, state를 줌으로

실시간성을 보장하며 동작하도록 하였다. 현재 **state**가 무엇인지 따라 매 주기 행동이 정해져있다. 이 과정을 반복하는 중에 Interrupt가 발생하는 경우 Service Routine에 따라 행동을 한다. 버튼(BTN1~BTN4)이 interrupt를 발생시킨 경우 현재 **state**가 새로 업데이트되고 타이머가 interrupt를 발생시킨 경우 습도 데이터를 Bluetooth Low Energy(BLE) 통신을 통해 사용자에게 전달한다. 이 과정은 주기적으로 반복된다.

모듈 설정 (*prj.conf*)

```
CONFIG_BT=y
CONFIG_BT_L2CAP_TX_BUF_COUNT=5
CONFIG_BT_PERIPHERAL=y
CONFIG_BT_DEVICE_NAME="SmartHome"
CONFIG_BT_DEVICE_APPEARANCE=962

CONFIG_HEAP_MEM_POOL_SIZE=2048

# This example requires more workqueue stack
CONFIG_SYSTEM_WORKQUEUE_STACK_SIZE=2048

# GPIO INT
CONFIG_SENSOR=y
CONFIG_DHT=y
CONFIG_GPIO=y
CONFIG_PRINTK=y
CONFIG_LOG=y

# HT16K33 LED
CONFIG_I2C=y
CONFIG_LED=y
CONFIG_KSCAN=y
CONFIG_KSCAN_INIT_PRIORITY=95
CONFIG_HT16K33_KEYSCAN=y

# Need float format support
CONFIG_NEWLIB_LIBC=y
CONFIG_NEWLIB_LIBC_FLOAT_PRINTF=y
CONFIG_CBPRINTF_FP_SUPPORT=y

# CO2
CONFIG_UART_INTERRUPT_DRIVEN=y

# sound
```

```
CONFIG_ADC=y
CONFIG_ADC_ASYNC=y
CONFIG_ADC_NRF52_SAADC=y
```

prj.conf 파일은 Zephyr RTOS(실시간 운영체제)와 같은 프로젝트에서 사용되는 구성 파일로, 프로그램의 빌드 및 실행에 필요한 설정을 정의합니다. 이 파일의 주요 역할은 다음과 같습니다:

1. 빌드 설정: 컴파일러 옵션, 빌드 플래그 등 빌드 과정에 필요한 다양한 설정을 지정합니다.
2. 커널 설정: 스레드, 메모리 관리, 스케줄링 등 커널 관련 설정을 구성합니다.
3. 드라이버 및 하드웨어 설정: 센서, 통신 모듈, **GPIO** 등 하드웨어와 관련된 설정을 정의합니다.
4. 애플리케이션 설정: 애플리케이션이 필요로 하는 기능과 라이브러리의 활성화 여부를 설정합니다.

prj.conf 파일을 통해 개발자는 프로젝트의 전반적인 설정을 중앙에서 관리할 수 있으며, 이를 통해 일관된 환경에서 빌드 및 실행이 이루어지도록 합니다.

위 그림에 나타난 코드는 본 프로젝트를 실행하는데 필요한 **dependencies**와 설정해야하는 정보들을 담은 *prj.conf* 파일이다. 대표적으로, **General Purpose I/O(GPIO)** 관련으로 **CONFIG_GPIO**이 존재하고, 블루투스 통신을 위해 **CONFIG_BT** 등이 존재한다.

주변장치

디바이스 트리 (devicetree overlay)

디바이스 트리 오버레이(Device Tree Overlay)는 임베디드 시스템에서 하드웨어 구성 요소와 그들의 속성을 정의하는 데 사용됩니다. 특히, Zephyr RTOS와 같은 시스템에서는 다음과 같은 역할을 수행합니다:

1. 하드웨어 구성 정의: 보드에 연결된 다양한 하드웨어 장치(센서, 통신 모듈, **GPIO** 등)의 구성을 명시합니다.
2. 드라이버 매핑: 각 하드웨어 장치에 적합한 드라이버를 지정하여, 운영체제가 해당 장치를 올바르게 인식하고 제어할 수 있도록 합니다.
3. 핀 설정: 마이크로컨트롤러의 특정 핀에 대한 설정(입출력 방향, 풀업/풀다운 저항 등)을 정의합니다.
4. 확장성 제공: 기본 디바이스 트리에서 추가적인 하드웨어 변경이나 추가를 쉽게 반영할 수 있도록 하여, 확장성과 유연성을 높입니다.

디바이스 트리 오버레이를 통해 개발자는 하드웨어 설정을 코드와 분리하여 관리할 수 있으며, 이를 통해 시스템의 설정 변경과 유지보수가 용이해집니다. 본 프로젝트에서 사용된 오버레이에는 DHT22 센서, ADC 채널, HT16K33 LED 매트릭스, UART 인터페이스 및 관련 핀 제어 설정에 대한 구성이 포함되어 있습니다. 오버레이의 정확한 구성은 다음과 같습니다(해당 코드 이미지와 함께 설명):

디바이스 트리 노드 및 구성

1. DHT22 센서 노드

```
dht22 {
    compatible = "aosong,dht";
    status = "okay";
    dio-gpios = <&gpio0 13 (GPIO_PULL_UP | GPIO_ACTIVE_LOW)>;
    dht22;
};
```

- a. **compatible:** 디바이스가 "aosong,dht" 드라이버와 호환됨을 나타냅니다.
- c. **status:** 디바이스가 활성 상태("okay")임을 표시합니다.
- d. **dio-gpios:** 데이터 입출력에 사용할 GPIO 구성을 지정하며, GPIO 포트 0, 핀 13을 풀업 및 액티브 로우 설정으로 사용합니다.

2. 사용자 IO 채널 노드

```
zephyr,user {
    io-channels = <&adc 4>;
};
```

- a. **io-channels:** 사용자 정의 입출력에 ADC 채널 4를 사용함을 나타냅니다.

3. I2C0 버스 및 HT16K33 LED 매트릭스 노드

```
&i2c0 {
    clock-frequency = <I2C_BITRATE_STANDARD>;

    You, 2 weeks ago | 1 author (You)
    ht16k33_ledmatrix: ht16k33@70 {
        compatible = "holtek,ht16k33";
        reg = <0x70>;

        You, 2 weeks ago | 1 author (You)
        keyscan {
            compatible = "holtek,ht16k33-keyscan";
        };
    };
};
```

- a. **clock-frequency:** I2C 버스의 클럭 주파수를 표준 비트 전송률로 설정합니다.
- c. **ht16k33_ledmatrix:** HT16K33 LED 매트릭스 장치 노드를 정의합니다.
 - i. **compatible:** 디바이스가 "holtek,ht16k33" 드라이버와 호환됨을 나타냅니다.
 - ii. **reg:** 디바이스의 I2C 주소를 0x70으로 설정합니다.
 - iii. **keyscan:** HT16K33의 키스캔 기능을 정의합니다.
 1. **compatible:** 디바이스가 "holtek,ht16k33-keyscan" 드라이버와

호환됨을 나타냅니다.

4. UART 인터페이스 노드

```
arduino_serial: &uart1 {
    status = "okay";
    compatible = "nordic,nrf-uart";
    current-speed = <9600>;
    pinctrl-0 = <&uart1_default>;
    pinctrl-1 = <&uart1_sleep>;
    pinctrl-names = "default", "sleep";
};
```

- a.
- b. **arduino_serial**: UART1 인터페이스를 정의합니다.
 - i. **status**: 인터페이스가 활성 상태("okay")임을 표시합니다.
 - ii. **compatible**: 디바이스가 "nordic,nrf-uart" 드라이버와 호환됨을 나타냅니다.
 - iii. **current-speed**: UART 통신 속도를 9600bps로 설정합니다.
 - iv. **pinctrl-0**: 기본 핀 제어 설정을 "uart1_default"로 지정합니다.
 - v. **pinctrl-1**: 슬립 모드 핀 제어 설정을 "uart1_sleep"로 지정합니다.
 - vi. **pinctrl-names**: 핀 제어 설정의 이름을 "default"와 "sleep"로 지정합니다.

5. 핀 제어 설정

```
&pinctrl {
    choijh4161@gmail.com, last week | 1 author (choijh4161@gmail.com)
    uart1_default: uart1_default {
        choijh4161@gmail.com, last week | 1 author (choijh4161@gmail.com)
        group1 {
            psels = <NRF_PSEL(UART_RX, 1, 10)>;
            bias-pull-up;
        };
        choijh4161@gmail.com, last week | 1 author (choijh4161@gmail.com)
        group2 {
            psels = <NRF_PSEL(UART_TX, 1, 8)>;
        };
    };

    choijh4161@gmail.com, last week | 1 author (choijh4161@gmail.com)
    uart1_sleep: uart1_sleep {
        choijh4161@gmail.com, last week | 1 author (choijh4161@gmail.com)
        group1 {
            psels = <NRF_PSEL(UART_RX, 1, 10)>,
                <NRF_PSEL(UART_TX, 1, 11)>;
            low-power-enable;
        };
    };
};
```

- a.
- b. **uart1_default**: UART1의 기본 핀 설정을 정의합니다.
 - i. **group1**: UART1의 RX 핀을 포트 1, 핀 10으로 설정하고 풀업 저항을 사용합니다.
 - ii. **group2**: UART1의 TX 핀을 포트 1, 핀 8로 설정합니다.
- c. **uart1_sleep**: UART1의 슬립 모드 핀 설정을 정의합니다.
 - i. **group1**: UART1의 RX 핀을 포트 1, 핀 10으로, TX 핀을 포트 1, 핀 11로 설정합니다. 저전력 모드를 활성화합니다.

6. ADC 채널 노드

```
&adc {
    #address-cells = <1>;
    #size-cells = <0>;

    choijh4161@gmail.com, 7 days ago | 1 author (choijh4161@gmail.com)
    channel@4 {
        reg = <4>;
        zephyr,gain = "ADC_GAIN_1";
        zephyr,reference = "ADC_REF_INTERNAL";
        zephyr,acquisition-time = <ADC_ACQ_TIME_DEFAULT>;
        zephyr,input-positive = <NRF_SAADC_AIN4>; /* P0.28 */
        zephyr,resolution = <10>;
    };
};
```

- a. `};`
- b. **#address-cells**: 주소 셀의 개수를 1로 설정합니다.
- c. **#size-cells**: 크기 셀의 개수를 0으로 설정합니다.
- d. **channel@4**: ADC 채널 4를 정의합니다.
 - i. **reg**: 채널 번호를 4로 설정합니다.
 - ii. **zephyr,gain**: ADC 이득을 1로 설정합니다.
 - iii. **zephyr,reference**: ADC 참조 전압을 내부 참조 전압으로 설정합니다.
 - iv. **zephyr,acquisition-time**: 데이터 획득 시간을 기본값으로 설정합니다.
 - v. **zephyr,input-positive**: 양극 입력을 NRF_SAADC_AIN4(P0.28)로 설정합니다.
 - vi. **zephyr,resolution**: ADC 해상도를 10비트로 설정합니다.

주변장치별 프로세서와 통신 표준 설명

다양한 센서와 주변장치를 프로세서와 어떻게 통신시키는지 설명합니다. 각각의 장치는 특정 통신 표준을 사용하여 데이터를 전송하며, 이를 통해 시스템이 실시간으로 공기질과 소음 수준을 모니터링할 수 있습니다. 여기서는 DHT22, CO2 센서, HT16K33 LED 매트릭스, 그리고 소음 센서의 통신 방식을 다룹니다.

DHT22 센서

통신 표준: GPIO

- 설명: DHT22는 온도와 습도를 측정하는 디지털 센서입니다.
- 통신 방식: GPIO(General-Purpose Input/Output) 핀을 통해 데이터를 전송합니다.
- 구성:
 - DHT22 센서는 단일 데이터 라인을 사용하여 마이크로컨트롤러와 통신합니다.
 - 데이터 라인은 일반적으로 풀업 저항과 함께 구성되어 신호의 안정성을 높입니다.
- 장점:

- 단순한 통신 방식으로 인해 구현이 쉽습니다.
- 낮은 전력 소비로 배터리 구동이 가능합니다.
- 응용: 스마트 홈 시스템에서 실내 온도와 습도 모니터링에 사용됩니다.

CO2 센서

통신 표준: UART

- 설명: CO2 센서는 실내의 이산화탄소 농도를 측정하는 장치입니다.
- 통신 방식: UART(Universal Asynchronous Receiver-Transmitter)를 통해 데이터를 전송합니다.
- 구성:
 - UART는 TX(송신)와 RX(수신) 라인을 사용하여 시리얼 데이터를 전송합니다.
 - 비동기 통신 방식으로 클럭 신호가 필요하지 않습니다.
- 장점:
 - 비교적 긴 거리에서도 신뢰성 있는 데이터 전송이 가능합니다.
 - 간단한 인터페이스로 다양한 프로세서와 쉽게 연결할 수 있습니다.
- 응용: 실내 공기질 모니터링 및 환기 시스템 제어에 사용됩니다.

HT16K33 LED 매트릭스

통신 표준: I2C

- 설명: HT16K33는 LED 매트릭스를 제어하기 위한 드라이버 IC입니다.
- 통신 방식: I2C(Inter-Integrated Circuit)를 통해 마이크로컨트롤러와 통신합니다.
- 구성:
 - I2C는 SDA(데이터)와 SCL(클럭) 두 개의 라인을 사용하여 다중 장치 간의 통신을 가능하게 합니다.
 - 각 장치는 고유한 주소를 가지며, 마스터-슬레이브 구조를 따릅니다.
- 장점:
 - 다중 장치와의 통신이 용이하며, 배선이 간단합니다.
 - 상대적으로 높은 데이터 전송 속도를 지원합니다.
- 응용: 스마트 홈 시스템에서 시각적 피드백을 제공하기 위해 사용됩니다.

소음 센서

통신 표준: ADC, GPIO

- 설명: 소음 센서는 주변 환경의 소음 수준을 측정하는 장치입니다.
- 통신 방식: 아날로그 신호를 ADC(Analog-to-Digital Converter)를 통해 디지털 신호로 변환하고,

GPIO를 사용하여 추가적인 신호 처리를 수행합니다.

- 구성:
 - 센서에서 출력되는 아날로그 신호는 ADC 핀을 통해 디지털 값으로 변환됩니다.
 - 특정 이벤트(예: 임계 소음 수준 초과)를 감지하기 위해 GPIO 핀을 사용합니다.
- 장점:
 - 아날로그 입력을 디지털 신호로 변환하여 정밀한 소음 측정이 가능합니다.
 - 추가적인 GPIO 사용으로 이벤트 기반 알람 설정이 가능합니다.
- 응용: 실내 소음 모니터링 및 소음 제어 시스템에 사용됩니다.

프로세스 및 인터럽트 관리

인터럽트 설정 (우선순위, ISR 등)

스마트 홈 시스템과 같은 임베디드 시스템에서는 다양한 센서와 주변장치에서 발생하는 이벤트를 효율적으로 처리하기 위해 프로세스 및 인터럽트 관리가 매우 중요합니다. 이 개요에서는 프로세스 관리, 인터럽트 설정, 우선순위 설정, 그리고 인터럽트 서비스 루틴(ISR)에 대해 설명합니다.

ISR은 인터럽트가 발생했을 때 실행되는 코드 블록입니다. ISR의 설계와 구현은 시스템의 성능과 안정성에 직접적인 영향을 미칩니다.

버튼 ISR

본 프로젝트는 사용자 인터페이스 중 입력을 가능하게 하는 입력 장치인 nRF52840 DK 보드 BTN 1, 2, 3, 4를 사용합니다. 각 BTN 1, 2, 3, 4 입력시 각각 고유 인터럽트를 발생시킵니다. 이들은 Zephyr RTOS에서 지원하는 K_WORK_QUEUE에 각각 고유 handler_callback을 추가합니다. RTOS 자체에서 준비되는데로 First In First Out(FIFO) 순서로 K_WORK_QUEUE에 담겨있는 callback 함수들을 호출하여 일을 실행합니다.

타이머 ISR

타이머 인터럽트는 위 버튼 ISR와 비슷하게 작동합니다. 다른점은 사용자에게서 입력 장치를 통해 입력을 받지 않고, Zephyr RTOS에서 지원하는 K_TIMER를 정의하여 5초 주기로 타이머 인터럽트를 발생시킵니다. 타이머가 만료될 때마다 실행되는 만료 함수를 사용하여 ISR를 관리합니다. 이 함수는 시스템 클럭 인터럽트 핸들러에 의해 실행됩니다.

본 프로젝트는 타이머 만료 함수를 이용하여 타이머 주기마다 BLE 통신을 통해 사용자에게 실시간으로 측정된 습도 값을 전달합니다.

구현 결과

사용자 입력 장치 사용방법 설명

- a) 1번 버튼 클릭시: 습도 출력
- b) 2번 버튼 클릭시: 온도 출력
- c) 3번 버튼 클릭시: co2 레벨 출력
- d) 4번 버튼 클릭시: 소음 레벨 출력
- e) 모바일 앱으로 절전모드 On/Off

동작시연 (데모 동영상과 추가 이미지를 사용하여 설명)

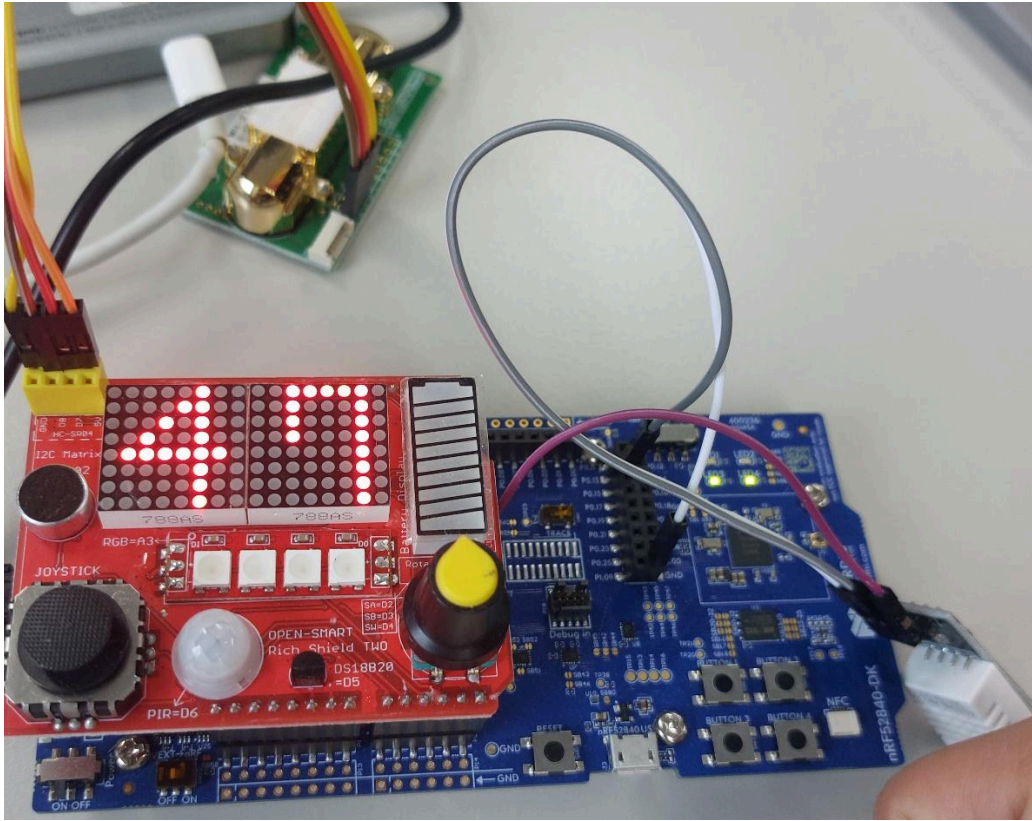
데모 동영상

<https://youtube.com/shorts/MJPMXxf1unI>

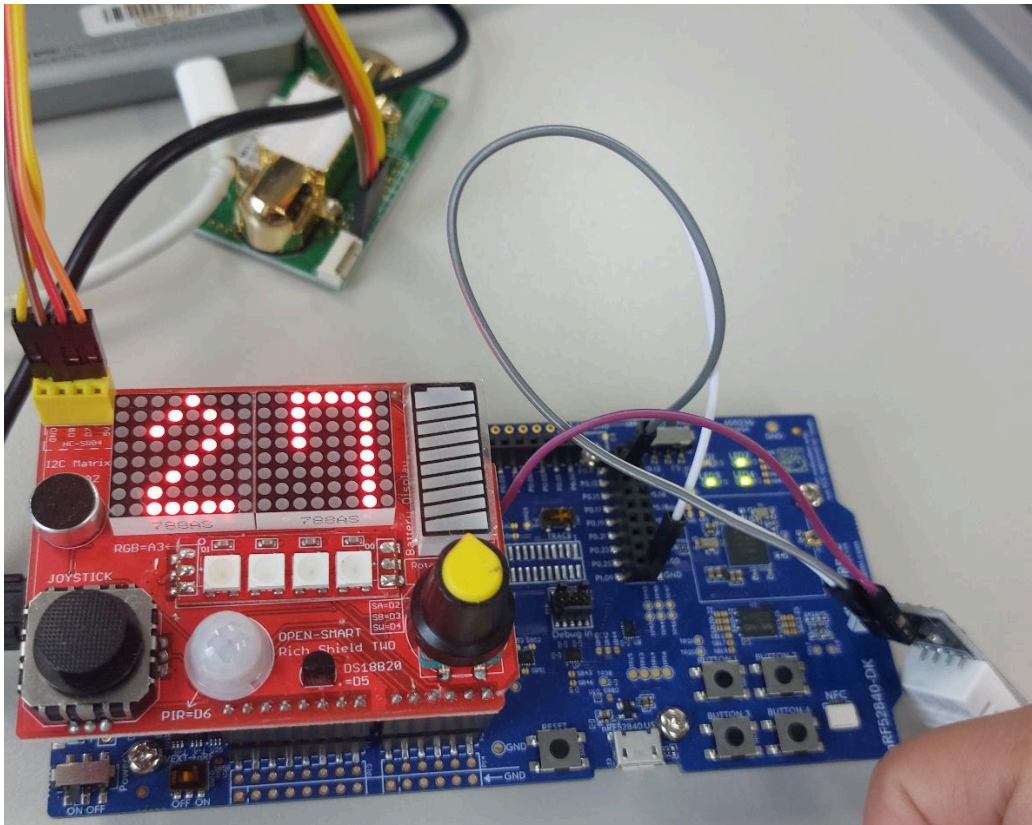
데모 동영상을 통해 동작시연을 합니다. 짧은 영상으로, 더 자세한 내용은 차근차근 설명하기 위해 영상 대신에 이미지와 설명을 대신하여 아래 작성하였습니다.

데모 이미지와 설명

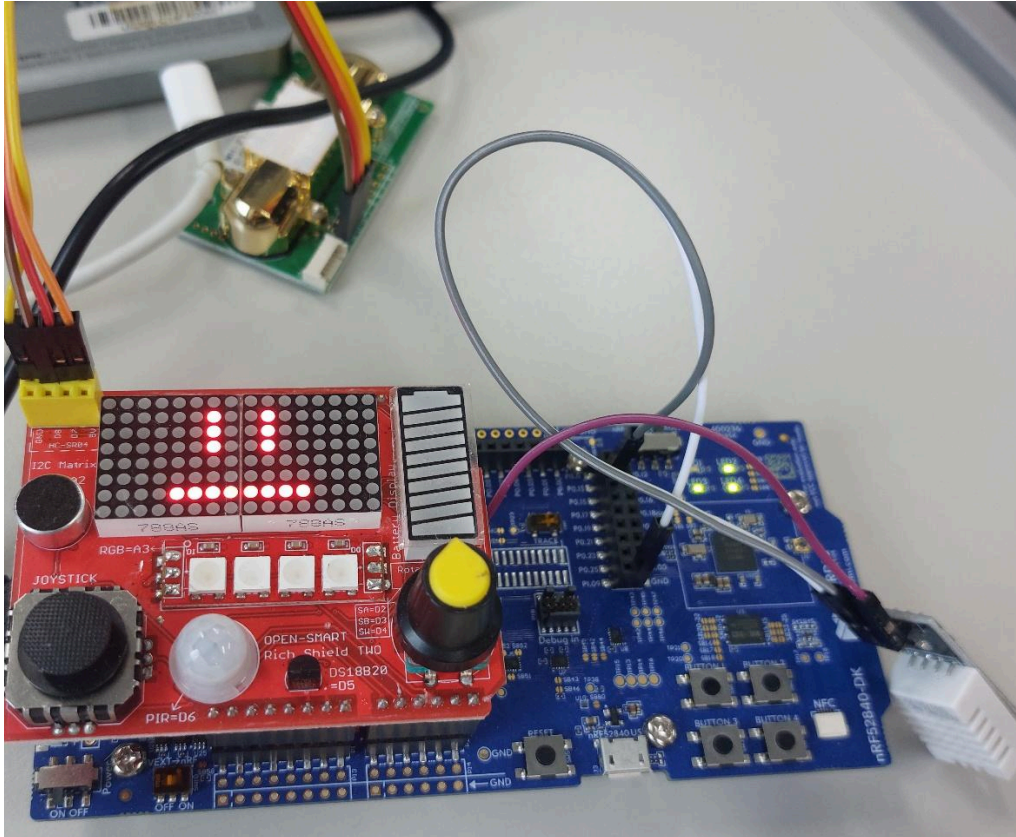
[1번 버튼 클릭시 습도 출력]



[2번 버튼 클릭시 온도 출력]

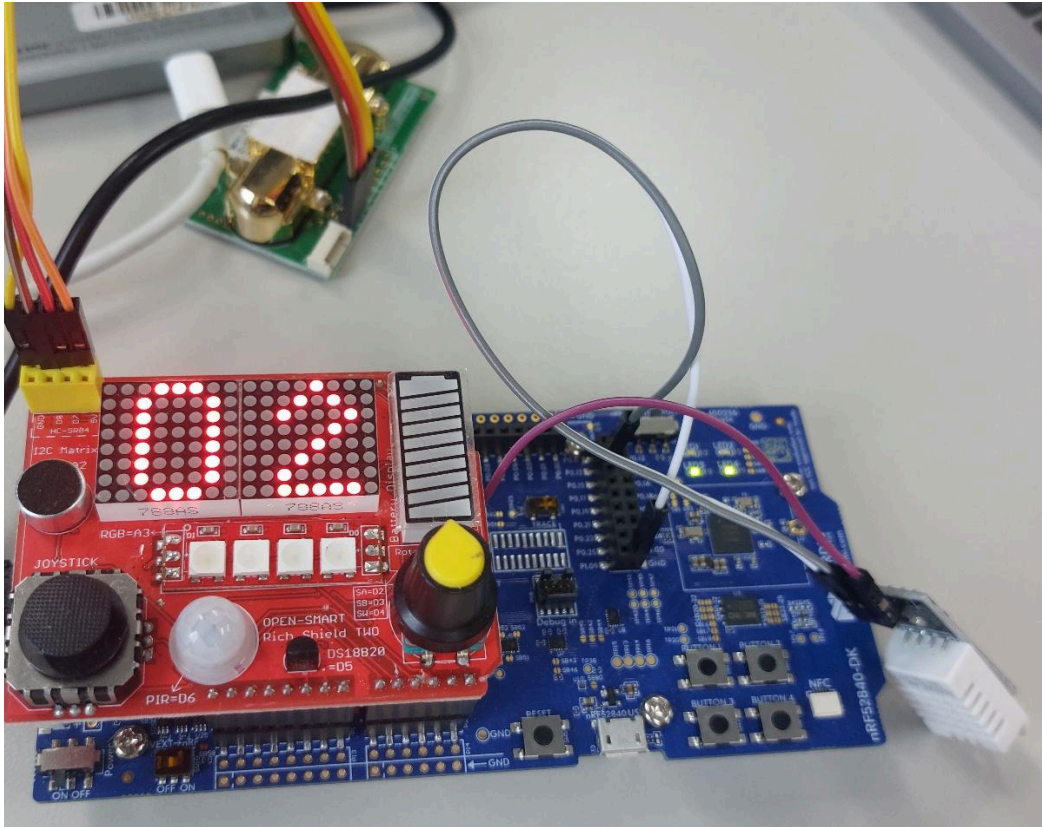


[3번 버튼 클릭시 co2 레벨 출력]



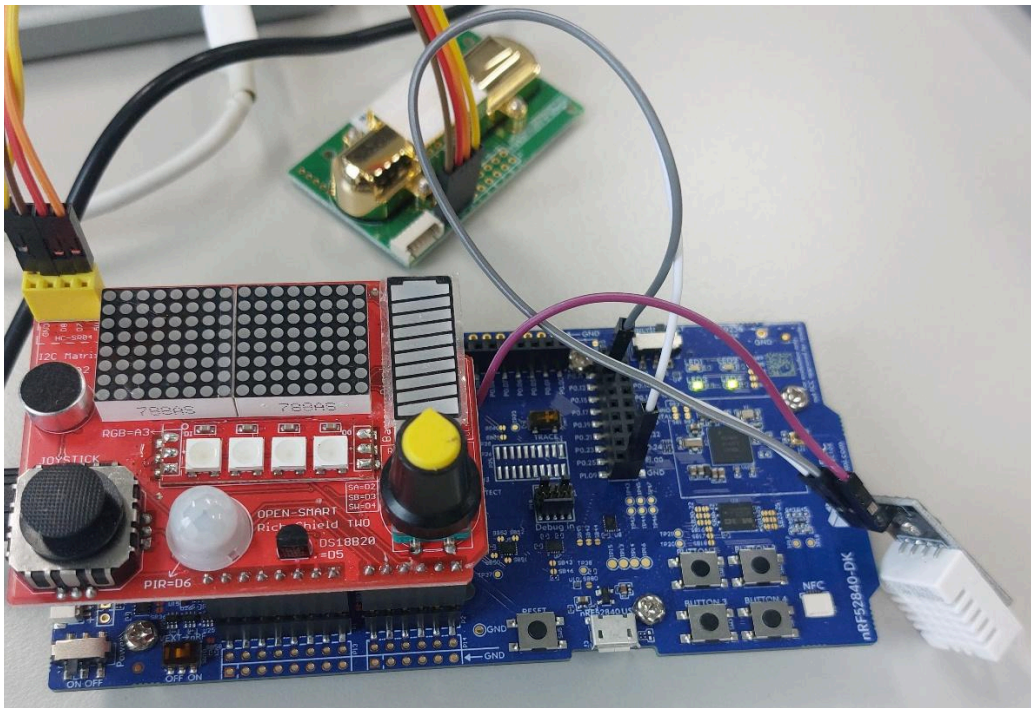
*기쁜 표정, 무표정, 기분 나쁜 표정 3가지 레벨로 표현됩니다.

[4번 버튼 클릭시 소음 레벨 출력]



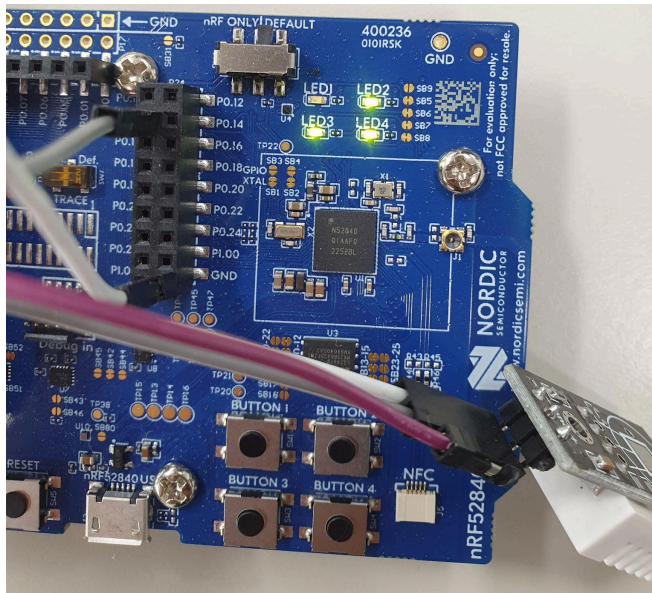
*0~8 단계까지 나누어 주변 소음 단계를 HT16K33 LED 매트릭스에 디스플레이 합니다.

[모바일 앱으로 절전 모드 On]

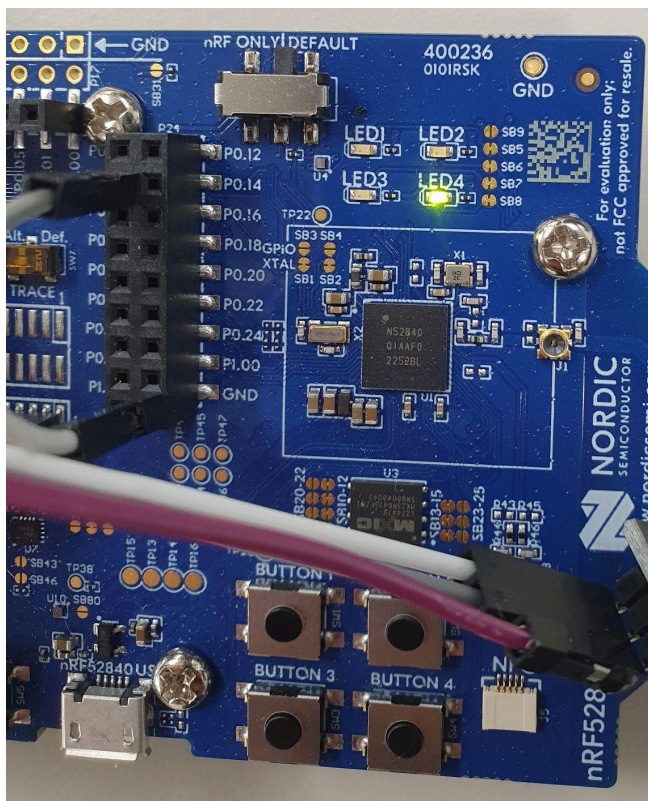


*절전 모드가 켜짐으로 HT16K33 LED 매트릭스의 디스플레이와 내부적으로 타이머도 종료시킵니다.

[BLE 연결됨 (led3 on)]



[BLE Off (led3 off)]



자체 평가

난이도 평가 (평가 기준에 근거하여 제시)

구현복잡도 및 완성도: 사용장치 수의 종류에 비례, 최종 데모의 성공 정도:

- 사용 장치의 종류(점수 * 개수 = 총점)
 - 새로운 IO 장치: 마이크, 온도도 센서 ($3 \times 2 = 6$)
 - 수업 시간에 다룬 장치: co2 센서, I2C LED 매트릭스 ($2 \times 2 = 4$)
 - 푸쉬 버튼 또는 단순 LED: LED 1,2,3,4, Button 1,2,3,4 ($1 \times 8 = 8$)
 - $6 + 4 + 8 = 18$
- 사용장치의 개수
 - 12개
- 최종 데모의 성공 정도
 - 모든 기능이 작동하면서 자연스러운 실시간성 보장: 10

기술 난이도: 입출력 프로토콜의 종류, 인터럽트 복잡도, 어셈블리어 사용 정도

- 주관적 문제의 복잡도
 - 아주
- 사용한 프로토콜 종류
 - BLE: 5
 - UART: 2
 - I2C: 2
 - GPIO: 1
- 인터럽트 복잡도: 인터럽트 개수(5) * 2 = 10
 - 버튼 4개 + 타이머 1개
- 인터럽트 우선순위에 의한 차이점 시연 또는 멀티 Thread 사용
 - 멀티 Thread 사용
 - HT16K33 LED Matrix 출력시 동시성을 지원해 출력
- 어셈블리어 사용 정도
 - 여러 함수 사용
 - map()
 - get_checksum()

어셈블리어 사용 함수를 만들기 위해 %[x] 형식의 어셈블리 코드에서 사용할 수 있는 자리 표시자(플레이스홀더)를 사용. 이는 GCC (GNU Compiler Collection)에서 인라인 어셈블리 코드를 작성할 때 사용되는 특정 구문으로 GCC 인라인 어셈블리에서 asm 블록 안에 변수의 위치를 지정하는 데

사용됨. 이 구문은 어셈블리 코드와 C 변수 간의 매핑을 정의함.

‘: [x] “r” (x)’의 경우 어셈블리 코드의 자리 표시자 x에 C 변수 x를 매핑하는 것을 의미함.

이때 “r”은 읽기 전용이고, “+r”은 입출력 모두 사용하는 경우에 사용

1. map 함수

```
long map(long x, long in_min, long in_max, long out_min, long out_max)
{
    long result;

    __asm volatile (
        "SUB %[x], %[x], %[in_min]\n\t"
        "SUB %[out_max], %[out_max], %[out_min]\n\t"
        "SUB %[in_max], %[in_max], %[in_min]\n\t"
        "MUL %[x], %[x], %[out_max]\n\t"
        "SDIV %[x], %[x], %[in_max]\n\t"
        "ADD %[x], %[x], %[out_min]"
        : [x] "+r" (x)
        : [in_min] "r" (in_min), [in_max] "r" (in_max), [out_min] "r" (out_min),
        [out_max] "r" (out_max)
    );

    return x;
}
```

2. getChecksum 함수

```
unsigned char getChecksum(char *packet) {
    unsigned char checksum = 0;

    long i;

    long temp;

    __asm volatile (
        "MOV %[i], #1\n\t"
        "MOV %[checksum], #0\n\t"
        "loop:\n\t"
```

```

    "LDRB %[temp], [%[packet], %[i]]\n\t"

    "ADD %[checksum], %[checksum], %[temp]\n\t"

    "ADD %[i], %[i], #1\n\t"

    "CMP %[i], #8\n\t"

    "BNE loop\n\t"

    "MVN %[checksum], %[checksum]\n\t"

    "ADD %[checksum], %[checksum], #1\n\t"

: [checksum] "+r" (checksum), [i] "+r" (i), [temp] "+r" (temp)

: [packet] "r" (packet)

);

return checksum;
}

```

계획대비 완성도

100%: 계획대로 진행되었습니다.

조원의 개인 기여 내용

김성빈:

- I2C LED 매트릭스 디자인 및 button 연결
- 온습도 센서 연결 및 버튼 1,2을 통해 데이터 display
- BTN ISR & TIMER ISR
- multi-thread 구현
- LED 1,2,3,4 연결
- BLE 통신 연결

최준혁:

- CO2 센서 연결 및 버튼 3을 통해 데이터 display
- 마이크 센서 연결 및 버튼 4을 통해 데이터 display

- 어셈블리어 함수 작성

부록

작성된 파일 목록 및 설명 (프로젝트 **README** 파일 내용)

- READEME.md
- main.c
 - main 프로그램이 실행되는 파일
 - uart를 이용한 co2 ppm 측정 및 co2 level로 변환
 - 사운드 센서를 통해 소리 측정 및 소음 level로 변환
- src/
 - buttons.c
 - 각 버튼마다 인터럽트 발생, I2C LED 매트릭스에 출력
 - button1: 습도 출력
 - button2: 온도 출력
 - button3: co2 level에 따른 표정 출력
 - button4: 소음 level 출력
 - dht22_sensor.c
 - 온습도 센서 관련 코드: 온습도 데이터 가져옴
 - ht16k33_led.c
 - I2C LED 매트릭스 관련 코드
 - my_service.c
 - BLE 관련 코드

코드 설명

온습도 센서 관련 코드

DHT22 장치 초기화

- DEVICE_DT_GET_ONE(aosong_dht) 매크로
 - 디바이스 트리에서 DHT22 센서를 검색하여 해당 장치의 포인터 반환

```
const struct device *const dht22 = DEVICE_DT_GET_ONE(aosong_dht);
```

init_dht

- DHT22 센서를 초기화
- device_is_ready 함수를 사용하여 센서가 준비되었는지 확인

```
int init_dht(void) {  
  
    if (!device_is_ready(dht22)) {  
  
        printk("Device %s is not ready\n", dht22->name);  
  
        return 1;  
  
    }  
  
    return 0;  
  
}
```

get_temperature (get_humidity)

- DHT22 센서에서 온도를 읽어옴
 - sensor_sample_fetch 함수를 사용하여 센서에서 샘플을 가져옴
 - 센서 데이터를 갱신
- sensor_channel_get 함수를 사용하여 갱신된 샘플에서 온도(습도) 값을 가져옴
 - 온도(습도) 값은 temperature(humidity) 구조체에 저장됩니다.
- 온도 값을 가져오는 데 실패하면 오류 메시지를 출력하고 -1을 반환
- 성공하면 온도 값을 출력하고 반환

```
int get_temperature(void) {  
  
    int rc = sensor_sample_fetch(dht22);  
  
  
    if (rc != 0) {  
  
        return -1;  
  
    }  
  
  
    struct sensor_value temperature;  
  
  
    rc = sensor_channel_get(dht22, SENSOR_CHAN_AMBIENT_TEMP, &temperature);  
  

```

```

    if (rc != 0) {

        printk("Sensor value get failed: %d\n", rc);

        return -1;

    } else {

        printk("Temperature: %d\n", temperature.val1);

    }

    return temperature.val1;

}

```

```

int get_humidity(void) {

    int rc = sensor_sample_fetch(dht22);

    if (rc != 0) {

        return -1;

    }

    struct sensor_value humidity;

    rc = sensor_channel_get(dht22, SENSOR_CHAN_HUMIDITY, &humidity);

    if (rc != 0) {

        printk("Sensor value get failed: %d\n", rc);

        return -1;

    } else {

        printk("Humidity: %d\n", humidity.val1);

    }

    return humidity.val1;

}

```

CO2 센서 관련 코드

ckeck_uart_fsm

- UART 통신에서 수신된 데이터를 기반으로 상태를 관리하는 상태 머신(FSM, Finite State Machine)
- 상태 머신을 사용하여 수신된 데이터를 단계별로 처리
- 필요한 경우 상태를 변경하여 다음 데이터를 준비
- UART_FSM_IDLE
 - 초기 상태
 - 수신된 데이터가 0xFF인 경우 UART_FSM_HEADER 상태로 전환
 - 그 외의 경우 상태를 유지
- UART_FSM_HEADER
 - 헤더를 수신하는 상태
 - 수신된 데이터가 0x86인 경우 UART_FSM_DATA 상태로 전환
 - 그 외의 경우 UART_FSM_IDLE 상태로 돌아감
- UART_FSM_DATA
 - 데이터를 수신하는 상태
 - 수신된 데이터의 위치(rx_buf_pos)가 메시지 크기(MSG_SIZE)에서 2번째 전 위치에 도달한 경우 UART_FSM_CHECKSUM 상태로 전환
- UART_FSM_CHECKSUM
 - 체크섬을 수신하는 상태
 - 수신된 데이터의 위치(rx_buf_pos)가 메시지 크기(MSG_SIZE)의 마지막 위치에 도달한 경우 UART_FSM_END 상태로 전환
- UART_FSM_END 상태
 - 메시지 수신이 완료된 상태
 - 수신 작업이 끝나면 다시 UART_FSM_IDLE 상태로 돌아감
- default
 - 정의되지 않은 상태일 경우 UART_FSM_IDLE 상태로 돌아감

```
uint8_t check_uart_fsm(uint8_t reset, uint8_t read_data) {  
  
    if(reset)  
  
        uart_fsm_state = UART_FSM_IDLE;  
  
    else  
  
        switch (uart_fsm_state) {  
  
            case UART_FSM_IDLE:
```

```

        if (read_data == 0xFF) {

            uart_fsm_state = UART_FSM_HEADER;

        }

        break;

    case UART_FSM_HEADER:

        if (read_data == 0x86) {

            uart_fsm_state = UART_FSM_DATA;

        } else {

            uart_fsm_state = UART_FSM_IDLE;

        }

        break;

    case UART_FSM_DATA:

        if (rx_buf_pos == MSG_SIZE - 2) {

            uart_fsm_state = UART_FSM_CHECKSUM;

        }

        break;

    case UART_FSM_CHECKSUM:

        if (rx_buf_pos == MSG_SIZE - 1) {

            uart_fsm_state = UART_FSM_END;

        }

        break;

    case UART_FSM_END:

        uart_fsm_state = UART_FSM_IDLE;

        break;

    default:

        uart_fsm_state = UART_FSM_IDLE;

        break;

}

return uart_fsm_state;

```

```
}
```

getChecksum

데이터 패킷의 체크섬을 계산

```
unsigned char getChecksum(char *packet) {

    unsigned char checksum = 0;

    long i;

    long temp;

    __asm volatile (

        "MOV %[i], #1\n\t"

        "MOV %[checksum], #0\n\t"

        "loop:\n\t"

        "LDRB %[temp], [%[packet], %[i]]\n\t"

        "ADD %[checksum], %[checksum], %[temp]\n\t"

        "ADD %[i], %[i], #1\n\t"

        "CMP %[i], #8\n\t"

        "BNE loop\n\t"

        "MVN %[checksum], %[checksum]\n\t"

        "ADD %[checksum], %[checksum], #1\n\t"

        : [checksum] "+r" (checksum), [i] "+r" (i), [temp] "+r" (temp)

        : [packet] "r" (packet)

    );

    return checksum;

}
```

serial_callback

- UART 인터럽트가 발생할 때 호출되는 콜백 함수
- UART 인터럽트 서비스 루틴(ISR)
- UART를 통해 수신된 데이터를 처리

- 수신된 데이터의 무결성을 체크섬을 통해 확인
- UART 수신 상태를 관리
- 수신된 CO2 데이터를 변환하여 저장

```
void serial_callback(const struct device *dev, void *user_data) {

    uint8_t c, high, low;

    char checksum_ok, value_calc_flag;

    int checksum;

    if (!uart_irq_update(uart_serial)) {

        printk("irq_update Error\n");

        return;

    }

    if (!uart_irq_rx_ready(uart_serial)) {

        printk("irq_ready: No data\n");

        return;

    }

    while (uart_fifo_read(uart_serial, &c, 1) == 1) {

        if (uart_fsm_state == UART_FSM_IDLE) {

            rx_buf_pos = 0;

        }

        check_uart_fsm(0,c);

        if (rx_buf_pos >= MSG_SIZE) {

            rx_buf_pos = 0;

        }

        rx_buf[rx_buf_pos++] = c;

    }

    if(uart_fsm_state == UART_FSM_END){
```



```

checksum = getChecksum(rx_buf);

checksum_ok = (checksum == rx_buf[8]);

if (checksum_ok) {

    value_calc_flag = (rx_buf_pos == MSG_SIZE);

    if (value_calc_flag) {

        high = rx_buf[2];

        low = rx_buf[3];

        co2_ppm = (high * CO2_MULTIPLIER) + low;

        for (int i = 0; i < MSG_SIZE; i+=1) {

            printk("%x ", rx_buf[i]);

        }

        printk("\n");

    }

}

else {

    printk("Checksum failed (%d == %d, index=%d)\n", checksum, rx_buf[8],
rx_buf_pos);

}

check_uart_fsm(1,0); // reset

}

}

```

serial_write

- UART를 통해 데이터를 전송
- 지정된 데이터 배열을 UART 송신 버퍼로 보냄
- UART 통신에서 데이터를 송신하기 위해 `uart_poll_out` 함수를 사용

```

void serial_write() {

    uint8_t tx_buf[MSG_SIZE] = {0xFF, 0x01, 0x86, 0x00, 0x00, 0x00, 0x00, 0x00,
0x79};

    for (int i = 0; i < MSG_SIZE; i+=1) {

```

```

        uart_poll_out(uart_serial, tx_buf[i]);

    }
}

```

사운드 센서 관련 코드

ADC 버퍼 및 시퀀스 구조체

```

struct adc_sequence sequence = {

    .buffer = &buf,

    .buffer_size = sizeof(buf),

};

```

ADC 채널 설정

```

for (size_t i = 0; i < ARRAY_SIZE(adc_channels); i++) {

    if (!adc_is_ready_dt(&adc_channels[i])) {

        printk("ADC controller device %s not ready\n", adc_channels[i].dev->name);

        return 0;

    }

    err = adc_channel_setup_dt(&adc_channels[i]);

    if (err < 0) {

        printk("Could not setup channel #%d (%d)\n", i, err);

        return 0;

    }

}
}

```

ADC 데이터 읽기

```

(void)adc_sequence_init_dt(&adc_channels[0], &sequence);

err = adc_read(adc_channels[0].dev, &sequence);

```

```

if (err < 0) {

    printk("Could not read (%d)\n", err);

    k_sleep(K_MSEC(100));

    continue;

}

```

사운드 값 처리

```

sound_value = (int32_t)buf;

    if(sound_value >= SENSOR_INVALID_VALUE){

        printk("sound_value: invalid data %" PRIu32 "\n", sound_value);

        k_sleep(K_MSEC(100));

        continue;

    }

    sound_level = map(sound_value, 0, MAX_SENSORVALUE, 0, MIN_SENSORVALUE);

    printk("sound_value: %" PRIu32 " sound_level : %d\n", sound_value,
sound_level);

```

BLE 관련 코드

데이터 수신 처리 함수

```

static ssize_t on_receive(struct bt_conn *conn, const struct bt_gatt_attr *attr,
const void *buf, uint16_t len, uint16_t offset, uint8_t flags)
{

    const uint8_t * buffer = buf;

    printk("Received data, handle %d, conn %p, data: 0x", attr->handle, conn);

    for(uint8_t i = 0; i < len; i++){

        printk("%02X", buffer[i]);

    }

    printk("\n");

```

```

    if (buffer[0] == 0) { // enter low-energy mode

        off_led_ht16k33();

        off_sensor_timer();

    } else {

        restart_sensor_timer();

    }

    return len;
}

```

데이터 송신 완료 처리 함수

```

static void on_sent(struct bt_conn *conn, void *user_data)
{
    ARG_UNUSED(user_data);

    const bt_addr_le_t * addr = bt_conn_get_dst(conn);

    printk("Data sent to Address 0x %02X %02X %02X %02X %02X %02X \n",
addr->a.val[0],  addr->a.val[1],  addr->a.val[2],  addr->a.val[3],  addr->a.val[4],
addr->a.val[5]);
}

```

CCCD 변경 처리 함수

```

void on_cccd_changed(const struct bt_gatt_attr *attr, uint16_t value)
{
    ARG_UNUSED(attr);

    switch(value)
    {
        {

            case BT_GATT_CCC_NOTIFY:

                break;

            case BT_GATT_CCC_INDICATE:

```

```

        break;

    case 0:

        break;

    default:

        printk("Error, CCCD has been set to an invalid value");

    }
}

```

GATT 서비스 및 특성 등록

- BT_GATT_SERVICE_DEFINE: GATT 서비스 정의 및 등록
- BT_GATT_PRIMARY_SERVICE: 기본 서비스로 등록.
- BT_GATT_CHARACTERISTIC: 각 특성을 정의
- BT_UUID_MY_SERVICE_RX:
 - 쓰기 권한
 - 데이터를 수신하면 on_receive 함수 호출
- BT_UUID_MY_SERVICE_TX:
 - 알림 권한
 - 데이터 송신
- BT_GATT_CCC
 - CCCD 정의
 - 값이 변경되면 on_cccd_changed 함수가 호출

```

BT_GATT_SERVICE_DEFINE(my_service,

BT_GATT_PRIMARY_SERVICE(BT_UUID_MY_SERVICE),

BT_GATT_CHARACTERISTIC(BT_UUID_MY_SERVICE_RX,

                        BT_GATT_CHRC_WRITE | BT_GATT_CHRC_WRITE_WITHOUT_RESP,

                        BT_GATT_PERM_READ | BT_GATT_PERM_WRITE,

                        NULL, on_receive, NULL),

BT_GATT_CHARACTERISTIC(BT_UUID_MY_SERVICE_TX,

                        BT_GATT_CHRC_NOTIFY,

                        BT_GATT_PERM_READ,

                        NULL, NULL, NULL),

```

```
BT_GATT_CCC(on_cccd_changed,

            BT_GATT_PERM_READ | BT_GATT_PERM_WRITE),

);
```

Multi-thread 관련 코드 (I2C LED 출력)

스레드 스택 선언

```
K_THREAD_STACK_DEFINE(tens_stack, STACK_SIZE);

K_THREAD_STACK_DEFINE(units_stack, STACK_SIZE);
```

스레드 데이터 구조 선언

```
struct k_thread tens_thread;

struct k_thread units_thread;
```

디스플레이 파라미터 구조체

```
struct display_params {

    int value;

    int start_idx;

};
```

디스플레이 스레드 함수

```
void display_digit_thread(void *param, void *unused1, void *unused2) {

    struct display_params *params = (struct display_params *)param;

    int value = params->value;

    int start_idx = params->start_idx;

    int num_arr_idx = 0;

    for (int i = 0; i < MAX_LED_MATRIX_NUM; i += 16) {
```

```

        for (int j = (i + start_idx); j < ((i + start_idx) + 8); j++) {

            if (led_matrix[value][num_arr_idx] == 1) {

                if (led_on(ht16k33_led, j) != 0) {

                    return -1;

                }

            } else {

                if (led_off(ht16k33_led, j) != 0) {

                    return -1;

                }

            }

            num_arr_idx++;

        }

    }

}

```

값을 디스플레이하기 위한 스레드 생성

```

int display_value_ht16k33(int value) {

    if (value < 0 || value > 999) {

        printk("Error: Invalid value\n");

        return -1;

    }

    if (value > 99) {

        value = value / 10;

    }

    int tens = value / 10;

    int units = value % 10;

    struct display_params tens_params = { .value = tens, .start_idx = 0 };

```

```

struct display_params units_params = { .value = units, .start_idx = 8 };

k_thread_create(&tens_thread, tens_stack, STACK_SIZE,
               display_digit_thread, &tens_params, NULL, NULL,
               PRIORITY, 0, K_NO_WAIT);

k_thread_create(&units_thread, units_stack, STACK_SIZE,
               display_digit_thread, &units_params, NULL, NULL,
               PRIORITY, 0, K_NO_WAIT);

k_thread_join(&tens_thread, K_FOREVER);

k_thread_join(&units_thread, K_FOREVER);

return 0;
}

```

개발과정에서 만났던 에러가 있는 경우, 에러메시지와 그 대처 방안 설명

개발과정 중 가장 해결하기 어려웠고 중요한 전환점이 되었던 에러가 있었다. **BLE** 통신과 관련된 에러인데 본 프로젝트가 작성된 흐름대로, 버튼 인터럽트가 들어오면 해당 고유 작업(센서 데이터 출력)을 실행하는 이 작업을 **BLE** 통신을 유지한 상태로 해야한다. 기존에는 센서 데이터 출력 작업을 **K_WORK_QUEUE** 없이 인터럽트가 발생한 직후 해당 콜백 함수에서 작업을 실행했다. 이는 해당 센서와 통신 뿐만 아니라 다양한 작업이 소요되었다. 이 과정 속 **BLE** 통신이 끊어지면서 보드 내부적으로 **stack corrupted** 에러가 발생했다.

이를 해결하기 위해 콜백 함수에서 곧바로 작업을 진행하는 것이 아니라 **BLE** 연결을 유지하면서 작업을 실행하기 위해 **K_WORK_QUEUE** 기능을 사용하여 OS 자체적으로 실시간으로 준비가 되었을때 작업을 실행 할 수 있도록 설계하여 문제를 해결하였다.

개인 소감

김성빈: Zephyr RTOS를 처음 경험할때 부족한 배경지식으로 어떤 기능들이 있는지, 어떻게

사용해야하는지 찾는데 시간과 노력이 많이 들었다. 하지만, 꾸준히 **documentation**을 읽고, **sample**위주로 코딩으로 하고 직접 수정하면서 변경하다보니 어느새 내것이 되었다. 이를 통해 단순히 **API documentation**과 **datasheet**를 가지고도 프로그래밍을 할수있겠다는 자신감이 생겼다. 추가로, **IoT**관련 주제를 다루면서 저전력 등 고려해야하는 상황이 많은 점에서 어려웠지만 재밌었고 만족스러운 결과를 냈다.

최준혁: 지금까지 많은 참조 자료들을 가지고 로직을 짜는 코딩만 해와서, 참조 자료가 없거나 거의 없는 환경에서 센서를 연결하고 활용하는 과정이 낯설고 힘들었습니다.

개발과정에서 참조 했던 리소스

(1) 블로그 등 **URL**, 하드웨어 업체 제공 공식 문헌,

<https://github.com/UmileVX/IoT-Development-with-Nordic-Zephyr>

- 다양한 **sample**과 참고자료

<https://github.com/zephyrproject-rtos/zephyr/tree/main/samples/sensor/dht>

- DHT22 센서 **sample**

<https://www.sparkfun.com/datasheets/Sensors/Temperature/DHT22.pdf>

- DHT22 센서 **datasheet**

<https://components101.com/sensors/dht22-pinout-specs-datasheet>

- DHT22 센서 **datasheet**

(2) Youtube, StackOverflow, 개발 **Community**, Chat-GPT 등의 활용

내역

ChatGPT

- BLE 통신 관련 대화

give code for both controller and peripheral where both are nrf52840dk boards connected via bluetooth. based on zephyr RTOS

when the controller has a button pressed, it should notify the peripheral board and the peripheral board should turn on its led. also, there is a k_timer in the peripheral board that periodically sends integer data back to the controller so that the controller prints it.



Sure, here's a basic example demonstrating how you can achieve this using Zephyr RTOS on two nRF52840 development boards connected via Bluetooth Low Energy (BLE). We'll create a simple controller and peripheral application.

Controller Application (controller/src/main.c):

```
c Copy code

#include <zephyr.h>
#include <device.h>
#include <bluetooth/bluetooth.h>
#include <bluetooth/gatt.h>

#define LED_CONTROLLER_PIN 13
#define BUTTON_CONTROLLER_PIN 17

static struct bt_conn *peripheral_conn;

static void button_pressed(struct device *dev, struct gpio_callback *cb,
                          uint32_t pins)
{
    if (peripheral_conn) {
        bt_gatt_notify(peripheral_conn, &button_service_attrs[1], &button_state, sizeof(button_state));
    }
}

static struct gpio_callback button_cb;

static void connected(struct bt_conn *conn, uint8_t err)
{
    if (err) {
        printk("Connection failed (err %u)\n", err);
    } else {
        peripheral_conn = bt_conn_ref(conn);
        printk("Connected\n");
    }
}


static void disconnected(struct bt_conn *conn, uint8_t reason)
{
    printk("Disconnected (reason %u)\n", reason);
    if (peripheral_conn) {
        bt_conn_unref(peripheral_conn);
        peripheral_conn = NULL;
    }
}

static struct bt_conn_cb conn_callbacks = {
    .connected = connected,
    .disconnected = disconnected,
};
```

- BLE Subscription Error Fix 대화

this code has errors even after connecting. errors occur when calling the `bt_gatt_notify()` function.

W: Device is not subscribed to characteristic
gatt notify error: -22

 The error **W: Device is not subscribed to characteristic** indicates that there are no active subscriptions to the characteristic you are trying to notify. This typically means that no connected device has registered for notifications on that characteristic.

In BLE, notifications can only be sent to a client if that client has explicitly enabled notifications for that characteristic. The error code **-22** corresponds to **EINVAL**, which usually signifies an invalid argument or that the operation is not allowed.

To resolve this, ensure the following:

1. **Enable Notifications on the Client Side:** Make sure the client device (e.g., a mobile app or another BLE device) subscribes to notifications for the characteristic in question.
2. **Correct Characteristic Properties:** Ensure the characteristic has the correct properties set for notifications.

Here's how you can make sure the characteristic is set up correctly and that notifications are enabled on the client side:

Define the Characteristic with Notification Capability

Make sure the GATT characteristic is defined with the **BT_GATT_CHRC_NOTIFY** property.

c
Copy code

```
BT_GATT_CHARACTERISTIC(&gatt_write_char_uuid.uuid,
    BT_GATT_CHRC_WRITE | BT_GATT_CHRC_NOTIFY,
    BT_GATT_PERM_WRITE,
    NULL, write_gatt_value, &gatt_write_value)
```

YouTube

- nRF5 SDK - Tutorial for Beginners Pt 41 - Nordic Soft Devices Basics
 - <https://www.youtube.com/watch?v=fzOg175s0Sg&list=PLiKJlJyEUIZj4z8RqJKTaoMpS qjLdBGO4&index=44>
- nRF5 SDK - Tutorial for Beginners Pt 33 - Low Power PWM using Low Frequency Clock
 - <https://www.youtube.com/watch?v=PrIboQluLRI&list=PLiKJlJyEUIZj4z8RqJKTaoMpSq jLdBGO4&index=35>
- How to communicate two board Nordic Bluetooth nrf52832 and nrf52840 .
 - <https://www.youtube.com/watch?v=PNwbGEYoQLk>
- Developing Bluetooth Low Energy products using nRF Connect SDK
 - https://www.youtube.com/watch?v=hY_tDext6zA
- nRF5 SDK - Tutorial for Beginners Pt 45 B - BLE Device Address Types
 - <https://www.youtube.com/watch?v=bR74NiqqbWQ&list=PLiKJlJyEUIZj4z8RqJKTaoMp SqjLdBGO4&index=61>
- nRF5 SDK - Tutorial for Beginners Pt 43 H - GATT Programming basics
 - <https://www.youtube.com/watch?v=2gxJ4aYPs60&list=PLiKJlJyEUIZj4z8RqJKTaoMp>

SqjLdBGO4&index=54

- nRF5 SDK - Tutorial for Beginners Pt 40 - Bluetooth Low Energy an Introduction to basics in BLE
 - <https://www.youtube.com/watch?v=AQu80Fdn3T8&list=PLiKJlJyEUIZj4z8RqJKTaoMpSqjLdBGO4&index=43>