SI-TA-PO

시계-타이머-뽀모도로 3 Mode 시간 관리 시스템

Linux kernel Device Driver Project

[Intel] 엣지 AI SW 아카데미 8기
류균봉, 이두현
발표일: 2025.12.29

Contents

Introduction

SI-TA-PO

식탁보

리눅스 커널 드라이버 제어를 통해 시계, 타이머, 뽀모도로 기능을 통합 구현한 고신뢰성 임베디드 제어 시스템

3 Mode

OLED 시각화

타이머 완료
LED 알림

Clock

Timer

Pomodoro

Goal

3 mode

TACT 스위치로 시계, 타이머, 뽀모도로 총 3가지 모드 전환

Clock

DS1302 시계 기능을 OLED에 Display
로터리 엔코더 및 버튼 제어로 시간 보정

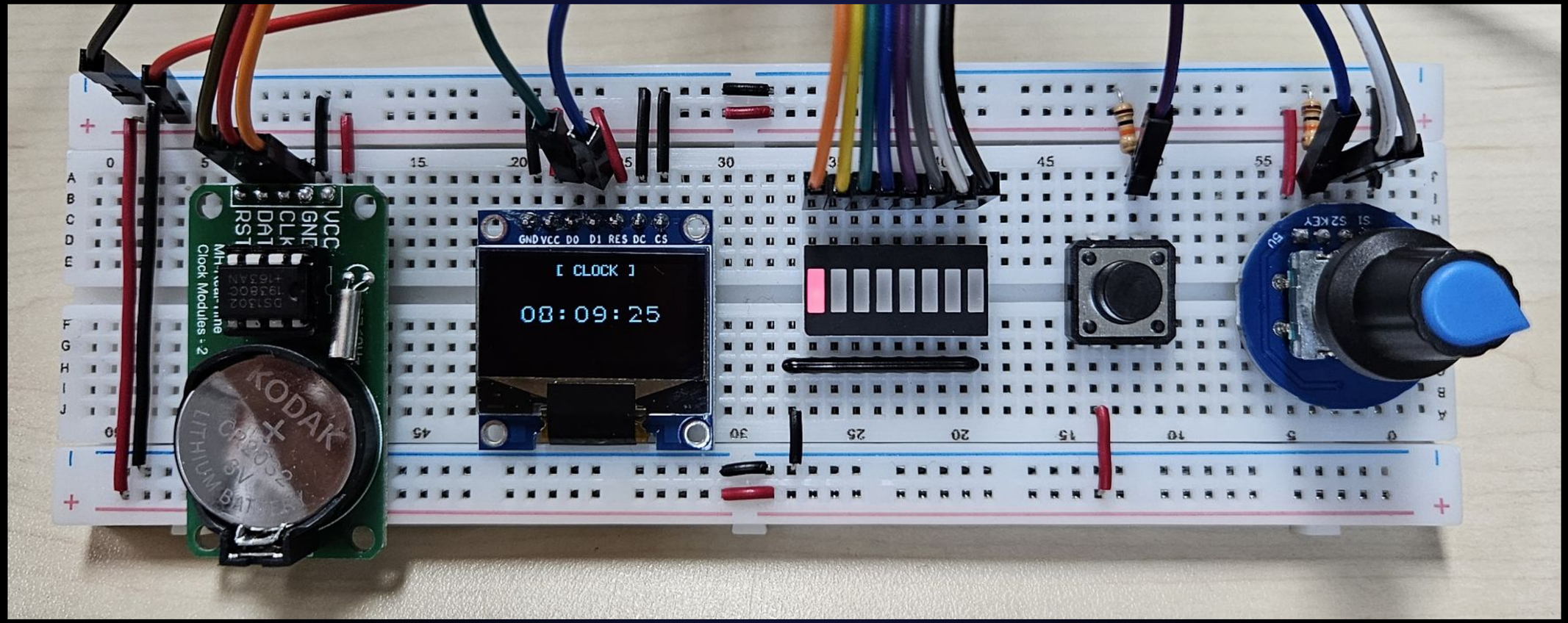
Timer

로터리 엔코더와 버튼 제어로 타이머 시간 설정
타이머 설정 및 동작 화면 OLED에 실시간으로 Display

Pomodoro

로터리 엔코더와 버튼 제어로 집중 시간, 쉬는 시간, 반복 횟수 설정
뽀모도로 설정 및 동작 화면 OLED에 실시간으로 Display

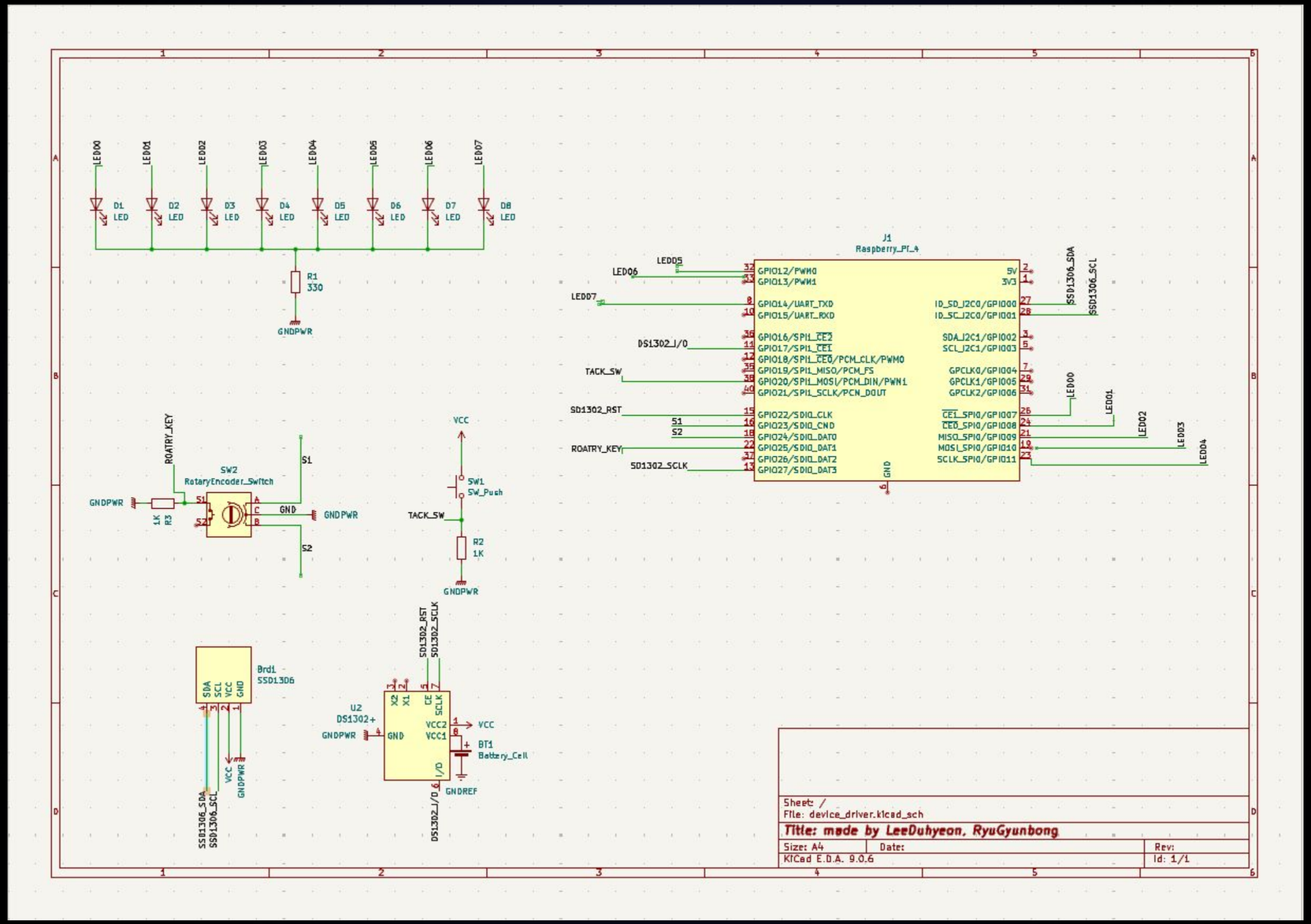
HW


VCC
GND
CLK
DAT
RST
KODAK
CR2032
3V
LITHIUM BAT
GND VCC D0 D1 RES DC CS
[CLOCK]
08:09:25
5V
S1 S2 KEY

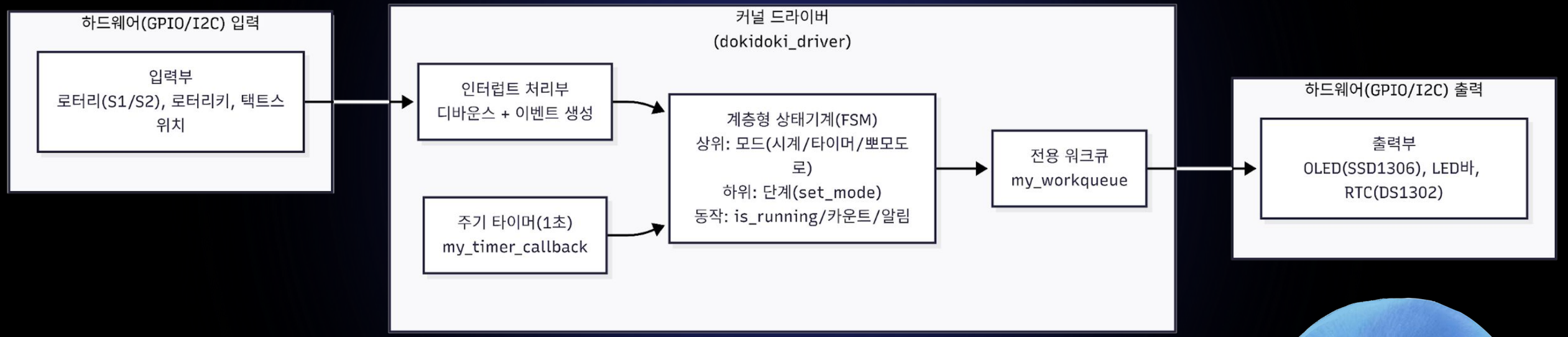
HW Bill of Material

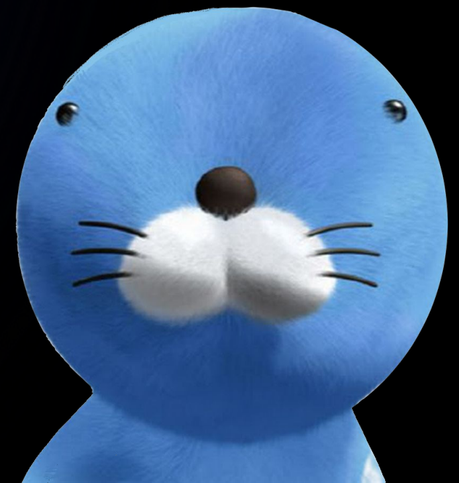
No	품목	규격/모델	수량	용도	비고
1	메인 보드	Raspberry Pi 4B	1	커널 드라이버 구동	OS: Debian 13.2 (trixie) / Kernel: 6.1.93-v8+ (aarch64)
2	로터리 엔코더	ec11 rotary encoder	1	값 조절/입력	S1/S2/KEY 사용
3	택트 스위치	tact push button 12X12	1	모드 변경	디바운스 적용
4	OLED 모듈	SSD1306	1	화면 출력	SDA/SCL
5	RTC 모듈	DS1302	1	시간 유지	CR2032 포함
6	코인 배터리	CR2032	1	RTC 백업 전원	3V
7	LED(또는 LED바)	8개(또는 모듈)	1	상태/알림 표시	GPIO7~14
8	저항	1KΩ	2	pull up 저항용	Rotary Key, Tact SW
9	점퍼선	M-F/M-M	다수	배선	
10	브레드보드	165X55	1	회로 구성	

HW Schematic Diagram

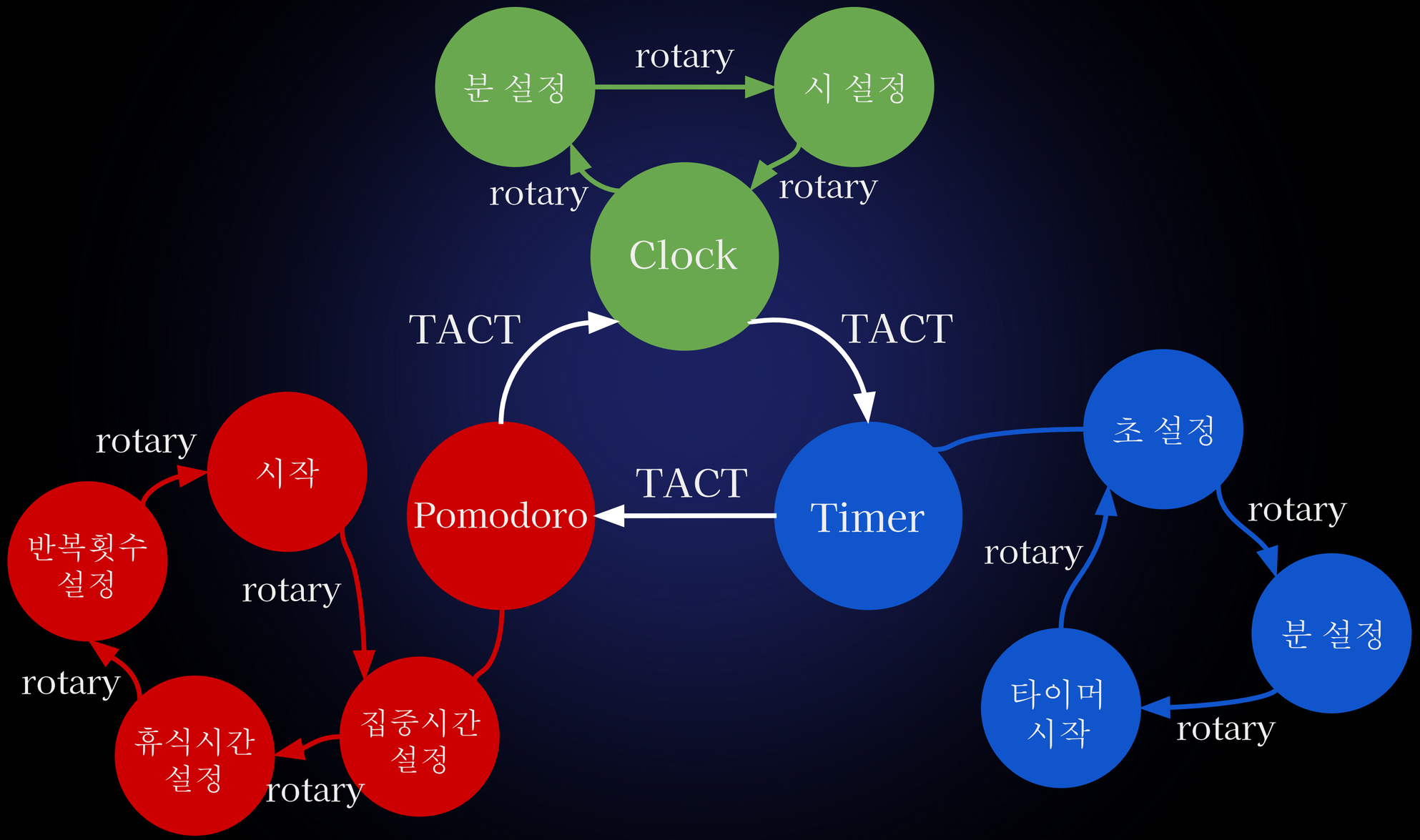


SW Block diagram





SW FSM (Hierarchical State Machine, HSM)



SW Code review

Rotary Encoder Rotation: Setting Value

```c
switch(mode) {
    case 0:
        clock_handler();
        break;
    case 1:
        timer_handler();
        break;
    case 2:
        pomodoro_handler();
        break;
}
```

```c
// ==============================clock_handler satrt
static void clock_handler(void)
{
    switch (set_mode) {

        case 0:                 // 시계 화면 상태 (RTC 업데이트 등)
            printk(KERN_INFO "Clock: Settings Saved\n");
            break;
        case 1:                         // 분 설정
        // set_num 은 1번째 파라미터를 2번째 파라미터 이하의 값으로 바꿔주는 함수
            set_num(&clock_m, 59);
            _ds1302_date.minutes = clock_m;
            _ds1302_date.seconds = 0;
            break;
        case 2:                         // 시 설수
            set_num(&clock_h, 23);
            _ds1302_date.hours = clock_h;
            _ds1302_date.seconds = 0;
            break;
    }
}
// ==============================clock_handler end
```

SW Code review

Rotary Encoder Rotation: Setting Value

```c
switch(mode) {
    case 0:
        clock_handler();
        break;
    case 1:
        timer_handler();
        break;
    case 2:
        pomodoro_handler();
        break;
}
```

```c
// ============================ timer_handler satrt
static void timer_handler(void)
{
    switch (set_mode) {
        case 0:                     // 초 설정
            is_running = 0;
            del_timer(&my_kernel_timer); // 설정 중엔 타이머 중지
            set_num(&timer_s, 59);
            break;
        case 1:                         // 분 설정
            set_num(&timer_m, 99);
            break;
        case 2:                         // 타이머 동작 시작
            is_running = 1;
            mod_timer(&my_kernel_timer, jiffies + HZ); // 1초 뒤 작동
            printk(KERN_INFO "Timer: Started\n");
            break;
    }
}
// ============================ timer_handler end
```

SW Code review

Rotary Encoder Rotation: Setting Value

```c
switch(mode) {
    case 0:
        clock_handler();
        break;
    case 1:
        timer_handler();
        break;
    case 2:
        pomodoro_handler();
        break;
}
```

```c
// ==================================pomodoro_handler start
static void pomodoro_handler(void)
{
    switch (set_mode) {
        case 0:                         // 집중 시간 설정 (최대 99분)
            is_running = 0;
            del_timer(&my_kernel_timer);    // 설정 중엔 타이머 중지
            set_num(&pomo_work, 99);
            break;
        case 1:                         // 쉬는 시간 설정 (최대 99분)
            set_num(&pomo_rest, 99);
            break;
        case 2:                         // 반복 횟수 설정 (최대 10번)
            set_num(&pomo_repeat, 10);
            break;
        case 3:                         // 뽀모도로 시작
            is_running = 1;
            pomo_state = 0;             // 집중(Work)부터 시작
            pomo_cur_repeat = 0;        // 반복 횟수 초기화
            timer_m = pomo_work;        // 사용자가 설정한 집중 시간으로 세팅
            timer_s = 0;                // 초는 0으로 시작
            mod_timer(&my_kernel_timer, jiffies + HZ); // 1초 뒤 작동
            printk(KERN_INFO "Pomodoro: Started\n");
            break;
    }
}
// ==================================pomodoro_handler end
```

SW Code review

Work Queue

전역 변수로 추가

```
// LED용 워크큐를 위한 구조체 공간
struct work_struct led_alert_work;
void led_alert_work_func(struct work_struct *work);
```

```
// led work queue init   dev init 부분에 추가
INIT_WORK(&led_alert_work, led_alert_work_func);
```

workqueue에 던질 작업(work_struct)을 초기화해서,
실행될 콜백 함수를 연결해주는 매크로

```
// ================================== led_alert_work_func start
void led_alert_work_func(struct work_struct *work)
{
    int i, j;
    for(i = 0; i < 3; i++){
        for(j = 3; j <= 7; j++){
            gpio_set_value(LED00+j, 1);
        }
        msleep(200);
        for(j = 3; j <= 7; j++){
            gpio_set_value(LED00+j, 0);
        }
        msleep(200);
    }
}
// ================================== led_alert_work_func end
```

SW Code review

Tact Switch IRQ: Debounce & Mode Update

```c
static irqreturn_t tact_irq_handler(int irq, void *dev_id) {
    unsigned long current_time = jiffies;       // jiffies는 jiffies.h에 선언되어 있음
    int cur_state;

    // 1. 초기 상태 변경: Pull-down이므로 안 눌렸을 때 0
    static int prev_state = 0;

    // 디바운싱
    if ((current_time - last_tact_time) < msecs_to_jiffies(DEBOUNCE_MS)) {
        return IRQ_HANDLED;
    }
    last_tact_time = current_time;

    // 현재 핀 상태 읽기 (Pull-down: 누르면 1, 떼면 0)
    cur_state = gpio_get_value(TACT_SW);

    // 버튼을 누르는 순간 (Rising Edge: 0 -> 1)
    if (cur_state == 1) { // Rising Edge
        // 모드를 0, 1, 2로 순환 (0:Clock, 1:Timer, 2:Pomo)
        mode = (mode + 1) % 3;
        set_mode = 0;
        rotary_value = 0;
        is_running = 0; // 모드 변경 시 타이머 정지
        del_timer(&my_kernel_timer);
```

SW Code review

Rotary Encoder Rotation: Debounce

```
// ================================================================ rotary_s1_irq_handler start
static irqreturn_t rotary_s1_irq_handler(int irq, void * dev_id)
{
    // 로터리 회전 시 채터링 발생 문제
    /* 시스템 워크큐가 아닌 전용 워크큐로 관리하면서 빠르게 회전할 때 발생했던 채터링은 해결됐으나
       천천히 돌릴 때의 채터링은 남음. 엔코더 신호의 전압이 변하는 경계선에서
       신호가 잘게 떨리면서 채터링이 발생하는 것을 제거하기 위해 양방향 S1, S2 값 검사 로직 추가
    */
    int val_s1, val_s2;

    unsigned long current_time = jiffies; // read current clock(Hz)
    unsigned long debounce_jiffies = msecs_to_jiffies(5);       // 1. 키 인터럽트와 달리 길게 주면 오히려 필요한 신호도 무시되므로 5ms 짧은 시간 설정

    if(time_before(current_time, last_rotary_s1_time + debounce_jiffies))
    {
        return IRQ_HANDLED;     //5ms 이하 디바운싱 처리
    }

    // 2. 전기적 안정화 대기
    udelay(2500);

    // 3. 안정된 후 S1 읽기
    // if s1 falling edge then INT occurs
    // S1(0) & S2(1) --> reverse clock(CCW)
    // S1(1) & S2(0) --> clock(CW)
    val_s1 = gpio_get_value(ROTARY_S1);

    // 4. 노이즈 무시를 위해 S1이 하강엣지 상태 즉, 0인지 확인
    if(val_s1 != 0) {
        return IRQ_HANDLED;
    }
```

Key Technologies

Finite State Machine (FSM)

(1) 개요

시스템이 가질 수 있는 유한한 상태(State)를 정의하고, 특정 조건(Event/Input)에 따라
상태 간 전이(Transition)를 제어하는 설계 모델

(2) 용도

복잡한 논리를 단순화해 관리에 용이

- Multi-Mode Control: '시계-타이머-뽀모도로' 간의 모드 전환 관리
- Operation Logic: 각 모드 내부의 세부 동작(설정 중, 동작 중, 종료 알람 등)을 논리적 오류 없이 제어

Key Technologies

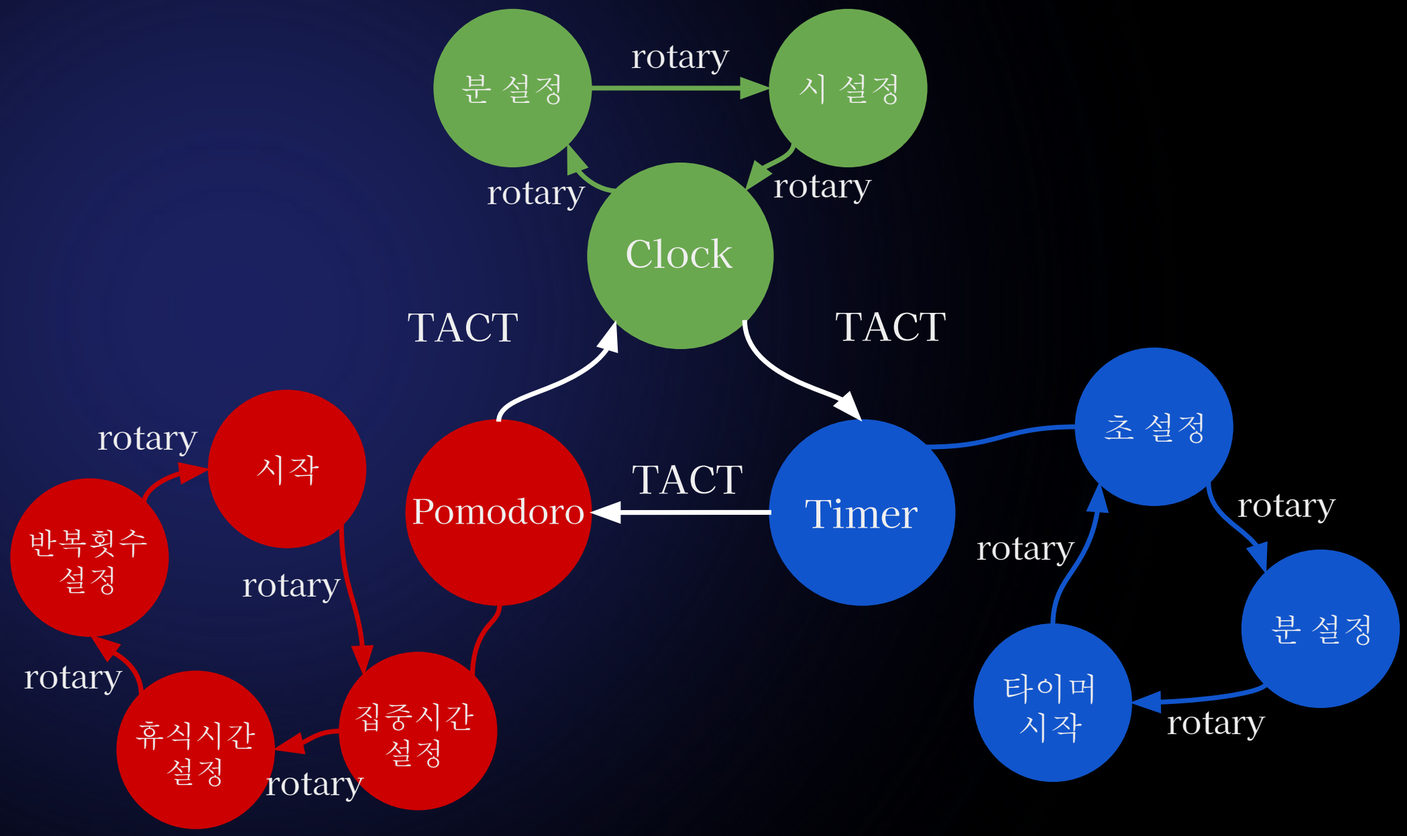
Finite State Machine (FSM)

(3) 동작 방식

입력 이벤트(Event) → 상태 전이

① TACT 스위치: 모드 변환

② Rotary Key: 모드 내 세부 동작 제어



Key Technologies

Finite State Machine (FSM)

(4) 특징 및 장단점

① 주요 특징

- 예측 가능성: 모든 동작이 사전에 정의된 상태 안에서만 발생하여 시스템의 행위가 명확

② 장점

- 가독성 향상: 시스템의 전체 흐름을 시각적으로 파악하기 용이(State Diagram)
- 신뢰성 및 안정성: 정의되지 않은 입력이나 잘못된 상태 전이를 원천 차단하여 논리적 오류 방지
- 유지보수 및 확장성: 기존 구조를 변경하지 않고 새로운 상태나 전이 규칙을 독립적으로 추가 가능

③ 단점

- 상태가 너무 많아지면 상태 전이도가 복잡해짐 → 계층형 상태 머신(HSM) 설계

Key Technologies

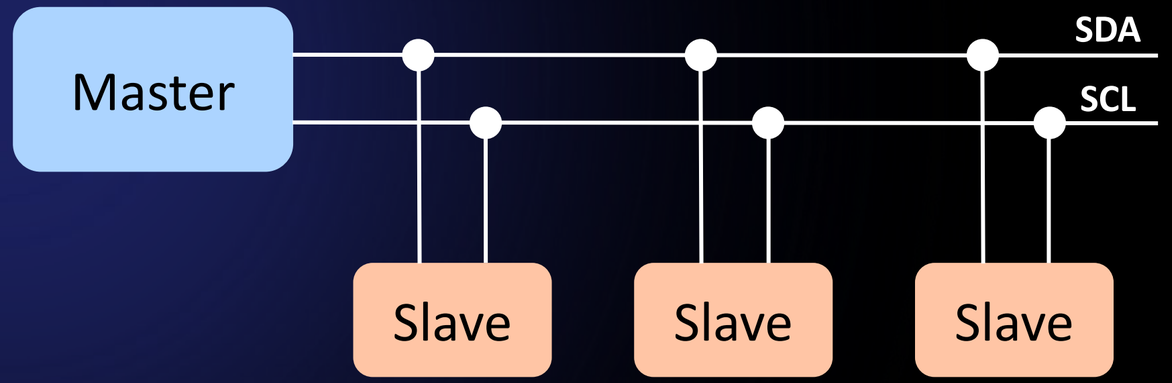
I2C

(1) 개요

SDA, SCL 두 개의 선으로 통신하는 동기식 직렬 통신 방식

(2) 용도

SSD1306 OLED 디스플레이에 실시간 시간 데이터 및
UI 그래픽을 전송하기 위해 사용



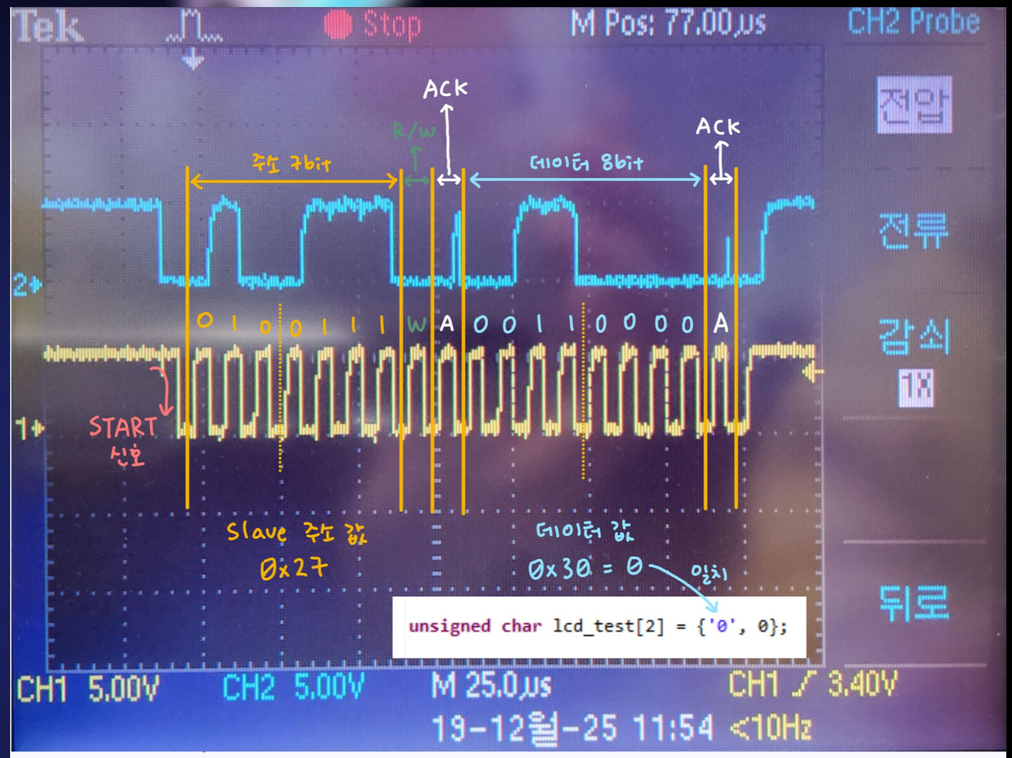
Key Technologies

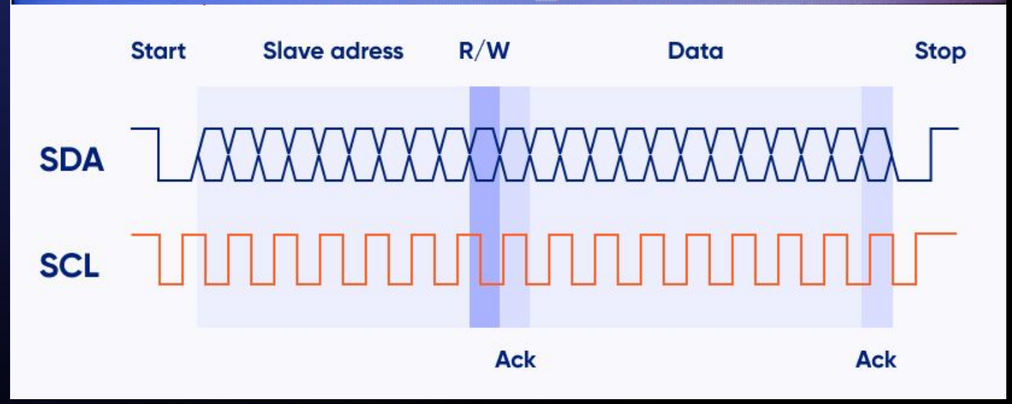
I2C

(3) 동작 방식

Start Condition → Slave 주소(7 bits) → R/W

→ ACK → 데이터(8 bits) → ACK → Stop Condition





Key Technologies

I2c

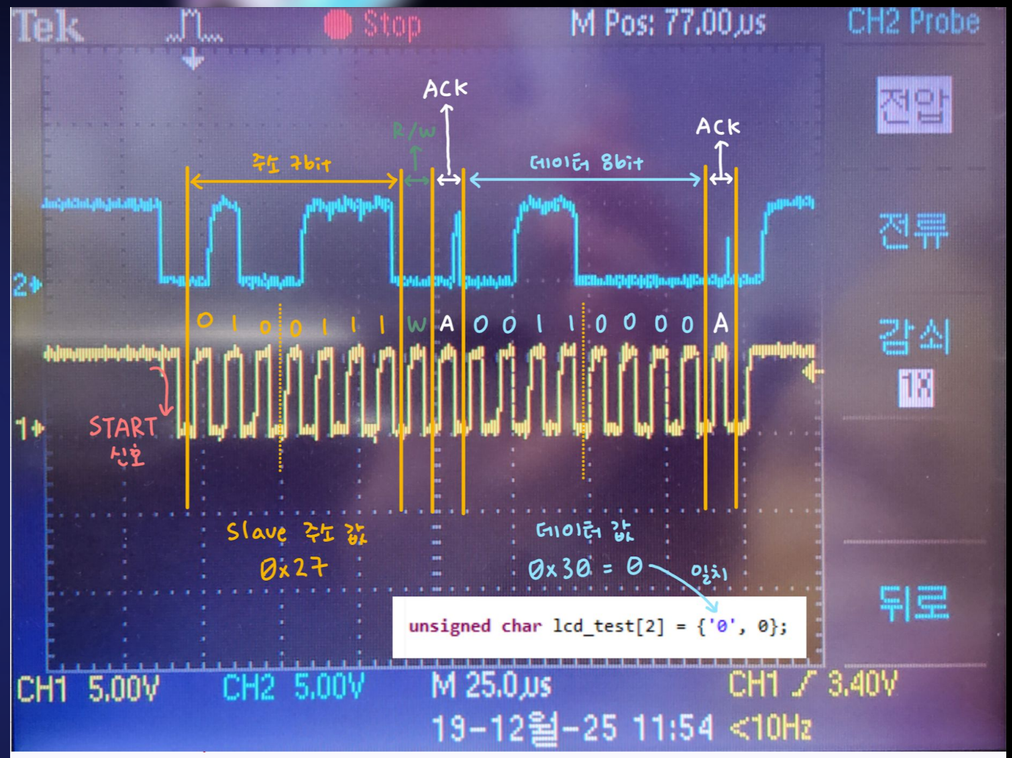
(4) 특징 및 장단점

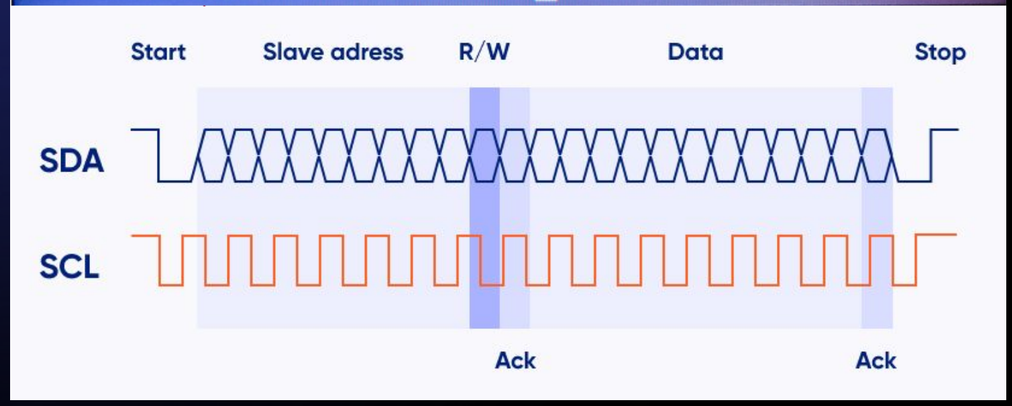
① 특징

- 2선식 인터페이스: SDA와 SCL 단 두 개만 사용
- 다중 마스터/슬레이브: 하나의 버스에 최대 127개 장치 연결 가능 (주소 방식)
- 동기식 통신: 클럭 신호에 맞춰 데이터 전송

② 장점: 배선이 간단, 여러 장치를 한 버스에 연결 가능

③ 단점: 통신 속도 제한적, 장거리 통신 시 노이즈에 취약





Key Technologies

Kernel Workqueue

(1) 개요

리눅스 커널에서 즉시 처리하기 어려운 무거운 작업을 후순위로 미뤄서 비동기적으로 처리하기 위한 메커니즘
인터럽트 핸들러가 점유하는 시간을 최소화하여 시스템의 응답성을 높이는 'Bottom Half' 처리 기법의 일종

(2) 용도

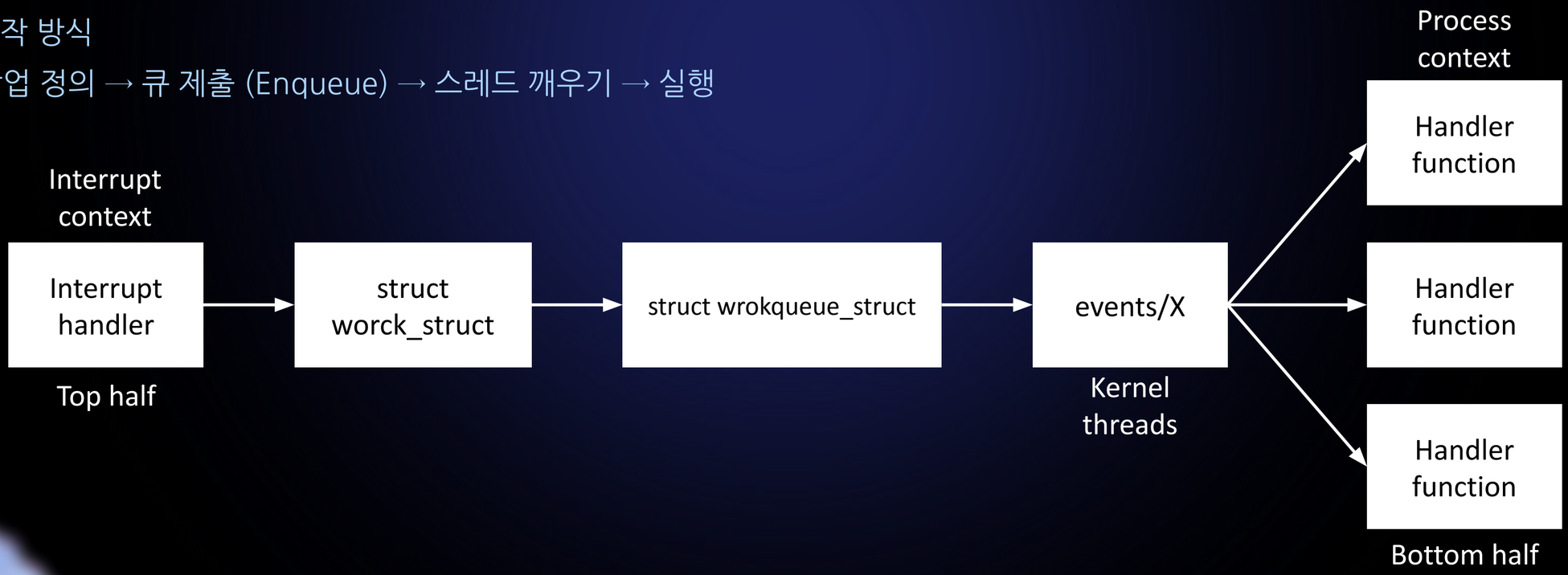
- 지연 처리(Deferred Work): 파일 시스템 입출력, 복잡한 네트워크 패킷 처리 등 시간이 오래 걸리는 작업 수행
- 비동기 성능 최적화: 메인 프로세스의 흐름을 끊지 않고 배경(Background)에서 작업을 병렬로 처리
- 전용 작업 큐 구축: 프로젝트 전용 워크큐를 생성하여 특정 작업의 우선순위와 실행 환경을 독립적으로 관리
- I2C 통신(OLED 업데이트) 및 RTC 읽기, LED 점멸 등에 적용하여 시스템 안정성 확보

Key Technologies

Kernel Workqueue

(3) 동작 방식

작업 정의 → 큐 제출 (Enqueue) → 스레드 깨우기 → 실행



Key Technologies

Kernel Workqueue

(4) 특징 및 장단점

① 특징

- 컨텍스트 전환: 인터럽트(H/W) 컨텍스트에서 프로세스(S/W) 컨텍스트로 작업을 이관
- 자율적 스케줄링: 커널 스케줄러가 시스템 부하에 따라 워커 스레드를 효율적으로 관리

② 장점

- 시스템 안정성: 인터럽트 금지 시간을 최소화하여 시스템 응답성 향상
- 유연성: msleep 등 블로킹 함수 사용 가능
- 독립성 (전용 큐 사용 시): 시스템 공유 큐를 쓰지 않고 전용 큐를 쓰면 타 작업의 간섭 없이 실시간성 확보 가능

③ 단점: 스케줄링 오버헤드가 발생할 수 있음 (전용 큐로 최적화 가능)

Key Technologies

Hardware Interrupts & Software Debouncing

(1) 개요

- 하드웨어 인터럽트(Hardware Interrupt)
  - 외부 장치(버튼, 센서 등)의 신호 변화 발생 시, CPU가 즉시 현재 작업을 멈추고 지정된 처리 루틴(ISR)을 실행하는 메커니즘
- 디바운싱(Debouncing)
  - 기계적 스위치가 눌릴 때 발생하는 미세한 떨림(Chattering) 현상을 제거해 한 번의 입력이 중복 인식되지 않도록 하는 기술

(2) 용도

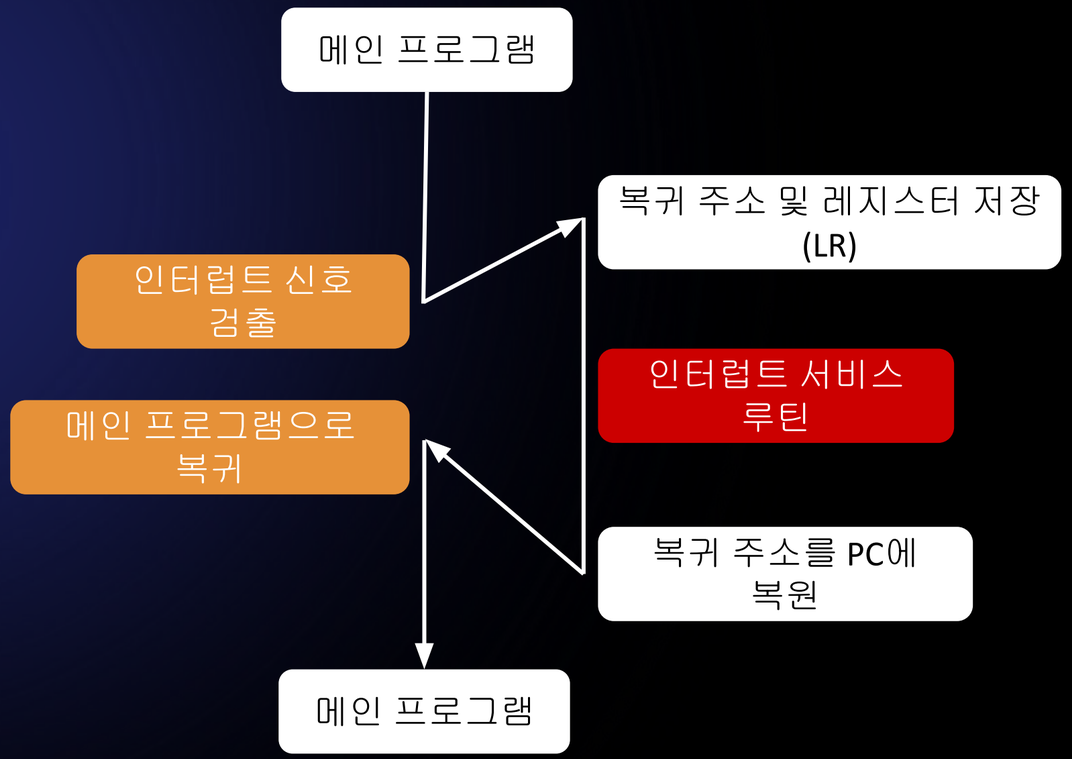
- 하드웨어 인터럽트: 시스템의 실시간 응답성 확보를 위해 사용 (예: 비상 정지 버튼, 센서 데이터 수신)
- 소프트웨어 디바운싱: 기계적 접점의 물리적 한계를 소프트웨어적으로 보완하여 데이터의 정확성↑ (예: 버튼 클릭)
- TACT 스위치 입력, 로터리 엔코더의 회전 신호 및 키 입력을 안정적으로 실시간 감지

Key Technologies

Hardware Interrupts & Software Debouncing

(3) 동작 방식

- 하드웨어 인터럽트
  신호 발생 → 인터럽트 감지 → ISR(Interrupt Service Routine)→ 복귀
- 소프트웨어 디바운싱
  시간 기반 1차 필터링 (Jiffies) → 전기적 안정화 대기(udelay)
  → 상태 재확인(S1이 0인지) → 방향 판별 및 확정



Key Technologies

Hardware Interrupts & Software Debouncing

(4) 특징 및 장단점

① 특징

- 즉시성: 이벤트 발생 즉시 대응
- 효율성: 신호가 없을 때는 CPU 자원 소모 X

② 장점

- 신뢰성: 디바운싱으로 하드웨어의 노이즈와 채터링을 차단해 오작동 방지
- 저전력: CPU가 신호를 계속 감시할 필요가 없어 전력 효율이 좋음

③ 단점

- 응답 지연: 디바운싱 시간만큼 아주 미세한 응답 지연 발생

Troubleshooting

Issue

SSD1306 화면 클리어 함수 실행 시 전송할 데이터를 1024 byte로 묶어서
i2c_master_send를 한 번만 호출해 효율을 높이고자 했으나
마지막 페이지 일부분이 클리어되지 않는 문제 발생

Cause

ftrace로 i2c 관련 이벤트를 추적, 로그를 확인한 결과 데이터는 잘리지 않고 온전히 전송됨
커널 메시지 출력 코드 추가한 결과 역시 1025 byte(제어 1 + 데이터 1024) 전부 전송된 것 확인
따라서, OLED 모듈 쪽에서 큰 데이터를 한 번에 입력받지 못하고 중간에 잘린 것으로 추정

Action

페이지 단위(128 byte)로 나누어 데이터 전송
그 결과, 1 byte씩 전송할 때보다 효율을 높이고 1024 byte로 전송할 때보다 안정적으로 화면
클리어 가능해짐

Troubleshooting

Issue

로터리 엔코더 회전 시 채터링 발생

Cause

시스템 워크큐가 아닌 전용 워크큐로 관리하면서 빠르게 회전할 때 발생했던 채터링은 해결됐으나 천천히 돌릴 때의 채터링은 남음. 엔코더 신호의 전압이 변하는 경계선에서 신호가 잘게 떨리는 것이 원인으로 보임

Action

5ms의 디바운싱 처리
전기적 안정화 대기를 위해 udelay 추가하고 이후 S1값 읽기
회전 초기 노이즈를 제거하기 위해 S1값이 0인지 확인하는 로직 추가(falling edge)
S1값이 0인 것을 확인한 후 S2값을 읽어 방향 확정

Troubleshooting

Issue

LED 점멸 기능을 추가 한 후 OLED 화면이 켜지지 않음

Cause

SSD1306, DS1302 모듈이 이미 리눅스 커널의 기본 워크큐(system_wq)에 등록 상태에서 LED 패턴 제어 워크큐를 추가함
커널 공유 워크큐의 한계로 병목현상이 발생, OLED, LED 워크 함수가 제대로 실행되지 못함

Action

사용자 전용 워크큐(my_workqueue)를 생성해 시스템 기본 큐와 분리된 전용 독립 통로를 구축
다른 커널 프로세스의 간섭 없이 프로젝트 전용 스레드가 즉시 할당되어 OLED 갱신 및 LED 점멸 기능 안정적으로 구동

Troubleshooting

Issue

DS1302 RTC 모듈을 GPIO2/3/4에 연결했을 때 통신 불가(시간 읽기 실패/에러값).

Cause

GPIO2/3는 I2C(SDA/SCL) 기본 핀으로 설정/풀업 및 커널 점유 가능성이 있어, 비트뱅잉 방식의 DS1302 신호 토글이 정상 동작하지 않을 수 있음.
또한 핀 기능(ALT) 또는 다른 오버레이(1-Wire 등)로 GPIO가 점유된 경우 충돌 가능.
혹은 안정화 시간, 통신 시간의 수정이 필요해 보임.

Action

DS1302 신호선을 I2C 기본 핀(GPIO2/3)에서 **일반 GPIO로 변경**하여 재배선함.

- 변경 전: SDA/SCL/CE = GPIO2/ 3/ 4
- 변경 후: SDA/SCL/RST = GPIO17/ 27/ 22

타이밍 또한 맞지 않는 것같이 딜레이값 수정

워크큐를 추가하여 사용

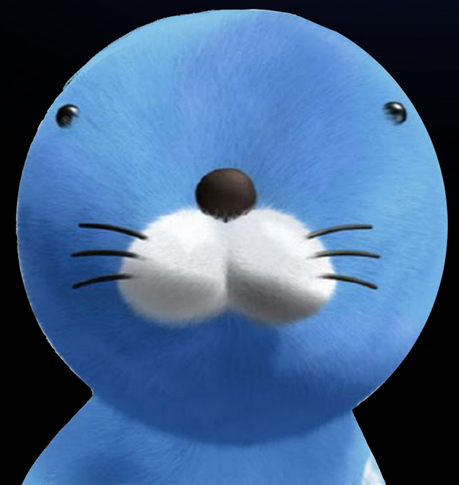
Troubleshooting

Issue

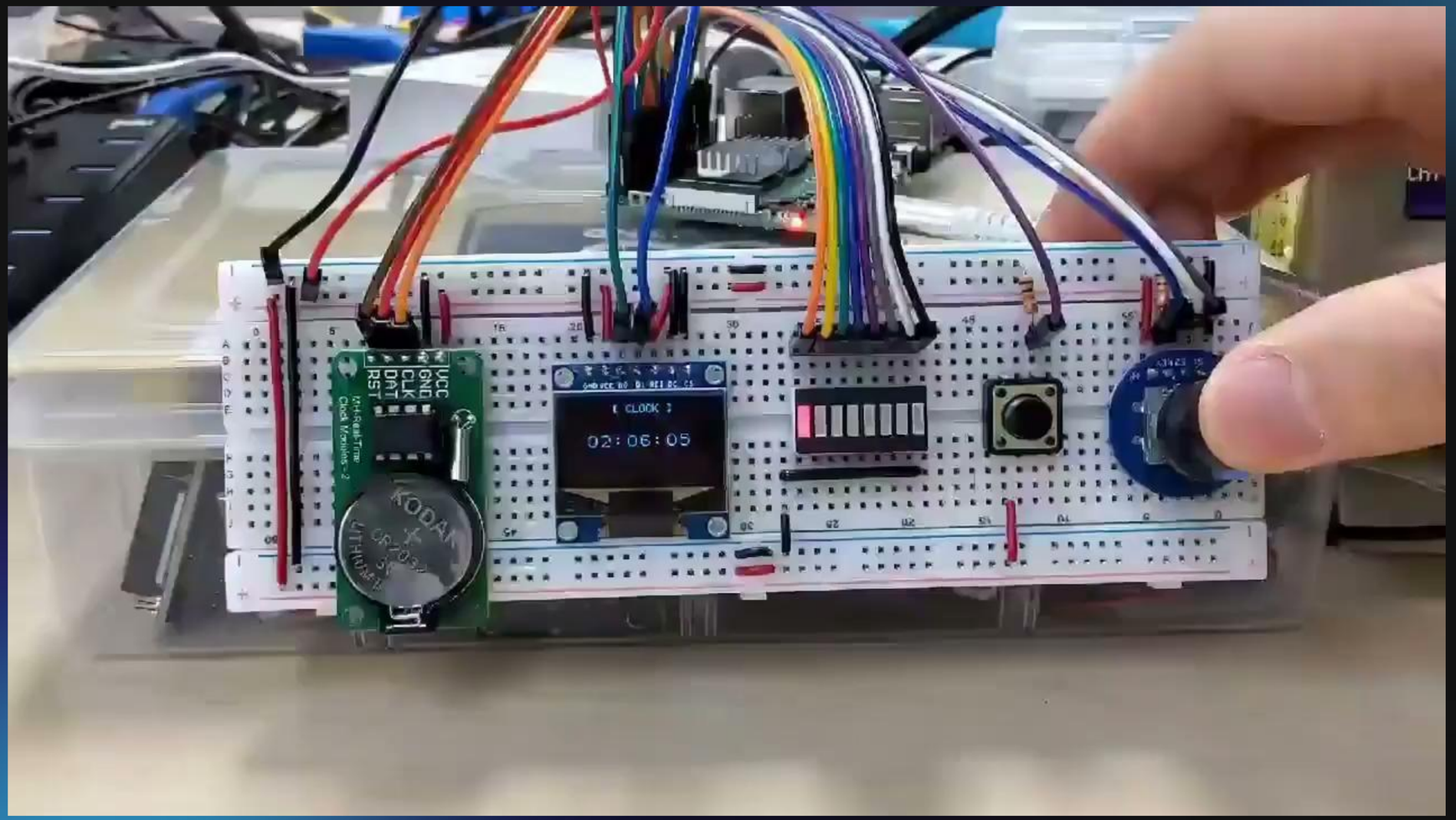
Human Error

Cause

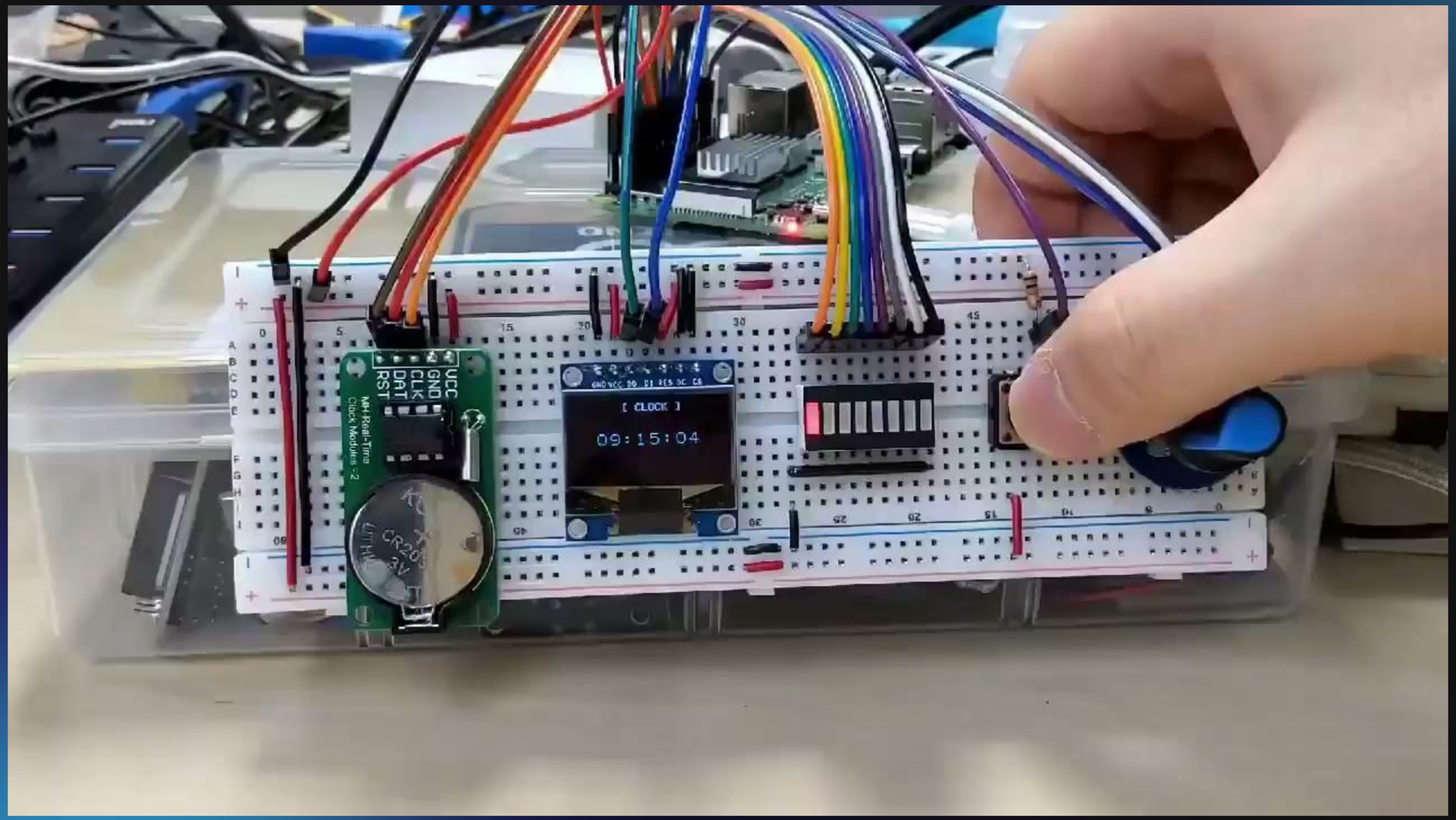
1. Make하고 최신화된 .ko File 안 넘기고 insmod 후 절규하기
2. 배선 밀려 꽂기
3. backup MakeFile 수정하고 왜 안 되지 외치기
4. 저항 연결 안 해서 LED 터트리기



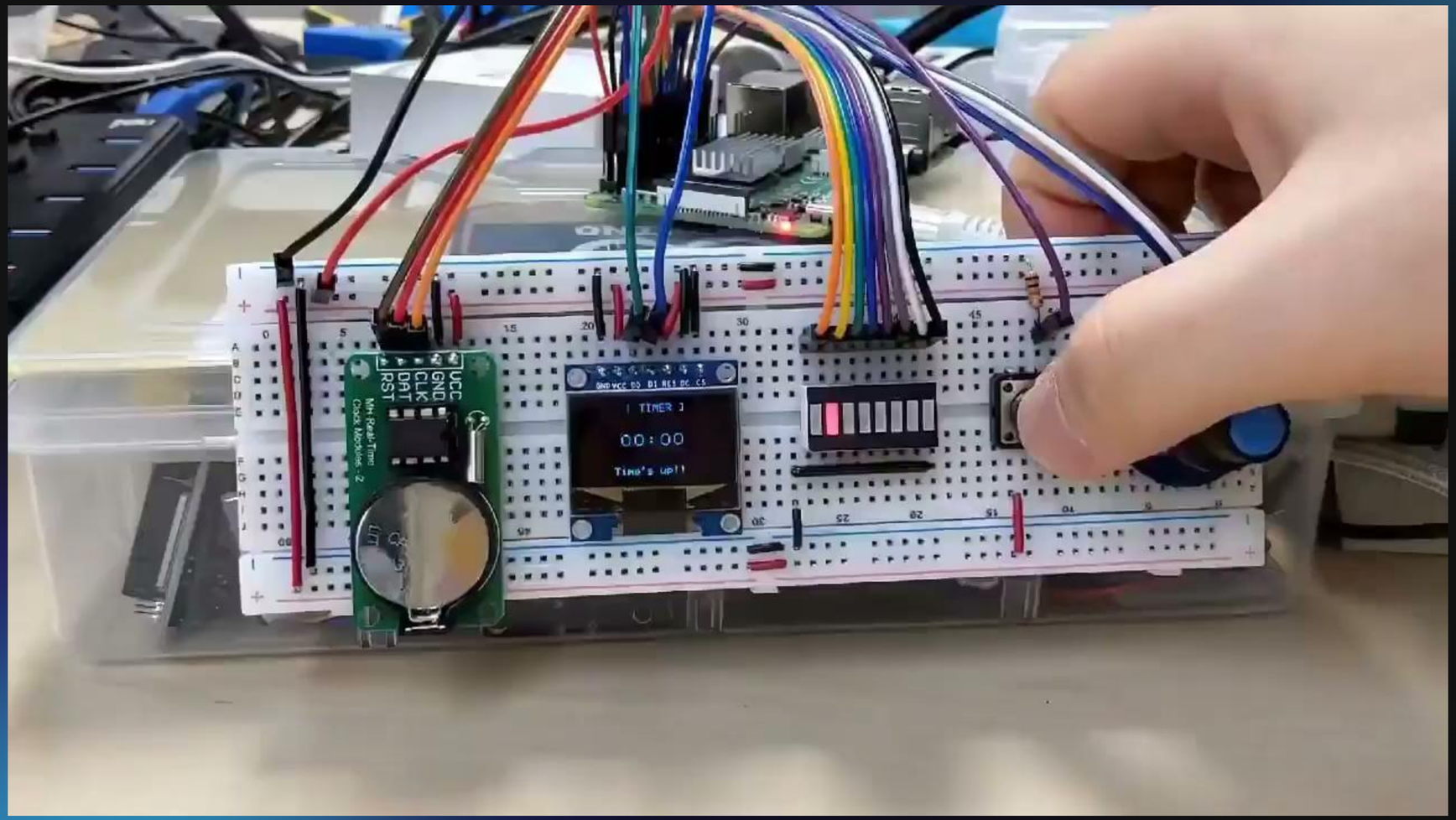
Demonstration video



Demonstration video



Demonstration video



Conclusion & Insights

Conclusion

(1) 목표 달성

리눅스 커널 드라이버를 통해 시계, 타이머, 뽀모도로 모드를 성공적으로 구현

- FSM을 사용한 모드 전환 및 동작 제어
- 로터리 엔코더와 Tact 스위치로 시간 설정 및 제어
- OLED 디스플레이로 시간과 상태 표시

(2) 주요 성과

- 모드 전환 및 상태 제어의 정확한 구현
- I2C 통신을 활용한 효율적인 데이터 전송
- OLED 디스플레이를 통한 실시간 시간 및 타이머 상태 등 표시
- Kernel Workqueue를 통해 안정성 및 응답성 향상

Conclusion & Insights

Insights

(1) 모듈화 및 FSM 설계
- 각 모드의 독립적 관리와 원활한 전환 구현
- 시스템의 가독성과 유지보수 용이성 강화

(2) 리눅스 커널과 하드웨어 통신 최적화: I2C 통신 및 데이터 전송 최적화로 효율성 증대

(3) 비동기 처리와 안정성: Kernel Workqueue 활용하여 시스템 응답성 및 안정성 향상

(4) 디바운싱 및 노이즈 처리: 하드웨어 신호 처리에서 발생한 채터링 문제 해결

(5) 하드웨어 유연성: 핀 충돌 문제 해결 및 하드웨어와 소프트웨어 간의 효율적 상호작용 관리

Thank You

Lee Duhyeon

Ryu Gyunbong