E L E V A T O R P r o j e c t

With STM32F411RE

(Telechips) AI 시스템반도체 SW 개발자

박성호 박준영

TABLE OF CONTENTS

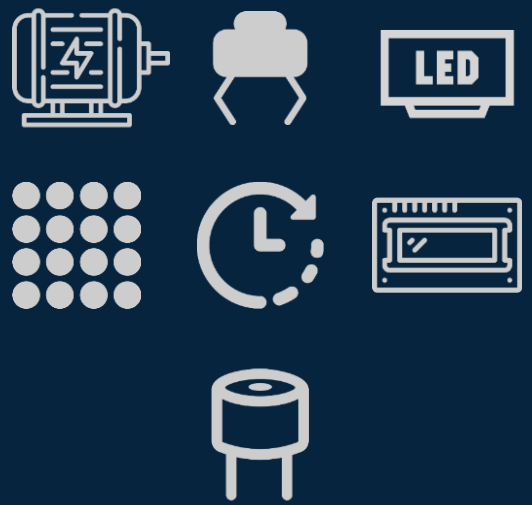
01 프로젝트 개요

프로젝트 목적

- RTOS 기반으로 실행되는 Elevator 구현

프로젝트 내용

- Step Motor를 이용하여 Elevator의 이동 구현
- Photoelectric Switch Sensor를 이용하여 Elevator의 층수 구별
- Button을 이용하여 층수 선택
- DS1302 & I2C LCD를 이용하여 날짜와 시간 출력
- Dotmatrix를 이용하여 Elevator의 위치(층수) 표시
- LED Bar를 이용하여 Elevator의 움직임을 가시적으로 표현
- Buzzer를 이용하여 Elevator의 도착 알림음 출력



01 프로젝트 개요

주요 부품

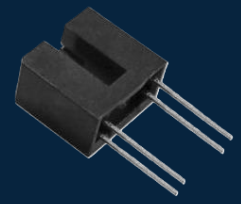
STM32 Nucleo-F411RE

DS1302

LCD1602

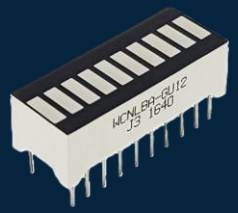
Step Motor

Photoelectric Switch



Dotmatrix

LED Bar



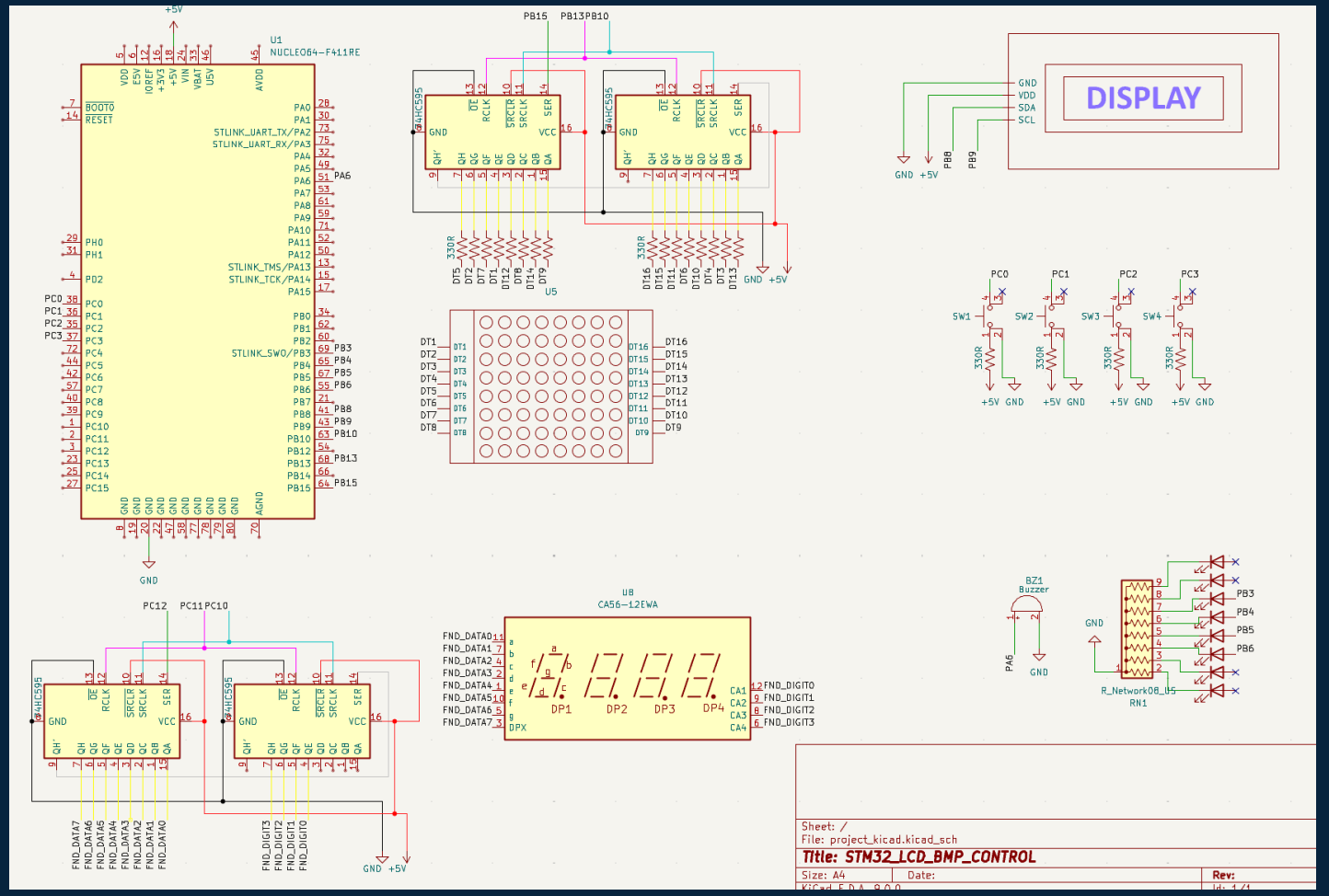
Buzzer

Trim Potentiometer

Button

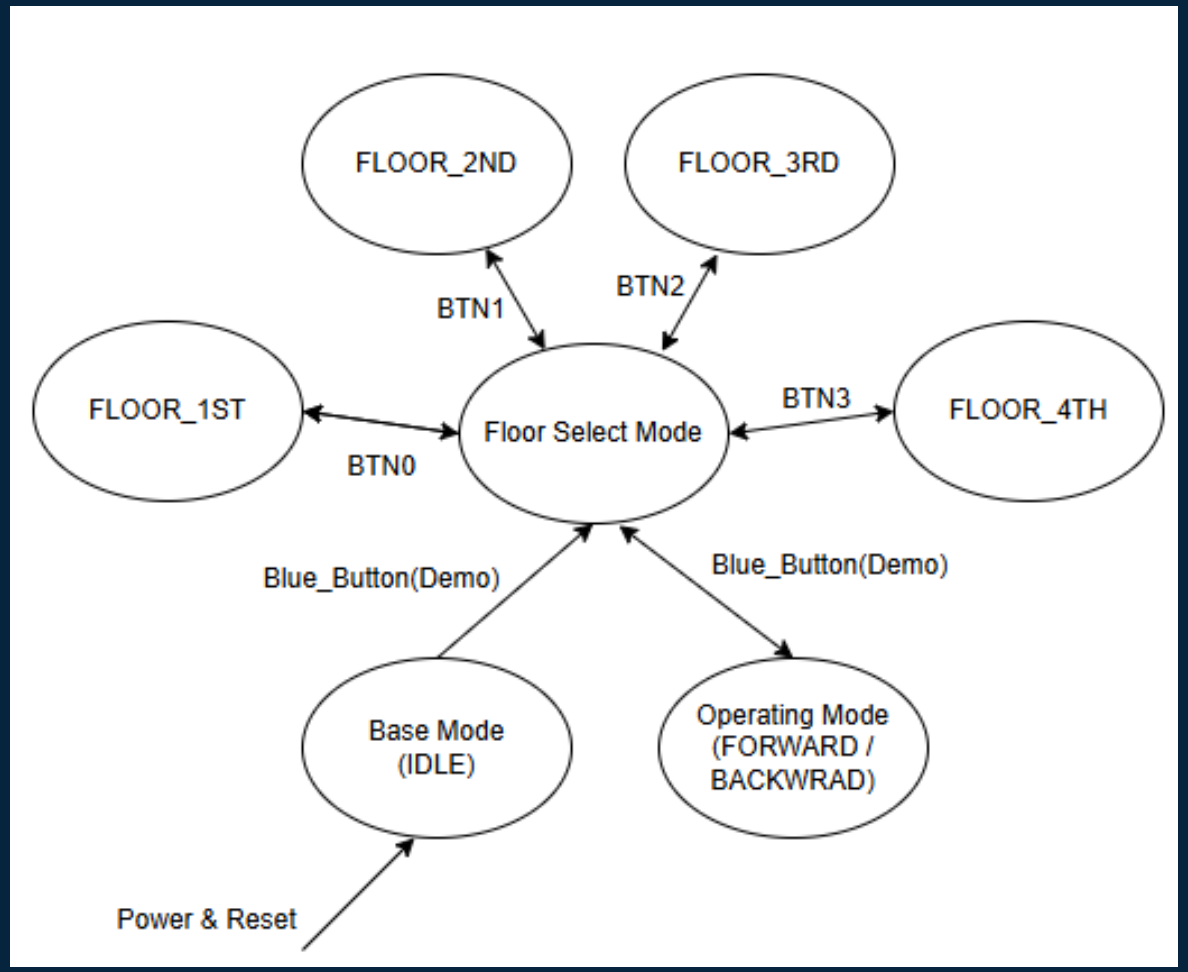
02 Schematic & FSM & Pinout View

Schematic (회로도)



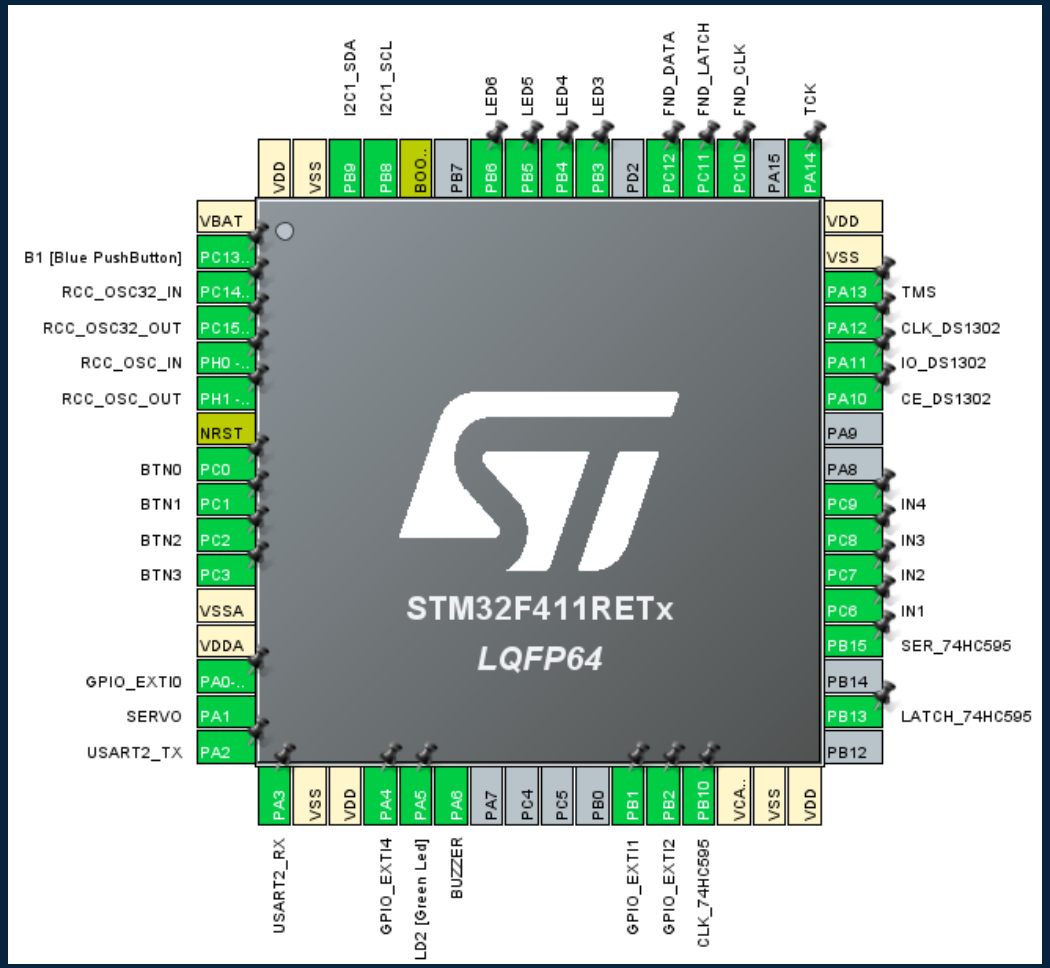
02 Schematic & FSM & Pinout View

FSM (상태천이도)



02 Schematic & FSM & Pinout View

Pinout View



03 세부 구현 기능

구현 기능 설명

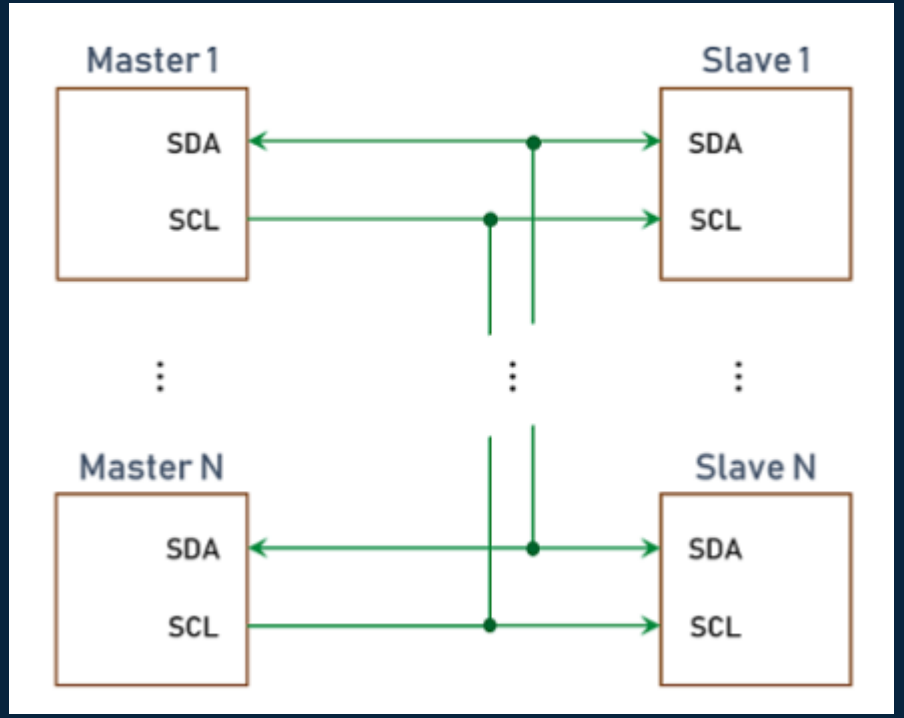
- Step Motor – IDLE(정지) / FORWARD(정방향 회전) / BACKWARD(역방향 회전) 상태에 따라 동작
- Photoelectric Switch – Step Motor를 통해 Elevator가 이동을 할 때
  각 층에 있는 Photoelectric Switch Sensor가 인식을 하여 현재 Elevator의 위치를 알 수 있음
- I2C LCD & DS1302 – I2C Protocol을 통해서 날짜와 시간을 LCD 화면에 출력
- Dotmatrix – Elevator의 기본 모드 / 층수 선택 모드 표시 및 Elevator의 현재 위치(층수) 출력
- Buzzer – Elevator가 최종 층수에 도착을 하면 도착 알림음 출력
- LED Bar – 층수 선택 모드에서 각 층별로 다른 LED 표시 (1층 : 1개, 2층 : 2개, 3층 : 3개, 4층 : 4개)
  Elevator가 올라갈 때는 LED 4개가 위로 shift되는 애니메이션 출력, 내려갈 때는 아래로 shift

03 I2C PROTOCOL

I2C PROTOCOL이란?

두 라인을 이용해 데이터를 교환하는 시리얼 통신 방식

I2C PROTOCOL의 특징
하나의 데이터 라인(SDA)과 하나의 클럭 라인(SCA)을 이용하는 동기식 방식
하나의 MASTER에서 여러 개의 SLAVE 제어(N by N 통신)



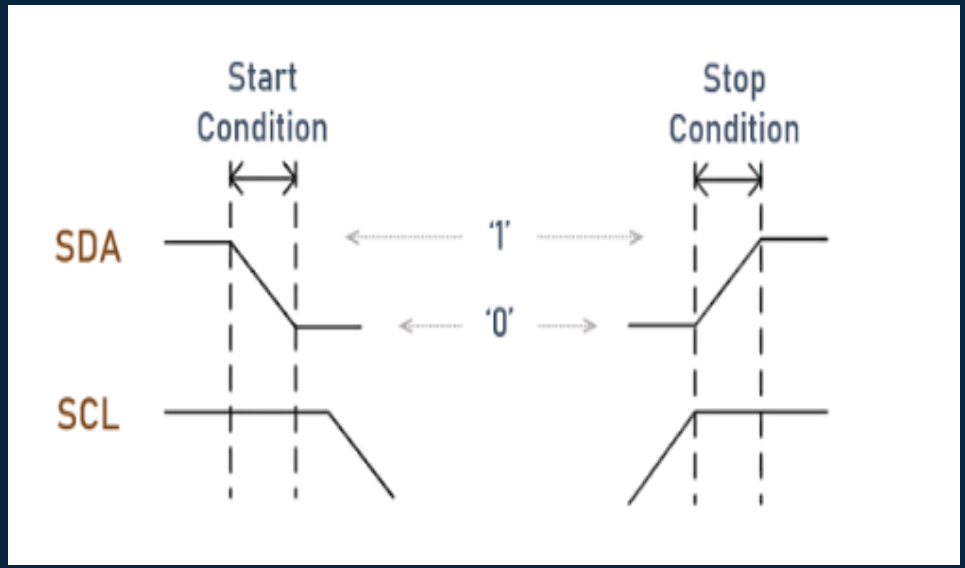
03 I2C PROTOCOL

I2C START/STOP

DATA 교환 전 => SCL,SDA = HIGH 상태 유지

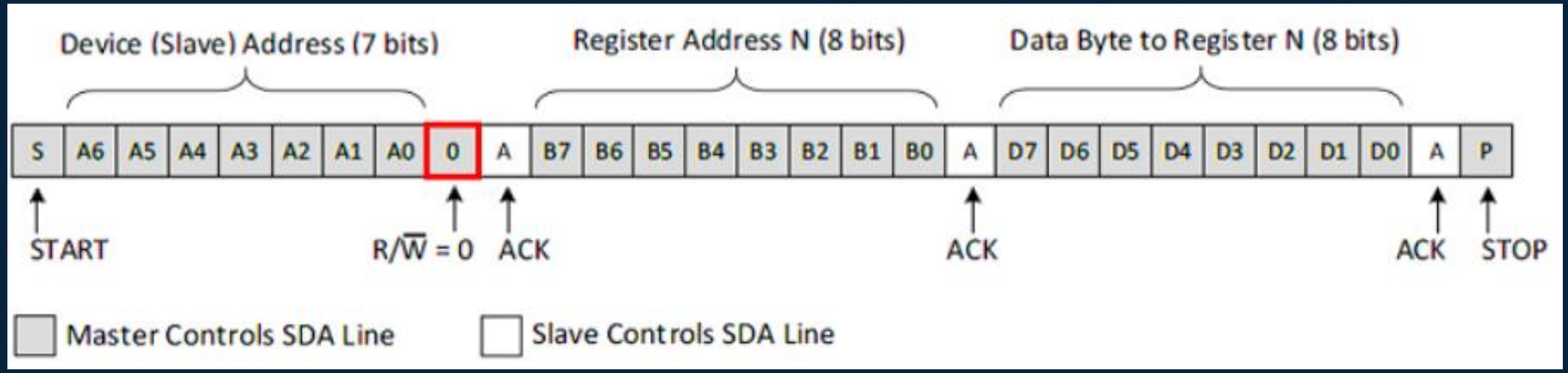
통신 START => SDA(DATA LINE)가 먼저 LOW 신호로 변환

통신 FINISH => SCL(CLOCK LINE)이 먼저 HIGH로 변환



03 I2C PROTOCOL

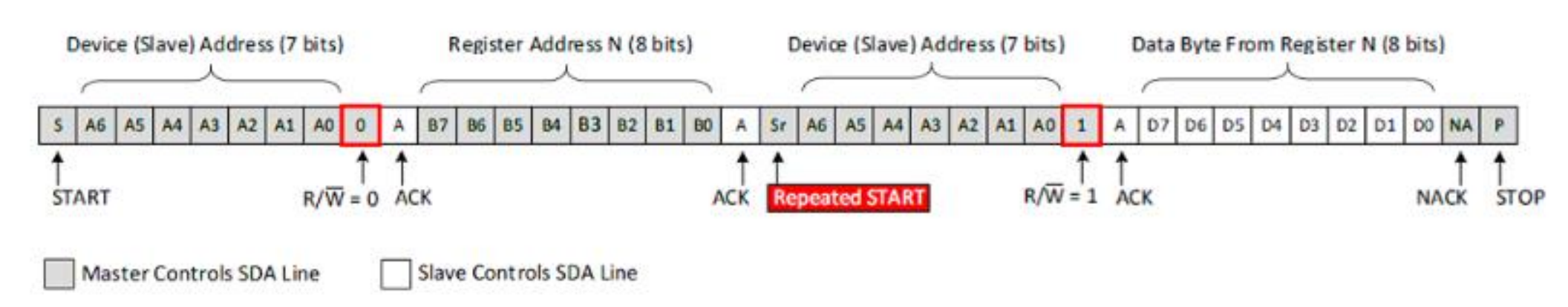
I2C Write Sequence



1. start 후 slave address 7bit, R(1)/W(0) 값 전송
2. Write하고자 하는 slave 내부 데이터 영역 선택을 위해 8bit resister 정보 전송
3. resister 선택 후 byte 단위로 전송이 끝날 때 마다 ACK 수신(MASTER)
4. WRITE 데이터가 없으면 STOP 후 I2C 통신 종료

03 I2C PROTOCOL

I2C Read Sequence



1. Resister 선택 지점 까지 write와 동일
2. Resister 선택 후, START 신호를 다시 전송
3. R/W값을 1로 전송, SLAVE에서 이전에 선택된 DATA값을 MASTER에 전송
4. Read 종료 시 master->slave로 NACK 신호 전송&STOP

03 I2C PROTOCOL(참고)

Clock Stretching

I2C 통신에서 슬레이브가 데이터 처리를 위해 시간이 더 필요할 때, SCL 라인을 강제로 Low로 유지시켜 마스터가 클럭을 더 보내지 못하도록 대기시키는 메커니즘

슬레이브가 처리 준비가 되기 전까지 **SCL을 Low로 유지**하면서 마스터의 클럭 전송을 지연시킴.

04 코드 설명 (i2c_protocol.c)

```c
#include "i2c_protocol.h"
#include "extern.h"

#define I2C_SDA 9
#define I2C_SCL 8

void i2c_start(void);
void i2c_stop(void);
void i2c_send_bit(uint8_t bit);
uint8_t i2c_send_byte(uint8_t data);
void init_i2c(void);

#define SDA_HIGH() (GPIOB->ODR |= (1 << I2C_SDA))
#define SDA_LOW()  (GPIOB->ODR &= ~(1 << I2C_SDA))
#define SCL_HIGH() (GPIOB->ODR |= (1 << I2C_SCL))
#define SCL_LOW()  (GPIOB->ODR &= ~(1 << I2C_SCL))
#define SDA_READ() ((GPIOB->IDR >> I2C_SDA) & 0x01)
```

I2C Protocol 방식

- GPIOB의 SDA, SCL 핀을 직접 제어해서 HIGH 또는 LOW를 만드는 매크로
- SDA와 SCL 선을 제어하는 매크로 정의
- GPIOB->ODR 레지스터를 직접 제어하여 해당 핀을 HIGH 혹은 LOW로 설정
- SDA와 SCL 제어를 위해 각각 비트 연산 사용
- GPIOB->IDR 레지스터로부터 SDA 핀의 상태를 읽음
- I2C_SDA만큼 shift한 후 마스크 연산을 수행하여 해당 핀의 비트만 추출

04 코드 설명 (i2c_protocol.c)

```c
void i2c_start(void)
{
    SDA_HIGH();
    delay_us(5);
    SDA_LOW();
    delay_us(5);
    SCL_LOW();
    delay_us(5);
}

void i2c_stop(void)
{
    SDA_LOW();
    delay_us(5);
    SCL_HIGH();
    delay_us(5);
    SDA_HIGH();
    delay_us(5);
}

void i2c_send_bit(uint8_t bit)
{
    if(bit)
       SDA_HIGH();
    else
       SDA_LOW();

    delay_us(5);
    SCL_HIGH();
    delay_us(5);
    SCL_LOW();
    delay_us(5);
}
```

void i2c_start(void)

- SDA가 HIGH -> LOW, SCL은 HIGH 상태여야 시작 조건, 버스에 데이터 전송을 시작하겠다는 신호

void i2c_stop(void)

- SDA가 LOW -> HIGH, SCL을 HIGH로 해서 슬레이브가 읽게 함

void i2c_send_bit(uint8_t bit)

- SDA에 비트를 설정하고 SCL을 HIGH로 해서 슬레이브가 읽게 함

04 코드 설명 (i2c_protocol.c)

```c
uint8_t i2c_send_byte(uint8_t data)
{
    for (int i = 7; i >= 0; i--) {
        i2c_send_bit((data >> i) & 0x01);
    }

    SDA_HIGH();
    delay_us(5);

    SCL_HIGH();
    delay_us(5);
    uint8_t ack = SDA_READ();
    SCL_LOW();
    delay_us(5);

    return ack;
}

void init_i2c(void)
{
    GPIOB->MODER &= ~((3 << (I2C_SDA * 2)) | (3 << (I2C_SCL * 2)));
    GPIOB->MODER |=  ((1 << (I2C_SDA * 2)) | (1 << (I2C_SCL * 2)));

    GPIOB->OTYPER |= ((1 << I2C_SDA) | (1 << I2C_SCL));

    GPIOB->PUPDR &= ~((3 << (I2C_SDA * 2)) | (3 << (I2C_SCL * 2)));
    GPIOB->PUPDR |=  ((1 << (I2C_SDA * 2)) | (1 << (I2C_SCL * 2)));
}
```

uint8_t i2c_send_byte(uint8_t data)

- 8비트를 순서대로 보내고 ACK(응답)를 받는 함수
- 왼쪽부터 1비트씩 i2c_send_bit() 함수로 전송하고 마지막에는 슬레이브가 잘 받았는지 확인 (SDA가 0이면ACK)

void init_i2c(void)

- GPIO 모드를 출력(Open-Drain), 풀업 설정, 핀들을 I2C 통신에 쓸 수 있게 설정한 단계

04 코드 설명 (extint.c)

```c
#include "extint.h"
#include "stepmotor.h"
volatile uint8_t stepmotor_state;
volatile uint8_t pin_state = 0;
volatile uint8_t current_state;

void HAL_GPIO_EXTI_Callback(uint16_t GPIO_Pin)
{
    switch(GPIO_Pin)
    {
    case GPIO_PIN_0:
        pin_state = HAL_GPIO_ReadPin(GPIOA, GPIO_PIN_0);
            if (pin_state) // LOW
            {
                current_state = FLOOR_4TH;
            }
        break;
    case GPIO_PIN_1:
        pin_state = HAL_GPIO_ReadPin(GPIOB, GPIO_PIN_1);
            if(pin_state)
            {
                current_state = FLOOR_3RD;
            }
        break;
    case GPIO_PIN_2:
        pin_state = HAL_GPIO_ReadPin(GPIOB, GPIO_PIN_2);
            if(pin_state)
            {
                current_state = FLOOR_2ND;
            }
        break;
    case GPIO_PIN_4:
        pin_state = HAL_GPIO_ReadPin(GPIOA, GPIO_PIN_4);
            if (pin_state) // LOW
            {
                current_state = FLOOR_1ST;
            }
        break;
    }
}
```

외부 인터럽트 이벤트에 따른 상태 변경

- 인터럽트를 유발한 GPIO 핀의 번호를 전달 받아 동작 실행
- GPIO_PIN_0 : PORT A의 PIN 0 상태 읽음
  - current_state = FLOOR_4TH(4층)
- GPIO_PIN_1 : PORT B의 PIN 1 상태 읽음
  - current_state = FLOOR_3RD(3층)
- GPIO_PIN_2 : PORT B의 PIN 2 상태 읽음
  - current_state = FLOOR_2ND(2층)
- GPIO_PIN_4 : PORT A의 PIN 4 상태 읽음
  - current_state = FLOOR_1ST(1층)

04 코드 설명 (stepmotor.c)

```c
void stepmotor_main(void)
{
   switch(stepmotor_state)
   {
   case IDLE:
      if (get_button(GPIOC, GPIO_PIN_13, BTN4) == BUTTON_PRESS)
      {
         if(floor_selection_mode == 0)
         {
            floor_selection_mode = 1;
            floor_count = 0;
            floor_index = 0;
         }
         else
         {
            floor_selection_mode = 0;

            if(floor_count > 0)
            {
               floor_state = selected_floors[0];

               if(current_state < floor_state)
                  stepmotor_state = FORWARD;
               else if(current_state > floor_state)
                  stepmotor_state = BACKWARD;
            }
         }
      }
```

Step Motor 제어 상태 머신 (IDLE 상태)

기본 모드일 때

- 버튼4 (STM32 blue push button)으로 모드 전환
- 기본 모드와 층수 선택 모드 전환
- 층수 선택 모드에서 selected_floors에 선택한 층 순서대로 저장

04 코드 설명 (stepmotor.c)

```
if(floor_selection_mode == 1)
{
   dotmatrix_main_test2();  // 선택 모드임을 dotmatrix에 표시

   if(get_button(GPIOC, GPIO_PIN_0, BTN0) == BUTTON_PRESS)
   {
       int found = 0;
       for (int i = 0; i < floor_count; i++)
       {
            if (selected_floors[i] == 1)
            {
               found = 1;

               for (int j = i; j < floor_count - 1; j++)
                   selected_floors[j] = selected_floors[j + 1];
               floor_count--;
               break;
            }
       }
       if (!found)
       {
            if (floor_count < MAX_FLOORS)
               selected_floors[floor_count++] = 1;  // 1층 추가
       }
       led_all_off();

       if (floor_count > 0 && selected_floors[floor_count - 1] == 1)
            led_one_on();

       dot_number = (floor_count > 0 ? selected_floors[floor_count - 1] : 0);
       dotmatrix_main_test();
   }
```

층수 선택 모드일 때 (IDLE 상태)

- BTN0 ~ BTN3을 통해 1층부터 4층까지 선택하거나 해제
- 층수가 이미 배열에 존재하는지 확인
  (처음에는 0으로 초기화하여 아직 층수 선택 안됨을 의미)
- 현재 저장된 모든 선택된 층을 순회하며 이미 선택된 경우에는 이를 표시
- 배열의 뒤쪽에 위치한 값들을 한 칸씩 앞으로 이동
- 결과적으로 배열에서 해당 요소 제거되고 배열 순서 유지
- 1층부터 4층까지 같은 내용 반복

04 코드 설명 (stepmotor.c)

```
case FORWARD:
   stepmotor_drive(FORWARD);
   set_rpm(13);
   dot_number = current_state;
   dotmatrix_main_test();

   if(current_state == floor_state)
   {
      beep(3);
      osDelay(10);
      floor_index++;
      if(floor_index < floor_count)
      {
         floor_state = selected_floors[floor_index];

         if(current_state < floor_state)
            stepmotor_state = FORWARD;
         else if(current_state > floor_state)
            stepmotor_state = BACKWARD;
      }
      else
      {
         stepmotor_state = IDLE;
      }
   }
   break;
```

층수 선택 모드일 때 (FORWARD 상태) (BACKWARD 상태도 동일)

- stepmotor_drive(FORWARD) 함수 : 모터 한 단계 전진
- set_rpm(13) : 분당 회전수 값 13에 해당하는 딜레이 삽입
- 현재 위치(층수)를 나타내는 current_state의 값을 Dotmatrix에 출력할 숫자 변수 dot_number에 저장 후 출력
- 현재 위치가 목표 층과 같으면 Elevator 도착 의미
- 도착 알림음 출력
- 도착 후 다음에 이동해야 할 층을 가리키기 위해 floor_index 증가
- 선택 층 남아있으면 배열에 저장되어 있는 다음 층수를 목표 층으로 업데이트
- 현재 위치와 새 목표 층 비교하여 Step Motor 이동 방향 결정
- 모든 층 도착 완료 후 IDLE 상태로 전환

04 코드 설명 (main.c)

```
/* EXTI interrupt init*/
HAL_NVIC_SetPriority(EXTI0_IRQn, 5, 0);
HAL_NVIC_EnableIRQ(EXTI0_IRQn);

HAL_NVIC_SetPriority(EXTI1_IRQn, 5, 0);
HAL_NVIC_EnableIRQ(EXTI1_IRQn);

HAL_NVIC_SetPriority(EXTI2_IRQn, 5, 0);
HAL_NVIC_EnableIRQ(EXTI2_IRQn);

HAL_NVIC_SetPriority(EXTI4_IRQn, 5, 0);
HAL_NVIC_EnableIRQ(EXTI4_IRQn);
```

외부 인터럽트 설정

- 외부 인터럽트의 우선 순위 설정, 해당 인터럽트 활성화

- EXTI0 인터럽트(PA0)를 우선순위 5, 서브 우선순위 0으로 설정
- EXTI0 인터럽트 활성화

- EXTI1 인터럽트(PB1)를 우선순위 5, 서브 우선순위 0으로 설정
- EXTI1 인터럽트 활성화

- EXTI2 인터럽트(PB2)를 우선순위 5, 서브 우선순위 0으로 설정
- EXTI2 인터럽트 활성화

- EXTI4 인터럽트(PA4)를 우선순위 5, 서브 우선순위 0으로 설정
- EXTI4 인터럽트 활성화

04 코드 설명 (main.c)

```
defaultTaskHandle = osThreadNew(StartDefaultTask, NULL, &defaultTask_attributes);

/* creation of myTask02 */
myTask02Handle = osThreadNew(StartTask02, NULL, &myTask02_attributes);

/* creation of myTask03 */
myTask03Handle = osThreadNew(StartTask03, NULL, &myTask03_attributes);

/* creation of myTask04 */
myTask04Handle = osThreadNew(StartTask04, NULL, &myTask04_attributes);

/* creation of myTask05 */
myTask05Handle = osThreadNew(StartTask05, NULL, &myTask05_attributes);
```

Task(Thread) 생성

- Task를 생성하여 여러 기능을 독립적으로 실행
- defaultTask, myTask02, myTask03, myTask04, myTask05 생성

04 코드 설명 (main.c)

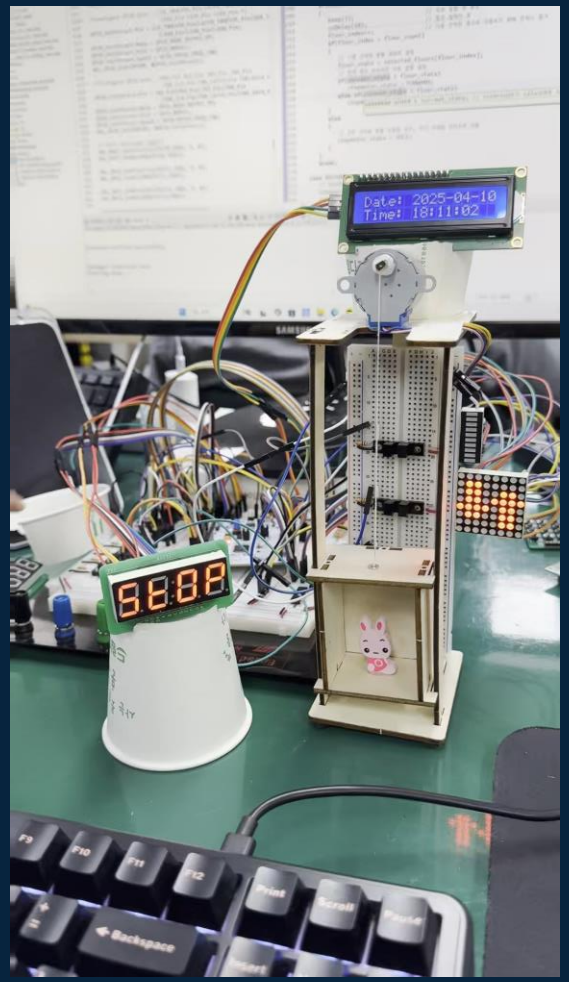
```c
void StartDefaultTask(void *argument)
{
  for(;;)
  {
      ds1302_main();
      osDelay(50);
  }
}

void StartTask02(void *argument)
{
  for(;;)
  {
      dotmatrix_main_test();
      osDelay(1);
  }
}

void StartTask03(void *argument)
{
  for(;;)
  {
    stepmotor_main();
    osDelay(1);
  }
}

void StartTask04(void *argument)
{
  for(;;)
  {
      if(stepmotor_state == FORWARD)
      {
          shift_right_ledon();
      }
      else if(stepmotor_state ==  BACKWARD)
      {
          shift_left_ledon();
      }

      osDelay(1);
  }
}

void StartTask05(void *argument)
{
  for(;;)
  {
    fnd_display();
    osDelay(1);
  }
}
```

Task(Thread) 개별 루프

- defaultTask
  - ds1302_main() 함수 실행
- myTask02
  - dotmatrix_main_test() 함수 실행
- myTask03
  - stepmotor_main() 함수실행
- myTask04
  - Step Motor의 상태에 따라 LED 효과 실행
  - Step Motor의 이동 방향에 따라 LED의 shift 애니메이션 출력
- myTask05
  - fnd_display() 함수 실행

05 결과 및 고찰

동작 영상 (https://youtube.com/shorts/RyurffD4WDc)



05 결과 및 고찰

고찰

- 시스템 확장으로 선택할 수 있는 층수가 늘어나거나 선택 항목에 대해 빈번한 동적 변경이 발생할 때는 배열보다 동적 자료구조 사용이 유리할 것 같습니다.
- 외부 인터럽트를 통해 빠른 이벤트 응답을 받고 태스크 기반 구조를 통해 전체 시스템의 복잡한 동작들을 효율적으로 관리하는 방법을 배울 수 있었습니다.
- I2C Protocol 방식을 HAL 드라이버 없이 구현하면서 레지스터 기반 GPIO 제어를 통해 하드웨어와 소프트웨어의 밀접한 동작 방식을 학습할 수 있었습니다.

Thank You

(Telechips) AI 시스템반도체 SW 개발자

박성호 박준영