부산대학교
PUSAN NATIONAL UNIVERSITY

임베디드 시스템 설계 및 실험 5조 텀프로젝트 보고서

■과 목 명 임베디드 시스템 설계 및 실험

■담당교수 김원석

■제 출 일 2024/12/23

■학 과 정보컴퓨터공학부

■조원 팀장:김대욱,
팀원:설종환,
박정민, 이승원

부산대학교
PUSAN NATIONAL UNIVERSITY

목차

1.목적

본 프로젝트는 스마트 블라인드 시스템을 통해 현대 가정 및 사무실 환경에서의 편리성, 에너지 절약, 사용자 경험 개선을 목표로 하며, 이를 통해 실용적인 스마트 홈 솔루션을 제공하고, 팀원의 기술적 역량을 강화하는 데 목적을 두고 있다.

2.개발내용

여러 센서와 블루투스 모듈을 활용하여 블라인드를 올리고 내리는 임베디드 시스템을 개발했다. 이 시스템은 환경 센서를 통해 조도 및 온도 데이터를 수집하고, 온도와 조도 값 및 블루투스 명령을 통해 블라인드를 제어한다. 사용자는 블라인드를 원격으로 제어하거나, 환경 조건에 따라 블라인드가 자동으로 작동하는 기능을 사용할 수 있다.

개발한 기능은 다음과 같다.

1.환경 센서 기반 블라인드 제어:

조도 센서(GL5537) 모듈을 사용하여 주변 광량을 측정하며, 임계값(4000 lux)을 초과할 경우 온도 센서를 활성화한다. 조도가 낮은 경우 햇빛이 없으므로 온도 또한 낮은 것으로 판단하여 온도 센서를 비활성화시킴으로써 에너지 소비를 절감한다.

온도 센서(LM35DZ) 모듈을 사용하여 주변 온도를 측정하고, 임계값(28 도) 이상일 경우 블라인드를 자동으로 내린다.

2.블루투스 및 유선 명령 수신

블루투스(UART2) 및 유선(UART1) 명령을 통해 수동으로 블라인드를 제어할 수 있다.
블루투스 모듈(UART2)를 이용하여 스마트폰 앱을 통해 블라인드를 원격으로 제어할 수 있다.
명령어 종류:
‘u’ 명령: 블라인드를 올린다.
‘d’ 명령: 블라인드를 내린다.
‘s’: 블라인드를 멈춘다.

또한 유선 명령(UART1)을 통해 PC 에서 블라인드 제어 명령을 전송한다.
명령 큐를 사용하여 다수의 명령이 동시에 입력되더라도 안정적으로 명령들을 처리할 수 있게 설계했다. UART1 과 UART2 명령은 독립적으로 처리되며, 상호 충돌 없이 블라인드를 제어할 수 있다.

3.LCD 디스플레이를 통한 정보 제공

블라인드 상태(“blind up”, “blind down”, “stopped”)를 실시간으로 표시한다.
현재 조도 및 온도 데이터를 실시간으로 전달받아 LCD 디스플레이에 표시하여, 사용자가 환경 상태를 쉽게 확인할 수 있다.
또한 에러 메세지(“Full Queue”, “Already Up”, Already Down”)를 통해 시스템 상태를 시각적으로 전달한다.

4.모터 제어 및 안전 장치

블라인드의 상하 제어는 모터 2 개와 L298N 모터 드라이브를 사용하여 제어한다.
배터리 홀더를 통해 모터에 전원을 공급하여 모터가 작동하도록 했다.
블라인드가 이미 올라가거나 내려간 경우, 똑같은 동작을 하는 상황(이미 올라간 상태에서 블라인드를 또 다시 올리는 등)을 방지하기 위해서 추가 명령은 무시한다.
블라인드의 작동 시간은 설정된 딜레이(BLIND_UP_PERIOD, BLIND_DOWN_PERIOD)로 제어된다.

5.에러 처리 및 안정성

명령 큐가 가득 찬 경우(enqueueCommand 에서 처리), LCD 에 “Full Queue” 메세지가 출력된다.
또한 블라인드가 이미 최상단/최하단에 있는 경우 추가 동작을 제한하고 LCD 에 메세지를 출력한다.

3. 기존 계획과 달라진 개발내용

기존 계획은 L298N 모터 드라이버의 EN1, EN2 핀을 통한 PWM 제어로 모터의 속도를 제어하여 블라인드 동작을 자연스럽게 구현하는 것이었다. 이를 위해 PB0(EN1), PB1(EN2), PB2(IN1), PB3(IN2), PB4(IN3), PB5(IN4)를 사용했지만 작동하지 않아, PWM 제어 없이 PB1(IN1), PB2(IN2), PD1(IN3), PD2(IN4)항상 최대 속도로 회전하도록 구현했다.

이 과정에서 보드의 PB3, PB4, PB5 핀을 통한 모터 제어가 잘 작동하지 않음을 알 수 있었다. 다른 핀을 이용해 원래 기획대로 PWM 을 제어를 하려 했으나, 시현 일자까지 남은 시간을 고려했을 때, 어려울 것 같아 항상 최대 속도로 모터를 제어하게 되었다.

4. 시나리오

실제 시나리오는 자동 제어 시나리오, 수동 제어 시나리오, 에러 처리 시나리오로 나뉜다.

시나리오 1: 자동 제어 시나리오:

1. 조도 센서가 주변의 광량을 측정한다. 조도 값이 4000 lux 를 초과하면 온도 센서가 활성화된다.
2. 온도 센서가 활성화되어 환경 온도를 측정한다. 온도가 28 도 이상일 경우 블라인드가 자동으로 내려간다. 또한 LCD 에는 “Blind Down” 상태가 표시된다.

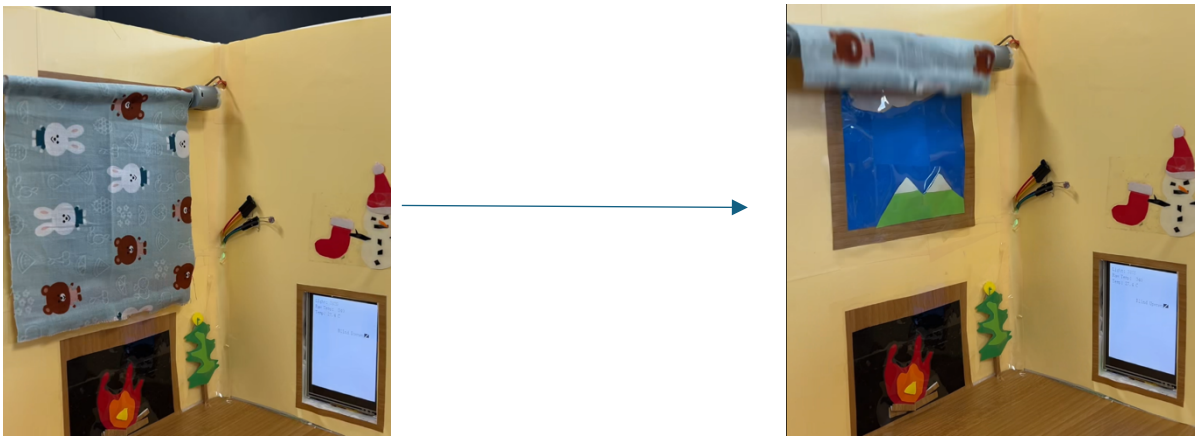
시나리오 2: 수동 제어 시나리오”

1. 사용자가 스마트폰 블루투스 앱을 통해 ‘u’ 명령을 전송한다. 블라인드가 상단으로 이동하고 LCD 에 “Blind Up” 상태가 표시된다.
2. ‘d’ 명령을 전송하면 블라인드가 하단으로 이동하고 LCD 에 “Blind Down” 상태가 표시된다.
3. ‘s’ 명령을 전송하면 블라인드 동작이 멈추고 LCD 에 “Stopped” 상태가 표시된다.
4. 블라인드가 이미 최상단/최하단에 있는 상태에서 추가 명령을 입력하면, LCD 에 “Already Up” 또는 “Already Down” 메세지가 표시되며 모터는 작동하지 않는다.

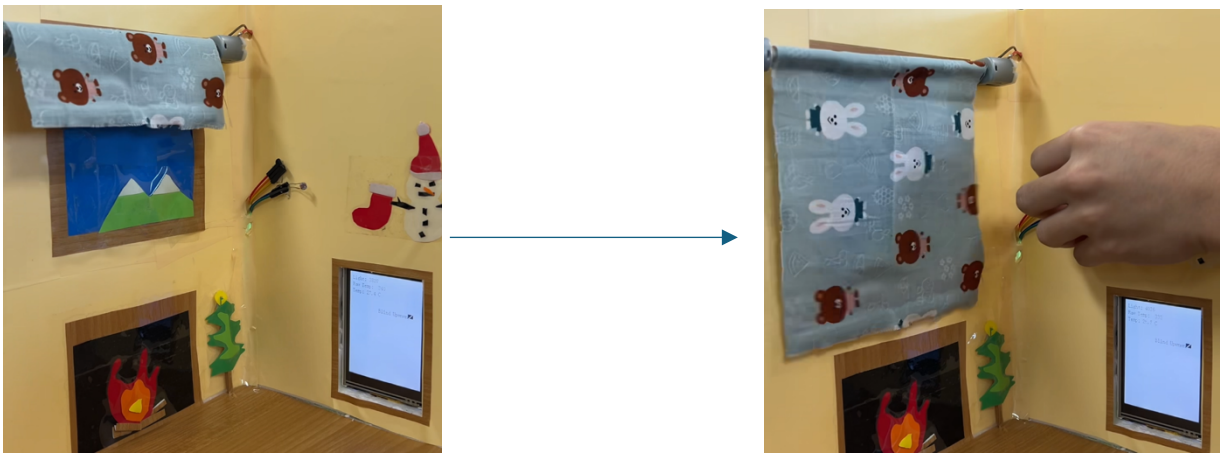
시나리오 3: 에러 처리 시나리오

1. 명령 큐가 가득 찬 경우: LCD 에 “Full Queue” 메세지가 표시된다. 추가 명령은 대기 상태로 전환된다.
2. UART 로 전송된 명령이 잘못된 경우: 명령은 무시되고 LCD 에 아무 메세지도 출력되지 않는다.

실제 동작 상태:



이 상태에서 'u' 명령 입력시, 블라인드가 올라간다.



말려 있는 상태에서 'd' 명령 입력시, 블라인드가 내려간다.

5.기술 구현 요약

기능	구현 방식
조도/온도 센서	ADC1 및 두 채널(조도, 온도) 기반 데이터 변환
블루투스 통신	UART2 인터럽트를 활용한 명령 수신 및 처리
유선 통신	UART1 인터럽트를 활용한 명령 수신 및 처리
LCD 상태 표시	LCD_ShowString 함수로 상태와 데이터를 실시간 표시
모터 제어	GPIO 기반 모터 방향 및 속도 제어
명령 큐	고정 크기의 큐(commandQueue)로 명령 처리 안정화
에러 처리	상태 메시지 및 LCD 출력으로 에러 상황 표시

6.코드 분석

```
void RCC_Configure(void);                    // 클럭 설정
void GPIO_Configure(void);                   // GPIO 설정
```

```
void USART1_Init(void);                    // UART1 초기화
void USART2_Init(void);                    // UART2 초기화
void NVIC_Configure(void);                 // 인터럽트 우선순위 설정
void ADC_Configure(void);                  // ADC 초기화 및 설정
```

우선 RCC 및 USART, GPIO, NVIC, ADC 의 설정을 configure 해주기 위한 함수를 선언한다. 이 함수들은 뒤에서 정의될 것이다.

```
void Motor_Control(char command);          // 모터 제어
void Blind_Up(void);                       // 블라인드 올리기
void Blind_Down(void);                     // 블라인드 내리기
void Blind_Stop(void);                     // 블라인드 정지
```

모터 제어 방식 및 모터 작동 방식을 구현한 함수들이다. 모터의 제어인 Motor_Control 의 경우에는 command 를 외부에서 받아 switch 를 통해 각 함수 Blind_Up, Blind_Down, Blind_Stop 을 실행한다.

```
void delay(uint32_t delayTime);            // 지연 함수
void sendDataUART1(uint16_t data);         // UART1 으로 데이터 전송
void sendDataUART2(uint16_t data);         // UART2 로 데이터 전송
```

delay 는 수업과 동일한 방식으로 for 문을 돌리는 방식으로 구현하였다. sendDataUART 함수는 UART1, 2 에 데이터를 전송하는 역할을 한다.

```
void delay(uint32_t delayTime) {
    for (volatile uint32_t i = 0; i < delayTime; i++);   // 단순 루프를 이용한 지연
}
```

delay 는 전술한 방식과 동일하게 delayTime 에 따라 for 문으로 지연 시간을 설정한다.

```
void enqueueCommand(char command) {
    int nextTail = (queueTail + 1) % COMMAND_QUEUE_SIZE; // 다음 테일 위치 계산
    if (nextTail == queueHead) {                         // 큐가 가득 찬 경우
        sprintf(stateString, "Full Queue");              // 상태 문자열 설정
        return;
```

```
    }
    commandQueue[queueTail] = command;                    // 큐에 명령 추가
    queueTail = nextTail;                                 // 테일 위치 갱신
}
```

```
char dequeueCommand(void) {
    if (queueHead == queueTail) {                         // 큐가 비어 있는 경우
        return 0;                                         // 0 반환
    }
    char command = commandQueue[queueHead];               // 헤드 위치의 명령 가져오기
    queueHead = (queueHead + 1) % COMMAND_QUEUE_SIZE;     // 헤드 위치 갱신
    return command;
}
```

enqueueCommand 는 Queue 를 이용해 command 를 저장한다. Queue 를 사용한 이유는 Command 가 연속적으로 들어올 경우 블루투스 간 UART 통신에서 Command 가 자주 무시되는 현상이 있었기 때문이다. 이는 UART 통신 과정에서 데이터 수신 속도와 처리속도가 불일치해 생기는 현상으로 추측된다. 이에 우리는 이것을 해결하기 위해 Command 를 Queueing 함으로서 Command 를 연속적으로 받아올 수 있었다. 만약 Queue 가 가득 찬 경우 stateString 을 “Full Queue”로 설정하여 TFT-LCD 에 출력해서 큐의 상태를 확인할 수 있도록 한다.

```
void RCC_Configure(void) {
    RCC_APB2PeriphClockCmd(RCC_APB2Periph_GPIOA, ENABLE); // GPIOA 클럭 활성화
    RCC_APB2PeriphClockCmd(RCC_APB2Periph_GPIOB, ENABLE); // GPIOB 클럭 활성화
    RCC_APB2PeriphClockCmd(RCC_APB2Periph_GPIOC, ENABLE); // GPIOC 클럭 활성화
    RCC_APB2PeriphClockCmd(RCC_APB2Periph_GPIOD, ENABLE); // GPIOD 클럭 활성화

    RCC_APB2PeriphClockCmd(RCC_APB2ENR_USART1EN, ENABLE); // USART1 클럭 활성화
    RCC_APB1PeriphClockCmd(RCC_APB1ENR_USART2EN, ENABLE); // USART2 클럭 활성화

    RCC_APB2PeriphClockCmd(RCC_APB2Periph_AFIO, ENABLE);  // AFIO 클럭 활성화
    RCC_APB2PeriphClockCmd(RCC_APB2Periph_ADC1, ENABLE);  // ADC1 클럭 활성화
}
```

RCC 에서는 우리가 사용할 포트인 A, B, C, D, 블루투스 통신을 위한 USART1, 2, 그리고 센서값의 ADC 변환을 위한 ADC1 에 클럭을 부여하였다. 포트 A 의 경우 UART1, 2 의 RX, TX 핀에 사용되었고 포트 B, D 는 모터를, 포트 C 는 센서를 설정하기 위해 사용되었다.

```
void GPIO_Configure(void) {
    GPIO_InitTypeDef GPIO_InitStructure;

    // UART1 TX(PA9), RX(PA10) 핀 설정
    GPIO_InitStructure.GPIO_Pin = GPIO_Pin_9;                // TX 핀
    GPIO_InitStructure.GPIO_Speed = GPIO_Speed_50MHz;    // 핀 속도 50MHz
    GPIO_InitStructure.GPIO_Mode = GPIO_Mode_AF_PP;       // AF 출력
    GPIO_Init(GPIOA, &GPIO_InitStructure);

    GPIO_InitStructure.GPIO_Pin = GPIO_Pin_10;               // RX 핀
    GPIO_InitStructure.GPIO_Mode = GPIO_Mode_IPU;          // 풀업 입력
    GPIO_Init(GPIOA, &GPIO_InitStructure);

    // UART2 TX(PA2), RX(PA3) 핀 설정
    GPIO_InitStructure.GPIO_Pin = GPIO_Pin_2;                // TX 핀
    GPIO_InitStructure.GPIO_Mode = GPIO_Mode_AF_PP;       // AF 출력
    GPIO_Init(GPIOA, &GPIO_InitStructure);

    GPIO_InitStructure.GPIO_Pin = GPIO_Pin_3;                // RX 핀
    GPIO_InitStructure.GPIO_Mode = GPIO_Mode_IPU;          // 풀업 입력
    GPIO_Init(GPIOA, &GPIO_InitStructure);

    // 모터 1(PB1, PB2) 핀 설정
    GPIO_InitStructure.GPIO_Pin = GPIO_Pin_1 | GPIO_Pin_2;
    GPIO_InitStructure.GPIO_Mode = GPIO_Mode_Out_PP;      // 출력
    GPIO_InitStructure.GPIO_Speed = GPIO_Speed_50MHz;    // 핀 속도 50MHz
    GPIO_Init(GPIOB, &GPIO_InitStructure);

    // 모터 2(PD1, PD2) 핀 설정
    GPIO_InitStructure.GPIO_Pin = GPIO_Pin_1 | GPIO_Pin_2;
    GPIO_InitStructure.GPIO_Mode = GPIO_Mode_Out_PP;      // 출력
    GPIO_Init(GPIOD, &GPIO_InitStructure);

    // 조도 센서(PC2), 온도 센서(PC1) 핀 설정
    GPIO_InitStructure.GPIO_Pin = GPIO_Pin_2 | GPIO_Pin_1;
```

```
    GPIO_InitStructure.GPIO_Mode = GPIO_Mode_AIN;          // 아날로그 입력
    GPIO_Init(GPIOC, &GPIO_InitStructure);
}
```

전술했듯이 각 핀마다 필요한 설정을 부여해서 Initializing 한다. 보드에서 신호를 보내줘야 하는 TX, 모터의 경우에는 Output 을, 그 외의 RX 나 센서의 경우 Input 을 설정해 주었다.

```
void ADC_Configure(void) {
    ADC_InitTypeDef ADC_InitStructure;

    ADC_DeInit(ADC1); // ADC 초기화
    ADC_InitStructure.ADC_Mode = ADC_Mode_Independent;  // 독립 모드 설정
    ADC_InitStructure.ADC_ScanConvMode = DISABLE;          // 단일 채널 모드
    ADC_InitStructure.ADC_ContinuousConvMode = DISABLE; // 소프트웨어 트리거 모드
    ADC_InitStructure.ADC_ExternalTrigConv = ADC_ExternalTrigConv_None;
    ADC_InitStructure.ADC_DataAlign = ADC_DataAlign_Right;
    ADC_InitStructure.ADC_NbrOfChannel = 1;                  // 단일 채널
    ADC_Init(ADC1, &ADC_InitStructure);

    ADC_RegularChannelConfig(ADC1, ADC_Channel_12, 1, ADC_SampleTime_239Cycles5);
// 초기 채널: 조도 센서
    ADC_ITConfig(ADC1, ADC_IT_EOC, ENABLE);                  // ADC 변환 완료
인터럽트 활성화
    ADC_Cmd(ADC1, ENABLE);                                      // ADC 활성화

    // ADC 보정
    ADC_ResetCalibration(ADC1);                          // 보정 초기화
    while (ADC_GetResetCalibrationStatus(ADC1)); // 보정 초기화 완료 대기
    ADC_StartCalibration(ADC1);                          // 보정 시작
    while (ADC_GetCalibrationStatus(ADC1));     // 보정 완료 대기
}
```

ADC 는 조도 센서 및 온도센서를 사용하기 위해 설정된다. 조도 센서는 온도센서와 함께 사용되지 않고 후술할 조도 센서의 특정 값을 트리거로 온도 센서가 활성화되도록 설계하였으므로 Initializing 과정에서 온도 센서는 사용되지 않는다. ADC 를 하나만 사용하게 되므로 ADC 는 독립 모드로 설정되며 마찬가지로 하나의 채널만 사용하므로 ScanConv 또한 Disable 되고 채널 또한 단일 채널로

설정된다. 센서는 외부 트리거 없이 소프트웨어만으로 조정할 것이므로 ExternalTrigConv 를 비활성화 한다. ADC1 에서 채널 12 를 선택하고 SamplingTime 을 239.5 사이클로 설정해준 뒤 USART 와 ADC 간의 인터럽트를 사용하기 위해 IT 를 활성화해준다. 마지막으로 ADC 를 보정하기 위해 보정을 초기화 한 뒤 보정을 수행한다. 각 과정이 완료될 때까지 대기를 While 문으로 수행한다.

```
void NVIC_Configure(void) {
	NVIC_InitTypeDef NVIC_InitStructure;

	NVIC_PriorityGroupConfig(NVIC_PriorityGroup_2);		// 우선순위 그룹 설정

	// ADC 인터럽트: 낮은 우선순위
	NVIC_InitStructure.NVIC_IRQChannel = ADC1_2_IRQn;
	NVIC_InitStructure.NVIC_IRQChannelPreemptionPriority = 2; // 낮은 우선순위
	NVIC_InitStructure.NVIC_IRQChannelSubPriority = 0;
	NVIC_InitStructure.NVIC_IRQChannelCmd = ENABLE;
	NVIC_Init(&NVIC_InitStructure);

	// USART1 인터럽트: 높은 우선순위
	NVIC_InitStructure.NVIC_IRQChannel = USART1_IRQn;
	NVIC_InitStructure.NVIC_IRQChannelPreemptionPriority = 0; // 높은 우선순위
	NVIC_InitStructure.NVIC_IRQChannelSubPriority = 1;
	NVIC_Init(&NVIC_InitStructure);

	// USART2 인터럽트: 높은 우선순위
	NVIC_InitStructure.NVIC_IRQChannel = USART2_IRQn;
	NVIC_InitStructure.NVIC_IRQChannelPreemptionPriority = 0; // 높은 우선순위
	NVIC_InitStructure.NVIC_IRQChannelSubPriority = 2;
	NVIC_Init(&NVIC_InitStructure);
}
```

NVIC 는 각 인터럽트간 우선순위를 부여해준다. USART 는 ADC 보다 실시간으로 통신 되어야 하므로 USART 는 항상 우선순위로 설정해준다.

```
void USART1_Init(void)
{
	USART_InitTypeDef USART1_InitStructure;

	// Enable the USART1 peripheral
	USART_Cmd(USART1, ENABLE);

	USART1_InitStructure.USART_BaudRate = 9600;
```

```
	USART1_InitStructure.USART_WordLength = (uint16_t) USART_WordLength_8b;
	USART1_InitStructure.USART_Mode = USART_Mode_Rx | USART_Mode_Tx;
	USART1_InitStructure.USART_Parity = (uint16_t) USART_Parity_No;
	USART1_InitStructure.USART_StopBits = (uint16_t) USART_StopBits_1;
	USART1_InitStructure.USART_HardwareFlowControl =
USART_HardwareFlowControl_None;

	USART_Init(USART1, &USART1_InitStructure);
	USART_ITConfig(USART1, USART_IT_RXNE, ENABLE);
}
```

```
void USART2_Init(void) {
	USART_InitTypeDef USART2_InitStructure;

	USART_Cmd(USART2, ENABLE);

	// USART2 초기화
	USART2_InitStructure.USART_BaudRate = 9600;
	USART2_InitStructure.USART_WordLength = (uint16_t)USART_WordLength_8b;
	USART2_InitStructure.USART_Mode = USART_Mode_Rx | USART_Mode_Tx;
	USART2_InitStructure.USART_Parity = (uint16_t)USART_Parity_No;
	USART2_InitStructure.USART_StopBits = (uint16_t)USART_StopBits_1;
	USART2_InitStructure.USART_HardwareFlowControl =
USART_HardwareFlowControl_None;

	USART_Init(USART2, &USART2_InitStructure);
	USART_ITConfig(USART2, USART_IT_RXNE, ENABLE); // RX 인터럽트 활성화
}
```

USART1 은 9600 의 baud rate, 8 비트의 워드길이를 가지며 패리티 비트를 가지지 않고, StopBit 는 1 로 설정되었다. RTS, CTS 를 사용하지 않고 소프트웨어로 Flow Control 을 할 수 있으므로 HardwareFlowControl 은 사용하지 않았다. 마지막에 RXNE 로 인터럽트를 호출할 수 있도록 설정하였다(IRQHandler 를 구현).
USART2 는 USART1 과 완전히 동일한 방식으로 설정되었다. USART1 은 PC 혹은 스마트폰에서 명령을 받아 PC → STM32 → 블루투스 모듈로 USART2 로 전달, USART2 는 블루투스 모듈에서 명령을 받아 블루투스 모듈 → STM32 → PC 로 USART1 로 전달한다.

```
void USART1_IRQHandler() {

	if(USART_GetITStatus(USART1,USART_IT_RXNE)!=RESET){
		flagUART1 = 1;
		// the most recent received data by the USART1 peripheral
		commandFromUART1 = USART_ReceiveData(USART1);
		// 큐에 명령 저장
```

```
            enqueueCommand(commandFromUART1);
            // clear 'Read data register not empty' flag
            USART_ClearITPendingBit(USART1,USART_IT_RXNE);
      }
}
```

```
void USART2_IRQHandler(void) {
      if (USART_GetITStatus(USART2, USART_IT_RXNE) != RESET) {
            flagUART2 = 1; // 플래그 설정
            commandFromUART2 = USART_ReceiveData(USART2); // 명령 수신
            enqueueCommand(commandFromUART2); // 큐에 명령 저장
            USART_ClearITPendingBit(USART2, USART_IT_RXNE); // 인터럽트 플래그 클리어
      }
}
```

각 IRQHandler 는 플래그를 설정하게 함으로써 후술할 방식으로 TX 와 RX 간의 데이터를 순차적으로 주고받을 수 있다. 그렇게 수신 받은 명령을 전술한 Queue 방식으로 받아 Command 를 수행한다. 수신이 완료된 뒤 인터럽트 플래그를 클리어해 인터럽트를 종료한다.

```
void ADC1_2_IRQHandler(void) {
      if (ADC_GetITStatus(ADC1, ADC_IT_EOC) != RESET) {
            uint32_t adcValue = ADC_GetConversionValue(ADC1);
```

ADC_GetITStatus 를 이용해서 ADC 의 변환 완료 시 발생하는 EOC 인터럽트를 확인한다. 변환이 완료된 상태이면 ADC1 으로부터 디지털 값으로 변환되는 센서의 아날로그 값을 GetConversionValue() 함수를 통해 추출한다.

```
if (!tempSensorActive) {
                  // 조도 센서 값 읽기
                  ADC_Values[0] = adcValue;

                  if (ADC_Values[0] > LIGHT_THRESHOLD) {
                        // 조도 값이 임계값 이상이면 온도 센서 활성화
                        tempSensorActive = true;
                        ADC_RegularChannelConfig(ADC1, ADC_Channel_11, 1,
ADC_SampleTime_239Cycles5);
```

온도센서가 작동 중이지 않을 시(초기 상태, 조도 센서가 threshold 를 넘지 않은 경우) 전역 함수인 ADC_Values 에 해당 값을 저장한다. 만약 ADC1 조도센서의 값이

설정한 Threshold(4000)를 넘을 시 온도 센서의 작동을 활성화한다. 이를 위해 온도센서의 ADC 변환 채널인 채널 11 로 전환시켜 준다(온도센서 작동).

```
// 채널 전환 후 첫 번째 데이터 무시
                    ADC_SoftwareStartConvCmd(ADC1, ENABLE);
                    while (!ADC_GetFlagStatus(ADC1, ADC_FLAG_EOC)); // 변환 완료 대기
                    (void)ADC_GetConversionValue(ADC1); // 첫 번째 값 버림

                    // 채널 설정 후 보정
                    ADC_ResetCalibration(ADC1);
                    while (ADC_GetResetCalibrationStatus(ADC1));
                    ADC_StartCalibration(ADC1);
                    while (ADC_GetCalibrationStatus(ADC1));
                }
            } else {
                // 온도 센서 값 읽기
                ADC_Values[1] = adcValue;

                // 온도 계산
                float temperatureCelsius = (ADC_Values[1] * 3.3 / 4096) * 100;
```

채널 전환 후 채널 변환이 완료된 뒤 처음 받은 값이 부정확하기 때문에 해당 값을 버린다. Configure 과정과 마찬가지로 해당 과정 후에도 calibration 을 수행한다. 다시 돌아와 만약 온도센서를 작동해야 할 경우(조도 센서의 값이 threshold 를 넘긴 후) 온도센서의 센서 값을 읽어와 변환과정을 거쳐 온도를 측정한다.

```
if (temperatureCelsius >= TEMP_THRESHOLD) {
                    // 온도가 임계값 이상이면 블라인드 내리기
                    Blind_Down();
                }

                // 조도 센서로 전환
                tempSensorActive = false;
                ADC_RegularChannelConfig(ADC1, ADC_Channel_12, 1,
ADC_SampleTime_239Cycles5);

                // 채널 전환 후 첫 번째 데이터 무시
                ADC_SoftwareStartConvCmd(ADC1, ENABLE);
                while (!ADC_GetFlagStatus(ADC1, ADC_FLAG_EOC)); // 변환 완료 대기
                (void)ADC_GetConversionValue(ADC1); // 첫 번째 값 버림
```

만약 온도 또한 지정한 Threshold 를 넘을 경우 설계한 대로 Blind 를 내려준다. 이 과정이 완료될 경우 조도 센서로 ADC 를 다시 사용하기 위해 Temp 를 비활성화 후 채널을 조도 센서로 변환해준다.

```
            // 채널 설정 후 보정
            ADC_ResetCalibration(ADC1);
            while (ADC_GetResetCalibrationStatus(ADC1));
            ADC_StartCalibration(ADC1);
            while (ADC_GetCalibrationStatus(ADC1));
        }

        // 다음 변환 시작
        ADC_SoftwareStartConvCmd(ADC1, ENABLE);
        // 인터럽트 플래그 클리어
        ADC_ClearITPendingBit(ADC1, ADC_IT_EOC);
    }
}
```

그 후 온도 센서와 동일하게 calibration 을 진행 후, 해당 과정들이 완료되면 계속해서 ADC 변환을 수행하기 위해 SoftwareStartConv() 함수로 소프트웨어 트리거를 활성화한다. 마지막으로 인터럽트를 클리어해 인터럽트를 종료한다.

```
void Motor_Control(char command) {
    // 모터 1 제어: PB1(IN1), PB2(IN2)
    // 모터 2 제어: PD1(IN3), PD2(IN4)
    switch (command) {
        case 'u': // 블라인드 올리기
            Blind_Up();
            break;
        case 'd': // 블라인드 내리기
            Blind_Down();
            break;
        case 's': // 블라인드 동작 정지
            Blind_Stop();
            break;
        default: // 잘못된 명령
            break;
    }
}
```

USART 로 받은 Command 를 사용해 모터를 조정하는 함수이다. 'u'의 경우에는 블라인드를 위로, 'd'의 경우에는 아래로, 's'는 모터를 모두 정지한다.

```
void Blind_Up(void) {
    if (blindPosition == BLIND_POSITION_TOP) {
        sprintf(stateString, "Already Up");
        return; // 블라인드가 이미 최상단에 있으면 동작하지 않음
    }

    blindPosition = BLIND_POSITION_MOVING; // 이동 중 상태로 설정

    // 모터 1 정방향
    GPIO_ResetBits(GPIOB, GPIO_Pin_1);
    GPIO_SetBits(GPIOB, GPIO_Pin_2);

    // 모터 2 역방향
    GPIO_ResetBits(GPIOD, GPIO_Pin_2);
    GPIO_SetBits(GPIOD, GPIO_Pin_1);
    sprintf(stateString, "Blind Up");

    delay(BLIND_UP_PERIOD);

    // 블라인드 다 올리면 멈추기
    Blind_Stop();

    blindPosition = BLIND_POSITION_TOP; // 최상단 상태로 설정
}
```

Blind 의 경우 Blind 의 현재 상태를 명시한 enum 변수를 통해 FSM 구조로 제어한다. 현재 블라인드의 상태가 더 이상 블라인드를 올릴 수 없는 '*BLIND_POSITION_TOP*'의 경우 TFT 에 보낼 문자열에 해당 상태를 저장한 후 return 한다. 그렇지 않다면 blind_stop 을 사용할 경우를 생각해 "*BLIND_POSITION_MOVING*"로 설정한다. 모터 1 은 정방향, 모터 2 는 역방향으로 회전하도록 모터드라이버의 IN1,2,3,4 핀 비트를 설정해준다. (각 모터는 서로를 마주보게 설계되어 있으므로 서로 역방향으로 돌아야 한다) 블라인드가 모두 올라가는데 걸리는 적절한 시간만큼의 delay 를 시행착오를 통해 알아냈다. 해당 delay 만큼 동안만 블라인드를 올리고, 그 후에는 모터를 정지시켜 블라인드를 올리는 동작을 수행한다. PWM 을 가용해 모터의 출력을 조정해 사용할 예정이었지만 알 수 없는 이유로 동작을 하지 않아, 해당 과정을 제쳐두고 모터 동작을 구현하였다. 마무리 과정에서 PWM 사용에 문제가 있었던 이유가 핀 및

포트의 문제였음을 알 수 있었지만, 시간 부족으로 인해 구현하지 못했다.

```
void Blind_Down(void) {
	if (blindPosition == BLIND_POSITION_BOTTOM) {
		sprintf(stateString, "Already Down");
		return; // 블라인드가 이미 최하단에 있으면 동작하지 않음
	}

	blindPosition = BLIND_POSITION_MOVING; // 이동 중 상태로 설정

	// 모터 1 역방향
	GPIO_ResetBits(GPIOB, GPIO_Pin_2);
	GPIO_SetBits(GPIOB, GPIO_Pin_1);

	// 모터 2 정방향
	GPIO_ResetBits(GPIOD, GPIO_Pin_1);
	GPIO_SetBits(GPIOD, GPIO_Pin_2);
	sprintf(stateString, "Blind Down");

	delay(BLIND_DOWN_PERIOD);

	// 블라인드 다 내리면 멈추기
	Blind_Stop();

	blindPosition = BLIND_POSITION_BOTTOM; // 최하단 상태로 설정
}
```

```
void Blind_Stop(void) {
	GPIO_SetBits(GPIOB, GPIO_Pin_1 | GPIO_Pin_2);
	GPIO_SetBits(GPIOD, GPIO_Pin_1 | GPIO_Pin_2);

	blindPosition = BLIND_POSITION_MOVING; // 이동 중 상태에서 정지로 변경
}
```

위 Blind_Up 과 마찬가지로 블라인드의 상태에 따라 동작을 결정한 뒤 블라인드를 움직여준다.
Blind_Stop 은 모든 핀의 bit 를 Set(모터 드라이브에서는 브레이크 모드)해줌으로서 블라인드의 이동을 정지한다.

```
void showLightAndTempValue(void) {
	char buffer[32];

	// 조도 센서 값 출력
```

```
    sprintf(buffer, "Light: %4lu", ADC_Values[0]);
    LCD_ShowString(10, 10, buffer, BLACK, WHITE);

    // 온도 센서 값 출력
    sprintf(buffer, "Raw Temp: %4lu", ADC_Values[1]);
    LCD_ShowString(10, 30, buffer, BLACK, WHITE);

    // 변환된 온도 출력
    float temperatureCelsius = (ADC_Values[1] * 3.3 / 4096) * 100;
    sprintf(buffer, "Temp: %2.1f C", temperatureCelsius);
    LCD_ShowString(10, 50, buffer, BLACK, WHITE);
}
```

showLightAndTempValue 는 TFT 에 현재 Light 와 Temperature 의 값을 출력해주는 함수이다. ADC_Values 는 전역 변수로 선언되어 있으며 각 Light 및 Temperature 의 값을 저장한다. 온도를 변환해준 후 buffer 을 이용해 숫자 데이터들을 문자열로 변형 후 수업 중 배운 방식으로 LCD 에 해당 값들을 출력해주었다.

```
RCC_Configure();            // 클럭 설정
GPIO_Configure();           // GPIO 핀 설정
ADC_Configure();            // ADC 설정

USART1_Init();              // USART1 초기화 (PC 와의 통신)
USART2_Init();              // USART2 초기화 (블루투스 통신)
NVIC_Configure();           // NVIC 인터럽트 설정
LCD_Init();                 // LCD 초기화
LCD_Clear(WHITE);           // LCD 화면 초기화 (배경 색상: 흰색)
```

우선 전술한 모든 Configure 과정을 수행해준다. TFT 의 Calibration 은 생략하였다.

```
    ADC_SoftwareStartConvCmd(ADC1, ENABLE); // 최초 소프트웨어 트리거
```

ADC1 의 변환을 시작하기 위해 소프트웨어 트리거를 실행한다. 현재 설정된 모드에서는 연속 변환이 비활성화되어 있으므로, 변환을 계속하려면 후술한 방식으로 소프트웨어 명령을 반복적으로 호출해야 한다.

```
while (1) {
        // USART1 명령 처리
        if (flagUART1 == 1) {
            // USART1 에서 명령 수신 시 처리
            char command;

            // 명령 큐에서 모든 명령 처리
            while ((command = dequeueCommand()) != 0) {
                sendDataUART2(command); // 수신한 명령을 USART2 로 전달
(블루투스 장치로 전송)
            }

            // 명령 처리가 끝나면 플래그 리셋
            flagUART1 = 0;
        }
```

USART1, 2 의 명령을 처리하는 과정이다. 우선 PC 나 스마트폰으로 명령을 받을 경우 IRQ 핸들러에서 flag 를 1 로 변환한다. Command 를 dequeueCommand() 함수로 받아 모든 Command 를 USART2 로 전달한다. 해당 과정이 끝나면 USART 의 flag 를 0 으로 초기화한다.

```
// USART2 명령 처리
        if (flagUART2 == 1) {
            // USART2 에서 명령 수신 시 처리
            char command;

            // 명령 큐에서 모든 명령 처리
            while ((command = dequeueCommand()) != 0) {
                sendDataUART1(command);   // 수신한 명령을 USART1 으로 전달
(PC 로 전송)
                Motor_Control(command); // 명령에 따라 모터 제어 (블라인드
조작)
                LCD_ShowString(100, 100, stateString, BLACK, WHITE); // LCD 에 현재
상태 표시
            }

            // 명령 처리가 끝나면 플래그 리셋
            flagUART2 = 0;
        }
```

블루투스 모듈에서 Command 를 수신해 flagUART2 가 1 이 될 경우 해당 Command 를 사용해 Motor 를 제어한다. 전술한 stateString(블라인드의 현재 상태)를 LCD 에 출력하고 마찬가지로 flag 를 초기화해준다. 해당 초기화를 통해 데이터 처리 완료를 알린다.

```
            // ADC 센서 값 표시 (조도 값 및 온도 값)
            showLightAndTempValue();
        }
}
```

마지막으로 TFT 에 각 센서의 정보들을 출력하는 함수인 showLightAndTempValue()까지 전 과정을 무한 루프로 반복함으로써 연속적으로 명령의 처리 및 TFT 의 출력을 처리해 줄 수 있다.