16. 부록

HAL 주요 헤더 분석 (`stm32f1xx_hal_conf.h`, `stm32f1xx_hal_def.h`)

STM32 HAL 라이브러리는 모든 주변장치 드라이버의 공통 설정과 내부 타입 정의를 포함하는 핵심 헤더 파일을 통해 구성된다.
특히 `stm32f1xx_hal_conf.h` 와 `stm32f1xx_hal_def.h` 는 HAL의 **기반 환경 설정**과 **핵심 타입 정의**를 담당하며, 모든 HAL 소스 파일이 이 두 헤더를 참조한다.

---

1. `stm32f1xx_hal_conf.h` — HAL 설정 구성 파일

이 파일은 **HAL 모듈의 활성화 여부**, **시스템 클럭 소스**, **Assert 옵션**, **외부 크리스털 설정** 등을 정의하는 "사용자 구성용 헤더"다.
CubeMX로 프로젝트를 생성하면 자동 생성되며, `Core/Inc/` 디렉터리에 위치한다.

주요 구성 항목

```
/* ########################## Module Selection ############################## */
#define HAL_MODULE_ENABLED
#define HAL_ADC_MODULE_ENABLED
#define HAL_GPIO_MODULE_ENABLED
#define HAL_I2C_MODULE_ENABLED
#define HAL_RTC_MODULE_ENABLED
#define HAL_UART_MODULE_ENABLED
#define HAL_TIM_MODULE_ENABLED
#define HAL_FLASH_MODULE_ENABLED
#define HAL_PWR_MODULE_ENABLED
```

> 사용할 HAL 모듈을 Enable/Disable — 미사용 드라이버를 비활성화하면 코드 크기 절감 가능

클럭 소스 및 설정

```
/* ########################### HSE/HSI Values adaptation ##################### */
#define HSE_VALUE    ((uint32_t)8000000U) /*!< External oscillator frequency */
#define HSI_VALUE    ((uint32_t)8000000U) /*!< Internal RC oscillator frequency */
#define LSE_VALUE    ((uint32_t)32768U)   /*!< External Low Speed oscillator (LSE) */
#define LSI_VALUE    ((uint32_t)40000U)   /*!< Internal Low Speed oscillator (LSI) */
```

> 시스템 클럭 및 RTC 기준 주파수를 지정 (LSE, LSI 등 실제 회로에 맞게 수정해야 함)

Assert 설정

```
/* ########################## Assert Selection ############################## */
#define USE_FULL_ASSERT 1U
```

> `assert_param()` 매크로를 활성화하면, 매개변수 검증 실패 시 `_Error_Handler()`로 이동
> 개발 중에는 1로 유지, 릴리스 빌드에서는 0으로 설정하여 코드 크기 절감

HAL 우선순위 설정

```
/* ################### System Configuration ######################### */
#define  VDD_VALUE                    ((uint32_t)3300U) /*!< VDD in mV */
#define  TICK_INT_PRIORITY            ((uint32_t)0U)    /*!< SysTick interrupt priority
*/
#define  USE_RTOS                     1U
#define  PREFETCH_ENABLE              1U
```

> 전원전압, SysTick 우선순위, RTOS 사용 여부 등을 지정
> FreeRTOS 사용 시 `USE_RTOS = 1` 로 설정 필요

외부 드라이버 헤더 참조

```
#ifdef HAL_RCC_MODULE_ENABLED
  #include "stm32f1xx_hal_rcc.h"
#endif
#ifdef HAL_GPIO_MODULE_ENABLED
  #include "stm32f1xx_hal_gpio.h"
#endif
```

> 각 모듈의 헤더를 조건부로 포함 — 활성화된 HAL만 컴파일됨

---

2. `stm32f1xx_hal_def.h` — HAL 기본 타입 및 구조체 정의

이 파일은 HAL의 **데이터형**, **상태 코드**, **Lock 관리 매크로**, **유틸리티 매크로**를 정의한다.
`stm32f1xx_hal.h` → `stm32f1xx_hal_conf.h` → `stm32f1xx_hal_def.h` 순으로 포함된다.

HAL 상태 코드 정의

```
typedef enum
{
  HAL_OK       = 0x00U,
  HAL_ERROR    = 0x01U,
  HAL_BUSY     = 0x02U,
  HAL_TIMEOUT  = 0x03U
} HAL_StatusTypeDef;
```

> HAL API 호출 시 항상 반환되는 기본 상태 코드
> `HAL_TIMEOUT` 은 통신 오류나 클럭 멈춤 시 자주 발생

HAL Lock 구조체 및 매크로

```
typedef enum
{
  HAL_UNLOCKED = 0x00U,
  HAL_LOCKED   = 0x01U
} HAL_LockTypeDef;

#define __HAL_LOCK(__HANDLE__)   do{ if((__HANDLE__)->Lock == HAL_LOCKED) return HAL_BUSY; \
                                     else (__HANDLE__)->Lock = HAL_LOCKED; }while (0)

#define __HAL_UNLOCK(__HANDLE__) ((__HANDLE__)->Lock = HAL_UNLOCKED)
```

> 주변장치 접근의 경쟁 조건을 방지하기 위한 **락 매커니즘**
> HAL 내부에서 자원(예: I²C 버스) 점유 시 자동으로 사용됨

HAL 매크로 및 유틸리티

```
#define UNUSED(x) ((void)(x))
#define HAL_MAX_DELAY 0xFFFFFFFFU
#define HAL_IS_BIT_SET(REG, BIT)  (((REG) & (BIT)) != RESET)
#define HAL_IS_BIT_CLR(REG, BIT)  (((REG) & (BIT)) == RESET)
```

> 미사용 변수 경고 제거, 비트 검사 매크로 등 공통 유틸리티
> `HAL_MAX_DELAY`는 Timeout 파라미터 기본값으로 사용됨

공용 핸들 구조 예시

모든 HAL 모듈은 다음과 같은 구조체를 기반으로 한다:

```
typedef struct
{
  I2C_TypeDef             *Instance;  /*!< I2C 레지스터 베이스 주소 */
  HAL_LockTypeDef         Lock;       /*!< Lock 상태 */
  __IO HAL_I2C_StateTypeDef State;    /*!< 통신 상태 */
  uint32_t                ErrorCode;  /*!< 에러 코드 */
} I2C_HandleTypeDef;
```

> HAL 함수가 받는 `&hi2c1`, `&htim2` 등은 모두 이러한 핸들 구조체

---

3. 포함 관계 요약

```
main.c
 └─> stm32f1xx_hal.h
      ├─ stm32f1xx_hal_conf.h   // 모듈 활성화 및 설정
      └─ stm32f1xx_hal_def.h    // 공용 타입 및 매크로
```

> `stm32f1xx_hal_conf.h` 는 사용자 수정 가능
> `stm32f1xx_hal_def.h` 는 절대 수정하지 않음 (HAL Core 내부 정의)

---

4. 실무 팁

- CubeMX 생성 후 **사용하지 않는 HAL 모듈 매크로를 주석 처리**하면 코드 크기 10~20% 절감
- `HAL_StatusTypeDef` 를 반환하는 함수는 항상 에러 처리 루틴 포함:

```
if (HAL_I2C_Master_Transmit(&hi2c1, addr, data, len, 100) != HAL_OK)
{
    Error_Handler();
}
```

- 커스텀 드라이버 작성 시 HAL Lock 매크로 활용으로 다중 Task 접근 방지 가능

요약:

`stm32f1xx_hal_conf.h` 는 HAL 기능을 "켜고 끄는 스위치",
`stm32f1xx_hal_def.h` 는 HAL의 "핵심 타입과 제어 매크로 집합"이다.
둘은 HAL 구조의 뼈대를 형성하며, 모든 HAL 기반 펌웨어의 공통 기반으로 작동한다.

CMSIS-Core 함수 정리

CMSIS (Cortex Microcontroller Software Interface Standard) 는 Arm Cortex-M 기반 MCU의 **레지스터 접근, 예외 처리, 인터럽트 관리, 코어 제어 함수** 등을 표준화한 핵심 인터페이스 집합이다.
STM32 HAL은 CMSIS-Core 위에서 동작하며, **레지스터 수준 접근과 코어 관리 기능을 HAL에 추상화하여 제공**한다.

---

1. 코어 제어 관련 함수

1.1 `__disable_irq()`, `__enable_irq()`

- 글로벌 인터럽트를 비활성화 또는 활성화한다.
- NVIC 레벨에서 모든 IRQ를 차단/허용한다.

```
__disable_irq();   // 모든 인터럽트 비활성화
__enable_irq();    // 인터럽트 복구
```

> 임계 구역(Critical Section) 보호 시 주로 사용됨.

1.2 `__set_PRIMASK(uint32_t value)` / `__get_PRIMASK()`

- PRIMASK 레지스터는 인터럽트 전역 차단 상태를 제어한다.

```
uint32_t primask = __get_PRIMASK();
__set_PRIMASK(1);  // IRQ 차단
__set_PRIMASK(primask);  // 복구
```

> `__disable_irq()`와 동일한 기능을 직접 제어할 때 사용.

1.3 `__set_BASEPRI(uint32_t value)` / `__get_BASEPRI()`

- **인터럽트 우선순위 임계값**을 설정한다.
- BASEPRI보다 낮은 우선순위의 IRQ는 마스크됨.

```
__set_BASEPRI(0x40);  // 우선순위 0x40 이하 인터럽트 마스크
```

> FreeRTOS의 **임계 구역 보호 매크로(portENTER_CRITICAL)** 내부에서 사용됨.

1.4 `__set_FAULTMASK()` / `__get_FAULTMASK()`

- 모든 예외(Fault 포함)를 마스크할지 여부를 제어.
- **HardFault** 제외 모든 인터럽트를 차단할 수 있다.

```
__set_FAULTMASK(1);  // 모든 인터럽트/Fault 차단
__set_FAULTMASK(0);  // 복구
```

1.5 `__WFI()` / `__WFE()`

- **WFI (Wait For Interrupt)**: 인터럽트 발생 시까지 CPU 슬립 진입
- **WFE (Wait For Event)**: 이벤트 또는 인터럽트 발생 시까지 대기

```
__WFI();  // 저전력 모드 진입 (Sleep, Stop 등)
```

> FreeRTOS의 Tickless Idle 모드에서 사용됨.

---

2. NVIC (Nested Vectored Interrupt Controller) 관련 함수

2.1 `NVIC_EnableIRQ(IRQn_Type IRQn)`

- 특정 인터럽트를 NVIC에서 활성화.

```
NVIC_EnableIRQ(EXTI0_IRQn);
```

2.2 `NVIC_DisableIRQ(IRQn_Type IRQn)`

- 해당 인터럽트를 NVIC에서 비활성화.

2.3 `NVIC_SetPriority(IRQn_Type IRQn, uint32_t priority)`

- 인터럽트 우선순위 설정 (0이 가장 높음).

```
NVIC_SetPriority(TIM2_IRQn, 3);
```

2.4 `NVIC_GetPriority(IRQn_Type IRQn)`

- 인터럽트 우선순위를 조회.

2.5 `NVIC_SetPendingIRQ(IRQn_Type IRQn)` / `NVIC_ClearPendingIRQ()`

- 인터럽트를 소프트웨어적으로 트리거하거나 대기 상태를 해제.

2.6 `NVIC_SystemReset()`

- 소프트웨어적으로 MCU를 리셋한다.

```
NVIC_SystemReset();
```

---

3. SysTick 관련 함수

3.1 `SysTick_Config(uint32_t ticks)`

- SysTick 타이머를 초기화하고 주기를 설정한다.

```
SysTick_Config(SystemCoreClock / 1000);  // 1ms 주기 설정
```

3.2 `SysTick_Handler()`

- 주기적 인터럽트 루틴 (HAL과 FreeRTOS에서 Tick 증가에 사용).
- HAL에서는 `HAL_IncTick()` 호출, FreeRTOS에서는 `xPortSysTickHandler()`로 연결됨.

---

4. SCB (System Control Block) 관련 함수

4.1 `SCB->VTOR`

- **Vector Table Offset Register**
- 인터럽트 벡터 테이블의 시작 주소 지정.

```
SCB->VTOR = FLASH_BASE | 0x2000;  // 부트 섹션 변경
```

4.2 `SCB->ICSR`

- 인터럽트 상태 및 펜딩 관리.
  - `SCB_ICSR_PENDSVSET_Msk` : PendSV 트리거
  - `SCB_ICSR_PENDSTCLR_Msk` : SysTick 클리어

4.3 `SCB->SCR`

- 슬립 모드 제어 비트 포함.
  - `SLEEPONEXIT` : 인터럽트 종료 후 자동 슬립
  - `SLEEPDEEP` : Deep Sleep 모드 진입

```
SCB->SCR |= SCB_SCR_SLEEPDEEP_Msk;
__WFI();
```

---

5. 레지스터 접근 매크로

5.1 `__IO`

- `volatile` + I/O 접근 지정자
- HAL 구조체 멤버에서 하드웨어 레지스터 변경 감지용으로 사용.

5.2 `__STATIC_INLINE`

- 성능 향상을 위해 CMSIS 함수 대부분을 inline으로 정의.

5.3 `__ASM`, `__NOP()`

- 인라인 어셈블리 지원 매크로.

```
__NOP();  // No Operation, 파이프라인 안정화
```

---

6. 예외 처리 관련 함수

6.1 `__get_MSP()`, `__set_MSP()`

- Main Stack Pointer 접근.

6.2 `__get_PSP()`, `__set_PSP()`

- Process Stack Pointer 접근.

> RTOS의 Task Context 전환 시 PSP 사용.

6.3 `__get_CONTROL()`, `__set_CONTROL()`

- 현재 실행 모드(Thread/Handler), 스택 모드(MSP/PSP) 제어.

```
uint32_t control = __get_CONTROL();
__set_CONTROL(control | 0x02);  // PSP 사용 모드로 전환
```

---

7. 시스템 함수 요약표

구분	주요 함수	설명
인터럽트 제어	`__enable_irq()`, `__disable_irq()`	글로벌 IRQ 제어
SysTick	`SysTick_Config()`	시스템 Tick 설정
NVIC	`NVIC_EnableIRQ()`, `NVIC_SetPriority()`	인터럽트 관리
슬립 제어	`__WFI()`, `__WFE()`	저전력 대기 진입
스택 제어	`__get_PSP()`, `__set_PSP()`	RTOS 컨텍스트 관리
리셋 제어	`NVIC_SystemReset()`	MCU 소프트 리셋
Fault 제어	`__set_FAULTMASK()`	Fault 차단 제어
벡터 테이블	`SCB->VTOR`	인터럽트 벡터 재배치

요약:

CMSIS-Core는 **Cortex-M3의 코어 레벨 제어, 예외 처리, 인터럽트 관리, 전력 제어**를 위한 표준화된 인터페이스다.
HAL과 RTOS는 모두 이 CMSIS-Core API를 기반으로 동작하며, 이를 통해 **플랫폼 간 호환성**과 **레지스터 접근의 일관성**을 확보한다.

NVIC 벡터 테이블 전체 목록

STM32F103 MCU는 **Cortex-M3 코어** 기반으로, ARM의 기본 예외 벡터 구조 + STM32 시리즈 고유의 **외부 인터럽트 (Peripheral IRQ)** 벡터를 함께 포함한다.
아래는 **벡터 번호, 예외명, 인터럽트 핸들러명, 설명**을 정리한 전체 표이다.

---

1. Cortex-M3 시스템 예외 벡터 (Core Exceptions)

벡터 번호	예외 이름	핸들러 이름	설명
0	초기 스택 포인터	`_estack`	초기화 시 스택의 최상단 주소
1	Reset	`Reset_Handler`	MCU 리셋 시 진입 지점
2	NMI	`NMI_Handler`	Non-Maskable Interrupt

벡터 번호	예외 이름	핸들러 이름	설명
3	HardFault	`HardFault_Handler`	심각한 오류 (Fault Mask 무시)
4	MemManage	`MemManage_Handler`	메모리 접근 오류
5	BusFault	`BusFault_Handler`	버스 전송 오류
6	UsageFault	`UsageFault_Handler`	명령어 실행 오류
7~10	Reserved	—	예약됨
11	SVCall	`SVC_Handler`	시스템 서비스 호출 (Supervisor Call)
12	Debug Monitor	`DebugMon_Handler`	디버그 이벤트
13	Reserved	—	예약됨
14	PendSV	`PendSV_Handler`	컨텍스트 스위칭용 (RTOS 등)
15	SysTick	`SysTick_Handler`	시스템 주기 인터럽트

2. 외부 인터럽트 벡터 (STM32F1xx Peripheral Interrupts)

벡터 번호 (IRQn +16)	IRQn 이름	핸들러 이름	설명
16	`WWDG_IRQn`	`WWDG_IRQHandler`	윈도우 워치독 타이머
17	`PVD_IRQn`	`PVD_IRQHandler`	전압 감시 인터럽트 (PVD)
18	`TAMPER_IRQn`	`TAMPER_IRQHandler`	RTC Tamper 이벤트
19	`RTC_IRQn`	`RTC_IRQHandler`	RTC 알람 인터럽트
20	`FLASH_IRQn`	`FLASH_IRQHandler`	Flash 메모리 오류
21	`RCC_IRQn`	`RCC_IRQHandler`	클록/리셋 컨트롤러 상태 변경
22	`EXTI0_IRQn`	`EXTI0_IRQHandler`	외부 인터럽트 라인 0
23	`EXTI1_IRQn`	`EXTI1_IRQHandler`	외부 인터럽트 라인 1
24	`EXTI2_IRQn`	`EXTI2_IRQHandler`	외부 인터럽트 라인 2
25	`EXTI3_IRQn`	`EXTI3_IRQHandler`	외부 인터럽트 라인 3
26	`EXTI4_IRQn`	`EXTI4_IRQHandler`	외부 인터럽트 라인 4
27	`DMA1_Channel1_IRQn`	`DMA1_Channel1_IRQHandler`	DMA1 채널 1 전송 완료

벡터 번호 (IRQn +16)	IRQn 이름	핸들러 이름	설명
28	`DMA1_Channel2_IRQn`	`DMA1_Channel2_IRQHandler`	DMA1 채널 2 전송 완료
29	`DMA1_Channel3_IRQn`	`DMA1_Channel3_IRQHandler`	DMA1 채널 3 전송 완료
30	`DMA1_Channel4_IRQn`	`DMA1_Channel4_IRQHandler`	DMA1 채널 4 전송 완료
31	`DMA1_Channel5_IRQn`	`DMA1_Channel5_IRQHandler`	DMA1 채널 5 전송 완료
32	`DMA1_Channel6_IRQn`	`DMA1_Channel6_IRQHandler`	DMA1 채널 6 전송 완료
33	`DMA1_Channel7_IRQn`	`DMA1_Channel7_IRQHandler`	DMA1 채널 7 전송 완료
34	`ADC1_2_IRQn`	`ADC1_2_IRQHandler`	ADC1, ADC2 변환 완료
35	`USB_HP_CAN1_TX_IRQn`	`USB_HP_CAN1_TX_IRQHandler`	USB 고속 전송 or CAN1 TX
36	`USB_LP_CAN1_RX0_IRQn`	`USB_LP_CAN1_RX0_IRQHandler`	USB 저속 수신 or CAN1 RX0
37	`CAN1_RX1_IRQn`	`CAN1_RX1_IRQHandler`	CAN1 RX1
38	`CAN1_SCE_IRQn`	`CAN1_SCE_IRQHandler`	CAN1 상태 변경/에러
39	`EXTI9_5_IRQn`	`EXTI9_5_IRQHandler`	외부 인터럽트 라인 5~9
40	`TIM1_BRK_IRQn`	`TIM1_BRK_IRQHandler`	TIM1 Break 이벤트
41	`TIM1_UP_IRQn`	`TIM1_UP_IRQHandler`	TIM1 업데이트 이벤트
42	`TIM1_TRG_COM_IRQn`	`TIM1_TRG_COM_IRQHandler`	TIM1 트리거/커뮤테이션
43	`TIM1_CC_IRQn`	`TIM1_CC_IRQHandler`	TIM1 캡처/비교
44	`TIM2_IRQn`	`TIM2_IRQHandler`	타이머 2 인터럽트
45	`TIM3_IRQn`	`TIM3_IRQHandler`	타이머 3 인터럽트
46	`TIM4_IRQn`	`TIM4_IRQHandler`	타이머 4 인터럽트
47	`I2C1_EV_IRQn`	`I2C1_EV_IRQHandler`	I²C1 이벤트
48	`I2C1_ER_IRQn`	`I2C1_ER_IRQHandler`	I²C1 에러
49	`I2C2_EV_IRQn`	`I2C2_EV_IRQHandler`	I²C2 이벤트

벡터 번호 (IRQn +16)	IRQn 이름	핸들러 이름	설명
50	`I2C2_ER_IRQn`	`I2C2_ER_IRQHandler`	I²C2 에러
51	`SPI1_IRQn`	`SPI1_IRQHandler`	SPI1 전송 완료
52	`SPI2_IRQn`	`SPI2_IRQHandler`	SPI2 전송 완료
53	`USART1_IRQn`	`USART1_IRQHandler`	UART1 수신/전송 완료
54	`USART2_IRQn`	`USART2_IRQHandler`	UART2 수신/전송 완료
55	`USART3_IRQn`	`USART3_IRQHandler`	UART3 수신/전송 완료
56	`EXTI15_10_IRQn`	`EXTI15_10_IRQHandler`	외부 인터럽트 라인 10~15
57	`RTCAlarm_IRQn`	`RTCAlarm_IRQHandler`	RTC 알람 이벤트
58	`USBWakeUp_IRQn`	`USBWakeUp_IRQHandler`	USB 웨이크업 이벤트
59~67	Reserved	—	예약됨
68	`TIM8_BRK_IRQn`	`TIM8_BRK_IRQHandler`	고급 타이머 TIM8 Break
69	`TIM8_UP_IRQn`	`TIM8_UP_IRQHandler`	TIM8 업데이트
70	`TIM8_TRG_COM_IRQn`	`TIM8_TRG_COM_IRQHandler`	TIM8 트리거/커뮤테이션
71	`TIM8_CC_IRQn`	`TIM8_CC_IRQHandler`	TIM8 캡처/비교
72	`ADC3_IRQn`	`ADC3_IRQHandler`	ADC3 변환 완료
73	`FSMC_IRQn`	`FSMC_IRQHandler`	외부 메모리 컨트롤러
74	`SDIO_IRQn`	`SDIO_IRQHandler`	SD 카드 인터페이스
75	`TIM5_IRQn`	`TIM5_IRQHandler`	타이머 5 인터럽트
76	`SPI3_IRQn`	`SPI3_IRQHandler`	SPI3 전송 완료
77	`UART4_IRQn`	`UART4_IRQHandler`	UART4
78	`UART5_IRQn`	`UART5_IRQHandler`	UART5
79	`TIM6_IRQn`	`TIM6_IRQHandler`	타이머 6 (DAC Trigger)
80	`TIM7_IRQn`	`TIM7_IRQHandler`	타이머 7
81	`DMA2_Channel1_IRQn`	`DMA2_Channel1_IRQHandler`	DMA2 채널 1
82	`DMA2_Channel2_IRQn`	`DMA2_Channel2_IRQHandler`	DMA2 채널 2

벡터 번호 (IRQn +16)	IRQn 이름	핸들러 이름	설명
83	`DMA2_Channel3_IRQn`	`DMA2_Channel3_IRQHandler`	DMA2 채널 3
84	`DMA2_Channel4_5_IRQn`	`DMA2_Channel4_5_IRQHandler`	DMA2 채널 4/5

---

3. 예시: `startup_stm32f103xb.s` (벡터 테이블 구조)

```
.section .isr_vector, "a", %progbits
g_pfnVectors:
  .word _estack
  .word Reset_Handler
  .word NMI_Handler
  .word HardFault_Handler
  .word MemManage_Handler
  .word BusFault_Handler
  .word UsageFault_Handler
  .word 0, 0, 0, 0              // Reserved
  .word SVC_Handler
  .word DebugMon_Handler
  .word 0                       // Reserved
  .word PendSV_Handler
  .word SysTick_Handler
  .word WWDG_IRQHandler
  .word PVD_IRQHandler
  .word TAMPER_IRQHandler
  .word RTC_IRQHandler
  ...
  .word DMA2_Channel4_5_IRQHandler
```

---

4. NVIC 관련 참고

- **IRQn_Type** 열거형은 `stm32f1xx.h` 에 정의됨
- HAL에서는 `HAL_NVIC_EnableIRQ(IRQn)` / `HAL_NVIC_SetPriority()` 로 제어
- 사용자 정의 ISR은 반드시 **핸들러 이름을 동일하게 선언**해야 한다.

```
void EXTI0_IRQHandler(void)
{
    HAL_GPIO_EXTI_IRQHandler(GPIO_PIN_0);
}
```

✅ 요약

- 벡터 0~15: Cortex-M3 시스템 예외
- 벡터 16~84: STM32F1 주변장치 인터럽트
- 각 IRQ는 `startup_stm32f10x.s` 에서 **핸들러 주소 테이블로 연결**
- 실제 동작은 NVIC 레지스터(`ISER`, `ICER`, `IPR`, `ICPR`)에서 Enable/Disable 및 우선순위로 제어됨.

Register-Level 접근 실습

1. 개요

HAL 드라이버는 하드웨어 접근을 추상화하여 사용 편의성을 제공하지만,
실제 동작 원리를 이해하기 위해서는 **레지스터 직접 접근(Register-Level Programming)** 이 필수적이다.
STM32의 모든 주변장치는 **메모리 맵(Memory-Mapped I/O)** 구조로,
특정 주소에 대응하는 레지스터를 읽고/쓰는 방식으로 제어된다.

---

2. 메모리 맵 구조

영역	주소 범위	설명
Code (Flash)	0x0800_0000 ~	프로그램 코드 저장
SRAM	0x2000_0000 ~	실행 중 변수 저장
Peripheral	0x4000_0000 ~	주변장치 레지스터
Cortex-M3 SCS	0xE000_E000 ~	NVIC, SysTick 등 코어 제어

STM32F103에서 예를 들어 **GPIOA** 는 `0x40010800` 주소에 매핑되어 있으며,
`GPIO_TypeDef` 구조체를 통해 각 레지스터에 접근 가능하다.

---

3. GPIO 예제 (레지스터 직접 접근)

(1) 기본 구조체

```
typedef struct
{
  __IO uint32_t CRL;     // 0x00 : Port configuration low register
  __IO uint32_t CRH;     // 0x04 : Port configuration high register
  __IO uint32_t IDR;     // 0x08 : Input data register
  __IO uint32_t ODR;     // 0x0C : Output data register
  __IO uint32_t BSRR;    // 0x10 : Bit set/reset register
  __IO uint32_t BRR;     // 0x14 : Bit reset register
  __IO uint32_t LCKR;    // 0x18 : Configuration lock register
} GPIO_TypeDef;
```

(2) 포인터 정의

```
#define PERIPH_BASE      0x40000000UL
#define APB2PERIPH_BASE  (PERIPH_BASE + 0x10000UL)
#define GPIOA_BASE       (APB2PERIPH_BASE + 0x0800UL)
#define GPIOA            ((GPIO_TypeDef *) GPIOA_BASE)
```

---

4. GPIOA PA5 출력 제어 (LED 토글)

```
// RCC 클록 활성화
RCC->APB2ENR |= (1 << 2);  // IOPAEN = Bit2

// GPIOA PA5 출력 설정 (CNF5[1:0]=00, MODE5[1:0]=11 → 50MHz Push-Pull)
GPIOA->CRL &= ~(0xF << (5 * 4));   // 5번 핀 설정 클리어
GPIOA->CRL |=  (0x3 << (5 * 4));   // MODE5 = 11
GPIOA->CRL |=  (0x0 << (5 * 4 + 2)); // CNF5 = 00

// LED On
GPIOA->BSRR = (1 << 5);

// LED Off
GPIOA->BRR = (1 << 5);
```

---

5. SysTick 타이머 직접 설정

```
#define SYSTICK_BASE  (0xE000E010UL)
#define SysTick       ((SysTick_Type *) SYSTICK_BASE)

SysTick->LOAD = 72000 - 1;   // 1ms 주기 (72MHz 기준)
SysTick->VAL  = 0;
SysTick->CTRL = 0x07;        // Enable, TickInt, ClockSource=Processor Clock
```

---

6. I²C 예제 (레지스터 기반 전송)

```
// START 조건
I2C1->CR1 |= I2C_CR1_START;
while (!(I2C1->SR1 & I2C_SR1_SB));

// 슬레이브 주소 전송
I2C1->DR = (slave_addr << 1);
while (!(I2C1->SR1 & I2C_SR1_ADDR));
(void)I2C1->SR2;  // ADDR 비트 클리어용 dummy read

// 데이터 송신
I2C1->DR = data;
while (!(I2C1->SR1 & I2C_SR1_TXE));

```

```
// STOP 조건
I2C1->CR1 |= I2C_CR1_STOP;
```

---

7. NVIC 직접 제어

```
#define NVIC_ISER0   (*(volatile uint32_t *)0xE000E100)
#define NVIC_ICER0   (*(volatile uint32_t *)0xE000E180)
#define NVIC_IPR_BASE (0xE000E400)

NVIC_ISER0 = (1 << EXTI0_IRQn);     // EXTI0 인터럽트 활성화
*(volatile uint8_t *)(NVIC_IPR_BASE + EXTI0_IRQn) = 0x20;  // 우선순위 설정
```

---

8. RTC 레지스터 접근 (Backup Domain)

```
RCC->BDCR |= RCC_BDCR_LSEON;                // LSE Enable
while(!(RCC->BDCR & RCC_BDCR_LSERDY));      // 안정화 대기

RCC->BDCR |= RCC_BDCR_RTCSEL_LSE;           // RTC 클럭 소스 선택
RCC->BDCR |= RCC_BDCR_RTCEN;                // RTC Enable

RTC->PRLL = 0x7FFF;                         // 1초 주기 (32768Hz / 32768)
```

---

9. 정리

항목	HAL 접근 방식	Register 접근 방식
코드 가독성	높음	낮음
실행 효율	중간	빠름
하드웨어 제어	제한적	완전 제어
디버깅 난이도	쉬움	복잡
학습 목적	사용 위주	원리 이해 위주

레지스터 접근 방식은 HAL의 내부 동작 구조를 이해하는 핵심 학습 단계이며,
 디버깅, 최적화, 하드웨어 디펜던시 최소화 등 고급 임베디드 설계에 필수적으로 사용된다.

보정 데이터 저장 예제 (EEPROM / Flash)

1. 개요

센서 보정(Calibration) 과정에서 계산된 **Offset**, **Scale**, **Baseline** 등의 파라미터는 전원이 꺼져도 유지되어야 한다. STM32F103은 내장 EEPROM이 없기 때문에 외부 I²C EEPROM(예: 24C02)을 사용하거나, 내부 Flash 메모리의 일부 영역을 데이터 저장용으로 활용한다.

두 방식 모두 비휘발성(NVM, Non-Volatile Memory)에 데이터를 저장할 수 있으며, **보정 데이터, 장비 설정, 로그 파라미터** 등을 유지하는 데 사용된다.

---

2. EEPROM 저장 방식 (I²C 24C02 예시)

(1) EEPROM 구조

- 주소 공간: 256 bytes (0x00 ~ 0xFF)
- 페이지 크기: 8 bytes (페이지 단위 쓰기)
- 쓰기 주기: 약 5ms (Write Cycle Time)

(2) 보정 데이터 구조체

```
typedef struct {
    float weight_offset;
    float weight_scale;
    float water_offset;
    float water_scale;
} CalibData_t;

CalibData_t calib_data;
```

(3) EEPROM 쓰기 함수

```
void EEPROM_Write(uint16_t memAddr, uint8_t *data, uint16_t len)
{
    HAL_I2C_Mem_Write(&hi2c1, 0xA0, memAddr, I2C_MEMADD_SIZE_8BIT, data, len, 100);
    HAL_Delay(5); // Write cycle time
}
```

(4) EEPROM 읽기 함수

```
void EEPROM_Read(uint16_t memAddr, uint8_t *data, uint16_t len)
{
    HAL_I2C_Mem_Read(&hi2c1, 0xA0, memAddr, I2C_MEMADD_SIZE_8BIT, data, len, 100);
}
```

(5) 보정 데이터 저장 / 로드

```
void SaveCalibration(void)
{
    EEPROM_Write(0x00, (uint8_t*)&calib_data, sizeof(CalibData_t));
}

void LoadCalibration(void)
{
    EEPROM_Read(0x00, (uint8_t*)&calib_data, sizeof(CalibData_t));
}
```

---

3. 내부 Flash 저장 방식

(1) Flash 데이터 영역 선택

STM32F103C8T6의 Flash 크기는 64KB이다.
일반적으로 **마지막 페이지(1KB)** 를 데이터 저장용으로 할당한다.

```
#define FLASH_USER_START_ADDR   0x0800FC00U  // 마지막 1KB
#define FLASH_USER_END_ADDR     0x0800FFFFU
```

(2) Flash 저장 함수

```
#include "stm32f1xx_hal_flash.h"

void Flash_Write(uint32_t address, uint32_t *data, uint16_t length)
{
    HAL_FLASH_Unlock();

    for(uint16_t i = 0; i < length; i++)
    {
        HAL_FLASH_Program(FLASH_TYPEPROGRAM_WORD, address, data[i]);
        address += 4;
    }

    HAL_FLASH_Lock();
}
```

(3) Flash 읽기 함수

```
void Flash_Read(uint32_t address, uint32_t *data, uint16_t length)
{
    for(uint16_t i = 0; i < length; i++)
    {
        data[i] = *(uint32_t*)address;
        address += 4;
    }
}
```

(4) 보정 데이터 저장 예시

```
void SaveCalibrationToFlash(void)
{
    uint32_t buffer[sizeof(CalibData_t)/4 + 1];
    memcpy(buffer, &calib_data, sizeof(CalibData_t));

    HAL_FLASH_Unlock();
    FLASH_EraseInitTypeDef EraseInitStruct;
    uint32_t PageError = 0;

    EraseInitStruct.TypeErase = FLASH_TYPEERASE_PAGES;
    EraseInitStruct.PageAddress = FLASH_USER_START_ADDR;
    EraseInitStruct.NbPages = 1;
    HAL_FLASHEx_Erase(&EraseInitStruct, &PageError);
    HAL_FLASH_Lock();

    Flash_Write(FLASH_USER_START_ADDR, buffer, sizeof(CalibData_t)/4 + 1);
}
```

(5) 보정 데이터 읽기 예시

```
void LoadCalibrationFromFlash(void)
{
    Flash_Read(FLASH_USER_START_ADDR, (uint32_t*)&calib_data, sizeof(CalibData_t)/4 +
1);
}
```

---

4. 데이터 무결성 및 버전 관리

보정 데이터 손상 방지를 위해 아래와 같은 메커니즘을 추가할 수 있다.

항목	설명
Checksum (CRC32)	데이터 일관성 검증용
버전 헤더 (Version Tag)	펌웨어/보정 버전 관리
백업 슬롯 (Backup Slot)	Flash 2페이지 번갈아 저장
마지막 성공 저장 시간 기록	RTC 타임스탬프 포함

```
typedef struct {
    uint32_t version;
    uint32_t crc;
    CalibData_t data;
} CalibPacket_t;
```

---

5. EEPROM vs Flash 비교

항목	EEPROM (24C02)	내부 Flash
인터페이스	I²C	메모리 직접 접근
저장 용량	256B~64KB	MCU에 따라 다름
쓰기 횟수	1백만 회 이상	약 1만 회
속도	느림 (I²C 통신)	빠름
코드 복잡도	낮음	높음 (Erase/Program 필요)
전력 소모	낮음	약간 높음

---

6. 정리

보정 데이터는 시스템 정확도와 신뢰성에 직접적인 영향을 미치며,
EEPROM 또는 Flash에 저장하여 전원 차단 후에도 유지된다.

- **소형 임베디드 시스템** → EEPROM 기반이 단순하고 안정적
- **Flash 내장형 MCU** → 별도 외부 칩 없이 구현 가능

실제 제품에서는 **데이터 백업 구조**, **CRC 검증**, **이중 페이지 플래시 저장** 기법을
조합하여 데이터 무결성을 확보하는 것이 바람직하다.

FreeRTOS 메모리 사용량 분석표

1. 개요

FreeRTOS 기반 시스템에서는 **태스크(Stack)**, **큐/세마포어**, **타이머 서비스**, **힙(Heap)** 등이
모두 **RAM 자원**을 소모한다.
특히 STM32F103C8T6처럼 **SRAM 20KB 제한 MCU**에서는
각 구성요소의 메모리 점유를 세밀하게 관리해야 한다.

---

2. 메모리 구성 요소

구분	항목	설명
커널 영역	Task Control Block (TCB)	각 Task의 상태, 우선순위, 스택 포인터 등 저장
	Idle Task / Timer Task	시스템 필수 태스크
사용자 영역	Task Stack	각 태스크의 지역변수 및 함수 호출 스택
	Queue / Semaphore	동기화 및 데이터 버퍼용
	Heap	동적 메모리(`pvPortMalloc`)로 관리
링커 관리 영역	전역 변수 / BSS	초기화된 전역 변수 및 배열

구분	항목	설명
	HAL/Driver Buffer	HAL 라이브러리에서 사용하는 버퍼

3. 태스크별 메모리 사용 예시

Task 이름	Stack 크기 (word)	Stack 크기 (byte)	주기(ms)	설명
IdleTask	128	512	-	시스템 대기 상태
SensorTask	256	1024	50	초음파/HX711 측정
ControlTask	256	1024	100	PID 제어 루프
DisplayTask	192	768	200	OLED 갱신
CommTask	256	1024	100	BLE/UART 통신
LoggingTask	256	1024	500	Flash/SD 기록
TimerTask	128	512	-	소프트웨어 타이머 관리
합계 (Task Stack)	**1,472 words**	**5,888 bytes (~5.75 KB)**		

4. Queue / Semaphore / Timer 메모리

구성요소	수량	단위 크기 (byte)	총합 (byte)	용도
Queue (Sensor → Control)	1	64	64	센서 데이터 전달
Queue (Control → Display)	1	32	32	상태 업데이트
Binary Semaphore (I²C Sync)	1	32	32	자원 보호
Mutex (OLED / UART)	2	48	96	동시 접근 제어
Software Timer	2	64	128	주기성 이벤트
합계	7		**352 bytes**	

5. 커널 및 관리 영역

항목	사용량 (byte)	비고
Idle Task + Timer Task Stack	1024	시스템 기본 포함
Kernel TCB/Queue Structures	768	태스크 관리용
Heap 관리 오버헤드	512	`heap_4.c` 사용 시

항목	사용량 (byte)	비고
소계	**2,304 bytes (~2.25 KB)**	

---

6. 총 RAM 사용 추정

구분	사용량 (byte)	비고
Task Stack	5,888	사용자 태스크
Queue / Semaphore	352	통신 동기화
Kernel / 관리	2,304	시스템 내부
FreeRTOS 합계	**8,544 (~8.35 KB)**	
전역 변수 (HAL, 센서 버퍼 등)	~4,000	드라이버 및 전역 변수
총합 (RAM 사용)	**≈12.5 KB / 20 KB (62%)**	

---

7. Heap (동적 메모리) 설정

FreeRTOS는 `FreeRTOSConfig.h`에서 힙 크기를 지정한다.

```
#define configTOTAL_HEAP_SIZE    (10 * 1024) // 10KB
```

> ⚙️ 실제 할당량은 `vPortGetFreeHeapSize()`로 확인 가능

```
printf("Free Heap: %lu bytes\n", xPortGetFreeHeapSize());
```

---

8. 메모리 최적화 전략

항목	방법	기대 효과
Stack 최소화	`uxTaskGetStackHighWaterMark()`로 여유 분석	불필요한 스택 감소
Static Allocation	`xTaskCreateStatic()` 사용	Heap 사용 제거
Queue 크기 조정	송신 주기 대비 버퍼 크기 최소화	RAM 절약
공유 버퍼 사용	센서/통신용 임시 버퍼 통합	중복 제거
DMA 사용	대용량 데이터 전송 시 CPU 부하 및 RAM 부담 완화	효율적 처리

---

9. 메모리 분석 예시 출력 (UART 로그)

```
===== FreeRTOS Memory Report =====
Task Count: 6
Heap Size: 10240 bytes
Free Heap: 3320 bytes
Idle Stack High Water: 112 words
Sensor Stack High Water: 180 words
Control Stack High Water: 160 words
Display Stack High Water: 140 words
Comm Stack High Water: 200 words
Logging Stack High Water: 220 words
==================================
```

---

10. 결론

- **FreeRTOS 핵심 오버헤드:** 약 2~3KB
- **태스크별 스택:** 전체 RAM 사용의 60~70% 차지
- **실제 여유 메모리:** 약 7KB 내외
- 정적 할당(`Static Allocation`) + 스택 최적화를 병행하면
  안정적 동작 + RAM 절약을 동시에 달성할 수 있다.

전력 소비 측정 리포트

1. 개요

본 섹션에서는 **STM32 기반 저전력 시스템(초음파 + 수위 + HX711 + RTC + OLED)**의
운영 모드별 전력 소비를 정량적으로 분석한다.
측정은 **전류 프로브(μA~mA 단위)** 또는 **INA219 전류 센서**를 이용해
VCC 라인에 직렬 연결한 상태에서 수행되었다.

---

2. 측정 환경

항목	사양
MCU	STM32F103C8T6 (72MHz, 3.3V)
전원 공급	USB 5V → LDO 3.3V
측정 장비	INA219 / Otii Arc / 멀티미터
샘플링 속도	10Hz (평균값 계산)
온도 조건	25 °C
펌웨어 구성	FreeRTOS + Tickless Idle + RTC Wakeup
Sleep 정책	STOP 모드 진입 후 알람 이벤트 복귀

3. 측정 항목 및 전류 소비

동작 모드	설명	평균 전류 (mA)	동작 시간 (s)	비고
Active (측정 루틴)	초음파 + HX711 + OLED + BLE 송신	**22.5 mA**	0.5	약 11 mJ 소모
Idle (대기 루프)	FreeRTOS IdleTask, Tick 유지	**5.3 mA**	2.0	
STOP 모드 (RTC 유지)	RTC/LSE ON, RAM 유지	**0.080 mA (80 µA)**	57.5	
Reset/Boot 구간	초기화 + 센서 설정	**25.0 mA**	0.2	비정상 상승 구간
평균 주기 (1분)	1분 주기 측정 기준	**0.27 mA**	-	= (Σ전력×시간)/60

4. 전력 계산

측정 주기: 1분
평균 전류: 0.27 mA
공급 전압: 3.3 V

$$P_{avg} = 3.3V \times 0.27mA = 0.891mW$$

5. 배터리 수명 예측

배터리	용량	이론 지속 시간	실제 추정 (효율 80%)
CR2032 (220 mAh)	0.22 Ah	0.22 / 0.00027 ≈ **814 h (34일)**	약 **27일**
18650 (2000 mAh)	2.0 Ah	2.0 / 0.00027 ≈ **7407 h (308일)**	약 **246일**
3.7V LiPo (1000 mAh)	1.0 Ah	1.0 / 0.00027 ≈ **3703 h (154일)**	약 **120일**

6. 세부 전류 분석 (Active 구간)

모듈	소비 전류 (mA)	비율 (%)	동작 시간 (ms)	비고
MCU Core (72MHz)	12.5	55%	500	연산, 제어
HX711 ADC	1.4	6%	500	측정 중 활성
초음파 모듈 (HC-SR04)	6.2	28%	250	TRIG+ECHO 구간
OLED (SSD1306)	2.0	9%	500	표시 갱신

모듈	소비 전류 (mA)	비율 (%)	동작 시간 (ms)	비고
BLE Tx	0.4	2%	100	UART 송신
합계	**22.5 mA**	100%	-	

---

7. 저전력 최적화 요약

개선 항목	조치 내용	절감 효과
MCU 클럭 스케일링	측정 중 72MHz → Sleep 시 8MHz	약 30% 감소
Tickless Idle 활성화	IdleTask → STOP 모드 진입	μA 단위 절전
OLED 절전모드 사용	Display OFF 명령 (`0xAE`)	대기 시 2 mA 절약
HX711 전원 차단	GPIO로 전원 제어	Sleep 중 1.5 mA 절감
BLE Tx 주기 조정	필요 시에만 전송	평균 5~10% 절전
RTC LSE 유지	32.768 kHz 오실레이터만 유지	안정적 Wake-up 보장

---

8. 실측 파형 예시 (INA219 Log)

```
Time(s) | Current(mA)
0.0     | 25.2   <-- Boot
0.2     | 22.4   <-- Sensor Active
0.7     | 5.3    <-- Idle
2.7     | 0.08   <-- STOP Mode
60.0    | 25.1   <-- Wake-up (RTC Alarm)
```

> 주기적 패턴으로 1분당 약 0.27 mA의 평균 소비가 계산됨.

---

9. 결론

- 주기 측정 구조에서 **대부분의 시간(>95%)은 STOP 모드**로 유지됨
- 실질 평균 소비전류는 **0.2~0.3 mA 수준**으로,
  1Ah 배터리 기준 약 4~5개월 지속 가능
- OLED 및 센서 전원 차단을 병행하면
  50% 이상 추가 절감 가능

---

10. 추가 분석 제안

분석 항목	도구	목적
전류 트레이스 분석	Otii Arc / Joulescope	동작 시퀀스별 전력 파형 시각화
FreeRTOS Idle Hook 활용	CPU 사용률 → 전류 상관 분석	부하 최적화
Power Profile 자동 로깅	INA219 + SD 카드	장기 평균 전력 추적