

1. STM32 마이크로컨트롤러 기본기

1.1 MCU 아키텍처 개요

• Cortex-M3 핵심 구조 (레지스터, 스택, 인터럽트 벡터)

(레지스터, 스택, 인터럽트 벡터)

1.1 개요

Cortex-M3는 ARMv7-M 아키텍처를 기반으로 하는 32-비트 RISC 프로세서 코어로, STM32F103 시리즈 MCU의 중앙 처리 장치로 사용된다.

이 코어는 **고성능, 저전력, 실시간 제어**를 목적으로 설계되었으며, Thumb-2 명령어 집합과 하드웨어 수준의 예외 관리 기능을 제공한다.

항목	내용
코어 이름	ARM Cortex-M3
아키텍처	ARMv7-M
데이터 폭	32비트
파이프라인 단계	Fetch, Decode, Execute (3-stage)
인터럽트 컨트롤러	NVIC (Nested Vectored Interrupt Controller)
스택 구조	Full-Descending (주소 감소 방향으로 성장)
특징	Thumb-2 ISA, 하드웨어 인터럽트 벡터, 예외 자동 저장 메커니즘

1.2 레지스터 구조

Cortex-M3 코어는 16개의 범용 레지스터(R0~R15)와 하나의 프로그램 상태 레지스터(xPSR)로 구성된다. 각 레지스터의 기능은 다음과 같다.

레지스터	이름	용도
R0 ~ R3	일반 레지스터	함수 인자, 연산 중간값 저장
R4 ~ R11	일반 레지스터	지역 변수 저장, 스택 보존용
R12	Intra-procedure (IP)	임시 레지스터
R13	Stack Pointer (SP)	스택의 현재 위치를 가리킴
R14	Link Register (LR)	함수 복귀 주소 저장
R15	Program Counter (PC)	현재 실행 명령어의 주소

레지스터	이름	용도
xPSR	Program Status Register	연산 결과, 예외 상태, 실행 제어 상태

(1) Stack Pointer (R13)

Cortex-M3는 두 개의 스택 포인터를 제공한다.

스택 포인터	용도
MSP (Main Stack Pointer)	시스템 초기화 및 인터럽트 처리용
PSP (Process Stack Pointer)	사용자 모드 또는 RTOS 태스크용

현재 활성화된 스택 포인터는 CONTROL 레지스터의 비트[1]로 결정된다.

CONTROL = 0 이면 MSP, CONTROL = 1 이면 PSP가 선택된다.

(2) Link Register (R14)

BL 명령으로 함수를 호출하면 복귀 주소가 LR에 저장된다.

예외 처리 중에는 LR에 특수한 복귀 코드를 저장하여 복귀 모드를 구분한다.

값	의미
0xFFFFFFFF9	예외 복귀 시 MSP 사용
0xFFFFFFFFD	예외 복귀 시 PSP 사용

(3) Program Counter (R15)

현재 실행 중인 명령어 주소를 나타내며,
명령어 실행 후 자동으로 2 또는 4바이트 단위로 증가한다.

(4) Program Status Register (xPSR)

xPSR은 APSR, EPSR, IPSR의 세 부분으로 구성된다.

서브레지스터	기능
APSR	연산 결과 플래그 (N, Z, C, V)
EPSR	Thumb 상태 등 실행 제어 비트
IPSR	현재 실행 중인 예외 번호

xPSR의 주요 비트는 다음과 같다.

비트	이름	설명
[31]	N	결과가 음수이면 1
[30]	Z	결과가 0이면 1
[29]	C	캐리 발생 시 1
[28]	V	오버플로 발생 시 1
[8:0]	ISR_NUMBER	현재 실행 중인 예외 번호

1.3 스택 구조

Cortex-M3는 **Full-Descending Stack** 구조를 사용한다.
스택은 주소가 감소하는 방향으로 확장되며,
스택에 저장되는 기본 프레임(Stack Frame)은 다음과 같다.



예외(인터럽트) 발생 시 위 8개의 레지스터가 **하드웨어에 의해 자동으로 푸시**된다.
복귀 시에는 `BX LR` 명령을 통해 자동으로 복원된다.

(1) MSP와 PSP의 구분

구분	사용 주체	주요 용도
MSP	시스템, 인터럽트	초기화, Fault, SysTick
PSP	사용자 태스크	RTOS Task, 스레드 실행

RTOS 환경에서는 각 태스크가 독립적인 PSP를 사용하며,
컨텍스트 전환 시 PSP 값만 교체함으로써 스택 교체가 이루어진다.

1.4 인터럽트 벡터 테이블

Cortex-M3의 예외 및 인터럽트 벡터는 메모리의 **0x0000_0000** 번지부터 시작한다.

각 엔트리는 32비트(4바이트) 주소로 구성되어 있으며,

해당 예외 또는 인터럽트가 발생했을 때 분기할 핸들러의 주소를 나타낸다.

오프셋	항목	설명
0x00	초기 스택 포인터 값	부팅 시 MSP 초기값
0x04	Reset_Handler	시스템 리셋 시 진입점
0x08	NMI_Handler	비마스크 예외 처리
0x0C	HardFault_Handler	하드 폴트 처리
0x10	MemManage_Handler	메모리 보호 예외
0x14	BusFault_Handler	버스 접근 예외
0x18	UsageFault_Handler	잘못된 명령어 예외
...	SysTick_Handler	시스템 틱 인터럽트
...	EXTIx_IRQHandler	외부 GPIO 인터럽트
...	TIMx_IRQHandler	타이머 인터럽트
...	USARTx_IRQHandler	UART 인터럽트

예시: STM32F103의 벡터 테이블 일부

```
1  .section .isr_vector
2      .word  _estack          /* 초기 스택 포인터 */
3      .word  Reset_Handler    /* Reset */
4      .word  NMI_Handler      /* NMI */
5      .word  HardFault_Handler /* HardFault */
6      .word  MemManage_Handler /* Memory Management */
7      .word  BusFault_Handler
8      .word  UsageFault_Handler
9      ...
10     .word  SysTick_Handler
11     .word  WWDG_IRQHandler
12     .word  PVD_IRQHandler
13     .word  EXTI0_IRQHandler
14     .word  EXTI1_IRQHandler
15     ...
```

1.5 NVIC (Nested Vectored Interrupt Controller)

NVIC는 Cortex-M3의 핵심 구성요소로,
모든 예외 및 인터럽트의 우선순위와 활성화 상태를 제어한다.

기능	설명
벡터 관리	최대 240개의 인터럽트 지원 (STM32F103은 약 60개 사용)
중첩 인터럽트	우선순위가 높은 인터럽트는 실행 중 다른 인터럽트를 중단 가능
우선순위 구조	Preemption Priority와 Sub Priority로 구성
레지스터	ISER, ICER, ISPR, ICPR, IPR 등

HAL 계층에서는 NVIC을 다음과 같은 API로 제어한다.

```
1 HAL_NVIC_SetPriority(IRQn_Type IRQn, uint32_t preempt, uint32_t sub);
2 HAL_NVIC_EnableIRQ(IRQn_Type IRQn);
3 HAL_NVIC_DisableIRQ(IRQn_Type IRQn);
```

1.6 예외 및 인터럽트 동작 순서

- 이벤트 발생 (GPIO, Timer, RTC 등)
- NVIC이 해당 예외 번호를 식별
- 현재 명령어 완료 후 Cortex-M3가 자동으로 컨텍스트 저장 (R0~R3, R12, LR, PC, xPSR을 스택에 푸시)
- LR에 복귀 코드 저장 (예: 0xFFFFFFF9)
- PC를 벡터 테이블의 ISR 주소로 변경
- ISR 실행
- BX LR** 명령 실행 → 스택에서 자동 복원 → 원래 코드로 복귀

이 전체 과정은 하드웨어 수준에서 자동으로 수행되며,
소프트웨어 개입 없이 컨텍스트 스위칭이 이루어진다.

1.7 FreeRTOS와의 연계 구조

FreeRTOS는 Cortex-M3의 이중 스택 구조(MSP/PSP)와 하드웨어 자동 푸시/복원 메커니즘을 활용한다.

기능	사용 자원
태스크 스택	PSP
인터럽트 스택	MSP
컨텍스트 저장	자동 (R0~R15 + xPSR)
PendSV	소프트웨어 예외를 통한 태스크 전환

기능	사용 자원
SysTick	주기적 스케줄러 틱 발생원

RTOS는 단순히 PSP 값을 교체하는 방식으로 태스크 간 전환을 수행하며, 이 덕분에 빠르고 안정적인 실시간 스케줄링이 가능하다.

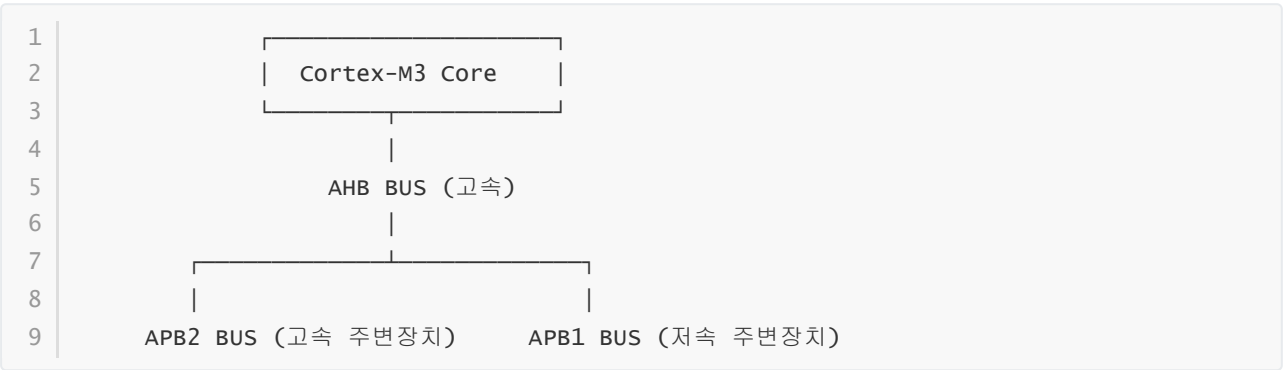
1.8 요약

항목	내용
레지스터 구성	R0~R15 + xPSR (총 17개)
스택 구조	Full-Descending, MSP/PSP 이중 포인터
인터럽트 벡터	0x0000_0000번지부터 시작하는 벡터 테이블
예외 처리	자동 푸시/복원 메커니즘
NVIC 기능	우선순위, 활성화, 중첩 제어
RTOS 활용	PSP 기반 태스크 스택 전환

• STM32F1 시리즈의 버스 구조 (AHB, APB1, APB2)

1.2.1 개요

STM32F1 시리즈는 **Cortex-M3 코어와 주변 장치(Peripheral)** 사이의 데이터 전송 효율을 높이기 위해 **다단 버스 계층 구조(Multi-Bus Architecture)**를 채택하고 있다.
이 구조는 **AHB → APB1 / APB2**로 분리된 버스 트리 형태이며, 각 버스는 클록 주파수와 연결된 주변 장치의 성격에 따라 구분된다.



1.2.2 AHB (Advanced High-Performance Bus)

AHB 버스는 시스템의 중심 고속 버스로, CPU, DMA, 플래시 메모리, SRAM, 버스 매트릭스 등이 이 버스에 연결된다. AHB는 32-비트 폭의 데이터 버스를 가지며, 단일 사이클 접근이 가능하다.

항목	내용
버스 폭	32비트
역할	CPU와 고속 주변장치 간 데이터 전송
주요 연결 장치	Cortex-M3, Flash, SRAM, DMA, RCC, NVIC, APB 브릿지
클록 속도	시스템 클록(HCLK) = SYSCLK

주요 구성 요소

구성 요소	설명
CPU Core Bus	명령어/데이터 접근용
DMA Controller	메모리 간 데이터 전송 담당
SRAM / Flash	내부 메모리 인터페이스
AHB-to-APB Bridge	저속 버스로 데이터 전송을 위한 브릿지

AHB 버스는 CPU의 메모리 접근, DMA, 인터럽트 처리 등 모든 핵심 데이터 흐름의 중심 경로를 제공한다.

1.2.3 APB1 (Advanced Peripheral Bus 1)

APB1 버스는 저속 주변 장치(저주파 클록 기반 디바이스)가 연결되는 버스이다. AHB보다 클록이 느리며, 전력 소비가 적다.

항목	내용
버스 폭	32비트
클록 주파수	HCLK의 최대 1/2 (36MHz 이하)
주요 연결 주변장치	USART2/3, I2C1/2, SPI2, CAN, TIM2~TIM7, DAC, PWR, BKP
클록 명칭	PCLK1

예시: APB1 주변장치

- **TIM2-TIM7** : 범용 타이머 (16-비트, 저속 타이밍)
- **USART2, USART3, UART4, UART5** : 시리얼 통신 장치
- **I2C1, I2C2** : I2C 버스 통신
- **SPI2** : 저속 SPI 통신
- **CAN** : 차량용 통신 버스
- **DAC** : 아날로그 출력 장치
- **PWR / BKP** : 전원 관리 및 백업 도메인

APB1은 저속 주변장치의 전력 효율을 높이기 위해 클록 속도를 낮게 유지하도록 설계되어 있다.

1.2.4 APB2 (Advanced Peripheral Bus 2)

APB2 버스는 고속 주변장치가 연결되는 버스이다.
이 버스는 CPU와 직접적인 인터페이스를 가지며, 높은 클록 주파수를 제공한다.

항목	내용
버스 폭	32비트
클록 주파수	HCLK (최대 72MHz)
주요 연결 주변장치	GPIOA~E, ADC1~3, SPI1, USART1, TIM1, AFIO
클록 명칭	PCLK2

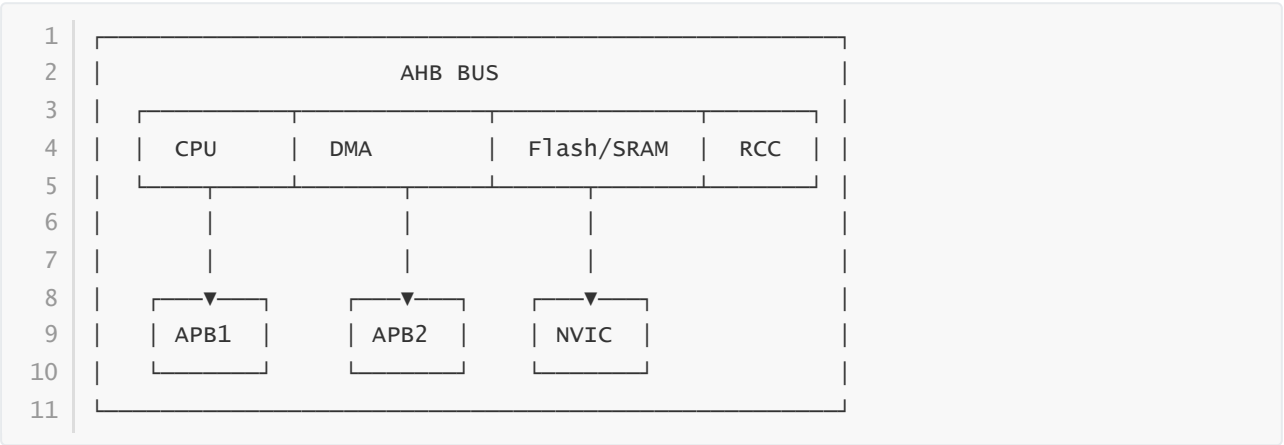
예시: APB2 주변장치

- **GPIOA~GPIOE** : 일반 목적 입출력 포트
- **ADC1~ADC3** : 고속 아날로그-디지털 변환기
- **TIM1** : 고급 타이머 (PWM, 캡처, 모터 제어 지원)
- **USART1** : 고속 UART 통신
- **SPI1** : 고속 SPI 통신
- **AFIO (Alternate Function I/O)** : 주변장치 핀 리매핑 기능 제공

APB2는 실시간 제어나 고속 샘플링이 필요한 장치가 주로 연결된다.

1.2.5 버스 간 연결 구조

STM32F1의 전체 버스 연결은 다음과 같이 표현된다.



1.2.6 버스 클록 관계

각 버스의 클록은 **RCC (Reset and Clock Control)** 모듈에 의해 설정된다.
아래는 일반적인 STM32F103의 클록 관계이다.

버스	클록 명칭	기본 관계	예시 (72MHz 시스템 클록 기준)
AHB	HCLK	SYSCLK / 1	72 MHz
APB2	PCLK2	HCLK / 1	72 MHz
APB1	PCLK1	HCLK / 2	36 MHz

특정 주변장치의 타이머(TIMx)는 클록이 두 배로 동작하도록 설계되어 있다.

주변장치	입력 클록
TIM1~TIM8 (APB2)	PCLK2 × 2
TIM2~TIM7 (APB1)	PCLK1 × 2

1.2.7 요약

버스	속도	주요 연결 장치	클록 명칭
AHB	고속	CPU, DMA, SRAM, Flash, RCC	HCLK
APB1	저속	USART2~5, I2C, SPI2, CAN, DAC, PWR	PCLK1
APB2	고속	GPIOA~E, ADC, TIM1, SPI1, USART1, AFIO	PCLK2

• 메모리 맵 구조 (Flash, SRAM, Peripheral)

1.3.1 개요

STM32F1 시리즈는 **ARM Cortex-M3** 코어를 기반으로 하며,
모든 메모리 및 주변장치는 **단일 4GB(0x0000_0000 ~ 0xFFFF_FFFF)** 주소 공간에 매핑된다.
이 구조를 "**메모리 맵(Memory Map)**" 이라고 하며,
CPU가 플래시, SRAM, 주변장치, 시스템 영역 등에 접근할 때
물리 주소 기반으로 직접 접근할 수 있도록 구성되어 있다.

이 통합 주소 공간은 **AHB / APB 버스 구조**와 연동되어 동작하며,
각 메모리 구역은 다음과 같이 구분된다.

1		0xFFFF FFFF
2	Peripheral (APB2)	0x4001 0000 ~ 0x4001 FFFF
3	Peripheral (APB1)	0x4000 0000 ~ 0x4000 FFFF
4	AHB Peripheral	0x5000 0000 ~ 0x5FFF FFFF
5	SRAM (Internal)	0x2000 0000 ~ 0x2000 FFFF
6	Flash Memory	0x0800 0000 ~ 0x080F FFFF
7	System Memory / Boot	0x1FFF F000 ~ 0x1FFF F7FF
8	Cortex-M3 Private Bus	0xE000 0000 ~ 0xE00F FFFF
9	Aliased / Reserved	기타 영역
10		0x0000 0000

1.3.2 주요 메모리 영역 요약

구분	주소 범위	크기	용도
Flash Memory	0x0800 0000 ~ 0x080F FFFF	최대 1MB	사용자 프로그램 저장
SRAM (Main RAM)	0x2000 0000 ~ 0x2000 FFFF	64KB	전역 변수, 스택, 런타임 데이터
System Memory	0x1FFF F000 ~ 0x1FFF F7FF	2KB	부트로더(Bootloader) 코드
Option Bytes	0x1FFFF800 ~ 0x1FFFF80F	16B	플래시 보호, 부팅 모드 설정
Peripheral (APB1, APB2, AHB)	0x4000 0000 ~ 0x5006 0000	-	주변장치 레지스터 매핑
Cortex-M3 Private Bus	0xE000 0000 ~ 0xE00F FFFF	-	NVIC, SysTick, SCB, MPU 등
Alias / Remap	0x0000 0000 ~ 0x000F FFFF	-	부팅 시 Flash/SRAM 재매핑 용

1.3.3 Flash Memory (프로그램 메모리)

Flash는 **비휘발성(Non-Volatile)** 메모리로, MCU 전원이 꺼져도 데이터가 유지된다. 사용자는 이 영역에 펌웨어(프로그램 코드)를 저장한다.

항목	설명
시작 주소	0x0800 0000
크기	64KB ~ 1MB (기종별 상이)
접근 속도	최대 72MHz
인터페이스	AHB 버스 연결

항목	설명
주요 용도	프로그램 코드, 상수 데이터 저장

특징

- Boot0 핀 설정에 따라 Flash, SRAM, System Memory 중 어느 영역에서 부팅할지 결정된다.
- Flash는 페이지 단위(1~2KB)로 지워지며,
HAL_FLASH_Program() 함수를 통해 런타임 중 쓰기 가능하다.

예시 - Flash 구조 (256KB MCU)

1	0x0800 0000	_____
2	Bootloader 영역 (optional)	
3	Application Code (Main)	
4	Constants / Calibration Data	
5	Flash 마지막 1~2KB → Option Bytes	
6	_____	0x0803 FFFF

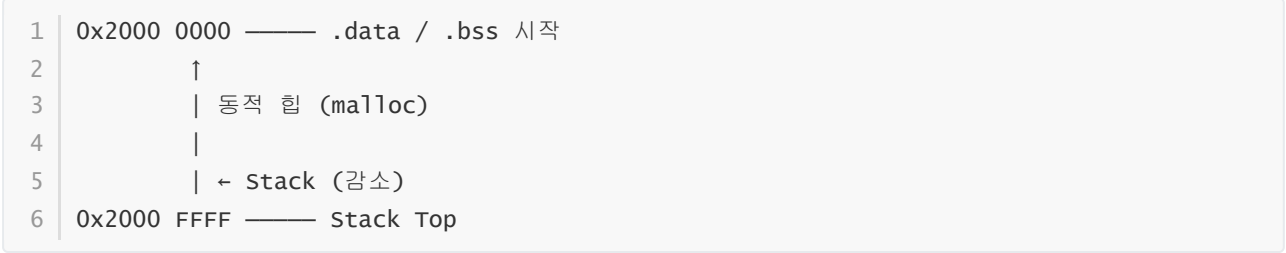
1.3.4 SRAM (Main Memory)

SRAM은 휘발성(Volatile) 메모리로, MCU가 실행 중일 때
전역 변수, 스택, 동적 메모리(malloc) 등의 저장 공간으로 사용된다.

항목	설명
시작 주소	0x2000 0000
크기	20KB ~ 64KB
접근 속도	1 CPU 사이클
연결 버스	AHB
주요 용도	전역 변수, 지역 변수, 스택, 힙

RAM 내부 영역 예시

영역	내용	비고
.data	초기화된 전역/정적 변수	Flash → RAM 복사
.bss	초기화되지 않은 전역/정적 변수	0으로 초기화
stack	함수 호출, 로컬 변수 저장	하단부터 감소 방향
heap	malloc() 할당 영역	증가 방향



1.3.5 System Memory (부트로더 영역)

System Memory는 **STMicroelectronics**가 제공하는 ROM 영역으로, 사용자가 직접 수정할 수 없다.
이 영역에는 **내장 부트로더**가 저장되어 있으며, UART, USB, CAN, I²C를 통한 펌웨어 업데이트(DFU)가 가능하다.

항목	설명
주소 범위	0x1FFF F000 ~ 0x1FFF F7FF
크기	2KB
역할	내장 부트로더, 펌웨어 업로드 인터페이스
접근 방식	Read-Only (ROM)

부팅 순서

1. Boot0 핀 상태 확인
2. Bootloader 활성화 여부 결정
3. System Memory 실행 시, UART/USB 등으로 대기

1.3.6 Peripheral Memory (주변장치 레지스터 영역)

모든 주변장치(ADC, GPIO, UART 등)는 **레지스터 집합 형태로 메모리 공간에 직접 매핑(Memory Mapped I/O)** 되어 있다.
즉, MCU는 특정 주소에 데이터를 쓰는 것만으로 하드웨어 제어를 수행한다.

버스 구분	주소 범위	주요 장치 예시
APB1	0x4000 0000 ~ 0x4000 FFFF	USART2~5, I ² C1/2, SPI2, CAN
APB2	0x4001 0000 ~ 0x4001 FFFF	GPIOA~E, ADC1/2, TIM1, SPI1, USART1
AHB	0x5000 0000 ~ 0x5006 0000	DMA1/2, RCC, FSMC, SDIO

예시 - 주변장치 레지스터 주소

장치	시작 주소	비고
GPIOA	0x40010800	APB2
GPIOB	0x40010C00	APB2
USART1	0x40013800	APB2
USART2	0x40004400	APB1
ADC1	0x40012400	APB2
RCC	0x40021000	AHB

1.3.7 Cortex-M3 Private Peripheral Bus

Cortex-M3 내부 전용 제어 모듈은 별도의 **Private Peripheral Bus (PPB)** 영역에 매핑되어 있다. 이 영역은 시스템 제어, 인터럽트, 디버깅 기능을 담당한다.

모듈	주소 범위	기능
NVIC	0xE000E100 ~ 0xE000E4FF	인터럽트 제어
SCB	0xE000ED00 ~ 0xE000ED8F	시스템 제어 (예외, 벡터 등)
SysTick	0xE000E010 ~ 0xE000E01F	시스템 타이머
MPU	0xE000ED90 ~ 0xE000EDFF	메모리 보호 단위

이 영역은 HAL이 아니라 **CMSIS(Core System Interface)** 계층에서 접근한다.

1.3.8 메모리 리매핑 (Boot Configuration)

STM32F1 시리즈는 부팅 시점에 다음 세 가지 메모리 중 하나를 주소 **0x0000_0000**에 매핑할 수 있다.

Boot 모드	매핑 대상	설명
Boot0 = 0, Boot1 = X	Flash	일반 사용자 부팅
Boot0 = 1, Boot1 = 0	System Memory	내장 부트로더 실행
Boot0 = 1, Boot1 = 1	SRAM	디버깅용, RAM에서 코드 실행

이는 `BOOT0` 핀 상태와 `Option Bytes` 설정에 의해 결정된다.

1.3.9 요약

영역	주소 범위	버스	주요 역할
Flash	0x0800 0000 ~	AHB	코드 저장
SRAM	0x2000 0000 ~	AHB	런타임 데이터
System Memory	0x1FFF F000 ~	AHB	내장 부트로더
Peripheral (APB1/2)	0x4000 0000 ~	APB	주변장치 제어
Cortex-M3 Private Bus	0xE000 0000 ~	내부	NVIC, SCB, SysTick 제어

• RCC(Reset and Clock Control) 동작원리

1.4.1 개요

STM32F1 시리즈의 **RCC(Reset and Clock Control)** 는 시스템 전체의 **클록(clock)** 및 **리셋(reset)** 동작을 관리하는 핵심 모듈이다.

RCC는 다음과 같은 기능을 담당한다.

- 시스템 클록 생성 및 분배
- 내부/외부 오실레이터 관리 (HSI, HSE, PLL 등)
- 주변장치 클록 공급 제어
- 각 버스(AHB, APB1, APB2)의 클록 분주 설정
- 전원 온/오프, 리셋 원인 감지

즉, **RCC는 MCU의 “심장부(Clock Generator)”** 역할을 수행한다.

모든 주변장치와 CPU의 동작 주기는 RCC가 설정한 클록 신호에 의해 결정된다.

1.4.2 시스템 클록(시스템 주파수) 개요

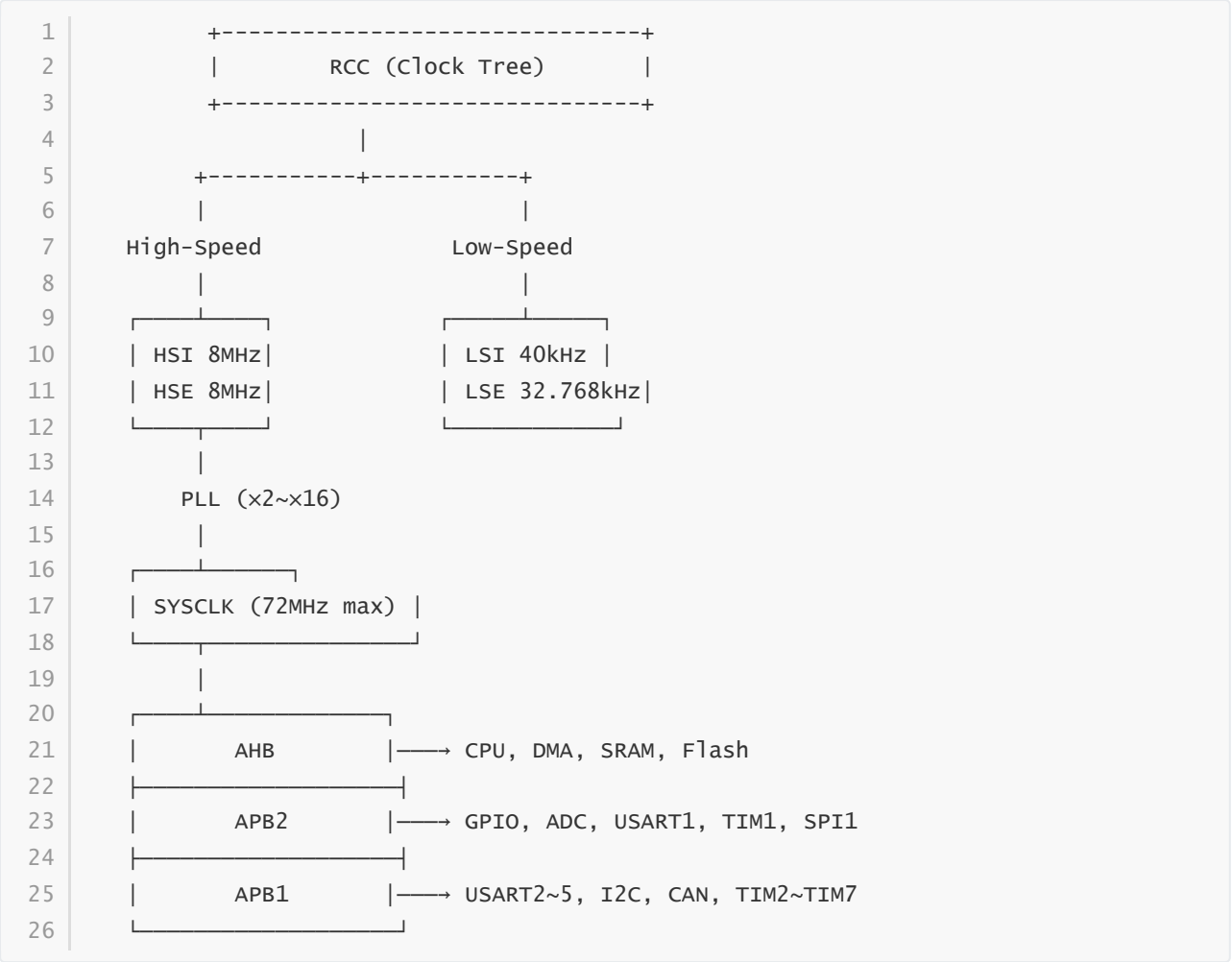
STM32F103 기준, 시스템 클록은 다음 네 가지 소스로부터 선택할 수 있다.

구분	이름	주파수	설명
내부 오실레이터	HSI (High-Speed Internal)	8 MHz	MCU 내장 RC 오실레이터
외부 오실레이터	HSE (High-Speed External)	4~16 MHz	외부 크리스털 또는 클록 입력
위상고정회로	PLL (Phase-Locked Loop)	최대 72 MHz	HSI 또는 HSE를 입력으로 증폭
저속 오실레이터	LSI / LSE	32.768 kHz	RTC 및 저전력 타이머용

최종적으로 **SYSCLK (System Clock)** 은 **HSI** , **HSE** , 또는 **PLL** 중 하나가 선택되어 CPU와 AHB 버스에 공급된다.

1.4.3 RCC의 전체 클록 분배 구조

RCC의 클록 트리는 다음과 같은 계층으로 구성된다.



1.4.4 클록 소스별 특성

클록 소스	설명	장점	단점
HSI (High-Speed Internal)	내장 RC 오실레이터 (8MHz)	외부 부품 불필요, 빠른 기동	정확도 낮음 ($\pm 1\%$)
HSE (High-Speed External)	외부 크리스털 또는 클록 입력	정확도 높음, 안정적	외부 부품 필요, 기동시간 길다
PLL (Phase-Locked Loop)	입력 신호를 정수배 증폭	고속 클록 생성 가능	발진기 의존, 잡음 민감
LSI / LSE	저속 클록(32kHz)	RTC 및 저전력 타이머 용	메인 클록으로는 불가

1.4.5 클록 소스 선택 및 전환 과정

시스템 클록(SYSCLK)을 설정하는 과정은 다음 순서로 이루어진다.

1. HSI 또는 HSE 발진기 활성화
 - RCC_CR 레지스터의 HSION 또는 HSEON 비트를 설정
 - HSIRDY / HSERDY 플래그가 1이 될 때까지 대기
2. PLL 구성 및 활성화
 - PLL 입력 소스 선택 (PLLXTPRE , PLLSRC)
 - PLL 배율 설정 (PLLMUL)
 - PLLON 비트 설정 후 PLLRDY 대기
3. 시스템 클록 전환
 - RCC_CFGR의 sw[1:0] 비트를 변경하여 SYSCLK 소스 선택 (00: HSI, 01: HSE, 10: PLL)
 - SWS[1:0] 플래그를 확인하여 전환 완료 확인
4. 버스 클록 분주 설정
 - AHB, APB1, APB2 클록 분주비 설정
 - 최종 SYSCLK이 각각 HCLK, PCLK1, PCLK2로 분배

1.4.6 주요 레지스터 요약

레지스터	이름	주요 기능
RCC_CR	Clock Control Register	발진기 On/Off, 준비 플래그
RCC_CFGR	Clock Configuration Register	클록 소스, 분주비, PLL 설정
RCC_APB1ENR	APB1 Peripheral Clock Enable Register	저속 주변장치 클록 공급 제어
RCC_APB2ENR	APB2 Peripheral Clock Enable Register	고속 주변장치 클록 공급 제어
RCC_AHBENR	AHB Peripheral Clock Enable Register	DMA, SRAM, FSMC 클록 제어
RCC_CIR	Clock Interrupt Register	클록 준비 인터럽트 관리

1.4.7 클록 분주 관계

클록 이름	공급원	최대 주파수	용도
SYSCLK	PLL, HSI, HSE	72 MHz	CPU, AHB, APB 등 메인 클록
HCLK	SYSCLK / Prescaler	72 MHz	AHB 버스, DMA, SRAM
PCLK1	HCLK / 2	36 MHz	APB1 주변장치 (USART2~5, I2C, CAN 등)

클럭 이름	공급원	최대 주파수	용도
PCLK2	HCLK / 1	72 MHz	APB2 주변장치 (ADC, GPIO, TIM1 등)
ADCCLK	PCLK2 / 6	12 MHz	ADC 변환용 클럭
USBCLK	PLL / 1.5	48 MHz	USB 통신용

1.4.8 주변장치 클럭 제어

모든 주변장치는 RCC를 통해 개별적으로 클럭 공급을 제어할 수 있다.
즉, 사용하지 않는 주변장치의 클럭을 차단하면 전력 소모를 줄일 수 있다.

예시)

GPIOA와 ADC1, USART1을 활성화할 경우:

```
1 __HAL_RCC_GPIOA_CLK_ENABLE();
2 __HAL_RCC_ADC1_CLK_ENABLE();
3 __HAL_RCC_USART1_CLK_ENABLE();
```

비활성화 시

```
1 __HAL_RCC_USART1_CLK_DISABLE();
```

이 매크로들은 내부적으로 `RCC_APB2ENR` 또는 `RCC_APB1ENR` 레지스터의 비트를 세트/클리어하여 클럭 신호를 차단하거나 공급한다.

1.4.9 리셋(Reset) 관리 기능

RCC는 클럭뿐만 아니라 시스템 리셋 신호도 제어한다.
이는 전원 인가, 소프트웨어 명령, 외부 핀 입력 등 여러 원인에 의해 발생할 수 있다.

리셋 종류	발생 원인	비고
POR (Power-On Reset)	전원 인가 시 자동 발생	초기화 필수
BOR (Brown-Out Reset)	전압이 임계치 이하로 떨어질 때	전원 보호 기능
NRST 핀 리셋	외부 하드웨어 리셋	수동 리셋
SW Reset	프로그램 내에서 강제 리셋	<code>NVIC_SystemReset()</code>
Watchdog Reset	타임아웃 감시 실패 시	독립 또는 윈도우 위치독

리셋 원인은 `RCC_CSR` 레지스터의 `RST_FLAG` 들을 통해 확인 가능하다.

1.4.10 RCC 설정 예시 (72MHz PLL 기반)

```
1 void SystemClock_Config(void)
2 {
3     RCC_OscInitTypeDef RCC_OscInitStruct = {0};
4     RCC_ClkInitTypeDef RCC_ClkInitStruct = {0};
5
6     /* 1. HSE 활성화 및 PLL 설정 */
7     RCC_OscInitStruct.OscillatorType = RCC_OSCILLATORTYPE_HSE;
8     RCC_OscInitStruct.HSEState = RCC_HSE_ON;
9     RCC_OscInitStruct.PLL.PLLState = RCC_PLL_ON;
10    RCC_OscInitStruct.PLL.PLLSource = RCC_PLLSOURCE_HSE;
11    RCC_OscInitStruct.PLL.PLLMUL = RCC_PLL_MUL9; // 8MHz × 9 = 72MHz
12    HAL_RCC_OscConfig(&RCC_OscInitStruct);
13
14    /* 2. 버스 클럭 설정 */
15    RCC_ClkInitStruct.ClockType = RCC_CLOCKTYPE_HCLK | RCC_CLOCKTYPE_SYSCLK |
16                                RCC_CLOCKTYPE_PCLK1 | RCC_CLOCKTYPE_PCLK2;
17    RCC_ClkInitStruct.SYSCLKSource = RCC_SYSCLKSOURCE_PLLCLK;
18    RCC_ClkInitStruct.AHBCLKDivider = RCC_SYSCLK_DIV1;
19    RCC_ClkInitStruct.APB1CLKDivider = RCC_HCLK_DIV2;
20    RCC_ClkInitStruct.APB2CLKDivider = RCC_HCLK_DIV1;
21    HAL_RCC_ClockConfig(&RCC_ClkInitStruct, FLASH_LATENCY_2);
22 }
```

결과:

- SYSCLK = 72 MHz
- HCLK = 72 MHz
- PCLK1 = 36 MHz
- PCLK2 = 72 MHz
- ADC = 12 MHz

1.4.11 요약

구분	역할
RCC_CR	발진기 활성화 및 준비 상태 확인
RCC_CFGR	클럭 소스 및 분주비 설정
RCC_APB1ENR / RCC_APB2ENR	주변장치 클럭 On/Off 제어
PLL	고속 클럭 생성 (최대 72MHz)
HSI/HSE	내부/외부 클럭 공급원
Reset 기능	전원, Watchdog, 외부 신호 등으로 리셋 제어

STM32F1의 RCC는 시스템 전체의 속도, 안정성, 전력 효율성을 결정짓는 핵심 제어 블록이다.
올바른 RCC 설정은 시스템의 성능과 안정성, 그리고 전력 소모를 모두 좌우한다.

부트 모드 (Boot0/Boot1, ISP, Flash 부팅)

1.5.1 개요

STM32F1 시리즈는 전원을 인가하거나 리셋이 발생했을 때,
어떤 메모리 영역에서 코드를 실행할지를 결정하는 부트 모드(Boot Mode) 메커니즘을 가진다.
이 동작은 **BOOT0** 핀, **BOOT1** 옵션 비트, 그리고 **내장 부트로더(System Memory)** 설정에 따라 결정된다.
즉, STM32는 전원 인가 시점에 다음 세 가지 중 하나의 영역에서 부팅할 수 있다:

부팅 대상	설명
Main Flash Memory (사용자 코드)	사용자가 작성한 펌웨어 실행
System Memory (ST 내장 부트로더)	UART, USB, I2C, CAN을 통한 펌웨어 다운로드
SRAM (RAM 실행)	디버깅, 테스트용 코드 실행

1.5.2 Boot 모드 결정 원리

부트 모드는 **BOOT0** 핀과 **BOOT1** 옵션 비트(또는 일부 MCU의 **BOOT1** 핀)의 조합으로 결정된다.

BOOT1	BOOT0	부팅 대상	설명
x	0	Main Flash Memory	일반 사용자 프로그램 실행
0	1	System Memory	내장 부트로더 실행 (ISP 모드)
1	1	SRAM	RAM에서 코드 실행 (디버깅용)

❄ BOOT1은 최신 STM32 시리즈에서는 **Option Byte**로 대체되어 있으며,
STM32F103에서는 **PB2** 핀이 BOOT1 역할을 겸할 수 있다.

1.5.3 Boot0 핀의 역할

BOOT0 핀은 MCU가 리셋될 때 어느 메모리 구역을 **0x0000_0000 (리셋 벡터)**로 맵핑할지를 결정한다.
즉, CPU는 부팅 직후 주소 **0x0000_0000**에서 명령어를 읽어 실행하기 때문에,
이 위치에 어떤 메모리가 연결되느냐가 “부트 대상”을 결정한다.

Boot0 상태	맵핑 메모리	실행 시작 주소
Low (0)	Main Flash	0x0800 0000
High (1)	System Memory or SRAM (Boot1 설정에 따름)	0x1FFF F000 또는 0x2000 0000

1.5.4 각 부트 모드 상세 설명

(1) Main Flash 부팅 모드

- 가장 일반적인 부트 모드로, 사용자 코드가 저장된 Flash에서 바로 실행된다.
- BOOT0 = 0 일 때 기본 활성화된다.
- MCU의 리셋 벡터가 Flash(0x0800 0000)에 매핑된다.
- CubeIDE, Keil, IAR 등에서 빌드한 펌웨어는 모두 이 위치로 다운로드된다.

실행 순서

1 | Reset → Boot0=0 감지 → Flash(0x0800_0000) 매핑 → Reset Handler 실행

특징

- 전원 인가 후 즉시 사용자 펌웨어 실행
- 대부분의 정상 동작 환경에서 사용

(2) System Memory 부팅 모드 (ISP / DFU 모드)

System Memory는 STMicroelectronics가 공장에서 미리 저장해 둔
내장 부트로더(ROM 부트로더) 코드 영역이다.

BOOT0 = 1, BOOT1 = 0 상태에서 진입할 수 있으며,

ISP(In-System Programming) 또는 DFU(Device Firmware Upgrade) 방식으로 펌웨어를 업로드한다.

인터페이스	지원 가능 여부	비고
USART1	O	표준 ISP (UART Bootloader)
USB	일부 기종	DFU 모드
CAN	O	산업용 통신 업데이트
I2C	O	확장 부트로더

실행 순서

1 | Reset → Boot0=1, Boot1=0 → System Memory(0x1FFF_F000) 매핑
2 | → 내장 부트로더 실행 → 외부 인터페이스 대기
3 | → PC에서 STM32CubeProgrammer로 펌웨어 다운로드

활용 예시

- 펌웨어 손상 시 복구
- 생산라인 펌웨어 업데이트
- UART/USB 기반 무선 OTA(Over The Air) 구현

(3) SRAM 부팅 모드

BOOT0 = 1, BOOT1 = 1 상태에서 진입한다.

이 모드는 Flash 대신 **내부 SRAM(0x2000 0000)** 을 리셋 벡터로 매핑하여 RAM에 로드된 코드를 직접 실행할 수 있다.

특징

- 일반적으로 **디버깅용** 또는 **RAM 실행 테스트용**으로 사용
- Flash를 사용하지 않고 메모리에서 바로 명령 실행 가능
- ST-Link, JTAG 디버깅 환경에서 주로 활용

실행 순서

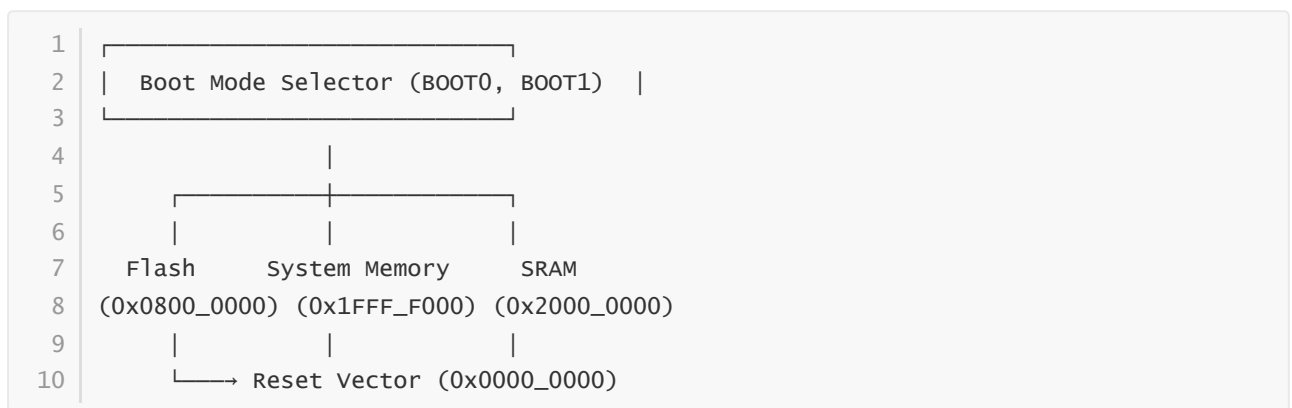
- 1 | Reset → Boot0=1, Boot1=1 → SRAM(0x2000_0000) 매핑
- 2 | → 디버거에서 Load & Run 수행

1.5.5 부트 메모리 매핑 구조

아래는 부트 모드에 따른 메모리 매핑 변화를 나타낸다.

부트 모드	주소 0x0000_0000에 매핑되는 메모리	실제 물리 주소
Flash 부팅	Flash	0x0800 0000
System Memory 부팅	System Memory	0x1FFF F000
SRAM 부팅	SRAM	0x2000 0000

개념도



1.5.6 Option Bytes를 이용한 부트 설정

BOOT1은 실제 핀 외에도 **플래시 Option Byte** 로 설정할 수 있다.

이 방식은 하드웨어 핀을 변경하지 않고도 부트 모드를 변경할 수 있게 해준다.

항목	설명
주소	0x1FFFF800
관련 비트	nBOOT1, nBOOT0
설정 방법	STM32CubeProgrammer → Option Bytes 메뉴
사용 목적	소프트웨어 기반 부팅 모드 전환 (예: OTA 시)

1.5.7 ISP (In-System Programming) 모드 동작

System Memory로 부팅 시, MCU는 자동으로 내장 부트로더를 실행하여 특정 통신 인터페이스(UART, I²C, USB 등)를 모니터링한다.

UART 부트로더 예시 (USART1)

- MCU 전원 인가
- UART RX 라인 감시
- 0x7F (Sync Byte) 수신 시 부트로더 활성화
- 이후 STMicroelectronics 부트 프로토콜에 따라 펌웨어 다운로드 수행

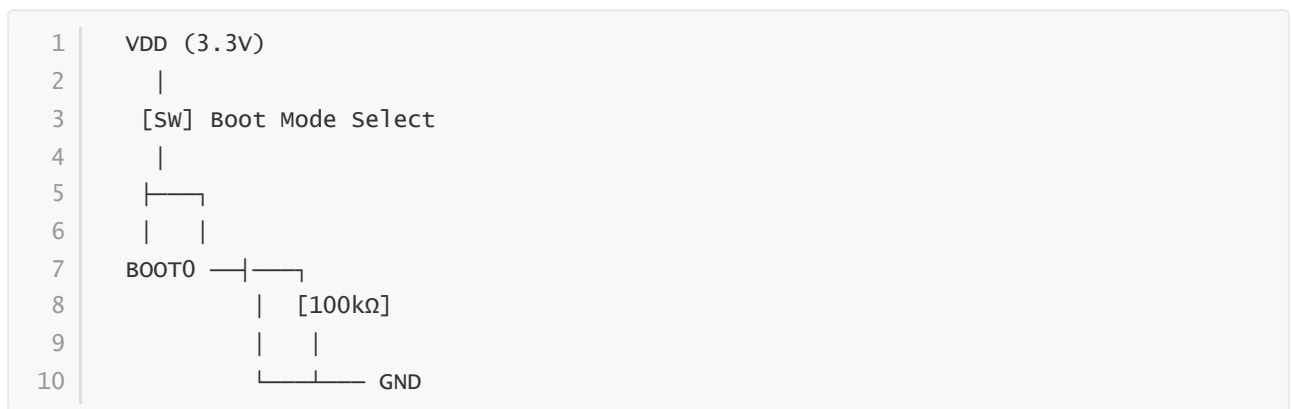
도구 예시

- STM32CubeProgrammer
- Flash Loader Demonstrator (구버전)

1.5.8 Boot0 하드웨어 설계 예시

일반적으로 Boot0 핀은 **풀다운 저항(100kΩ 정도)**으로 기본 Flash 부팅 상태를 유지하고, 필요할 때만 점퍼나 스위치를 이용해 HIGH로 전환하도록 설계한다.

회로 예시:



- 스위치 OFF → Boot0=0 → Flash 부팅
- 스위치 ON → Boot0=1 → ISP/System Memory 진입

1.5.9 Boot 관련 레지스터

레지스터	이름	역할
FLASH_OBR	Option Byte Register	부트 관련 옵션 상태 확인
RCC_CSR	Reset Cause Register	리셋 원인 확인
SYSCFG_MEMRMP	Memory Remap Register	부트 후 메모리 리매핑 제어 (일부 MCU)

1.5.10 요약

Boot0	Boot1	실행 메모리	주요 용도
0	X	Flash	사용자 펌웨어 실행
1	0	System Memory	ISP, DFU 펌웨어 업데이트
1	1	SRAM	디버깅 및 테스트 실행

STM32의 부트 모드 시스템은 “개발 → 생산 → 유지보수” 전 과정을 하나의 MCU에서 완결할 수 있게 설계되어 있다. 이를 통해 외부 프로그래머 없이도 UART/USB를 통한 펌웨어 다운로드 및 복구가 가능하며, 내장 Bootloader를 활용하면 완전한 무인(無人) 펌웨어 관리 시스템을 구현할 수 있다.

1.2 펌웨어 프로젝트 구성

- startup_stm32f1xx.s — Reset Vector & 초기화 루틴

1.6.1 개요

startup_stm32f1xx.s 파일은 STM32 프로젝트의 부팅 진입점(Entry Point) 을 정의하는 어셈블리 코드다. 이 파일은 Reset 후 MCU가 처음 실행하는 코드를 포함하며, C 코드(main() 함수)가 실행되기 전의 저수준 초기화 절차를 담당한다.

즉, 이 파일은 MCU가 전원을 켜거나 리셋될 때 스택 초기화 → 벡터 테이블 설정 → 데이터 섹션 복사 → BSS 초기화 → main() 호출 순으로 시스템을 준비한다.

1.6.2 파일 구성 요약

구역	주요 역할
벡터 테이블(Vector Table)	예외 및 인터럽트 핸들러 주소 목록 정의
Reset_Handler	리셋 발생 시 실행되는 초기화 루틴
데이터 초기화 루틴	.data 섹션 복사, .bss 섹션 클리어

구역	주요 역할
SystemInit 호출	클럭/PLL 설정, 하드웨어 초기화
main() 진입	사용자 애플리케이션 실행 시작

1.6.3 Reset Vector와 벡터 테이블 구조

MCU의 부팅 주소(보통 0x0800_0000)에는 **Interrupt Vector Table** 이 존재하며, 여기에는 각 예외(Interrupt)에 대한 진입 주소가 순서대로 나열되어 있다.

예시:

1	.section .isr_vector,"a",%progbits
2	g_pfnVectors:
3	.word _estack /* 초기 스택 포인터 */
4	.word Reset_Handler /* Reset */
5	.word NMI_Handler /* NMI */
6	.word HardFault_Handler /* HardFault */
7	.word MemManage_Handler
8	.word BusFault_Handler
9	.word UsageFault_Handler
10	.word 0, 0, 0, 0 /* Reserved */
11	.word SVC_Handler
12	.word DebugMon_Handler
13	.word 0 /* Reserved */
14	.word PendSV_Handler
15	.word SysTick_Handler
16	/* 이후 외부 인터럽트들 */
17	.word WWDG_IRQHandler
18	.word PVD_IRQHandler
19	.word TAMPER_IRQHandler
20	.word RTC_IRQHandler
21	...

주요 특징

- 첫 번째 항목(`_estack`)은 초기 스택 포인터(Stack Pointer) 주소를 나타낸다.
MCU는 리셋 직후 이 값을 **SP 레지스터**에 로드한다.
- 두 번째 항목은 **Reset_Handler** 주소이다.
MCU는 리셋이 발생하면 자동으로 이 핸들러를 실행한다.

1.6.4 Reset_Handler (핵심 초기화 루틴)

`Reset_Handler` 는 MCU가 리셋된 직후 실행되는 첫 번째 코드이며, 다음과 같은 단계를 순차적으로 수행한다.

예시 (단순화된 어셈블리)

```

1 Reset_Handler:
2     ldr    sp, =_estack          /* 1. 스택 초기화 */
3     bl     SystemInit           /* 2. 시스템 클럭/PLL 설정 */
4     bl     __libc_init_array    /* 3. C++ 전역 생성자 호출 */
5     bl     main                 /* 4. 사용자 코드 진입 */
6     bx     lr                   /* 5. 종료 (일반적으로 실행되지 않음) */

```

실제 동작 절차 (상세):

1. `_estack` 주소를 SP 레지스터에 로드하여 스택 초기화
2. `SystemInit()` 호출 — 시스템 클럭, PLL, Flash Wait State, 벡터 테이블 리맵 설정
3. `.data` 섹션을 Flash → SRAM으로 복사
4. `.bss` 섹션을 0으로 초기화
5. C/C++ 전역 객체 생성자 호출 (`__libc_init_array`)
6. `main()` 호출 → 애플리케이션 코드로 진입

1.6.5 데이터 초기화 섹션

C 코드에서 사용하는 전역 변수들은 Flash에 저장되어 있다가 부팅 시 SRAM으로 복사되어야 한다.

이 작업을 어셈블리에서 다음과 같이 수행한다.

```

1  /* Copy .data section from Flash to SRAM */
2  ldr    r0, =_sdata             /* SRAM 내 .data 시작 */
3  ldr    r1, =_edata             /* SRAM 내 .data 끝 */
4  ldr    r2, =_sdata             /* Flash 내 초기값 시작 */
5  Loop_Copy_Data:
6  cmp    r0, r1
7  ittt   lt
8  ldr!t  r3, [r2], #4
9  str!t  r3, [r0], #4
10  blt    Loop_Copy_Data

```

- `_sdata`: Flash에 저장된 초기화된 데이터의 시작 주소
- `_sdata`: SRAM에 복사될 대상 주소
- `_edata`: SRAM 내 데이터 끝 주소

1.6.6 BSS 섹션 초기화

초기화되지 않은 전역/정적 변수(`.bss`)는 모두 0으로 초기화되어야 한다.

```

1  /* Zero fill the .bss section */
2  ldr    r0, =_sbss
3  ldr    r1, =_ebss
4  movs   r2, #0
5  Loop_Zero_BSS:
6  cmp    r0, r1
7  it     lt
8  strlt  r2, [r0], #4
9  blt    Loop_Zero_BSS

```

이 과정이 완료되면 C 언어 표준에 따라
전역/정적 변수가 올바르게 초기화된 상태가 된다.

1.6.7 SystemInit() 함수 호출

`SystemInit()` 함수는 `system_stm32f1xx.c` 파일에 정의되어 있으며,
하드웨어 클럭 관련 설정을 수행한다.

예를 들어:

- HSE / PLL 활성화
- SYSCLK 소스 변경
- Flash Wait State 설정
- AHB/APB 분주비 적용
- Vector Table Offset 설정

⚙ 즉, `Reset_Handler`는 저수준 초기화 담당,
`SystemInit()` 은 클럭 및 하드웨어 설정 담당으로 역할이 분리되어 있다.

1.6.8 main() 호출 및 프로그램 실행

모든 초기화가 완료되면,
`main()` 함수를 호출하여 사용자가 작성한 애플리케이션으로 제어를 넘긴다.

```

1  bl main
2  Infinite_Loop:
3  b Infinite_Loop /* main()이 종료되면 무한 루프 대기 */

```

일반적으로 임베디드 프로그램은 `main()` 이 종료되지 않으며,
종료 시 MCU는 위 루프에 진입한다.

1.6.9 예외 핸들러 기본 구조

모든 인터럽트 핸들러는 기본적으로 약한 심볼(`.weak`)로 정의되어 있다.
사용자가 동일한 이름의 함수를 정의하면 자동으로 오버라이드된다.

```

1  .weak NMI_Handler
2  .thumb_set NMI_Handler, Default_Handler
3
4  .weak HardFault_Handler
5  .thumb_set HardFault_Handler, Default_Handler

```

즉, 사용자가 `void HardFault_Handler(void)` 를 C에서 정의하면, 어셈블리의 기본 핸들러 대신 사용자의 함수가 실행된다.

1.6.10 스택 및 힙 설정

링커 스크립트(`STM32F103C8_FLASH.ld`)에서 스택(Stack)과 힙(Heap)의 시작 및 크기를 정의한다.

```

1  _estack = 0x20005000;    /* Stack top address */
2  _Min_Heap_Size = 0x200; /* 512 bytes */
3  _Min_Stack_Size = 0x400; /* 1 KB */

```

`_estack` 은 벡터 테이블의 첫 번째 항목으로 사용되어 리셋 시 SP 초기값으로 로드된다.

1.6.11 전체 부트 시퀀스 요약

단계	수행 모듈	동작 내용
1	하드웨어 (RCC, Boot0)	부트 메모리 결정 (Flash/System/SRAM)
2	Cortex-M3	SP ← <code>_estack</code> 로드
3	Cortex-M3	PC ← <code>Reset_Handler</code> 로드
4	<code>startup_stm32f1xx.s</code>	데이터 복사, BSS 초기화
5	<code>startup_stm32f1xx.s</code>	<code>SystemInit()</code> 호출
6	<code>startup_stm32f1xx.s</code>	<code>__libc_init_array()</code> 실행 (C++용)
7	<code>startup_stm32f1xx.s</code>	<code>main()</code> 진입
8	<code>main.c</code>	사용자 코드 실행 시작

1.6.12 요약

구성 요소	역할
Vector Table	모든 예외 및 인터럽트 진입점 정의
Reset_Handler	부팅 직후 전체 초기화 절차 수행

구성 요소	역할
SystemInit()	클럭 및 PLL 설정
데이터 복사 루틴	Flash → SRAM 데이터 전송
BSS 초기화 루틴	미초기화 변수 0으로 세팅
main() 호출	애플리케이션 진입
Default_Handler	미구현 인터럽트 무한 루프 처리

👉 정리하자면,

`startup_stm32f1xx.s` 는 MCU의 생명주기를 여는 가장 첫 번째 코드다.

이 코드를 이해하면 STM32의 실행 구조, 메모리 초기화, 인터럽트 동작 원리를 모두 이해할 수 있다.

• system_stm32f1xx.c — PLL 및 클럭 초기화

1.7.1 개요

`system_stm32f1xx.c` 는 STM32F1 시리즈 MCU의 **시스템 클럭(System Clock)** 과 **PLL(Phase-Locked Loop)** 초기화를 담당하는 핵심 소스 파일이다.

이 파일은 **Reset 이후, C 런타임 진입 전에 호출되는 `SystemInit()` 함수**를 포함하며 MCU의 동작 속도, 버스 클럭 분주비, Flash 접근 대기시간, PLL 배율 설정 등을 구성한다.

즉, `system_stm32f1xx.c` 는 MCU가 “얼마나 빠르게, 어떤 기준으로” 작동할지를 결정하는 **클럭 인프라의 초기 설정 파일**이다.

1.7.2 주요 역할

기능	설명
HSE/HSI 클럭 소스 선택	외부 크리스탈(HSE) 또는 내부 RC 오실레이터(HSI) 사용 결정
PLL 설정	입력 주파수를 증폭시켜 최대 72 MHz 시스템 클럭 생성
AHB/APB 분주비 설정	CPU, 주변장치 버스 클럭 비율 조정
Flash Wait State 설정	CPU 속도에 따른 Flash 접근 타이밍 보정
시스템 클럭 전환	PLL 출력을 SYSCLK 소스로 설정
SystemCoreClock 변수 갱신	CMSIS에서 사용하는 글로벌 클럭 주파수 변수 업데이트

1.7.3 주요 전역 변수

```

1 uint32_t SystemCoreClock = 72000000; // 현재 시스템 클럭 (Hz 단위)
2 const uint8_t AHBPrescTable[16] = {0,0,0,0,1,2,3,4,6,7,8,9,10,11,12,13};
3 const uint8_t APBPrescTable[8]  = {0,0,0,0,1,2,3,4};

```

이 변수들은 CMSIS 드라이버 계층에서

`HAL_GetTickFreq()`, `HAL_RCC_GetSysClockFreq()` 등의 함수가 사용할 수 있도록 유지된다.

1.7.4 SystemInit() 함수의 전체 구조

```
1 void SystemInit(void)
2 {
3     /* 1. FPU 설정 (F1에는 없음, 후속 시리즈용) */
4
5     /* 2. Reset 이후 기본 클록 상태 복원 */
6     RCC->CR |= (uint32_t)0x00000001;           // HSI ON
7     RCC->CFGR = 0x00000000;                     // 기본 분주 설정 (1:1)
8     RCC->CR &= (uint32_t)0xFE6FFFFF;           // HSE/PLL OFF
9     RCC->PLLCFGR = 0x00000000;                  // PLL 설정 초기화
10    RCC->CIR = 0x00000000;                       // 클록 인터럽트 OFF
11
12    /* 3. 시스템 클록 구성 */
13    SetSysClock();                               // 사용자 정의 PLL/HSE 설정 루틴
14
15    /* 4. 벡터 테이블 리맵 설정 */
16    #ifdef VECT_TAB_SRAM
17        SCB->VTOR = SRAM_BASE | VECT_TAB_OFFSET;
18    #else
19        SCB->VTOR = FLASH_BASE | VECT_TAB_OFFSET;
20    #endif
21 }
```

`SystemInit()` 은 MCU 부팅 시 `Reset_Handler` 에서 호출되며,
RCC 레지스터를 초기화하고, `SetSysClock()` 함수를 통해 PLL을 구성한다.

1.7.5 SetSysClock() 내부 구조

이 함수는 실제로 HSE, PLL, 분주비 등을 조정하여
최종적으로 **SYSCLK = 72 MHz** 를 달성한다.

일반적인 STM32F103 구성 예시:

```
1 static void SetSysClock(void)
2 {
3     RCC_OscInitTypeDef RCC_OscInitStruct;
4     RCC_ClkInitTypeDef RCC_ClkInitStruct;
5
6     /* 1. 외부 크리스탈(HSE) 활성화 */
7     RCC_OscInitStruct.OscillatorType = RCC_OSCILLATORTYPE_HSE;
8     RCC_OscInitStruct.HSEState = RCC_HSE_ON;
9     RCC_OscInitStruct.HSEPredivValue = RCC_HSE_PREDIV_DIV1;
10
11    /* 2. PLL 설정: HSE(8 MHz) × 9 = 72 MHz */
12    RCC_OscInitStruct.PLL.PLLState = RCC_PLL_ON;
13    RCC_OscInitStruct.PLL.PLLSource = RCC_PLLSOURCE_HSE;
```

```

14     RCC_OscInitStruct.PLL.PLLMUL = RCC_PLL_MUL9;
15     HAL_RCC_OscConfig(&RCC_OscInitStruct);
16
17     /* 3. 버스 클럭 설정 */
18     RCC_ClkInitStruct.ClockType = RCC_CLOCKTYPE_SYSCLK |
19                                     RCC_CLOCKTYPE_HCLK |
20                                     RCC_CLOCKTYPE_PCLK1 |
21                                     RCC_CLOCKTYPE_PCLK2;
22     RCC_ClkInitStruct.SYSCLKSource = RCC_SYSCLKSOURCE_PLLCLK;
23     RCC_ClkInitStruct.AHBCLKDivider = RCC_SYSCLK_DIV1;    // HCLK = 72 MHz
24     RCC_ClkInitStruct.APB1CLKDivider = RCC_HCLK_DIV2;    // PCLK1 = 36 MHz
25     RCC_ClkInitStruct.APB2CLKDivider = RCC_HCLK_DIV1;    // PCLK2 = 72 MHz
26     HAL_RCC_ClockConfig(&RCC_ClkInitStruct, FLASH_LATENCY_2);
27 }

```

1.7.6 PLL 구성 원리

PLL 입력 및 출력 관계

1 $PLLCLK = (\text{입력 클럭} / PREDIV) \times MUL$

STM32F103의 PLL 설정 제한사항:

항목	값
PLL 입력 주파수	4 MHz ~ 16 MHz
PLL 출력 주파수	최대 72 MHz
PLL 배율 (MUL)	$\times 2 \sim \times 16$
PLL 입력 소스	HSI/2 또는 HSE

예시 (HSE 8 MHz 사용):

```

1  입력: 8 MHz → PLLMUL  $\times 9$  → 출력: 72 MHz
2  SYSCLK = 72 MHz
3  HCLK = 72 MHz
4  PCLK1 = 36 MHz
5  PCLK2 = 72 MHz

```

1.7.7 Flash Wait State 설정

CPU 클럭이 48 MHz 이상일 경우, Flash 메모리 접근에 대기 사이클(Wait State)을 추가해야 한다.

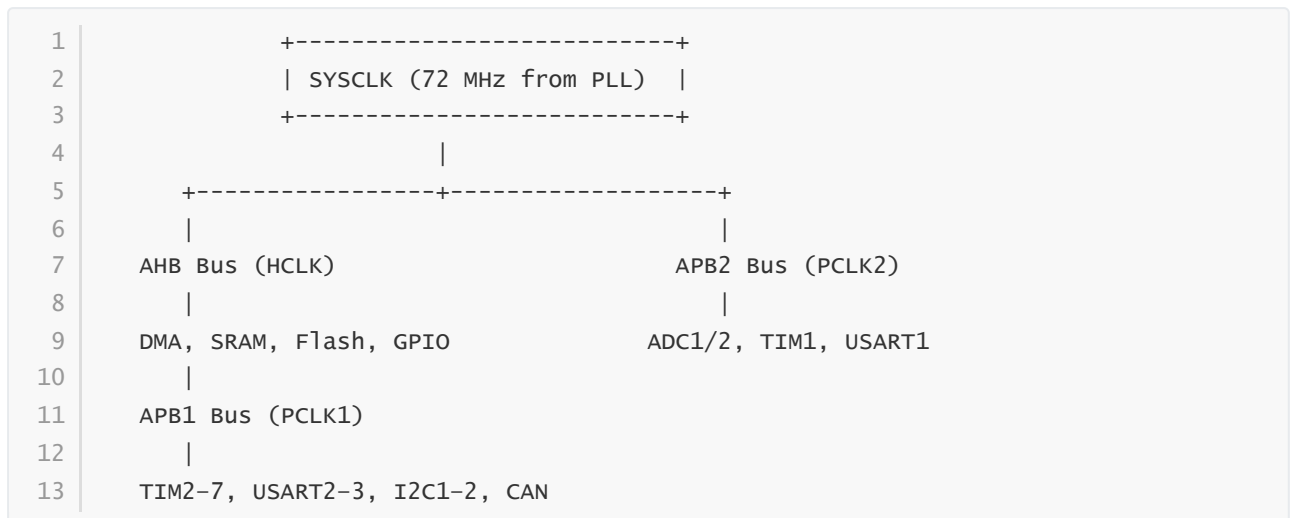
FLASH->ACR 레지스터를 통해 설정한다:

SYSCLK 주파수	Wait State	ACR 설정
0 – 24 MHz	0 WS	FLASH_LATENCY_0
24 – 48 MHz	1 WS	FLASH_LATENCY_1
48 – 72 MHz	2 WS	FLASH_LATENCY_2

HAL_RCC_ClockConfig() 함수가 내부적으로 이 설정을 자동 수행한다.

1.7.8 버스 클록 분주 구조

클록 트리는 다음과 같이 분배된다.



버스	분주비	주파수	용도
AHB (HCLK)	÷1	72 MHz	CPU, DMA, SRAM
APB1 (PCLK1)	÷2	36 MHz	저속 주변장치
APB2 (PCLK2)	÷1	72 MHz	고속 주변장치

1.7.9 HSE 대체 시나리오

HSE(8 MHz 크리스털)가 존재하지 않거나 고장 난 경우,
시스템은 자동으로 내부 RC 오실레이터(HSI, 8 MHz)를 사용하도록 설계할 수 있다.

```

1  if (HAL_RCC_OscConfig(&RCC_OscInitStruct) != HAL_OK)
2  {
3      /* HSE 실패 시 HSI 사용 */
4      RCC_OscInitStruct.OscillatorType = RCC_OSCILLATORTYPE_HSI;
5      RCC_OscInitStruct.HSISState = RCC_HSI_ON;
6      RCC_OscInitStruct.HSICalibrationValue = RCC_HSICALIBRATION_DEFAULT;
7      HAL_RCC_OscConfig(&RCC_OscInitStruct);
8  }

```

이렇게 하면 외부 크리스탈 고장 시에도 MCU가 안전하게 부팅된다.

1.7.10 SystemCoreClockUpdate() 함수

`SystemCoreClock` 변수는 CMSIS 표준 변수로,
현재 CPU 클럭 주파수를 실시간으로 저장한다.

`HAL_Delay()` 나 `SysTick_Config()` 함수는
이 값을 참조하여 타이밍을 계산한다.

```
1 void SystemCoreClockUpdate(void)
2 {
3     uint32_t tmp, pllmu11, pllsource;
4
5     tmp = RCC->CFGR & RCC_CFGR_SWS; // SYSCLK 소스 확인
6     switch (tmp)
7     {
8         case RCC_CFGR_SWS_HSI: SystemCoreClock = HSI_VALUE; break;
9         case RCC_CFGR_SWS_HSE: SystemCoreClock = HSE_VALUE; break;
10        case RCC_CFGR_SWS_PLL:
11            pllmu11 = (RCC->CFGR & RCC_CFGR_PLLMULL) >> 18;
12            pllsource = (RCC->CFGR & RCC_CFGR_PLLSRC);
13            if (pllsource == 0x00)
14                SystemCoreClock = (HSI_VALUE >> 1) * pllmu11;
15            else
16                SystemCoreClock = (HSE_VALUE / 1) * pllmu11;
17            break;
18        default:
19            SystemCoreClock = HSI_VALUE;
20            break;
21    }
22
23    /* AHB 분주 적용 */
24    tmp = AHBPrescTable[((RCC->CFGR & RCC_CFGR_HPRE) >> 4)];
25    SystemCoreClock >>= tmp;
26 }
```

이 함수는 클럭 설정이 변경된 후 반드시 호출해야
HAL 레이어의 시간 관련 함수들이 정확히 동작한다.

1.7.11 PLL 설정 예시 요약

입력 클럭	PLL 소스	배율	SYSCLK	Flash Wait State
8 MHz (HSE)	HSE	×9	72 MHz	2 WS
8 MHz (HSI/2)	HSI/2	×16	64 MHz	2 WS
12 MHz (HSE)	HSE	×6	72 MHz	2 WS

입력 클럭	PLL 소스	배율	SYSCLK	Flash Wait State
16 MHz (HSE)	HSE	×4.5	72 MHz	2 WS

1.7.12 요약

항목	역할
<code>SystemInit()</code>	Reset 이후 시스템 클럭 초기화 진입점
<code>SetSysClock()</code>	PLL 및 분주비 설정 루틴
<code>SystemCoreClock</code>	현재 CPU 클럭 주파수 저장 변수
<code>SystemCoreClockUpdate()</code>	클럭 변경 후 주파수 갱신
<code>FLASH_LATENCY</code>	Flash 접근 대기 사이클 설정
<code>RCC_CFGR</code>	클럭 소스 및 버스 분주비 제어
<code>RCC_CR</code>	오실레이터 활성화 제어 (HSI/HSE/PLL)

STM32F1의 `system_stm32f1xx.c` 는

시스템 전체의 동작 타이밍을 결정하는 근본 제어 모듈이다.

이 파일을 명확히 이해하면, 클럭 안정성, 전력 소비, 성능 최적화를 하드웨어 수준에서 직접 제어할 수 있다.

• stm32f1xx_it.c — 인터럽트 핸들러 정의

1.8.1 개요

`stm32f1xx_it.c` 파일은 STM32F1 시리즈의

예외(Exception) 및 외부 인터럽트(Interrupt) 핸들러 함수를 정의하는 소스 파일이다.

이 파일은 `startup_stm32f1xx.s` 의 벡터 테이블(Vector Table)에 등록된 각 인터럽트 엔트리의 함수 본체(실제 동작 내용)를 포함한다.

즉,

`startup_stm32f1xx.s` 가 “인터럽트 이름과 위치”를 지정한다면,
`stm32f1xx_it.c` 는 “그 인터럽트가 발생했을 때 실제 수행할 코드”를 구현한다.

1.8.2 파일 구조

기본 형태:

```

1  #include "main.h"
2  #include "stm32f1xx_it.h"
3
4  /* 외부 선언: main.c 또는 HAL 드라이버에서 정의된 핸들들 */

```

```

5 extern TIM_HandleTypeDef htim1;
6 extern ADC_HandleTypeDef hadc1;
7 extern UART_HandleTypeDef huart1;
8
9 /*****
10 /*          Cortex-M3 Processor Interruption and Exception Handlers          */
11 /*****
12 void NMI_Handler(void) {}
13 void HardFault_Handler(void)
14 {
15     while (1) {}    /* 시스템 치명적 오류 발생 시 무한 루프 */
16 }
17 void MemManage_Handler(void) { while (1) {} }
18 void BusFault_Handler(void) { while (1) {} }
19 void UsageFault_Handler(void) { while (1) {} }
20 void SVC_Handler(void) {}
21 void DebugMon_Handler(void) {}
22 void PendSV_Handler(void) {}
23 void SysTick_Handler(void)
24 {
25     HAL_IncTick();          /* HAL Tick 증가 */
26 }
27
28 /*****
29 /* STM32F1xx Peripheral Interrupt Handlers          */
30 /*****
31 void TIM1_UP_IRQHandler(void)
32 {
33     HAL_TIM_IRQHandler(&htim1); /* HAL의 타이머 인터럽트 처리 루틴 호출 */
34 }
35
36 void ADC1_2_IRQHandler(void)
37 {
38     HAL_ADC_IRQHandler(&hadc1); /* ADC 변환 완료 인터럽트 처리 */
39 }
40
41 void USART1_IRQHandler(void)
42 {
43     HAL_UART_IRQHandler(&huart1); /* UART 통신 인터럽트 처리 */
44 }
45
46 void EXTI9_5_IRQHandler(void)
47 {
48     HAL_GPIO_EXTI_IRQHandler(GPIO_PIN_8); /* EXTI 라인(예: 초음파 ECHO 핀) */
49 }

```

1.8.3 Cortex-M3 코어 예외 핸들러

(1) NMI_Handler()

- **Non-Maskable Interrupt (마스크 불가 인터럽트)**
- 주로 외부 하드웨어 Fault (예: 클럭 실패, Watchdog 등)에 의해 발생
- 디버깅용으로 사용되며 일반적으로 비워둔다.

(2) HardFault_Handler()

- 시스템 내에서 복구 불가능한 오류 발생 시 진입
(예: 잘못된 메모리 접근, 0으로 나누기 등)
- MCU가 즉시 중단되고, 디버깅 중이라면
HardFault 스택 프레임 분석을 통해 원인 추적 가능

예시:

```
1 void HardFault_Handler(void)
2 {
3     printf("HardFault 발생!\n");
4     while (1); // 시스템 멈춤
5 }
```

(3) MemManage_Handler(), BusFault_Handler(), UsageFault_Handler()

- 각각 메모리 관리, 버스 접근, 명령어 사용 오류 시 발생
- Cortex-M3의 Fault subsystem에 의해 트리거됨
- 대부분의 프로젝트에서는 무한 루프로 처리

(4) SysTick_Handler()

- HAL의 시간 관리 루틴에서 핵심 역할 수행
- 1ms 주기로 호출되어 `HAL_IncTick()` 을 실행함으로써
`HAL_Delay()` 및 FreeRTOS Tick 기반 시간 관리가 동작

1.8.4 외부 인터럽트 핸들러 (EXTI)

STM32는 GPIO 핀에 대해 **외부 인터럽트 라인(EXTI Line)** 을 제공한다.

`stm32f1xx_it.c`에서는 이러한 핀 트리거 이벤트를 처리한다.

예시 (HC-SR04 ECHO 핀 감지):

```

1 void EXTI9_5_IRQHandler(void)
2 {
3     HAL_GPIO_EXTI_IRQHandler(GPIO_PIN_8);
4 }
5
6 void HAL_GPIO_EXTI_Callback(uint16_t GPIO_Pin)
7 {
8     if (GPIO_Pin == GPIO_PIN_8)
9     {
10         /* 초음파 Echo Rising/Falling 처리 */
11         handle_echo_signal();
12     }
13 }

```

- `EXTI9_5_IRQHandler()` → NVIC가 호출
- `HAL_GPIO_EXTI_IRQHandler()` → HAL이 콜백으로 전달
- `HAL_GPIO_EXTI_Callback()` → 사용자가 처리

⚙ 콜백 기반 구조 덕분에, 사용자 코드는 인터럽트 내부에서 직접 레지스터를 다루지 않아도 된다.

1.8.5 타이머 인터럽트 핸들러

타이머(TIM)는 업데이트 이벤트(Overflow) 나 입력 캡처(Input Capture) 시 인터럽트를 발생시킨다.

예시:

```

1 void TIM1_UP_IRQHandler(void)
2 {
3     HAL_TIM_IRQHandler(&htim1);
4 }
5
6 void HAL_TIM_PeriodElapsedCallback(TIM_HandleTypeDef *htim)
7 {
8     if (htim->Instance == TIM1)
9     {
10         /* 1ms 주기 LED 토글 */
11         HAL_GPIO_TogglePin(GPIOC, GPIO_PIN_13);
12     }
13 }

```

- 인터럽트 벡터 → `TIM1_UP_IRQHandler()` 호출
- HAL 내부에서 상태 비트 확인 → 콜백 호출
- `HAL_TIM_PeriodElapsedCallback()` 에서 사용자 동작 수행

1.8.6 ADC 인터럽트 핸들러

ADC 변환 완료(End Of Conversion) 시 인터럽트를 발생시킨다.

```
1 void ADC1_2_IRQHandler(void)
2 {
3     HAL_ADC_IRQHandler(&hadc1);
4 }
5
6 void HAL_ADC_ConvCpltCallback(ADC_HandleTypeDef *hadc)
7 {
8     if (hadc->Instance == ADC1)
9     {
10         uint16_t value = HAL_ADC_GetValue(hadc);
11         process_voltage(value);
12     }
13 }
```

- 변환 완료 플래그(EOC)가 세트되면 `ADC1_2_IRQHandler()` 호출
- HAL이 내부적으로 상태 확인 후 사용자 콜백 호출

1.8.7 UART 인터럽트 핸들러

UART 송신/수신, 에러, IDLE 감지 등 다양한 인터럽트를 관리한다.

```
1 void USART1_IRQHandler(void)
2 {
3     HAL_UART_IRQHandler(&huart1);
4 }
5
6 void HAL_UART_RxCpltCallback(UART_HandleTypeDef *huart)
7 {
8     if (huart->Instance == USART1)
9     {
10         process_rx_data(rx_buffer);
11         HAL_UART_Receive_IT(huart, rx_buffer, 1); // 재수신 활성화
12     }
13 }
```

핵심 흐름

```
1 | USART1_IRQHandler → HAL_UART_IRQHandler → HAL_UART_RxCpltCallback
```

UART의 인터럽트 기반 수신(`HAL_UART_Receive_IT`)은 CPU가 대기하지 않고 즉시 다음 데이터 처리를 가능하게 한다.

1.8.8 기타 인터럽트 예시

인터럽트	함수 이름	주요 동작
DMA 전송 완료	<code>DMA1_ChannelX_IRQHandler()</code>	<code>HAL_DMA_IRQHandler()</code>
I2C 이벤트	<code>I2C1_EV_IRQHandler()</code>	<code>HAL_I2C_EV_IRQHandler()</code>
I2C 에러	<code>I2C1_ER_IRQHandler()</code>	<code>HAL_I2C_ER_IRQHandler()</code>
RTC 알람	<code>RTC_Alarm_IRQHandler()</code>	<code>HAL_RTC_Alarm_IRQHandler()</code>
USB / CAN	<code>USB_LP_CAN1_RX0_IRQHandler()</code>	<code>HAL_PCD_IRQHandler()</code> / <code>HAL_CAN_IRQHandler()</code>

1.8.9 HAL 콜백 구조 요약

모든 HAL 기반 인터럽트는 다음 패턴으로 동작한다.

```
1 | NVIC → [Handler()] → HAL_xxx_IRQHandler() → HAL_xxx_Callback()
```

단계	함수	정의 위치	설명
①	<code>_IRQHandler()</code>	<code>stm32f1xx_it.c</code>	실제 인터럽트 엔트리
②	<code>HAL_xxx_IRQHandler()</code>	HAL 드라이버 내부	상태 플래그 검사 및 클리어
③	<code>HAL_xxx_Callback()</code>	사용자 코드	애플리케이션 동작 수행

예시

```
1 | EXTI → EXTI9_5_IRQHandler() → HAL_GPIO_EXTI_IRQHandler() → HAL_GPIO_EXTI_Callback()
```

1.8.10 사용자 정의 인터럽트 추가 절차

1. NVIC에 인터럽트 활성화

```
1 | HAL_NVIC_SetPriority(EXTI9_5_IRQn, 1, 0);
2 | HAL_NVIC_EnableIRQ(EXTI9_5_IRQn);
```

2. 인터럽트 핸들러 함수 작성

```
1 | void EXTI9_5_IRQHandler(void)
2 | {
3 |     HAL_GPIO_EXTI_IRQHandler(GPIO_PIN_8);
4 | }
```

3. 콜백 정의

```
1 void HAL_GPIO_EXTI_Callback(uint16_t GPIO_Pin)
2 {
3     if (GPIO_Pin == GPIO_PIN_8)
4         handle_echo_signal();
5 }
```

1.8.11 예외 핸들러 동작 요약

핸들러	역할	기본 동작
NMI_Handler	비마스크 인터럽트	일반적으로 무시
HardFault_Handler	치명적 오류	무한 루프 정지
MemManage_Handler	MPU 관련 오류	무한 루프
BusFault_Handler	버스 접근 오류	무한 루프
UsageFault_Handler	잘못된 명령/상황	무한 루프
SysTick_Handler	시스템 틱 증가	HAL_IncTick() 호출

1.8.12 요약

구성 요소	역할
Cortex-M3 예외 핸들러	시스템 레벨 Fault 및 Tick 관리
주변장치 인터럽트 핸들러	TIM, ADC, UART 등 하드웨어 이벤트 처리
HAL_xxx_IRQHandler()	HAL 드라이버 내 상태 확인 및 콜백 트리거
사용자 콜백 함수	애플리케이션 단의 실제 동작 수행
NVIC 설정	인터럽트 우선순위 및 활성화 제어

결론적으로 `stm32f1xx_it.c` 는 STM32 시스템의 “이벤트 반응 레이어(Event Reaction Layer)” 이며, 모든 하드웨어 신호가 소프트웨어 동작으로 연결되는 핵심 경로다. 이 파일을 정확히 이해하면 실시간 반응형 MCU 제어 구조의 근본을 완전히 파악할 수 있다.

• syscalls.c / sysmem.c — printf, malloc 등 재구현

1.9.1 개요

`syscalls.c`와 `sysmem.c`는 C 표준 라이브러리(newlib)를 STM32의 Bare-metal 환경에 맞게 재정의(Stub)한 파일이다.

임베디드 MCU는 일반적인 운영체제(OS) 없이 동작하기 때문에,

`printf`, `malloc`, `free`, `exit`, `read`, `write` 등과 같은 시스템 콜(System Call)을 커널 대신 개발자가 직접 정의해 주어야 한다.

즉,

이 두 파일은 “운영체제가 없는 마이크로컨트롤러 환경에서 표준 C 함수를 정상적으로 작동시키기 위한 최소한의 OS 대체 인터페이스”이다.

1.9.2 역할 요약

파일	주요 기능	대표 함수
<code>syscalls.c</code>	<code>printf</code> , <code>scanf</code> , <code>exit</code> 등과 같은 표준 입출력/프로세스 함수 연결	<code>_write()</code> , <code>_read()</code> , <code>_close()</code> , <code>_exit()</code>
<code>sysmem.c</code>	동적 메모리(<code>malloc</code> , <code>free</code>) 관리	<code>_sbrk()</code>

1.9.3 시스템 콜(System Call)과 Newlib

C 표준 라이브러리(`stdio`, `stdlib`, `string`)는 내부적으로 OS 커널 호출을 사용한다.

예를 들어, PC 환경에서는 `printf()`가 `write()` 시스템 콜을 통해 커널에 전달된다.

하지만 STM32에는 OS가 없으므로

이 호출을 직접 HAL(UART, SWO 등)로 연결해야 한다.

이를 가능하게 하는 것이 바로

`syscalls.c`와 `sysmem.c`이다.

1.9.4 `syscalls.c` 구조 개요

```
1 #include <sys/stat.h>
2 #include <stdlib.h>
3 #include <errno.h>
4 #include <stdio.h>
5 #include <signal.h>
6 #include <time.h>
7 #include <sys/time.h>
8 #include <sys/times.h>
9
10 extern int __io_putchar(int ch) __attribute__((weak));
11 extern int __io_getchar(void) __attribute__((weak));
```

- `__io_putchar()` : UART, SWO 등으로 1바이트 출력하는 사용자 함수
- `__io_getchar()` : 입력 스트림에서 1바이트 읽는 사용자 함수
- STM32CubeIDE는 기본적으로 `printf`를 **UART로 리디렉션**하기 위해 `_write()` 함수에서 `__io_putchar()`를 호출하도록 설정한다.

1.9.5 `_write()` — `printf()`의 실질 구현부

```

1  __attribute__((weak)) int _write(int file, char *ptr, int len)
2  {
3      (void)file;
4      int DataIdx;
5
6      for (DataIdx = 0; DataIdx < len; DataIdx++)
7      {
8          __io_putchar(*ptr++);    // UART로 1바이트씩 전송
9      }
10     return len;
11 }
```

동작 원리:

```

1  printf() → _write() → __io_putchar() → HAL_UART_Transmit()
```

즉, `printf("Hello")`를 호출하면
문자열이 한 바이트씩 UART로 전송된다.

예시 (main.c)

```

1  int __io_putchar(int ch)
2  {
3      HAL_UART_Transmit(&huart1, (uint8_t*)&ch, 1, HAL_MAX_DELAY);
4      return ch;
5  }
```

이로써 `printf()`는 UART1 터미널로 출력된다.

1.9.6 `_read()` — `scanf()` 구현

```

1  __attribute__((weak)) int _read(int file, char *ptr, int len)
2  {
3      (void)file;
4      for (int i = 0; i < len; i++)
5      {
6          *ptr++ = __io_getchar();    // UART 입력 대기
7      }
8      return len;
9  }
```

동작 원리

```
1 | scanf() → _read() → __io_getchar() → HAL_UART_Receive()
```

예시:

```
1 | int __io_getchar(void)
2 | {
3 |     uint8_t ch;
4 |     HAL_UART_Receive(&huart1, &ch, 1, HAL_MAX_DELAY);
5 |     return ch;
6 | }
```

이렇게 하면 터미널에서 키보드 입력을 받을 수 있다.

1.9.7 `_sbrk()` — `malloc()`의 핵심 메모리 관리자

`malloc()` 과 `free()` 는 런타임 시 힙(Heap)을 사용한다.

그러나 Bare-metal 환경에서는 OS 메모리 관리자가 없으므로,

`_sbrk()` 함수가 그 역할을 대신한다.

system.c 내부 예시:

```
1 | static uint8_t *__sbrk_heap_end = NULL;
2 |
3 | void *_sbrk(ptrdiff_t incr)
4 | {
5 |     extern uint8_t _end;           // RAM 내 .bss 끝 (링커가 정의)
6 |     extern uint8_t _estack;        // 스택 최상단
7 |     extern uint32_t _Min_Stack_Size;
8 |     const uint32_t stack_limit = (uint32_t)&_estack - (uint32_t)&_Min_Stack_Size;
9 |     const uint8_t *max_heap = (uint8_t *)stack_limit;
10 |    uint8_t *prev_heap_end;
11 |
12 |    if (__sbrk_heap_end == NULL)
13 |    {
14 |        __sbrk_heap_end = &_end;    // 초기 힙 시작점 설정
15 |    }
16 |
17 |    if (__sbrk_heap_end + incr > max_heap)
18 |    {
19 |        errno = ENOMEM;              // 메모리 부족
20 |        return (void *)-1;
21 |    }
22 |
23 |    prev_heap_end = __sbrk_heap_end;
24 |    __sbrk_heap_end += incr;
25 |
26 |    return (void *)prev_heap_end;
27 | }
```

구조 설명

심볼	의미	정의 위치
<code>_end</code>	<code>.bss</code> 끝 주소 (힙 시작점)	링커 스크립트
<code>_estack</code>	스택의 최상단 주소	링커 스크립트
<code>_Min_Stack_Size</code>	최소 스택 크기 (보호 영역)	링커 스크립트

힙은 `_end` 부터 `_estack - _Min_Stack_Size` 사이에 존재하며, `malloc()` 은 `_sbrk()` 를 통해 이 영역을 점진적으로 확장한다.

1.9.8 힙과 스택의 메모리 배치

1	
2	Flash (Program)
3	.text, .rodata, .data_init
4	
5	SRAM 영역
6	
7	
8	.data → 전역 초기화 변수
9	.bss → 전역 0 초기화 변수
10	heap → malloc() 공간
11	↑ grows upward
12	
13	stack → 함수 호출/지역변수
14	↓ grows downward
15	

- `malloc()` 이 호출되면 힙이 위로 확장된다.
- 함수 호출이 반복되면 스택이 아래로 확장된다.
- 둘이 충돌하면 HardFault가 발생한다.

1.9.9 기타 함수 구현 요약 (`syscalls.c`)

함수	설명
<code>_getpid()</code>	항상 PID = 1 리턴 (프로세스 개념 없음)
<code>_kill()</code>	OS가 없으므로 항상 실패 반환
<code>_exit()</code>	종료 후 무한 루프 (임베디드 환경에서 프로세스 종료 불가)
<code>_fstat()</code> , <code>_isatty()</code>	표준입출력 장치 특성 반환 (항상 성공)
<code>_close()</code> , <code>_lseek()</code>	파일시스템 없음 → 더미 반환

1.9.10 printf 동작 경로 (요약 다이어그램)

```
1 printf("Hello")
2   ↓
3 libc: write()
4   ↓
5 syscalls.c: _write()
6   ↓
7 __io_putchar()
8   ↓
9 HAL_UART_Transmit()
10  ↓
11 USART TX
```

1.9.11 malloc 동작 경로 (요약 다이어그램)

```
1 malloc(128)
2   ↓
3 libc: _sbrk(128)
4   ↓
5 system.c: __sbrk_heap_end 증가
6   ↓
7 SRAM 힙 영역 확보
```

1.9.12 사용자 정의 리디렉션 예시

`printf()` 를 UART 대신 **SWO(Debug Console)** 로 보내고 싶다면 다음과 같이 수정할 수 있다.

```
1 int __io_putchar(int ch)
2 {
3     ITM_SendChar(ch);    // SWO 포트에 전송
4     return ch;
5 }
```

또는 **USB CDC 가상 COM 포트**를 사용할 수도 있다.

1.9.13 요약

항목	역할
<code>syscalls.c</code>	<code>printf</code> , <code>scanf</code> , <code>exit</code> 등 시스템 콜 대체 구현
<code>_write()</code>	<code>printf()</code> 가 UART로 출력되도록 연결
<code>_read()</code>	<code>scanf()</code> 가 UART에서 입력받도록 연결

항목	역할
<code>system.c</code>	<code>malloc()</code> 힙 영역 관리
<code>_sbrk()</code>	힙 확장 구현 (RAM 내 동적 메모리 관리)
<code>__io_putchar()</code>	UART 전송 인터페이스
<code>__io_getchar()</code>	UART 수신 인터페이스
<code>errno</code>	POSIX 오류 코드 전달용 변수

1.9.14 결론

`syscalls.c`와 `system.c`는 STM32의 **표준 C 라이브러리**와 **MCU 하드웨어** 사이의 **인터페이스 계층**이다.

이 두 파일 덕분에 임베디드 환경에서도 `printf`, `scanf`, `malloc`, `free` 같은 고수준 C 함수를 안전하게 사용할 수 있다.

즉,

운영체제가 없는 STM32에서도

“OS가 있는 것처럼” 동작하도록 만들어주는 숨은 커널의 역할을 수행한다.

• `main.c` 구조 및 함수 호출 흐름 (`HAL_Init()`, `SystemClock_Config()`)

1.10.1 개요

`main.c`는 STM32 프로젝트의 **실행 진입점(entry point)**이며,

펌웨어의 초기화, 클럭 설정, 주변장치 구성, 사용자 애플리케이션 루프를 담당하는 **최상위 제어 루틴**이다.

즉, `startup_stm32f1xx.s`의 `Reset_Handler`가 호출된 이후 프로그램 제어권은 `main()` 함수로 전달된다.

`main.c`는 “프로그램의 논리적 시작점”이자

HAL 기반 펌웨어의 전체 구조를 결정하는 핵심 파일이다.

1.10.2 기본 구조

CubeIDE 또는 CubeMX를 통해 생성된 STM32F1 프로젝트의 `main.c`는 다음과 같은 기본 골격을 갖는다.

```

1  #include "main.h"
2
3  int main(void)
4  {
5      /* 1. HAL 및 MCU 초기화 */
6      HAL_Init();
7
8      /* 2. 시스템 클럭 설정 */

```

```

9   SystemClock_Config();
10
11  /* 3. 주변장치 초기화 (GPIO, UART, ADC, etc.) */
12  MX_GPIO_Init();
13  MX_USART1_UART_Init();
14  MX_ADC1_Init();
15  MX_TIM1_Init();
16
17  /* 4. 사용자 변수 및 애플리케이션 초기화 */
18  Sensor_Init();
19  Display_Init();
20
21  /* 5. 메인 루프 */
22  while (1)
23  {
24      Read_Sensor();
25      Process_Data();
26      HAL_Delay(100);
27  }
28  }

```

1.10.3 호출 순서 및 역할 요약

단계	함수	역할
①	<code>Reset_Handler()</code>	startup 파일에서 호출, RAM 초기화 및 <code>main()</code> 진입
②	<code>HAL_Init()</code>	HAL 레이어 및 SysTick 초기화
③	<code>SystemClock_Config()</code>	PLL 및 버스 클럭 설정
④	<code>MX_GPIO_Init()</code> 등	각 주변장치 초기화 (CubeMX 생성 코드)
⑤	사용자 함수	센서, 로직, FreeRTOS 등 애플리케이션 로직
⑥	<code>while(1)</code>	무한 루프 내에서 메인 기능 수행

1.10.4 `HAL_Init()` 내부 구조

`HAL_Init()` 은 HAL 드라이버 계층의 전역 초기화를 담당하며, 다음 순서로 동작한다.

```

1  HAL_StatusTypeDef HAL_Init(void)
2  {
3      /* 1. 플래시 프리페치 버퍼, 인터럽트 우선순위 그룹 초기화 */
4      HAL_InitTick(TICK_INT_PRIORITY);
5      HAL_MspInit();
6      return HAL_OK;
7  }

```

내부 구성 흐름:

1. HAL_Msplnit()

- MCU 지원 패키지(MSP) 레벨 초기화
(NVIC 설정, 클록 소스 활성화 등)
- `stm32f1xx_hal_msp.c` 파일 내에서 재정의 가능.

2. SysTick 타이머 초기화

- 1ms 주기로 동작하도록 SysTick 설정.
→ 이후 `HAL_Delay()` 와 `HAL_GetTick()` 사용 가능.

3. 중단점

- 이 시점 이후 HAL의 모든 API를 안전하게 사용할 수 있다.

즉, `HAL_Init()` 은 “HAL 레이어의 전원 공급 + SysTick 기반 시간 시스템 구성” 단계이다.

1.10.5 SystemClock_Config() — 시스템 클록 설정

`SystemClock_Config()` 은 `system_stm32f1xx.c` 에서 정의된

하드웨어 초기화 이후, PLL과 버스 클록을 구성하는 함수이다.

CubeMX에서 자동 생성되며, RCC 구조체 기반의 HAL API를 사용한다.

예시 (STM32F103 @ 72 MHz)

```
1 void SystemClock_Config(void)
2 {
3     RCC_OscInitTypeDef RCC_OscInitStruct = {0};
4     RCC_ClkInitTypeDef RCC_ClkInitStruct = {0};
5
6     /* 1. 외부 크리스탈(HSE) 활성화 */
7     RCC_OscInitStruct.OscillatorType = RCC_OSCILLATORTYPE_HSE;
8     RCC_OscInitStruct.HSEState = RCC_HSE_ON;
9     RCC_OscInitStruct.HSEPredivValue = RCC_HSE_PREDIV_DIV1;
10
11     /* 2. PLL 설정: 8MHz × 9 = 72MHz */
12     RCC_OscInitStruct.PLL.PLLState = RCC_PLL_ON;
13     RCC_OscInitStruct.PLL.PLLSource = RCC_PLLSOURCE_HSE;
14     RCC_OscInitStruct.PLL.PLLMUL = RCC_PLL_MUL9;
15     HAL_RCC_OscConfig(&RCC_OscInitStruct);
16
17     /* 3. 클록 트리 구성 */
18     RCC_ClkInitStruct.ClockType = RCC_CLOCKTYPE_HCLK |
19                                     RCC_CLOCKTYPE_SYSCLK |
20                                     RCC_CLOCKTYPE_PCLK1 |
21                                     RCC_CLOCKTYPE_PCLK2;
22     RCC_ClkInitStruct.SYSCLKSource = RCC_SYSCLKSOURCE_PLLCLK;
23     RCC_ClkInitStruct.AHBCLKDivider = RCC_SYSCLK_DIV1;
24     RCC_ClkInitStruct.APB1CLKDivider = RCC_HCLK_DIV2;
25     RCC_ClkInitStruct.APB2CLKDivider = RCC_HCLK_DIV1;
26
27     /* 4. 최종 설정 반영 및 Flash wait state 지정 */
```

1.10.6 클록 설정 흐름

1.10.7 MX_GPIO_Init() 등 주변장치 초기화

1.10.8 함수 호출 흐름 전체 다이어그램

```

1 |
2 |             Reset_Handler()
3 | → 초기 RAM, Stack 설정
4 | → SystemInit() (PLL/Vector Table 설정)
5 | → main() 진입
6 |
7 |             ↓
8 |
9 |             main()
10 | -----
11 | HAL_Init()

```

```

12 |   └─ HAL_MspInit()
13 |   └─ SysTick_Config()
14 |
15 |   SystemClock_Config()
16 |   └─ HAL_RCC_OscConfig()
17 |   └─ HAL_RCC_ClockConfig()
18 |   └─ SystemCoreClockUpdate()
19 |
20 |   MX_GPIO_Init()
21 |   MX_UART_Init(), MX_ADC_Init(), ...
22 |
23 |   while(1)
24 |   └─ Sensor_Read()
25 |   └─ Data_Process()
26 |   └─ HAL_Delay(100)
27 |

```

1.10.9 SysTick 기반 HAL 시간 시스템

`HAL_Init()` 에서 SysTick이 설정되면,
1ms마다 `SysTick_Handler()` 가 호출되어 내부 tick 카운터를 증가시킨다.

```

1 void SysTick_Handler(void)
2 {
3     HAL_IncTick(); // 내부 tick 증가
4 }

```

이를 통해 다음 함수들이 정확한 시간 제어를 수행한다.

- `HAL_Delay(ms)`
- `HAL_GetTick()`
- `HAL_Timeout` 기반 I/O 함수 (예: `HAL_I2C_Master_Transmit()`)

1.10.10 사용자 애플리케이션 초기화 구역

`main.c` 내에는 **USER CODE BEGIN / END** 블록이 제공된다.

```

1 /* USER CODE BEGIN 2 */
2 Sensor_Init();
3 Display_Init();
4 /* USER CODE END 2 */

```

이 구간에 사용자의 초기화 코드를 작성하면
CubeMX 재생성 시 코드가 보존된다.

1.10.11 오류 및 디버깅 포인트

증상	원인	해결책
UART, ADC 값 이상	PLL 배율/클록 설정 오류	<code>SystemClock_Config()</code> 확인
<code>HAL_Delay()</code> 작동 안함	<code>HAL_Init()</code> 누락	반드시 첫 호출로 배치
HardFault 발생	초기화 누락 or Stack Overflow	디버깅 시 Call Stack 추적
FreeRTOS Tick 불안정	SysTick 중복 설정	RTOS Tick Source 조정

1.10.12 요약

함수	역할
<code>HAL_Init()</code>	HAL 시스템 초기화, SysTick 설정
<code>SystemClock_Config()</code>	PLL 및 버스 클록 구성
<code>MX_GPIO_Init()</code>	GPIO 핀모드 설정
<code>MX_UART_Init()</code>	UART 설정 (보레이트, 모드)
<code>MX_ADC_Init()</code>	ADC 샘플링 설정
<code>while(1)</code>	사용자 메인 루프 (센서, 로직, 통신 등)
<code>HAL_Delay()</code>	ms 단위 지연 함수
<code>HAL_GetTick()</code>	현재 Tick 시간 반환

1.10.13 결론

`main.c`는 STM32 펌웨어의 **초기화** → **실행** → **유지 루프** 전체를 담당하는 핵심 진입점이다.

특히 `HAL_Init()` 과 `SystemClock_Config()` 의 호출 순서는 모든 주변장치의 동작 타이밍과 안정성에 직접적인 영향을 미친다.

따라서 STM32 프로젝트를 분석할 때,

`main.c`의 **함수 호출 순서와 초기화 흐름**을 정확히 이해하는 것은 임베디드 시스템의 구조를 해석하는 첫 단계이자, FreeRTOS 및 복합 센서 시스템 개발의 기반이 된다.

1.3 CMSIS와 HAL 구조

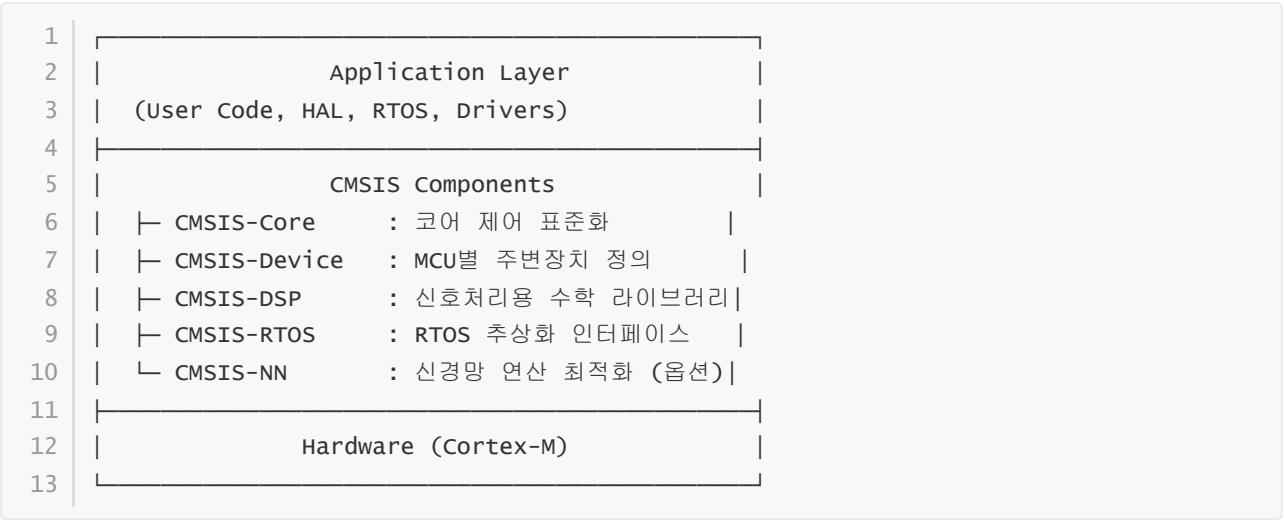
• CMSIS-Core, Device, DSP의 역할

1.11.1 개요

CMSIS (Cortex Microcontroller Software Interface Standard) 는 ARM에서 정의한 표준 펌웨어 인터페이스 규격으로, Cortex-M 계열 MCU에서 **하드웨어 접근, 레지스터 제어, DSP, RTOS 통합**을 일관된 방식으로 수행할 수 있도록 만든 소프트웨어 프레임워크이다.

STM32 HAL도 내부적으로 **CMSIS**를 기반으로 작성되어 있으며, 모든 Cortex-M MCU는 CMSIS 계층을 통해 “ARM 코어 공통 기능”과 “벤더별 주변장치 레지스터”를 구분한다.

1.11.2 CMSIS 전체 구성



1.11.3 CMSIS-Core — 코어 제어 표준화 계층

(1) 역할

CMSIS-Core는 Cortex-M 프로세서 코어의 **레지스터 접근과 제어**를 **표준화**한 계층이다. ARM이 직접 제공하며, 모든 MCU 제조사는 이를 기반으로 자신의 드라이버를 구성한다.

(2) 주요 기능

기능	설명
레지스터 매핑	NVIC, SysTick, SCB(System Control Block) 등 Cortex-M 공통 레지스터 정의
인터럽트 관리	NVIC_EnableIRQ(), NVIC_SetPriority() 등 인터럽트 제어 API
시스템 제어	SystemInit(), SystemCoreClockUpdate() 등 클럭 관리
비트밴드(Bit-Band)	원자적 메모리 접근 지원 매크로

기능	설명
Intrinsic 함수	<code>__enable_irq()</code> , <code>__disable_irq()</code> , <code>__WFI()</code> , <code>__NOP()</code> 등 어셈블리 래퍼 제공

(3) 파일 구성 예시 (STM32F103 기준)

파일	역할
<code>core_cm3.h</code>	Cortex-M3 레벨의 공통 정의 (레지스터 구조체, NVIC 등)
<code>system_stm32f1xx.c</code>	SystemInit() 구현, 클럭 기본 설정
<code>cmsis_gcc.h</code> / <code>cmsis_armclang.h</code>	컴파일러별 인라인 어셈블리 정의

(4) 예시 코드

```

1  #include "core_cm3.h"
2
3  void Example_NVIC_Config(void)
4  {
5      NVIC_SetPriority(SysTick_IRQn, 1);
6      NVIC_EnableIRQ(USART1_IRQn);
7  }
```

HAL의 모든 인터럽트 초기화 함수 (`HAL_NVIC_SetPriority`, `HAL_NVIC_EnableIRQ`)
내부는 CMSIS-Core의 NVIC 레지스터 접근을 기반으로 구현되어 있다.

1.11.4 CMSIS-Device — MCU 제조사별 하드웨어 정의 계층

(1) 역할

CMSIS-Device는 **MCU 벤더(ST, NXP, TI 등)** 가 제공하는
주변장치 레지스터 구조체와 벡터 테이블 정의를 포함한다.

CMSIS-Core가 코어 레벨(CPU)에 집중한다면,
CMSIS-Device는 **칩 레벨(Peripherals)** 에 집중한다.

(2) 주요 구성

항목	설명
레지스터 구조체 정의	GPIO, RCC, TIM, ADC 등 주변장치 구조체 선언
인터럽트 벡터 테이블	startup_stm32f1xx.s 에 정의된 IRQ 엔트리 매핑
시스템 초기화	SystemInit(), SystemCoreClock 등 공통 초기화 변수
디바이스 헤더 파일	<code>stm32f103xb.h</code> , <code>stm32f1xx.h</code> 등에서 MCU별 정의 제공

(3) 예시: GPIO 구조체 정의 (stm32f103xb.h)

```
1 typedef struct
2 {
3     __IO uint32_t CRL;
4     __IO uint32_t CRH;
5     __IO uint32_t IDR;
6     __IO uint32_t ODR;
7     __IO uint32_t BSRR;
8     __IO uint32_t BRR;
9     __IO uint32_t LCKR;
10 } GPIO_TypeDef;
11
12 #define GPIOA ((GPIO_TypeDef *) GPIOA_BASE)
```

이를 통해 다음과 같이 직접 레지스터 제어도 가능하다.

```
1 GPIOA->ODR |= (1 << 5); // PA5 핀 HIGH 출력
```

(4) 구성 파일 예시

파일	역할
stm32f1xx.h	CMSIS-Core + Device 종합 헤더
stm32f103xb.h	MCU 레벨의 하드웨어 정의
system_stm32f1xx.c	클럭 초기화 코드 및 SystemInit()
startup_stm32f1xx.s	인터럽트 벡터 테이블

1.11.5 CMSIS-DSP — 디지털 신호처리 라이브러리

(1) 역할

CMSIS-DSP는 ARM이 제공하는 **Cortex-M용 고성능 DSP(신호처리) 라이브러리**이다.

하드웨어 Multiply-Accumulate (MAC) 연산, SIMD 명령어를 활용하여

FFT, FIR, IIR, Matrix, Vector, Statistics 등을 최적화된 속도로 처리한다.

(2) 주요 기능 그룹

기능 그룹	대표 함수
수학 연산 (Math)	arm_sin_f32(), arm_cos_f32(), arm_sqrt_f32()
통계 (Statistics)	arm_mean_f32(), arm_var_f32(), arm_std_f32()
벡터/행렬 (Matrix)	arm_mat_mult_f32(), arm_mat_inverse_f32()
필터 (Filters)	arm_fir_f32(), arm_biquad_cascade_df1_f32()

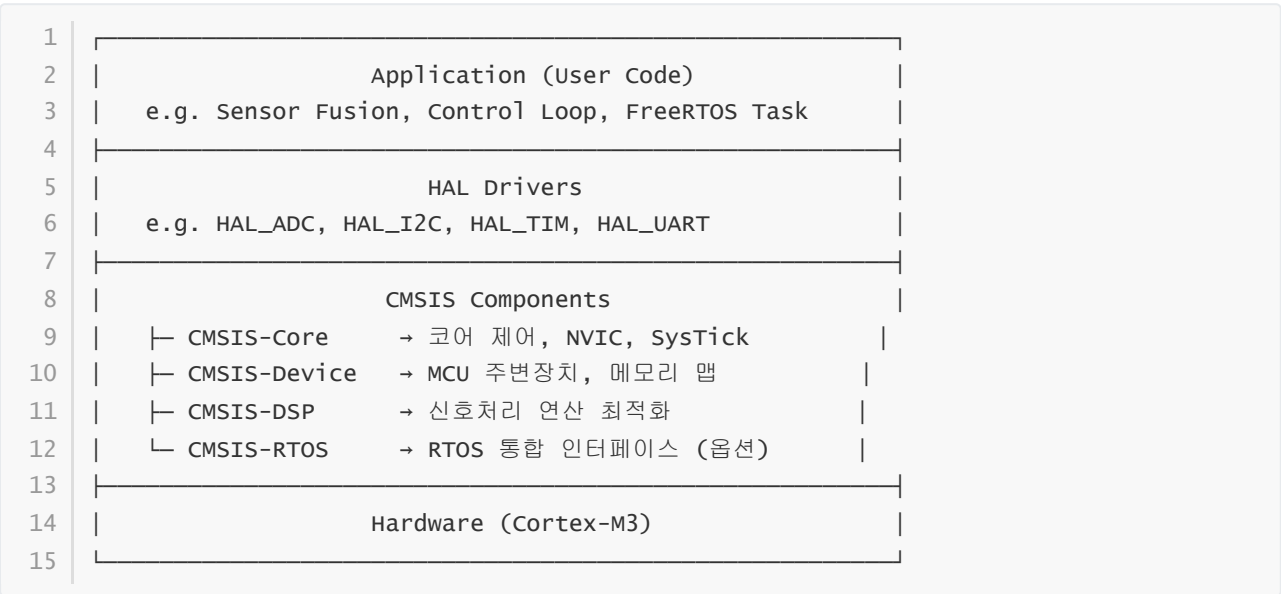
기능 그룹	대표 함수
변환 (Transforms)	<code>arm_rfft_fast_f32()</code> , <code>arm_cfft_f32()</code>
통계 분석 (Correlation)	<code>arm_correlate_f32()</code>
PID 제어 (Controller)	<code>arm_pid_init_f32()</code> , <code>arm_pid_compute_f32()</code>

(3) 예시: FIR 필터 적용

```
1  #include "arm_math.h"
2
3  #define BLOCK_SIZE 32
4  #define NUM_TAPS   16
5
6  float32_t firStateF32[BLOCK_SIZE + NUM_TAPS - 1];
7  float32_t firCoeffs32[NUM_TAPS] = { /* 계수 */ };
8  arm_fir_instance_f32 S;
9
10 void DSP_Example(void)
11 {
12     arm_fir_init_f32(&S, NUM_TAPS, firCoeffs32, firStateF32, BLOCK_SIZE);
13     arm_fir_f32(&S, inputSignal, outputSignal, BLOCK_SIZE);
14 }
```

Cortex-M3 수준에서도 부동소수점 연산을 최적화하여
고속 신호 처리나 센서 데이터 평활화에 사용된다.

1.11.6 CMSIS 간 계층 관계



1.11.7 STM32F1 프로젝트에서의 실제 연계

계층	예시 파일	역할
CMSIS-Core	<code>core_cm3.h</code>	코어 제어 함수, 인터럽트 설정
CMSIS-Device	<code>stm32f103xb.h</code> / <code>system_stm32f1xx.c</code>	MCU 주변장치 정의
CMSIS-DSP	<code>arm_math.h</code>	DSP 연산 라이브러리
HAL	<code>stm32f1xx_hal_gpio.c</code> / <code>stm32f1xx_hal_uart.c</code>	하드웨어 추상화 계층
Application	<code>main.c</code> / <code>sensor.c</code>	사용자 로직 구현

1.11.8 요약

구성 요소	제공 주체	주요 역할	대표 파일
CMSIS-Core	ARM	Cortex-M 공통 코어 기능 (NVIC, SysTick, SCB)	<code>core_cm3.h</code>
CMSIS-Device	MCU 제조사 (STMicroelectronics)	주변장치 레지스터 매핑, 클록 초기화, 인터럽트 벡터	<code>stm32f103xb.h</code> , <code>system_stm32f1xx.c</code>
CMSIS-DSP	ARM	고성능 수학 및 신호처리 함수	<code>arm_math.h</code>
CMSIS-RTOS (옵션)	ARM	RTOS 호환 API (v1/v2)	<code>cmsis_os.h</code>

1.11.9 결론

CMSIS는 STM32 펌웨어의 **기초 표준 계층**으로서, HAL·RTOS·DSP 모듈들이 동일한 코어 인터페이스를 통해 상호 작용하도록 한다.

- **CMSIS-Core** : 코어 레벨 표준화 (인터럽트, 시스템 제어)
- **CMSIS-Device** : 칩 레벨 표준화 (주변장치 및 레지스터 매핑)
- **CMSIS-DSP** : 수학 연산 및 신호처리 최적화

즉,

HAL과 FreeRTOS, 그리고 센서 신호처리가
모두 CMSIS 위에서 통일된 언어로 협력하도록 만드는
“임베디드 소프트웨어 표준의 중심축”이 바로 CMSIS이다.

HAL 계층과 LL (Low Layer) 비교

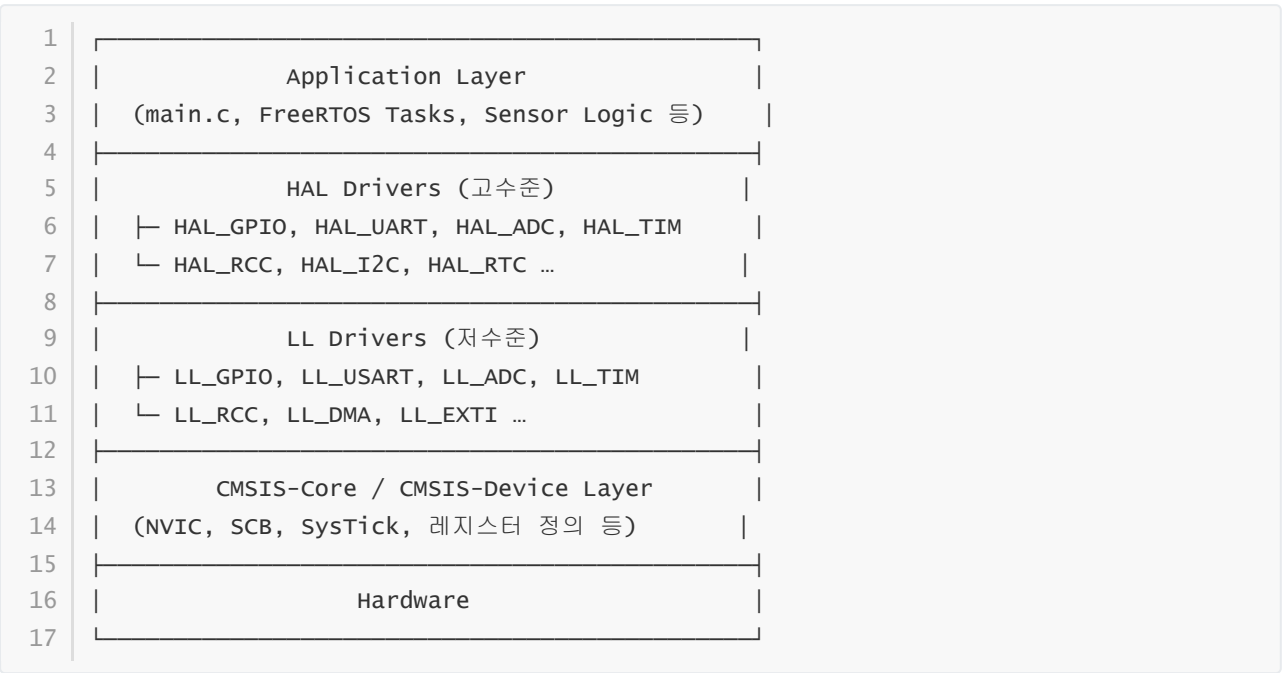
1.12.1 개요

STM32Cube 프레임워크는 하드웨어 접근 수준에 따라 두 가지 주요 드라이버 계층을 제공한다:

- 1. HAL (Hardware Abstraction Layer)
→ 하드웨어 세부 구조를 추상화하여 고수준 API로 제공하는 계층
- 2. LL (Low Layer)
→ 레지스터에 직접 접근할 수 있는 저수준 제어 계층

두 계층은 동일한 주변장치(Peripheral) 레벨에서 동작하지만, 코드 가독성, 성능, 제어 세밀도 측면에서 상반된 특징을 가진다.

1.12.2 계층적 관계



1.12.3 HAL (Hardware Abstraction Layer)

(1) 특징

HAL은 STM32 MCU의 모든 하드웨어 동작을 함수 기반으로 추상화한다. 직접 레지스터를 다루지 않고, 구조체 기반으로 설정을 처리한다.

(2) 주요 특성 요약

항목	설명
목적	코드 이식성, 이해도, 유지보수성 향상
접근 방식	구조체 기반 API (HAL_GPIO_Init, HAL_UART_Transmit)

항목	설명
추상화 수준	높음 — 하드웨어 세부정보 숨김
성능	상대적으로 느림 (함수 오버헤드 존재)
RTOS 친화성	높음 — FreeRTOS와 자연스럽게 통합 가능
코드 크기	큼 — 함수 호출 계층 다수
디버깅 난이도	쉬움 — 고수준 API 중심

(3) 예시

```

1  GPIO_InitTypeDef GPIO_InitStruct = {0};
2  __HAL_RCC_GPIOC_CLK_ENABLE();
3
4  GPIO_InitStruct.Pin = GPIO_PIN_13;
5  GPIO_InitStruct.Mode = GPIO_MODE_OUTPUT_PP;
6  GPIO_InitStruct.Speed = GPIO_SPEED_FREQ_LOW;
7  HAL_GPIO_Init(GPIOC, &GPIO_InitStruct);
8
9  HAL_GPIO_WritePin(GPIOC, GPIO_PIN_13, GPIO_PIN_SET);

```

HAL은 함수 이름만으로도 동작을 직관적으로 이해할 수 있으며, **CubeMX 자동 생성 코드**와 완벽히 호환된다.

1.12.4 LL (Low Layer)

(1) 특징

LL은 HAL과 동일한 주변장치를 제어하지만, **레지스터 접근 수준에 가까운 매크로 기반 인터페이스**를 제공한다. 즉, 하드웨어의 속도와 효율성을 극대화하고자 할 때 사용된다.

(2) 주요 특성 요약

항목	설명
목적	성능 최적화, 빠른 실행, 미세 제어
접근 방식	인라인 함수 및 매크로 (LL_GPIO_SetOutputPin)
추상화 수준	낮음 — 레지스터 비트 직접 접근
성능	매우 빠름 (오버헤드 거의 없음)
RTOS 친화성	중간 — 인터럽트 처리 중심
코드 크기	작음 — 간단한 매크로 기반
디버깅 난이도	높음 — 하드웨어 이해 필요

(3) 예시

```
1 LL_APB2_GRP1_EnableClock(LL_APB2_GRP1_PERIPH_GPIOC);
2 LL_GPIO_SetPinMode(GPIOC, LL_GPIO_PIN_13, LL_GPIO_MODE_OUTPUT);
3 LL_GPIO_SetOutputPin(GPIOC, LL_GPIO_PIN_13);
```

HAL보다 훨씬 빠르지만, **하드웨어 동작을 개발자가 직접 책임져야** 한다.
잘못된 설정은 즉시 MCU 오동작으로 이어질 수 있다.

1.12.5 HAL vs LL 비교표

구분	HAL (Hardware Abstraction Layer)	LL (Low Layer)
개발 수준	고수준 API (함수 기반)	저수준 API (매크로/인라인)
성능	느림 (함수 호출 오버헤드)	빠름 (직접 레지스터 접근)
코드 크기	큼	작음
가독성	높음 (초보자 친화적)	낮음 (레지스터 지식 필요)
이식성	높음 (MCU 간 호환)	낮음 (MCU별 차이 있음)
디버깅 난이도	쉬움	어려움
CubeMX 지원	완전 지원	일부만 지원
RTOS 통합	매우 용이	직접 구현 필요
대표 파일	stm32f1xx_hal_*.c	stm32f1xx_ll_*.h
용도	교육, 프로토타입, 일반 제품	실시간 제어, 고속 응용, 임베디드 최적화

1.12.6 혼용 구조 (Hybrid Approach)

실제 산업용 STM32 프로젝트에서는 **HAL + LL 혼용 구조**가 흔히 사용된다.

- 초기화 및 설정 → HAL
(CubeMX 자동 생성 활용)
- 실시간 제어 루프, 인터럽트 내부 → LL
(지연 없는 하드웨어 제어 필요 시)

예시

```
1 void Motor_Control_ISR(void)
2 {
3     /* LL 사용: 빠른 핀 토글 */
4     LL_GPIO_SetOutputPin(GPIOA, LL_GPIO_PIN_5);
5     LL_GPIO_ResetOutputPin(GPIOA, LL_GPIO_PIN_5);
6
7     /* HAL 사용: 주기적 로깅 */
8     HAL_UART_Transmit(&huart1, (uint8_t*)"Pulse!\r\n", 8, 100);
9 }
```

이처럼 HAL의 관리성과 LL의 속도를 절충하여
“초기화는 HAL, 제어는 LL” 구조로 최적화할 수 있다.

1.12.7 코드 성능 및 크기 비교

항목	HAL	LL
GPIO Toggle	약 0.6 μ s	약 0.1 μ s
PWM 설정	약 40 μ s	약 6 μ s
Flash Size 증가량	+15~25 KB	+3~5 KB

(STM32F103 @72MHz, -O2 빌드 기준)

1.12.8 실제 사용 전략

개발 목적	권장 계층	설명
교육 및 프로토타입	HAL	빠른 개발, 유지보수 용이
실시간 제어 시스템 (모터, 통신)	LL	지연 최소화, 성능 극대화
복합 프로젝트 (RTOS + Sensor)	HAL + LL	초기화 HAL, 제어 LL
저전력/소형 MCU	LL	코드 사이즈 절감
대형 프로젝트 (FreeRTOS, IoT)	HAL	모듈 간 일관성 중요

1.12.9 결론

- HAL은 안정성과 개발 효율성을 위해 설계된 고수준 프레임워크이며, 복잡한 하드웨어를 단일 API로 다루도록 만든다.
- LL은 속도와 제어권을 극대화한 저수준 인터페이스로, 타이밍이 중요한 펌웨어(모터 제어, 통신, 인터럽트 루프)에 적합하다.
- 두 계층은 상호 보완적이다.

STM32 개발의 원칙은

“초기화는 HAL로, 제어는 LL로”,

즉 유지보수성과 성능의 균형을 맞추는 것이다.

• CubeMX가 생성하는 초기화 코드 분석

1.13.1 개요

STM32CubeMX는 STMicroelectronics에서 제공하는
MCU 초기화 자동 코드 생성 도구이다.

개발자는 GUI 환경에서 **핀 매핑, 클럭 설정, 주변장치 구성**을 수행하고,
CubeMX가 이를 기반으로 **HAL 라이브러리 기반의 초기화 코드**를 자동 생성한다.

이 코드의 핵심은 다음 세 가지 함수로 요약된다:

1. `HAL_Init()` — HAL 레이어 및 기본 시스템 설정
2. `SystemClock_Config()` — PLL, HCLK, PCLK 설정
3. `MX_<Peripheral>_Init()` — 각 주변장치 초기화

이들은 `main.c` 내부에서 순차적으로 호출되며,
프로젝트의 **실행 환경(Initialization Stage)** 을 구성한다.

1.13.2 main.c 구조

CubeMX로 생성된 `main.c` 는 기본적으로 다음 구조를 갖는다:

```
1  int main(void)
2  {
3      /* 1. HAL 초기화 */
4      HAL_Init();
5
6      /* 2. 시스템 클럭 설정 */
7      SystemClock_Config();
8
9      /* 3. 주변장치 초기화 */
10     MX_GPIO_Init();
11     MX_ADC1_Init();
12     MX_TIM2_Init();
13     MX_USART1_UART_Init();
14     MX_I2C1_Init();
15
16     /* 4. 사용자 코드 시작 */
17     while (1)
18     {
19         // Application Loop
20     }
21 }
```

이 순서는 **모든 STM32 HAL 기반 프로젝트의 공통 템플릿**이며,
CubeMX는 각 `MX_*.c` 파일에 해당하는 초기화 함수들을 자동 생성한다.

1.13.3 HAL_Init()

(1) 정의 위치

stm32f1xx_hal.c 내부

(2) 주요 기능

단계	설명
① HAL_MspInit() 호출	NVIC 우선순위, SysTick, RCC 기본 설정 수행
② SysTick 설정	1ms 단위 Tick 생성 (HAL_Delay() 기반)
③ NVIC 그룹 설정	NVIC_PRIORITYGROUP_4
④ 내부 변수 초기화	HAL 상태, 타이머, 내부 플래그 등

(3) 핵심 코드

```
1 void HAL_Init(void)
2 {
3     HAL_MspInit();
4     HAL_InitTick(TICK_INT_PRIORITY);
5     __HAL_RCC_AFIO_CLK_ENABLE();
6     __HAL_RCC_PWR_CLK_ENABLE();
7 }
```

즉, HAL_Init()은 HAL 계층의 기반을 세팅하는 “시스템 프레임워크 초기화 루틴”이다.

1.13.4 SystemClock_Config()

(1) 위치

main.c 혹은 system_stm32f1xx.c 와 연결되어 있으며, CubeMX가 자동 생성한 함수이다.

(2) 기능 요약

항목	설정 내용
PLL Source	HSE or HSI divided input
PLL Multiplier	×2 ~ ×16 설정
AHB Prescaler	SYSCLK → HCLK
APB1 Prescaler	HCLK/2 (저속 버스)
APB2 Prescaler	HCLK (고속 버스)

항목	설정 내용
Flash Latency	클록 주파수에 맞게 조정
SysTick Source	HCLK / 8 or HCLK 직접

(3) 예시

```

1 void SystemClock_Config(void)
2 {
3     RCC_OscInitTypeDef RCC_OscInitStruct = {0};
4     RCC_ClkInitTypeDef RCC_ClkInitStruct = {0};
5
6     RCC_OscInitStruct.OscillatorType = RCC_OSCILLATORTYPE_HSE;
7     RCC_OscInitStruct.HSEState = RCC_HSE_ON;
8     RCC_OscInitStruct.PLL.PLLState = RCC_PLL_ON;
9     RCC_OscInitStruct.PLL.PLLSource = RCC_PLLSOURCE_HSE;
10    RCC_OscInitStruct.PLL.PLLMUL = RCC_PLL_MUL9;
11    HAL_RCC_OscConfig(&RCC_OscInitStruct);
12
13    RCC_ClkInitStruct.ClockType = RCC_CLOCKTYPE_HCLK |
14                                RCC_CLOCKTYPE_SYSCLK |
15                                RCC_CLOCKTYPE_PCLK1 |
16                                RCC_CLOCKTYPE_PCLK2;
17    RCC_ClkInitStruct.SYSCLKSource = RCC_SYSCLKSOURCE_PLLCLK;
18    RCC_ClkInitStruct.AHBCLKDivider = RCC_SYSCLK_DIV1;
19    RCC_ClkInitStruct.APB1CLKDivider = RCC_HCLK_DIV2;
20    RCC_ClkInitStruct.APB2CLKDivider = RCC_HCLK_DIV1;
21    HAL_RCC_ClockConfig(&RCC_ClkInitStruct, FLASH_LATENCY_2);
22 }
```

STM32F103에서 일반적으로 PLL ×9로 동작시,
8 MHz HSE → 72 MHz SYSCLK이 구성된다.

1.13.5 주변장치 초기화 (MX_*.c)

CubeMX는 선택된 각 주변장치에 대해

MX_<Peripheral>_Init() 함수를 자동 생성한다.

예시 ① — GPIO

```
1 void MX_GPIO_Init(void)
2 {
3     __HAL_RCC_GPIOC_CLK_ENABLE();
4
5     GPIO_InitTypeDef GPIO_InitStruct = {0};
6     GPIO_InitStruct.Pin = GPIO_PIN_13;
7     GPIO_InitStruct.Mode = GPIO_MODE_OUTPUT_PP;
8     GPIO_InitStruct.Pull = GPIO_NOPULL;
9     GPIO_InitStruct.Speed = GPIO_SPEED_FREQ_LOW;
10    HAL_GPIO_Init(GPIOC, &GPIO_InitStruct);
11 }
```

예시 ② — ADC

```
1 void MX_ADC1_Init(void)
2 {
3     hadc1.Instance = ADC1;
4     hadc1.Init.ScanConvMode = ADC_SCAN_DISABLE;
5     hadc1.Init.ContinuousConvMode = ENABLE;
6     hadc1.Init.DataAlign = ADC_DATAALIGN_RIGHT;
7     hadc1.Init.NbrOfConversion = 1;
8     HAL_ADC_Init(&hadc1);
9 }
```

예시 ③ — Timer

```
1 void MX_TIM2_Init(void)
2 {
3     htim2.Instance = TIM2;
4     htim2.Init.Prescaler = 72 - 1;
5     htim2.Init.CounterMode = TIM_COUNTERMODE_UP;
6     htim2.Init.Period = 1000 - 1;
7     HAL_TIM_Base_Init(&htim2);
8 }
```

예시 ④ — UART

```
1 void MX_USART1_UART_Init(void)
2 {
3     huart1.Instance = USART1;
4     huart1.Init.BaudRate = 115200;
5     huart1.Init.WordLength = UART_WORDLENGTH_8B;
6     huart1.Init.StopBits = UART_STOPBITS_1;
7     huart1.Init.Parity = UART_PARITY_NONE;
8     huart1.Init.Mode = UART_MODE_TX_RX;
9     HAL_UART_Init(&huart1);
10 }
```

모든 `MX_*.c` 함수들은 HAL 기반 구조체(`*_HandleTypeDef`)를 초기화하고,
해당 포트의 클럭 활성화 → GPIO 설정 → 주변장치 설정 순으로 구성된다.

1.13.6 사용자 코드 보호 블록

CubeMX가 재생성할 때 코드를 덮어쓰지 않도록
“사용자 코드 보호 블록”이 삽입된다.

```
1  /* USER CODE BEGIN 0 */
2  // 여기에 사용자가 직접 작성한 코드 삽입
3  /* USER CODE END 0 */
```

이 영역 내부의 코드는 **재생성 시에도 유지**되므로,
사용자 함수, 변수 선언, 로직 추가를 이 안에 배치해야 한다.

1.13.7 MspInit() 구조

CubeMX는 `stm32f1xx_hal_msp.c` 에
주변장치별 클럭 및 핀 설정 코드를 생성한다.

```
1  void HAL_UART_MspInit(UART_HandleTypeDef* huart)
2  {
3      if(huart->Instance == USART1)
4      {
5          __HAL_RCC_USART1_CLK_ENABLE();
6          __HAL_RCC_GPIOA_CLK_ENABLE();
7          GPIO_InitTypeDef GPIO_InitStruct = {0};
8
9          GPIO_InitStruct.Pin = GPIO_PIN_9;
10         GPIO_InitStruct.Mode = GPIO_MODE_AF_PP;
11         GPIO_InitStruct.Speed = GPIO_SPEED_FREQ_HIGH;
12         HAL_GPIO_Init(GPIOA, &GPIO_InitStruct);
13     }
14 }
```

HAL_Init() → `HAL_MspInit()` → 주변장치별 `HAL_*_MspInit()`
순서로 호출되며, 실제 하드웨어 클럭 및 핀 설정이 이루어진다.

1.13.8 전체 초기화 순서 요약

```
1  main()
2  │  HAL_Init()
3  │  │  HAL_MspInit()
4  │  SystemClock_Config()
5  │  │  PLL, AHB, APB 설정
6  │  MX_GPIO_Init()
7  │  MX_ADC1_Init()
8  │  MX_TIM2_Init()
9  │  MX_USART1_UART_Init()
10 │  while(1)
11 │      Application Loop
```

1.13.9 CubeMX 코드의 특징

항목	특징
자동화 수준	매우 높음 — 레지스터 세팅 자동 처리
구조	함수 분리형 (각 Peripheral별 파일)
유지보수성	CubeIDE 재생성 시 일관성 유지
성능	HAL 오버헤드 존재 (성능 중요 시 LL 혼용 필요)
학습 효과	HAL 함수 호출 순서 및 구조체 초기화 학습에 유용

1.13.10 결론

CubeMX가 생성하는 초기화 코드는

STM32의 **시스템 부팅, 클럭 설정, 주변장치 활성화**를 완전히 자동화한다.

개발자는 하드웨어 레벨의 복잡한 초기화를 직접 다루지 않아도 되며,

`main()` 이후부터 **응용 로직에 집중**할 수 있다.

즉, CubeMX의 초기화 코드는

STM32 HAL 기반 프로젝트의 “**표준 진입점 (Standard Entry Point)**”이며,

펌웨어 구조를 자동으로 일관성 있게 유지해주는 핵심 프레임워크이다.

• HAL 함수 구조 및 리턴 코드 이해

1.14.1 개요

HAL(Hardware Abstraction Layer)은 STM32Cube HAL 라이브러리에서

모든 하드웨어 제어를 **함수(Function) 단위로 추상화한 계층**이다.

모든 HAL 함수는 일관된 구조와 리턴 규약을 따르며,

호출 결과를 통해 **성공/실패/타임아웃/오류 상태**를 판별할 수 있다.

HAL 함수의 설계 원칙:

“직접 레지스터 접근 대신, 안전한 함수 호출을 통해 하드웨어를 제어한다.”

1.14.2 HAL 함수의 기본 구조

모든 HAL 함수는 다음과 같은 구조적 패턴을 가진다.

```
1 HAL_StatusTypeDef HAL_<Peripheral>_<Operation>(<HandleTypeDef> *handle, ...);
```

예시:

```
1 HAL_StatusTypeDef HAL_ADC_Start(ADC_HandleTypeDef *hadc);
2 HAL_StatusTypeDef HAL_UART_Transmit(UART_HandleTypeDef *huart, uint8_t *pData,
   uint16_t Size, uint32_t Timeout);
3 HAL_StatusTypeDef HAL_I2C_Master_Transmit(I2C_HandleTypeDef *hi2c, uint16_t
   DevAddress, uint8_t *pData, uint16_t Size, uint32_t Timeout);
```

- `HAL_` 접두사 : HAL 계층 함수임을 명시
- `<Peripheral>` : 주변장치 이름 (GPIO, ADC, I2C, UART 등)
- `<Operation>` : 수행 기능 (Init, Start, Stop, Transmit 등)
- `HandleTypeDef` : 주변장치의 내부 상태를 저장하는 구조체 포인터

1.14.3 Handle 구조체의 역할

모든 HAL 함수는 `Handle` 을 통해
주변장치 인스턴스 및 내부 상태를 추적한다.

예시 - `ADC_HandleTypeDef` :

```
1 typedef struct
2 {
3     ADC_TypeDef           *Instance;      // ADC1, ADC2 등 하드웨어 인스턴스
4     ADC_InitTypeDef       Init;          // 초기화 파라미터
5     DMA_HandleTypeDef     *DMA_Handle;   // DMA 연계 시 사용
6     HAL_LockTypeDef       Lock;          // 동시 접근 방지
7     __IO HAL_ADC_StateTypeDef State;     // 현재 ADC 상태
8     uint32_t              ErrorCode;     // 오류 코드
9 } ADC_HandleTypeDef;
```

즉, `Handle` 은 주변장치의 소프트웨어적 “객체(Object)” 역할을 한다.

HAL 함수는 항상 이 객체를 인자로 받아, 내부 상태를 안전하게 갱신한다.

1.14.4 HAL의 상태 및 리턴 코드

모든 HAL 함수는 `HAL_StatusTypeDef` 열거형을 리턴한다.

```

1  typedef enum
2  {
3      HAL_OK          = 0x00U,  // 정상 수행
4      HAL_ERROR       = 0x01U,  // 오류 발생
5      HAL_BUSY        = 0x02U,  // 자원(Peripheral)이 사용 중
6      HAL_TIMEOUT     = 0x03U   // 응답 지연 또는 타임아웃
7  } HAL_StatusTypeDef;

```

(1) HAL_OK

- 함수가 정상적으로 실행됨
- 하드웨어 동작 완료 및 내부 상태 �신 완료

(2) HAL_ERROR

- 하드웨어 설정 실패 또는 잘못된 인자 전달
- 예: 잘못된 채널, 클록 미활성화, NULL 포인터 등

(3) HAL_BUSY

- 다른 HAL 함수가 동일한 주변장치를 이미 점유 중
- 예: DMA 전송 중에 다시 전송 시도

(4) HAL_TIMEOUT

- 지정한 Timeout(ms) 내에 하드웨어 응답이 없음
- 주로 통신 계열(I²C, UART, SPI 등)에서 발생

HAL 함수의 리턴 코드를 반드시 확인해야 한다.
무시하면 하드웨어 충돌이나 무한 루프의 원인이 될 수 있다.

1.14.5 HAL 함수의 내부 흐름 예시

(예: I²C 송신 함수)

```

1  HAL_StatusTypeDef HAL_I2C_Master_Transmit(I2C_HandleTypeDef *hi2c,
2                                             uint16_t DevAddress,
3                                             uint8_t *pData,
4                                             uint16_t Size,
5                                             uint32_t Timeout)
6  {
7      if(hi2c->State != HAL_I2C_STATE_READY)
8          return HAL_BUSY;
9
10     hi2c->State = HAL_I2C_STATE_BUSY_TX;
11     hi2c->ErrorCode = HAL_I2C_ERROR_NONE;
12
13     // (1) Start 조건 생성
14     I2C_GenerateSTART(hi2c->Instance);
15

```

```

16 // (2) Address 전송 및 ACK 확인
17 if(!I2C_waitOnFlag(hi2c, FLAG_ADDR, Timeout))
18     return HAL_TIMEOUT;
19
20 // (3) 데이터 송신 루프
21 for(uint16_t i = 0; i < Size; i++)
22 {
23     if(!I2C_waitOnTXEFlag(hi2c, Timeout))
24         return HAL_TIMEOUT;
25     hi2c->Instance->DR = pData[i];
26 }
27
28 // (4) stop 조건 생성 및 종료
29 I2C_GenerateSTOP(hi2c->Instance);
30 hi2c->State = HAL_I2C_STATE_READY;
31
32 return HAL_OK;
33 }

```

이처럼 HAL 함수는

“상태 확인 → 플래그 대기 → 동작 수행 → 리턴 코드” 순서로 설계된다.

1.14.6 HAL Lock 메커니즘

다중 스레드나 인터럽트 환경에서 동일 주변장치 접근 충돌을 방지하기 위해 HAL은 내부적으로 Lock을 사용한다.

```

1 __HAL_LOCK(huart); // 주변장치 잠금
2 __HAL_UNLOCK(huart); // 잠금 해제

```

예시:

```

1 if(huart->Lock == HAL_LOCKED)
2     return HAL_BUSY;
3 else
4     huart->Lock = HAL_LOCKED;

```

이 구조 덕분에 FreeRTOS나 다중 인터럽트 환경에서도 HAL 함수의 **재진입성(Reentrancy)**을 보장할 수 있다.

1.14.7 HAL ErrorCode 구조

각 Handle에는 `ErrorCode` 필드가 존재하며, 함수 실패 시 구체적인 오류 유형을 기록한다.

```

1 #define HAL_UART_ERROR_NONE      (0x00000000U)
2 #define HAL_UART_ERROR_PE        (0x00000001U)
3 #define HAL_UART_ERROR_NE        (0x00000002U)
4 #define HAL_UART_ERROR_FE        (0x00000004U)
5 #define HAL_UART_ERROR_ORE        (0x00000008U)
6 #define HAL_UART_ERROR_DMA        (0x00000010U)

```

예:

```

1 if (HAL_UART_Transmit(&huart1, data, len, 1000) != HAL_OK) {
2     printf("UART Error: %lx\r\n", huart1.ErrorCode);
3 }

```

HAL은 단순한 `HAL_ERROR` 리턴 외에도
ErrorCode를 통해 하드웨어 레벨의 오류 원인까지 추적할 수 있게 설계되어 있다.

1.14.8 Timeout 매개변수의 역할

HAL의 통신 계열 함수들은 대부분 `Timeout` 인자를 가진다.
이는 **소프트웨어 대기 루프의 최대 시간(ms)**을 의미하며,
시스템의 안정성을 결정짓는 중요한 요소이다.

```

1 HAL_I2C_Master_Transmit(&hi2c1, addr, data, len, 1000);

```

- `Timeout = 1000` → 최대 1초 동안 응답 대기
- 타임아웃 발생 시 `HAL_TIMEOUT` 반환

실시간 시스템에서는 지나치게 긴 Timeout 설정을 피해야 한다.

1.14.9 HAL 상태 머신 (State Machine)

HAL 내부에는 모든 주변장치의 상태를 표현하는
`HAL_<PERIPH>_StateTypeDef`가 존재한다.

예: `HAL_UART_StateTypeDef`

```

1 typedef enum
2 {
3     HAL_UART_STATE_RESET      = 0x00U,
4     HAL_UART_STATE_READY      = 0x01U,
5     HAL_UART_STATE_BUSY       = 0x02U,
6     HAL_UART_STATE_BUSY_TX    = 0x12U,
7     HAL_UART_STATE_BUSY_RX    = 0x22U,
8     HAL_UART_STATE_TIMEOUT    = 0x03U,
9     HAL_UART_STATE_ERROR      = 0x04U
10 } HAL_UART_StateTypeDef;

```

이 상태는 Handle 내부에서 관리되며,
함수 실행 중 BUSY/READY 상태 전환이 자동으로 수행된다.

1.14.10 리턴 코드 활용 예시

```
1  HAL_StatusTypeDef status;  
2  
3  status = HAL_I2C_Master_Transmit(&hi2c1, addr, buf, len, 100);  
4  
5  switch (status)  
6  {  
7      case HAL_OK:  
8          printf("I2C Transmit success.\n");  
9          break;  
10  
11     case HAL_BUSY:  
12         printf("I2C Bus is busy.\n");  
13         break;  
14  
15     case HAL_TIMEOUT:  
16         printf("I2C Timeout.\n");  
17         break;  
18  
19     case HAL_ERROR:  
20     default:  
21         printf("I2C Error. Code: %lx\n", hi2c1.ErrorCode);  
22         break;  
23 }
```

1.14.11 결론

구분	설명
HAL 함수 구조	표준화된 함수명과 Handle 기반 구조
리턴 코드	HAL_OK, HAL_ERROR, HAL_BUSY, HAL_TIMEOUT
핸들(Handle)	주변장치 상태 및 오류 추적용 구조체
Lock / Timeout	충돌 방지 및 안정성 확보 메커니즘
ErrorCode	하드웨어 세부 오류 디버깅용 코드

HAL 함수의 리턴 코드와 상태 관리를 올바르게 이해하면,
STM32 펌웨어의 **신뢰성, 재진입성, 예외 처리 능력**을 극대화할 수 있다.

• HAL_Delay(), HAL_GetTick(), SysTick Handler 이해

1.15.1 개요

STM32 HAL 라이브러리에서 **시간 지연(Delay)** 과 **시스템 시간 관리**는 **SysTick 타이머(System Tick Timer)** 를 기반으로 수행된다.

- `HAL_Delay()` : 지정한 밀리초(ms) 동안 지연
- `HAL_GetTick()` : 시스템이 시작된 이후 경과한 시간(ms) 반환
- `SysTick_Handler()` : 1ms마다 호출되는 인터럽트 서비스 루틴

이 세 함수는 HAL의 **타임베이스(Time Base)** 메커니즘을 구성하며, 모든 HAL 모듈의 Timeout, Timer, Delay 계산에 사용된다.

1.15.2 SysTick 타이머의 개념

SysTick (System Tick) 은 Cortex-M3 코어 내부에 내장된 **24비트 다운카운터 타이머**로, 운영체제나 HAL의 시간 기반 이벤트를 제공한다.

항목	내용
타이머 크기	24bit 다운카운터
클록 소스	HCLK 또는 HCLK/8
인터럽트 발생 주기	사용자가 설정 (기본: 1ms)
인터럽트 벡터	<code>SysTick_Handler()</code>
레지스터	<code>CTRL</code> , <code>LOAD</code> , <code>VAL</code> , <code>CALIB</code>

SysTick은 HAL에서 **1ms마다 인터럽트를 발생시켜**, 시스템의 “시간 틱(Tick)”을 관리한다.

1.15.3 HAL_GetTick() — 시스템 시간 조회

```
1 | uint32_t HAL_GetTick(void);
```

(1) 기능

시스템이 시작된 이후 경과한 시간을 **밀리초(ms)** 단위로 반환한다.

이 값은 SysTick 인터럽트가 1ms마다 증가시키는 **전역 tick 변수(uwTick)** 을 참조한다.

(2) 내부 동작

```
1  __IO uint32_t uwTick;  
2  
3  uint32_t HAL_GetTick(void)  
4  {  
5      return uwTick;  
6  }
```

(3) 예시

```
1  uint32_t t0 = HAL_GetTick();  
2  while(HAL_GetTick() - t0 < 1000); // 1초 대기
```

`HAL_GetTick()` 은 실시간 지연 루프뿐 아니라,
Timeout 검사, 센서 주기 제어, RTOS Tickless Idle 등
다양한 타임베이스 로직의 기준이 된다.

1.15.4 HAL_Delay() — 소프트웨어 지연 함수

```
1  void HAL_Delay(uint32_t Delay);
```

(1) 기능

입력한 시간(밀리초 단위) 동안 **블로킹(Blocking)** 지연을 수행한다.

(2) 내부 구조

```
1  void HAL_Delay(uint32_t Delay)  
2  {  
3      uint32_t tickstart = HAL_GetTick();  
4      uint32_t wait = Delay;  
5  
6      while((HAL_GetTick() - tickstart) < wait)  
7      {  
8          // Polling wait  
9      }  
10 }
```

즉, `HAL_GetTick()` 값이 증가할 때까지 루프를 돌며 대기한다.

(3) 특징

항목	내용
기반 타이머	SysTick
단위	밀리초(ms)

항목	내용
정확도	± 1 Tick (약 1ms 오차)
CPU 점유율	100% (Busy Wait 방식)
대안	FreeRTOS 사용 시 <code>vTaskDelay()</code> 권장

HAL_Delay()는 단순 지연에는 편리하지만,
CPU가 루프 대기 중 다른 작업을 수행하지 못한다는 한계가 있다.

1.15.5 SysTick_Handler() — 시스템 틱 인터럽트

SysTick 타이머는 설정된 주기(기본 1ms)마다
`SysTick_Handler()` 인터럽트를 발생시킨다.

(1) 기본 구조

`stm32f1xx_it.c` 내부:

```
1 void SysTick_Handler(void)
2 {
3     HAL_IncTick();
4 }
```

(2) HAL_IncTick() 정의

`stm32f1xx_hal.c` 내부:

```
1 void HAL_IncTick(void)
2 {
3     uwTick += uwTickFreq; // 기본적으로 1ms 단위 증가
4 }
```

즉, 1ms마다 전역 변수 `uwTick` 을 +1 증가시킨다.
→ `HAL_GetTick()` 이 이 변수를 읽어 경과 시간을 계산한다.

(3) 흐름 요약

```
1 SysTick Timer (1ms)
2   ↓
3 SysTick_Handler()
4   ↓
5 HAL_IncTick()
6   ↓
7 uwTick++
8   ↓
9 HAL_GetTick() → uwTick 읽음
10  ↓
11 HAL_Delay() → uwTick 차이 계산
```

1.15.6 SysTick 주기 설정

SysTick 주기는 HAL 초기화 시 자동 설정된다.

`HAL_InitTick()` 내부에서 설정 코드가 실행된다:

```
1 HAL_StatusTypeDef HAL_InitTick(uint32_t TickPriority)
2 {
3     HAL_SYSTICK_Config(HAL_RCC_GetHCLKFreq()/1000U); // 1ms
4     HAL_NVIC_SetPriority(SysTick_IRQn, TickPriority, 0U);
5     return HAL_OK;
6 }
```

- `HAL_RCC_GetHCLKFreq()` : 현재 HCLK 주파수 조회
- `/1000U` : 1ms 단위로 인터럽트 발생

예: HCLK = 72MHz → 72,000,000 / 1000 = **72,000 카운트마다 인터럽트 발생**

1.15.7 SysTick 관련 주요 매크로

매크로	설명
<code>SysTick->LOAD</code>	타이머 리로드 값 (인터럽트 주기 설정)
<code>SysTick->VAL</code>	현재 카운터 값
<code>SysTick->CTRL</code>	타이머 제어 비트 (Enable, TickInt, ClockSource 등)

예시:

```
1 SysTick->LOAD = (HAL_RCC_GetHCLKFreq() / 1000) - 1; // 1ms
2 SysTick->VAL  = 0;
3 SysTick->CTRL = SysTick_CTRL_CLKSOURCE_Msk |
4                 SysTick_CTRL_TICKINT_Msk   |
5                 SysTick_CTRL_ENABLE_Msk;
```

1.15.8 사용자 정의 Tick 주기 변경

Tick 주기를 1ms → 10ms로 바꾸려면

`HAL_InitTick()` 또는 `HAL_SYSTICK_Config()` 호출 시 나눗값을 변경한다.

```
1 HAL_SYSTICK_Config(HAL_RCC_GetHCLKFreq() / 100); // 10ms 단위
```

또는 HAL 전역 변수 변경:

```
1 uwTickFreq = 10U; // 10ms per tick
```

단, Tick 주기를 변경하면 Delay, Timeout 등의 모든 HAL 함수 시간 계산이 동일하게 변하므로 주의해야 한다.

1.15.9 FreeRTOS와 SysTick

FreeRTOS를 사용할 경우 SysTick은 HAL이 아닌 커널 타이머로 재할당된다.

- HAL의 `SysTick_Handler()` 대신 FreeRTOS의 `xPortSysTickHandler()` 가 호출된다.
- HAL 함수(`HAL_Delay()`)는 RTOS의 Tick Hook으로 동작하도록 재정의된다.

```
1 void SysTick_Handler(void)
2 {
3     osSysTickHandler(); // FreeRTOS 전용 SysTick 핸들러
4 }
```

FreeRTOS 환경에서는 `HAL_Delay()` 대신 `vTaskDelay()` 또는 `osDelay()` 사용이 권장된다.

1.15.10 정리

구성 요소	설명	역할
SysTick Timer	Cortex-M3 내장 24bit 타이머	1ms마다 인터럽트 발생
SysTick_Handler()	인터럽트 루틴	HAL_IncTick() 호출
HAL_IncTick()	내부 Tick 변수 증가	시스템 시간 갱신
HAL_GetTick()	현재 Tick 값 반환	시간 조회
HAL_Delay()	지정 시간 대기	Busy Wait 방식 지연

1.15.11 결론

`HAL_Delay()`, `HAL_GetTick()`, `SysTick_Handler()` 는 HAL의 모든 시간 기반 함수들의 **기초 인프라**를 형성한다.

즉, 이 세 함수는 **STM32 HAL의 시간 제어의 핵심 축**으로, 지연(Delay), 타임아웃, 태스크 스케줄링, 주기적 이벤트 처리 등 모든 동작의 “시간 기준(Time Base)”을 제공한다.