4. ADC (Analog to Digital Converter)

4.1 ADC 구조

- ## 샘플링, 홀드, 변환 과정

1. 개요

STM32의 **ADC(Analog-to-Digital Converter)** 는
입력 전압(아날로그 신호)을 일정 시간 간격으로 측정하여
디지털 이진수(0~4095) 로 변환한다.

이 변환 과정은 세 단계로 구분된다.

```
Analog Input → [샘플링] → [홀드] → [변환(Conversion)] → Digital Value
```

각 단계는 ADC 하드웨어 내부의 **Sample-and-Hold 회로**와
Successive Approximation Register (SAR) 에 의해 수행된다.

---

2. 샘플링(Sampling)

정의

입력 신호(전압)를 짧은 시간 동안 ADC 내부의 **샘플링 커패시터(Csample)** 에 충전하는 과정이다.

원리

- 입력 핀(AINx)은 내부적으로 스위치를 통해 커패시터에 연결된다.
- 이 커패시터가 입력 전압에 따라 충전되는 동안 신호를 “샘플링”한다.
- 이 시간이 충분히 짧으면 입력 신호의 “순간 값”을 정확히 포착할 수 있다.

수식

$$t_{sample} = (Sampling\,Cycles + 12.5) \times \frac{1}{f_{ADC}}$$

> Sampling Time은 CubeMX에서 설정 가능
> (예: 1.5, 7.5, 13.5, 28.5, 41.5, 55.5, 71.5, 239.5 cycles 등)

예시

- ADC 클록: 12 MHz
- Sampling Cycles: 55.5

$$t_{sample} = (55.5 + 12.5)/12\,MHz \approx 5.7\,\mu s$$

---

3. 홀드(Hold)

정의

샘플링이 끝나면 스위치가 차단되어 커패시터에 전하가 고정(“홀드”)된다.
 이 동안에는 ADC가 입력값을 고정시킨 채 내부 변환 연산을 수행한다.

역할

- 입력 신호가 변하더라도 변환 중에는 고정된 값만을 참조
- 정확한 변환을 위해 반드시 필요
- 커패시터 용량이 너무 작거나 소스 임피던스가 높으면 충전이 불완전해져 오차 발생

주의

입력 임피던스(Rin)와 소스 저항(Rsource)에 따라 샘플링 시간이 충분히 길어야 함.

$$t_{sample} > 5 \times R_{source} \times C_{sample}$$

---

4. 변환(Conversion)

정의

고정된 전압값을 기준으로 SAR(Successive Approximation Register) 방식으로
 디지털 이진수로 변환하는 단계.

SAR 구조

- **비교기(Comparator)**: 기준전압(Vref)과 샘플된 전압 비교
- **DAC**: 단계적으로 전압을 생성하여 입력 전압과 비교
- **레지스터(SAR)**: 각 비트를 하나씩 결정 (MSB → LSB)

과정 예시 (12-bit ADC)

단계	DAC 전압	비교 결과	비트 결정
1	Vref/2	Vin < Vref/2	b11 = 0
2	Vref/4	Vin > Vref/4	b10 = 1
3	…	…	…
12	최종 비교	→ 12-bit Digital Output	

> 한 번의 변환에는 **12.5 ADC 클록 사이클**이 소요됨.

---

5. 전체 동작 흐름

```
┌──────────────────────────────────────────┐
|            ADC 변환 순서도               |
├──────────────────────────────────────────┤
| ① 입력 전압 샘플링 (Sampling)            |
| ② 커패시터 충전 및 스위치 오픈 (Hold)    |
| ③ SAR 비교 및 비트 결정 (Conversion)     |
| ④ 결과 레지스터에 저장 (DR Register)     |
| ⑤ 다음 채널 변환으로 이동                |
└──────────────────────────────────────────┘
```

---

6. 변환 시간 계산 예시

항목	값
ADC 클록	12 MHz
Sampling Cycles	55.5
변환 Cycles	12.5
총 변환시간	(55.5 + 12.5) / 12 MHz = **5.67 µs**
변환속도	1 / 5.67µs ≈ **176 kSPS** (샘플/초)

---

7. 변환 결과

- ADC 결과는 **12비트 (0~4095)** 로 표현됨
- 입력전압 V_{in} 과 출력값 D의 관계:

$$D = \frac{V_{in}}{V_{ref}} \times 4095$$

예:

- $V_{ref} = 3.3V$
- $D = 2048$

$$V_{in} = \frac{2048}{4095} \times 3.3V \approx 1.65V$$

---

8. 전체 개념 요약

단계	이름	동작	하드웨어 구성	시간단위
①	샘플링	입력 신호를 커패시터에 충전	스위치 + Csample	수 µs
②	홀드	충전 전압 고정	Csample 유지	수 µs

단계	이름	동작	하드웨어 구성	시간단위
③	변환	SAR 비교로 디지털값 계산	Comparator + DAC	수 μs

9. 시스템 설계 시 고려사항

항목	설명
ADC 클록 주파수	너무 높으면 샘플링 불완전, 너무 낮으면 속도 저하
입력 임피던스	높은 소스 임피던스 → 샘플링 시간 증가 필요
Vref 안정성	기준 전압이 불안정하면 전체 측정값 오차 발생
DMA 연속 변환	다채널 계측 시 CPU 부하 감소에 효과적

✅ 핵심 요약

항목	개념	설명
Sampling	신호를 커패시터에 충전	입력 전압 포착
Hold	충전된 전압을 유지	변환 중 안정성 확보
Conversion	SAR로 디지털 값 변환	12비트 결과 생성
총 변환시간	(Sampling + 12.5) / fADC	예: 5.6μs
결과값	0~4095	Vin = D × Vref / 4095

> STM32의 ADC는 내부 샘플링-홀드-변환 과정을 자동으로 수행하므로
> 사용자는 단지 **Sampling Time과 클록 속도**만 적절히 조정하면 된다.
> 이 세 과정의 균형이 ADC의 **정확도와 속도**를 결정짓는다.

- # 분해능(Resolution), 변환속도, 채널 선택

1. 개요

STM32의 **ADC(Analog-to-Digital Converter)** 는
하나의 MCU 내부에서 여러 아날로그 신호를 디지털 값으로 변환하기 위해
다양한 **분해능(Resolution)**, **속도(Conversion Speed)**,
채널 선택(Channel Selection) 기능을 제공한다.

이 세 요소는 ADC 성능을 결정하는 핵심 파라미터이다.

2. 분해능 (Resolution)

(1) 정의

> 분해능이란 ADC가 구분할 수 있는 **최소 전압 단위(LSB, Least Significant Bit)** 의 크기를 의미한다.
> 즉, 입력 아날로그 전압을 몇 단계의 디지털 값으로 표현할 수 있는지를 나타낸다.

(2) STM32F103C8T6의 분해능

- 12비트 ADC 사용 → **0~4095 (2^{12}–1)**
- 입력 전압 범위: 0 ~ Vref (보통 3.3V)

(3) 변환 공식

$$\text{Digital Output (D)} = \frac{V_{in}}{V_{ref}} \times (2^{Resolution} - 1)$$

(4) 예시

Resolution	Digital Range	1 LSB(3.3V 기준)	설명
8-bit	0~255	12.94 mV	빠르지만 정밀도 낮음
10-bit	0~1023	3.22 mV	중간 수준
12-bit	**0~4095**	**0.805 mV**	STM32F1 기본
16-bit	0~65535	0.05 mV	고정밀 (외부 ADC 사용 시)

> STM32F103의 ADC는 고정 12-bit이므로,
> 결과값은 0~4095 범위의 정수로 표현된다.

---

3. 변환속도 (Conversion Speed)

(1) 정의

> ADC가 아날로그 입력을 디지털로 변환하는 데 걸리는 시간.
> 즉, 초당 샘플링 가능한 횟수(Samples per Second, SPS)를 나타낸다.

(2) 기본 공식

$$t_{conv} = \frac{(T_{sample} + 12.5)}{f_{ADC}}$$

- `T_sample` : 샘플링 클록 수 (1.5~239.5 cycles)
- `12.5` : SAR 변환에 필요한 고정 클록 수
- `f_ADC` : ADC 클록 주파수 (최대 14 MHz, STM32F1 기준)

(3) 예시

Sampling Cycles	fADC (MHz)	변환시간 (μs)	샘플링 속도 (kSPS)
1.5	12	1.17 μs	855 kSPS
7.5	12	1.67 μs	598 kSPS
55.5	12	5.67 μs	176 kSPS
239.5	12	21 μs	47 kSPS

> Sampling Cycles가 짧을수록 속도는 빠르지만,
> 입력 임피던스가 높거나 노이즈가 많은 회로에서는 정확도가 떨어진다.

---

(4) ADC 클록 설정

- ADC 클록은 APB2 클록을 분주하여 설정됨
  (예: 72 MHz → /6 → 12 MHz)
- CubeMX 또는 레지스터에서 설정 가능
  (`RCC_CFGR_ADCPRE = 0b10` → divide by 6)

```
// 실제 클록 확인 (HAL 코드 예시)
uint32_t adc_clk = HAL_RCC_GetPCLK2Freq() / 6; // 12 MHz
```

---

(5) 샘플링 시간과 정확도의 관계

Sampling Time	특징
1.5~13.5 cycles	빠른 변환, 저임피던스 신호에 적합
55.5~239.5 cycles	안정적 변환, 고임피던스 센서(저항분압 등)에 적합
권장값	55.5 또는 71.5 (일반 센서 계측 시)

---

4. 채널 선택 (Channel Selection)

(1) ADC 채널 구조

STM32F103C8T6에는 **16개 입력 채널**이 있다.

구분	채널 번호	핀
외부 입력	ADC_IN0 ~ ADC_IN15	PA0~PA7, PB0~PB1, PC0~PC5
내부 채널	ADC_IN16, ADC_IN17	온도 센서, Vrefint

> 각 채널은 멀티플렉서(MUX)를 통해 하나의 ADC에 순차적으로 연결된다.

(2) 단일 변환 모드 (Single Conversion Mode)

- 한 번에 하나의 채널만 변환
- 변환 완료 시 `EOC(End Of Conversion)` 플래그 세트
- 폴링 또는 인터럽트로 결과 확인

```
HAL_ADC_Start(&hadc1);
HAL_ADC_PollForConversion(&hadc1, 10);
uint32_t val = HAL_ADC_GetValue(&hadc1);
```

(3) 스캔 모드 (Scan Conversion Mode)

- 여러 채널을 **순차 변환(Sequential Conversion)**
- 내부적으로 ADC가 MUX를 자동 변경
- 각 채널별로 순차 결과를 DMA 또는 인터럽트로 읽기 가능

```
ADC_ChannelConfTypeDef sConfig = {0};

sConfig.Channel = ADC_CHANNEL_0;
sConfig.Rank = 1;
HAL_ADC_ConfigChannel(&hadc1, &sConfig);

sConfig.Channel = ADC_CHANNEL_1;
sConfig.Rank = 2;
HAL_ADC_ConfigChannel(&hadc1, &sConfig);

HAL_ADC_Start_DMA(&hadc1, (uint32_t*)adc_buf, 2);
```

(4) 내부 채널

채널	기능	설명
ADC_IN16	온도 센서	내부 다이 온도 측정 (`T = (V25 - Vsense) / Avg_Slope + 25°C`)
ADC_IN17	Vrefint	내부 기준전압 측정 (보정 및 배터리 감지용)

(5) 다중 ADC 연동 (Dual ADC Mode)

STM32F1 시리즈의 일부 MCU는 ADC1, ADC2를 동시에 구동하여
샘플링 속도를 두 배로 높일 수 있다.

- **Regular Simultaneous Mode** : 동시 변환
- **Injected Simultaneous Mode** : 이벤트 트리거 기반
- **Interleaved Mode** : 샘플링 간격을 반씩 나눔 (속도 2배)

5. 예시 — 단일채널 전압 측정

```c
float Read_ADC_Voltage(void)
{
  HAL_ADC_Start(&hadc1);
  HAL_ADC_PollForConversion(&hadc1, HAL_MAX_DELAY);
  uint32_t value = HAL_ADC_GetValue(&hadc1);
  return (value * 3.3f) / 4095.0f;
}
```

---

6. 예시 — 다채널 DMA 변환

```c
uint32_t adc_buffer[3]; // CH0: 전압, CH1: 전류, CH2: 온도

void ADC_Read_All(void)
{
  HAL_ADC_Start_DMA(&hadc1, adc_buffer, 3);
  HAL_Delay(10);
  float v = adc_buffer[0] * 3.3f / 4095.0f;
  float i = adc_buffer[1] * 3.3f / 4095.0f;
  float t = ((1.43 - adc_buffer[2]*3.3f/4095.0f) / 0.0043) + 25.0f;
}
```

---

7. 요약

항목	내용	예시
분해능(Resolution)	변환된 디지털 값의 비트 수	12-bit → 0~4095
LSB 크기	$V_{ref}/(2^{Resolution}-1)$	3.3V/4095 ≈ 0.805mV
변환속도	$t_{conv} = (T_{sample} + 12.5)/f_{ADC}$	약 5μs
채널 수	16개 외부 + 2개 내부	ADC_IN0~17
모드	Single / Scan / DMA / Dual	다채널 계측 가능

✅ 핵심 요약

구분	설명
Resolution	ADC의 정밀도 (12bit = 4096 단계)
Conversion Speed	샘플링 및 변환에 소요되는 시간 (μs 단위)
Channel Selection	여러 아날로그 입력 중 선택적 측정 가능

구분	설명
설계 포인트	고속 → 짧은 샘플링, 고정밀 → 긴 샘플링
결론	STM32F103의 12bit ADC는 0.8mV 해상도, 약 170kSPS 속도, 16채널 입력을 지원한다.

> 정리하자면,
> **Resolution**은 ADC의 정밀도를,
> **Conversion Speed**는 처리 속도를,
> **Channel Selection**은 입력 확장성을 의미한다.
> 이 세 요소의 조합이 시스템의 **정확도, 응답성, 효율성**을 결정한다.

- # 내부 온도센서 및 Vref 측정

1. 개요

STM32F103 시리즈의 ADC에는
외부 센서 외에도 내부적으로 **온도센서(Temperature Sensor)** 와
기준전압(Vrefint, Internal Reference Voltage) 이 내장되어 있다.

이들은 별도의 핀 연결 없이 ADC 채널을 통해 직접 측정할 수 있으며,
시스템 온도 보정, **전압 안정성 확인**, **배터리 모니터링** 등의 용도로 사용된다.

---

2. 내부 온도센서 (Temperature Sensor)

(1) 채널 및 구조

항목	내용
ADC 채널	`ADC_IN16`
신호 출력	내부 다이(Die) 온도에 비례한 전압
전압 범위	약 1.43V (25°C 기준)
민감도	4.3mV / °C (평균값)
동작 온도 범위	-40°C ~ +125°C

온도센서는 **밴드갭(Bandgap) 다이오드 전압 변화**를 이용해 칩의 내부 온도를 감지한다.
변환 결과는 ADC를 통해 읽을 수 있으며, 아래의 보정 식으로 실제 온도로 환산한다.

---

(2) 변환 공식

$$T(^\circ C) = (V_{25} - V_{SENSE})/AvgSlope + 25$$

파라미터	의미	STM32F1 Typical 값
V25	25°C에서의 출력 전압	1.43 V

파라미터	의미	STM32F1 Typical 값
Avg_Slope	온도당 전압 변화율	4.3 mV/°C
VSENSE	ADC 측정 전압	(ADC_Value × 3.3 / 4095)

즉,

$$T = (1.43 - (ADC \times 3.3/4095))/0.0043 + 25$$

---

(3) 예시 코드

```
float Read_Internal_Temperature(void)
{
  ADC_ChannelConfTypeDef sConfig = {0};
  sConfig.Channel = ADC_CHANNEL_TEMPSENSOR;  // ADC_IN16
  sConfig.Rank = 1;
  sConfig.SamplingTime = ADC_SAMPLETIME_239CYCLES_5; // 충분한 시간 필요
  HAL_ADC_ConfigChannel(&hadc1, &sConfig);

  HAL_ADC_Start(&hadc1);
  HAL_ADC_PollForConversion(&hadc1, HAL_MAX_DELAY);

  uint32_t adc_val = HAL_ADC_GetValue(&hadc1);
  float vsense = (adc_val * 3.3f) / 4095.0f;
  float temp = ((1.43f - vsense) / 0.0043f) + 25.0f;

  return temp;
}
```

> 온도센서는 내부 회로의 출력 임피던스가 높으므로,
> 반드시 **긴 샘플링 시간(≥ 239.5 cycles)** 을 설정해야 한다.

---

3. 내부 기준전압 (Vrefint)

(1) 개요

항목	내용
ADC 채널	`ADC_IN17`
출력 전압	약 **1.20V** (정확도 ±1%)
용도	전원 전압(Vdd) 모니터링, ADC 보정
특징	온도 및 전압 변화에 따라 거의 일정

Vrefint는 **ADC 기준전압(Vref)** 이 변하더라도
내부 고정 기준(약 1.2V)을 측정해 이를 보정하는 데 사용된다.

---

(2) 측정 공식

$$V_{DD} = 1.20V \times \frac{4095}{ADC_{Vrefint}}$$

> `ADC_Vrefint` : Vrefint 채널의 변환 결과 값

즉, Vrefint가 낮게 측정되면 실제 Vdd가 높다는 뜻이다.

---

(3) 예시 코드

```
float Read_Vdd(void)
{
  ADC_ChannelConfTypeDef sConfig = {0};
  sConfig.Channel = ADC_CHANNEL_VREFINT;     // ADC_IN17
  sConfig.Rank = 1;
  sConfig.SamplingTime = ADC_SAMPLETIME_239CYCLES_5;
  HAL_ADC_ConfigChannel(&hadc1, &sConfig);

  HAL_ADC_Start(&hadc1);
  HAL_ADC_PollForConversion(&hadc1, HAL_MAX_DELAY);

  uint32_t adc_val = HAL_ADC_GetValue(&hadc1);
  float vdd = 1.20f * 4095.0f / (float)adc_val;

  return vdd; // 실제 MCU 동작 전압(V)
}
```

---

4. 온도센서 + Vrefint 동시 측정 예시 (DMA 방식)

```
uint32_t adc_buf[2]; // [0] = Temp, [1] = Vref

void ADC_Read_Internal(void)
{
  ADC_ChannelConfTypeDef sConfig = {0};

  sConfig.SamplingTime = ADC_SAMPLETIME_239CYCLES_5;

  sConfig.Channel = ADC_CHANNEL_TEMPSENSOR;
  sConfig.Rank = 1;
  HAL_ADC_ConfigChannel(&hadc1, &sConfig);

  sConfig.Channel = ADC_CHANNEL_VREFINT;
  sConfig.Rank = 2;
  HAL_ADC_ConfigChannel(&hadc1, &sConfig);

  HAL_ADC_Start_DMA(&hadc1, adc_buf, 2);
}
```

> DMA를 사용하면 두 내부 채널을 자동으로 순차 변환하여
> CPU 개입 없이 실시간으로 데이터 수집이 가능하다.

---

5. 계산 예시

항목	ADC 결과	계산식	결과
온도	1720	$(1.43 - 1720 \times 3.3/4095)/0.0043 + 25$	약 36°C
Vrefint	1470	$1.20 \times 4095/1470$	약 3.34V

---

6. 설계 시 고려사항

항목	설명
샘플링 시간	반드시 239.5 cycles 이상
ADC 클록	12 MHz 이하 권장
온도센서 보정	STM32F1은 공장 보정값이 없음 → 소프트웨어 보정 필요
Vrefint 안정성	VDD 안정화 후 최소 5μs 대기 필요
동시 측정	내부 채널 측정 시 외부 채널과 혼합 가능

---

7. 응용 예시

용도	활용 예시
온도 보정	온도 변화에 따른 센서 오차 보정
전원 모니터링	배터리 전압 저하 감지
자체 진단(Self-Diagnostic)	내부 온도 상승 감지 → 시스템 보호
전력 최적화	Vrefint 기반 전압 보정 후 Sleep 제어

---

8. 요약

항목	채널	전압	주요 용도
온도센서	ADC_IN16	1.43V @ 25°C	내부 온도 감지
Vrefint	ADC_IN17	약 1.20V	VDD 보정, 전압 측정
필요 샘플링 시간	≥ 239.5 cycles	—	—
ADC 클록	≤ 14 MHz	—	—

✅ 핵심 요약

구분	공식	설명
온도 계산	`T = (1.43 - Vsense) / 0.0043 + 25`	내부 다이 온도
VDD 계산	`VDD = 1.20 × 4095 / ADC_Vrefint`	MCU 동작 전압
주의	긴 샘플링 시간, 안정화 시간 확보 필요	측정 오차 방지

> 요약하자면,
> STM32F103의 **ADC 내부 채널**은
> 칩의 "**자기 상태**(온도, 전압)"를 직접 감지할 수 있게 해주는 강력한 기능이다.
> 외부 센서 없이도 **온도 감시 + 전압 보정**을 수행할 수 있으며,
> 고신뢰 시스템(산업용, IoT, 배터리 기기)에서 특히 유용하다.

4.2 HAL ADC 함수

- `HAL_ADC_Start()`, `HAL_ADC_PollForConversion()`

1. 개요

STM32 HAL 라이브러리에서 ADC(Analog to Digital Converter)는
3단계 절차로 동작한다:

```
① 변환 시작 → ② 변환 완료 대기 → ③ 결과 읽기
```

이 과정에서 사용되는 대표적인 두 함수가
`HAL_ADC_Start()`와 `HAL_ADC_PollForConversion()`이다.

- `HAL_ADC_Start()` → **ADC 변환 시작(트리거)**
- `HAL_ADC_PollForConversion()` → **변환 완료 대기 및 상태 확인**

---

2. `HAL_ADC_Start()`

(1) 기능

> ADC 변환을 **시작(Start Conversion)** 시키는 함수.
> 내부적으로 ADC의 `ADON` 비트를 활성화하여 변환 프로세스를 시작한다.

(2) 함수 원형

```
HAL_StatusTypeDef HAL_ADC_Start(ADC_HandleTypeDef *hadc);
```

매개변수	설명
`hadc`	ADC 핸들 구조체 포인터 (예: &hadc1)

(3) 내부 동작 요약

1. `ADON` 비트를 설정 (ADC 전원 ON)
2. 변환 준비 상태 확인
3. `SWSTART` 비트를 세트하여 소프트웨어 변환 시작
4. `EOC` (End of Conversion) 플래그가 세트될 때까지 진행

(4) 주의사항

- ADC는 반드시 **초기화 이후**(`HAL_ADC_Init()`)에 호출해야 한다.
- 동일 ADC를 반복 측정 시에는 변환 종료 후 다시 호출해야 한다.

(5) 예시

```
HAL_ADC_Start(&hadc1);  // ADC1 변환 시작
```

> 이후 `HAL_ADC_PollForConversion()` 또는 인터럽트/DMA 방식으로 결과를 읽는다.

---

3. `HAL_ADC_PollForConversion()`

(1) 기능

> 변환 완료 플래그(`EOC`)가 세트될 때까지 **폴링(Polling)** 방식으로 대기한다.
> 주어진 Timeout 내에 완료되지 않으면 `HAL_TIMEOUT` 반환.

(2) 함수 원형

```
HAL_StatusTypeDef HAL_ADC_PollForConversion(ADC_HandleTypeDef *hadc, uint32_t Timeout);
```

매개변수	설명
`hadc`	ADC 핸들 구조체
`Timeout`	변환 완료를 기다릴 최대 시간 (ms 단위)

(3) 반환값

반환값	의미
`HAL_OK`	변환 완료
`HAL_TIMEOUT`	시간 초과

반환값	의미
`HAL_ERROR`	변환 중 오류 발생

(4) 내부 동작

1. `EOC` (End Of Conversion) 플래그 확인 루프
2. Timeout 경과 시 반환
3. 완료 시 플래그 클리어 및 다음 변환 준비

---

4. 일반적인 사용 흐름

```
HAL_ADC_Start(&hadc1);                          // 변환 시작
HAL_ADC_PollForConversion(&hadc1, 10);          // 변환 완료 대기 (10ms 제한)
uint32_t adc_val = HAL_ADC_GetValue(&hadc1);    // 결과 읽기
```

> 위 3단계는 **Polling 모드 단일 변환(Single Conversion)** 의 기본 패턴이다.

---

5. 전체 코드 예시

```
float Read_ADC_Voltage(void)
{
  HAL_ADC_Start(&hadc1);                            // ① 변환 시작
  HAL_ADC_PollForConversion(&hadc1, HAL_MAX_DELAY);  // ② 완료 대기
  uint32_t value = HAL_ADC_GetValue(&hadc1);         // ③ 결과 읽기

  float voltage = (value * 3.3f) / 4095.0f;          // 12bit 기준
  return voltage;
}
```

> 이 함수는 아날로그 입력 전압(0~3.3V)을 실수 형태로 반환한다.

---

6. 다중 채널 변환 예시

```
float voltages[3];
uint32_t adc_channels[3] = {ADC_CHANNEL_0, ADC_CHANNEL_1, ADC_CHANNEL_2};

for (int i = 0; i < 3; i++)
{
  ADC_ChannelConfTypeDef sConfig = {0};
  sConfig.Channel = adc_channels[i];
  sConfig.Rank = ADC_REGULAR_RANK_1;
  sConfig.SamplingTime = ADC_SAMPLETIME_55CYCLES_5;
  HAL_ADC_ConfigChannel(&hadc1, &sConfig);

  HAL_ADC_Start(&hadc1);
```

```
    HAL_ADC_PollForConversion(&hadc1, HAL_MAX_DELAY);

    uint32_t raw = HAL_ADC_GetValue(&hadc1);
    voltages[i] = (raw * 3.3f) / 4095.0f;
}
```

> `HAL_ADC_Start()`와 `HAL_ADC_PollForConversion()`을 반복 호출하여
> 다채널 입력을 순차 측정할 수 있다.

---

7. 인터럽트 / DMA 방식과 비교

모드	특징	함수
Polling	CPU가 변환 완료까지 대기	`HAL_ADC_Start()` + `HAL_ADC_PollForConversion()`
Interrupt	변환 완료 시 자동 콜백 호출	`HAL_ADC_Start_IT()` + `HAL_ADC_ConvCpltCallback()`
DMA	연속 변환 → 메모리에 자동 저장	`HAL_ADC_Start_DMA()`

> Polling 방식은 구현이 간단하지만 CPU 점유율이 높다.
> 실시간 시스템에서는 **DMA 또는 Interrupt 방식**이 권장된다.

---

8. 내부 레지스터 변화

단계	주요 레지스터	설명
`HAL_ADC_Start()`	`CR2.ADON=1`, `CR2.SWSTART=1`	변환 시작
`HAL_ADC_PollForConversion()`	`SR.EOC` 감시	변환 완료 확인
`HAL_ADC_GetValue()`	`DR` 읽기	디지털 변환 결과 획득

---

9. 디버깅 포인트

문제	원인	해결책
변환 결과가 0	ADC 클록 미설정	RCC_CFGR_ADCPRE 확인
Polling Timeout 발생	SamplingTime 너무 길거나 클록 저하	Timeout 증가 또는 Sampling 조정
이상한 값 출력	채널 설정 누락	`HAL_ADC_ConfigChannel()` 호출 확인
불안정한 값	Floating 핀 입력	풀다운 저항 또는 안정된 신호 연결

---

10. 요약

항목	함수	역할
변환 시작	`HAL_ADC_Start()`	ADC 트리거
완료 대기	`HAL_ADC_PollForConversion()`	변환 종료 감시
결과 읽기	`HAL_ADC_GetValue()`	변환된 디지털 값
해상도	12-bit (0~4095)	0.805 mV 단위
응용 예시	전압계, 센서 입력, 내부 온도 측정	—

✅ 핵심 요약

함수	기능	특징
`HAL_ADC_Start()`	ADC 변환 시작	내부 SWSTART 트리거
`HAL_ADC_PollForConversion()`	변환 완료 대기	Timeout 내 완료 확인
결합 사용	Polling 모드에서 기본 구조	단일 채널, 저속 계측에 적합
대안	`HAL_ADC_Start_IT()`, `HAL_ADC_Start_DMA()`	고속, 비동기 처리용

> 결론적으로,
> `HAL_ADC_Start()`와 `HAL_ADC_PollForConversion()`은
> **ADC Polling 모드의 핵심 2단계 함수**이며,
> 간단한 단일 센서 측정에는 이상적이지만,
> 실시간 멀티센서 시스템에서는 DMA 방식으로의 확장이 필수적이다.

- # `HAL_ADC_GetValue()`, `HAL_ADC_Stop()`

1. 개요

STM32 HAL 라이브러리에서 ADC(Analog to Digital Converter)의 변환은
`Start → PollForConversion → GetValue → Stop` 의 순서로 진행된다.

이 중

- `HAL_ADC_GetValue()` : **ADC 변환 결과(디지털 값)** 를 읽는 함수
- `HAL_ADC_Stop()` : **ADC 변환을 중단(ADC 종료)** 시키는 함수

로, 변환 루틴의 마무리 단계에서 매우 중요하다.

---

2. `HAL_ADC_GetValue()`

(1) 기능

> 변환이 완료된 후, **ADC 데이터 레지스터(DR)** 에 저장된
> 디지털 값을 읽어 반환한다.

(2) 함수 원형

```
uint32_t HAL_ADC_GetValue(ADC_HandleTypeDef *hadc);
```

매개변수	설명
`hadc`	ADC 핸들 구조체 포인터 (예: &hadc1)

(3) 반환값

타입	의미
`uint32_t`	변환된 디지털 값 (0 ~ 4095, 12비트)

> 결과는 **12비트 정수값**이지만, 리턴 타입은 32비트로 확장되어 있다.
> 따라서 추가 연산 없이 그대로 실수 변환 가능하다.

---

(4) 내부 동작

1. `SR.EOC` (End Of Conversion) 플래그가 세트되어야 호출 가능
2. `DR` (Data Register)의 값을 읽고 반환
3. `EOC` 플래그가 자동으로 클리어됨

---

(5) 예시 코드

```
HAL_ADC_Start(&hadc1);
HAL_ADC_PollForConversion(&hadc1, HAL_MAX_DELAY);
uint32_t raw = HAL_ADC_GetValue(&hadc1);
float voltage = (raw * 3.3f) / 4095.0f; // 12비트 → 전압 변환
```

---

(6) 다중 채널 (Scan Mode) 예시

```
uint32_t adc_val[3];

for (int i = 0; i < 3; i++)
{
  HAL_ADC_Start(&hadc1);
  HAL_ADC_PollForConversion(&hadc1, 10);
  adc_val[i] = HAL_ADC_GetValue(&hadc1);
}

float v0 = adc_val[0] * 3.3f / 4095.0f;
float v1 = adc_val[1] * 3.3f / 4095.0f;
float v2 = adc_val[2] * 3.3f / 4095.0f;
```

> 각 채널별로 반복 측정 시 `HAL_ADC_GetValue()`를 이용해 결과를 저장한다.

---

(7) 실수 전압 변환 공식

$$V_{in} = \frac{ADC_{value}}{2^{Resolution} - 1} \times V_{ref}$$

예:
Resolution = 12bit, Vref = 3.3V, ADC = 2048
$\rightarrow 2048/4095 \times 3.3V = 1.65V$

---

3. `HAL_ADC_Stop()`

(1) 기능

> ADC 변환을 **종료(Stop Conversion)** 시키는 함수.
> 내부적으로 변환 트리거, DMA, 인터럽트를 비활성화하고
> ADC를 Idle 상태로 되돌린다.

(2) 함수 원형

```
HAL_StatusTypeDef HAL_ADC_Stop(ADC_HandleTypeDef *hadc);
```

매개변수	설명
`hadc`	ADC 핸들 구조체

(3) 반환값

반환값	의미
`HAL_OK`	정상적으로 종료
`HAL_ERROR`	변환 중 오류 발생

반환값	의미
`HAL_TIMEOUT`	내부 타임아웃 발생

---

(4) 내부 동작 요약

단계	설명
1	`CR2.ADON` 비트를 클리어 → ADC 비활성화
2	DMA 및 인터럽트 비활성화
3	변환 중이던 채널 즉시 종료
4	ADC 상태를 `READY`로 복귀

---

(5) 단일 변환 후 종료 예시

```
HAL_ADC_Start(&hadc1);
HAL_ADC_PollForConversion(&hadc1, 10);
uint32_t adc_val = HAL_ADC_GetValue(&hadc1);
HAL_ADC_Stop(&hadc1);
```

> ADC를 반복적으로 사용할 경우, 변환 종료 후 반드시 `Stop()`을 호출해야 한다.
> 그렇지 않으면 다음 변환 시 "busy" 상태로 인해 값이 갱신되지 않거나
> `HAL_BUSY` 오류가 발생할 수 있다.

---

(6) DMA / IT 모드에서의 역할

모드	종료 함수
Polling	`HAL_ADC_Stop()`
Interrupt	`HAL_ADC_Stop_IT()`
DMA	`HAL_ADC_Stop_DMA()`

> 각 모드에 맞는 종료 함수를 사용해야 한다.
> 예: DMA 모드에서 `HAL_ADC_Stop()`을 호출하면 DMA 채널이 비정상 종료될 수 있다.

---

4. Polling 기반 ADC 전체 시퀀스

```
float Read_Analog(void)
{
  HAL_ADC_Start(&hadc1);                                  // ① 변환 시작
  HAL_ADC_PollForConversion(&hadc1, HAL_MAX_DELAY);  // ② 변환 완료 대기
  uint32_t raw = HAL_ADC_GetValue(&hadc1);            // ③ 결과 읽기
  HAL_ADC_Stop(&hadc1);                                   // ④ 변환 종료

  return (raw * 3.3f) / 4095.0f;                      // ⑤ 전압 변환
}
```

> 위 함수는 하나의 ADC 입력을 안정적으로 측정하고 종료하는 **표준 구조**이다.
> FreeRTOS 환경에서도 그대로 사용 가능하다.

---

5. 레지스터 기반 동작 흐름

순서	레지스터	설명
`HAL_ADC_Start()`	`CR2.ADON=1`, `CR2.SWSTART=1`	변환 시작
`HAL_ADC_PollForConversion()`	`SR.EOC` 감시	변환 완료 확인
`HAL_ADC_GetValue()`	`DR` 읽기	결과 획득
`HAL_ADC_Stop()`	`CR2.ADON=0`	변환 종료 및 Idle 복귀

---

6. 실제 응용 예시 — 전압 측정 루프

```
while (1)
{
  HAL_ADC_Start(&hadc1);
  HAL_ADC_PollForConversion(&hadc1, 10);
  uint32_t raw = HAL_ADC_GetValue(&hadc1);
  HAL_ADC_Stop(&hadc1);

  float voltage = raw * 3.3f / 4095.0f;
  printf("ADC Voltage: %.3f V\r\n", voltage);
  HAL_Delay(500);
}
```

→ 결과 예시:

```
ADC Voltage: 1.652 V
ADC Voltage: 1.651 V
ADC Voltage: 1.653 V
```

---

7. 비교 요약

함수	역할	특징
`HAL_ADC_GetValue()`	변환 결과 읽기	DR 레지스터 반환 (0~4095)
`HAL_ADC_Stop()`	변환 중단 및 종료	ADC 비활성화, 다음 변환 준비
실행 시점	PollForConversion 후	변환 완료 후
비고	Stop을 생략하면 Busy 상태 유지	반복 변환 시 필수

✅ 핵심 요약

항목	함수	설명
결과 읽기	`HAL_ADC_GetValue()`	변환된 디지털 값 반환
변환 종료	`HAL_ADC_Stop()`	ADC 종료 및 Idle 복귀
사용 순서	Start → Poll → Get → Stop	Polling 모드 기본 구조
결과 범위	0~4095 (12bit)	Vref = 3.3V 기준
주의사항	Stop 생략 시 Busy 플래그 지속	변환 후 반드시 호출

> 결론적으로,
> `HAL_ADC_GetValue()`는 **측정값을 읽는 역할**,
> `HAL_ADC_Stop()`은 **ADC 하드웨어를 안정적으로 종료**시키는 역할을 수행한다.
> 두 함수는 단일 변환(Polling) 루틴에서 반드시 함께 사용되어야 하며,
> 이후 DMA나 FreeRTOS 환경에서도 이 구조가 그대로 확장된다.

- # Polling, Interrupt, DMA 방식 비교

1. 개요

STM32의 ADC(Analog to Digital Converter)는
아날로그 입력을 디지털 값으로 변환하는 장치로,
결과를 CPU로 전달하는 방식에 따라 세 가지 운용 모드를 제공한다.

모드	특징	대표 함수
Polling 모드	CPU가 변환 완료를 직접 대기	`HAL_ADC_Start()` + `HAL_ADC_PollForConversion()`
Interrupt 모드	변환 완료 시 콜백 자동 호출	`HAL_ADC_Start_IT()` + `HAL_ADC_ConvCpltCallback()`
DMA 모드	변환 결과를 메모리로 자동 전송	`HAL_ADC_Start_DMA()`

이 세 가지 방식은 **응답성, CPU 점유율, 처리 효율** 면에서 차이가 있다.
적절한 모드를 선택하는 것이 실시간 제어 성능의 핵심이다.

---

2. Polling 모드

(1) 구조

CPU가 변환 완료 플래그(`EOC`)를 직접 확인하면서 기다리는 방식이다.

(2) 흐름

```
HAL_ADC_Start(&hadc1);
HAL_ADC_PollForConversion(&hadc1, HAL_MAX_DELAY);
uint32_t val = HAL_ADC_GetValue(&hadc1);
HAL_ADC_Stop(&hadc1);
```

(3) 특징

항목	내용
CPU 점유율	매우 높음 (대기 중 다른 작업 불가)
구현 난이도	가장 간단
속도	변환 주기마다 Blocking
응용 예시	저속 센서, 테스트, 단일 측정

(4) 장점 / 단점

장점	단점
구조 단순, 디버깅 용이	변환 완료까지 CPU가 대기
타이밍 제어 용이	실시간 멀티태스킹에 부적합

---

3. Interrupt 모드

(1) 구조

ADC 변환이 완료되면 하드웨어가 자동으로 인터럽트를 발생시키고,
콜백 함수 `HAL_ADC_ConvCpltCallback()` 이 호출된다.

(2) 흐름

```
HAL_ADC_Start_IT(&hadc1);   // 변환 시작 (인터럽트 방식)

void HAL_ADC_ConvCpltCallback(ADC_HandleTypeDef* hadc)
{
    uint32_t val = HAL_ADC_GetValue(hadc);
    float voltage = val * 3.3f / 4095.0f;
    printf("ADC Voltage: %.2fV\r\n", voltage);
}
```

(3) 특징

항목	내용
CPU 점유율	낮음 (변환 중 다른 작업 가능)
응답성	빠름 (EOC 시 즉시 인터럽트 발생)
사용 난이도	중간
응용 예시	FreeRTOS Task Notify, 주기적 센서 갱신

(4) 장점 / 단점

장점	단점
비동기 동작, CPU 효율적 사용	ISR 설계 복잡
실시간 제어에 적합	고속 연속 변환 시 과부하 위험

---

4. DMA(Direct Memory Access) 모드

(1) 구조

ADC 변환 결과를 CPU 개입 없이 **DMA 컨트롤러가 메모리 버퍼로 자동 전송**한다.
CPU는 단순히 버퍼를 읽어 처리만 수행한다.

(2) 흐름

```
uint32_t adc_buf[4];  // 4채널 데이터 버퍼

HAL_ADC_Start_DMA(&hadc1, adc_buf, 4);  // DMA 기반 연속 변환 시작

void HAL_ADC_ConvCpltCallback(ADC_HandleTypeDef* hadc)
{
    printf("ADC1: %lu | ADC2: %lu | ADC3: %lu | ADC4: %lu\r\n",
           adc_buf[0], adc_buf[1], adc_buf[2], adc_buf[3]);
}
```

(3) 특징

항목	내용
CPU 점유율	매우 낮음 (하드웨어 전송)
응답성	일정 주기적 데이터 확보
적합 환경	연속 다채널 측정, 실시간 제어, FreeRTOS 시스템
사용 난이도	높음 (DMA 버퍼 관리 필요)

(4) 장점 / 단점

장점	단점
CPU 부하 최소화	초기 설정 복잡
연속/다채널 측정 가능	버퍼 관리, 동기화 필요
고속 ADC 읽기 가능	디버깅 난이도 높음

---

5. 모드별 비교 요약

구분	Polling	Interrupt	DMA
CPU 점유율	높음	낮음	매우 낮음
응답 속도	느림 (Blocking)	빠름 (즉시 ISR)	매우 빠름 (하드웨어)
구현 난이도	낮음	중간	높음
추천 환경	단순, 저속	이벤트 기반	고속, 연속 측정
FreeRTOS 호환성	낮음 (Blocking)	좋음	매우 좋음
대표 함수	`HAL_ADC_PollForConversion()`	`HAL_ADC_Start_IT()`	`HAL_ADC_Start_DMA()`
콜백 함수	없음	`HAL_ADC_ConvCpltCallback()`	`HAL_ADC_ConvCpltCallback()`
데이터 흐름	CPU 직접	ISR 처리	DMA → Memory 자동

---

6. 실제 시스템 선택 기준

조건	권장 모드	이유
단일 센서, 테스트용	Polling	코드 단순, 빠른 구현
일정 주기로 센서 측정	Interrupt	CPU 병행 작업 가능
다채널 / 고속 측정	DMA	하드웨어 전송, 효율적 처리
FreeRTOS 사용	Interrupt / DMA	Task 병행 실행
배터리 시스템	DMA	CPU Sleep 가능

---

7. FreeRTOS 환경 예시

(1) Interrupt 방식

```
void HAL_ADC_ConvCpltCallback(ADC_HandleTypeDef* hadc)
{
  BaseType_t xHigherPriorityTaskWoken = pdFALSE;
  vTaskNotifyGiveFromISR(xSensorTaskHandle, &xHigherPriorityTaskWoken);
  portYIELD_FROM_ISR(xHigherPriorityTaskWoken);
}
```

→ 변환 완료 시 `SensorTask`를 깨워 데이터 처리 수행.

(2) DMA 방식

```
void SensorTask(void *argument)
{
  for(;;)
  {
    HAL_ADC_Start_DMA(&hadc1, adc_buffer, 4);
    ulTaskNotifyTake(pdTRUE, portMAX_DELAY);
    Process_ADC_Data(adc_buffer);
  }
}
```

→ DMA 변환 완료 후 Task에 알림 → 데이터 후처리.

---

8. 성능 비교 (예시)

항목	Polling	Interrupt	DMA
100Hz 변환 시 CPU 사용률	약 60%	10~20%	<5%
최대 처리 속도 (12MHz ADC 기준)	약 15kS/s	약 200kS/s	최대 1MS/s
전력 소모	높음	중간	최소
코드 복잡도	낮음	중간	높음

---

9. 실무 팁

- **테스트 단계**에서는 Polling이 가장 빠르고 안정적이다.
- **제품 단계**에서는 DMA 방식이 거의 필수적이다.
- **FreeRTOS 환경**에서는 Polling 금지 (Blocking으로 Task 정지 위험).
- **DMA + Interrupt 조합**은 고속/저전력 시스템의 표준 구조이다.

✅ 핵심 요약

구분	Polling	Interrupt	DMA
설명	CPU가 직접 대기	변환 완료 시 인터럽트	DMA가 자동 전송
CPU 부하	높음	낮음	매우 낮음
FreeRTOS 적합성	낮음	중간	최고
응용 예시	단일 ADC 측정	이벤트 트리거형 센서	고속 다채널 / 지속 계측
추천	디버깅, 단순	일반 센서	산업 제어, IoT, 로깅

> 결론적으로,
>
> - **Polling**은 단순하지만 Blocking 구조로 CPU 효율이 낮고,
> - **Interrupt**는 실시간 반응성에 유리하며,
> - **DMA**는 고속·다채널·저전력 계측에 가장 적합하다.

STM32F103에서는 실전 프로젝트 단계(FreeRTOS 기반 Smart-Tank 등)에서
DMA + Interrupt 하이브리드 구조가 최적의 선택이다.

4.3 전압 측정 실습

- # 분압회로 설계 (10k / 2.2k)

1. 개요

ADC(Analog to Digital Converter)는 일반적으로
입력 전압 범위가 **0 ~ Vref (보통 3.3 V)** 로 제한된다.
따라서 5 V 이상의 전압을 측정할 경우,
전압을 안전하게 낮추기 위해 **분압회로(Voltage Divider)** 를 사용한다.

STM32F103의 ADC 입력은 최대 3.6 V를 초과하면 손상될 수 있으므로,
5 V → 3.3 V 이하로 낮추기 위한 분압 설계가 필요하다.

---

2. 분압 원리

두 개의 저항 **R₁**, **R₂**를 직렬로 연결하여
전압의 비율을 나누는 가장 기본적인 회로다.

```
Vin ---[R1]---┬---[R2]--- GND
              |
            Vout  →  ADC 입력
```

$$V_{out} = V_{in} \times \frac{R_2}{R_1 + R_2}$$

---

3. 10 kΩ / 2.2 kΩ 조합 예시

(1) 설계 조건

- 입력전압 (**Vin**) = 5.0 V
- ADC 기준전압 (**Vref**) = 3.3 V
- 목표 출력전압 (**Vout**) ≤ 3.3 V

(2) 계산

$$V_{out} = 5.0 \times \frac{2.2}{10 + 2.2} = 5.0 \times 0.177 = 0.885\,V$$

이는 **비율이 너무 작다** → 5 V가 0.885 V로 낮아져서 ADC 해상도 활용이 비효율적이다.

---

4. 올바른 비율 계산

목표가 5 V → 3.3 V이므로

$$\frac{V_{out}}{V_{in}} = \frac{3.3}{5.0} = 0.66$$

예를 들어 R_2 = 10 kΩ일 때 R_1 ≈ 5.1 kΩ이면
5 V → 3.3 V로 정확히 분압된다.

---

5. 10 kΩ / 2.2 kΩ 조합의 실제 활용

10 k / 2.2 k는 5 V → 0.885 V처럼 낮은 비율이지만,
12 V 전원 감시용으로는 매우 적절하다.

$$V_{out} = 12 \times \frac{2.2}{10 + 2.2} = 12 \times 0.177 = 2.12V$$

즉, **12 V 시스템 전압을 3.3 V 이하로 안전하게 줄여서 ADC로 측정** 가능하다.

---

6. 전류 및 전력 검토

$$I = \frac{V_{in}}{R_1 + R_2} = \frac{12}{12.2\,k\Omega} = 0.983\,mA$$

→ ¼ W(250 mW) 정격 저항으로 충분히 안전하다.

---

7. 실제 회로 예시

```
      +12V (측정대상 전압)
           |
           |
         [R1] 10kΩ
           |
           ├────→  ADC 입력 (Vout ≈ 2.1V)
           |
         [R2] 2.2kΩ
           |
         GND
```

---

8. 코드 예시 (ADC 변환값을 전압으로 환산)

```
float Read_Voltage(void)
{
  HAL_ADC_Start(&hadc1);
  HAL_ADC_PollForConversion(&hadc1, HAL_MAX_DELAY);
  uint32_t adc_val = HAL_ADC_GetValue(&hadc1);

  // ADC 값 → 입력전압 변환
  float vout = (adc_val * 3.3f) / 4095.0f;
  float vin  = vout * ((10.0f + 2.2f) / 2.2f); // 분압비 보정

  return vin; // 실제 입력전압 (≈ 12V)
}
```

---

9. 분압비 보정식 정리

항목	공식	설명
분압출력	$V_{out} = V_{in} \times \frac{R_2}{R_1+R_2}$	기본 분압식
입력전압 복원	$V_{in} = V_{out} \times \frac{R_1+R_2}{R_2}$	ADC 값 보정용
예시 (10k/2.2k)	$V_{in} = V_{out} \times 5.545$	12 V → 2.16 V

---

10. 선택 저항의 기준

항목	권장값	이유
$R_1 + R_2$ 합	10 kΩ ~ 50 kΩ	ADC 입력 임피던스 대비 안정
R_1:R_2 비율	전압 비율에 따라 조정	Vref(3.3 V) 초과 방지
저항 허용오차	±1% 권장	오차 누적 방지

항목	권장값	이유
바이패스 커패시터	100 nF (ADC 입력단)	노이즈 필터링

---

11. 정전용량 보정 (RC 필터 추가)

노이즈가 심한 환경에서는 R_2 하단에 커패시터를 추가하여
저역통과 필터(LPF) 형태로 안정화시킬 수 있다.

```
Vin ── R1 ─┬── R2 ── GND
           |
           └── C = 100nF → ADC 입력
```

필터 차단 주파수:

$$f_c = \frac{1}{2\pi R_{eq} C} \approx \frac{1}{2\pi (2.2k\Omega)(100nF)} \approx 723Hz$$

---

12. 요약

항목	값	설명
R1	10 kΩ	상단 저항
R2	2.2 kΩ	하단 저항
Vin (측정 대상)	12 V	입력 전압
Vout (ADC 입력)	2.1 V	3.3 V 이하 안전
분압비	1 : 0.177	약 5.5배 감쇠
ADC 계산식	`Vin = Vout * 5.545`	보정식

✅ **핵심 요약**

항목	공식	예시 (10k/2.2k)
분압 출력	$V_{out} = V_{in} \times \frac{2.2}{12.2}$	12 V → 2.16 V
ADC 보정	$V_{in} = V_{out} \times 5.545$	2.16 V × 5.545 ≈ 12 V
ADC 변환	$V_{out} = ADC \times 3.3/4095$	—
최종 식	$V_{in} = (ADC \times 3.3/4095) \times 5.545$	완성

> 결론적으로,
> **10 kΩ / 2.2 kΩ 분압회로는 12 V 계통 감시용으로 매우 안정적이며**,
> STM32F103의 ADC 입력 보호, 정확도, 안전성을 모두 확보할 수 있다.

- # `Read_Voltage()` 함수 구조

1. 개요

`Read_Voltage()` 함수는 **STM32F103의 ADC 채널에서 전압을 측정하고**,
그 값을 **실제 입력 전압(Vin)** 으로 변환하는 핵심 함수다.

이 함수는 다음 과정을 수행한다:

1. ADC 변환 시작 (`HAL_ADC_Start`)
2. 변환 완료 대기 (`HAL_ADC_PollForConversion`)
3. 변환 결과 읽기 (`HAL_ADC_GetValue`)
4. ADC 결과 → 전압 변환
5. 분압비 적용 (10k / 2.2k 보정)
6. ADC 종료 (`HAL_ADC_Stop`)

---

2. 함수 구조

```
float Read_Voltage(void)
{
    // ① ADC 변환 시작
    HAL_ADC_Start(&hadc1);

    // ② 변환 완료 대기 (무한 대기 또는 Timeout 설정)
    HAL_ADC_PollForConversion(&hadc1, HAL_MAX_DELAY);

    // ③ 변환된 디지털 값 읽기 (0~4095)
    uint32_t adc_val = HAL_ADC_GetValue(&hadc1);

    // ④ ADC 중단 (Busy 방지)
    HAL_ADC_Stop(&hadc1);

    // ⑤ ADC 결과를 실제 전압(Vout)으로 변환 (3.3V 기준)
    float vout = (adc_val * 3.3f) / 4095.0f;

    // ⑥ 분압비 보정 (R1=10k, R2=2.2k → 보정계수 ≈ 5.545)
    float vin = vout * ((10.0f + 2.2f) / 2.2f);

    // ⑦ 측정 전압 반환
    return vin;
}
```

---

3. 각 단계별 상세 설명

단계	함수 / 처리	설명
①	`HAL_ADC_Start()`	ADC 변환 개시
②	`HAL_ADC_PollForConversion()`	변환 완료 대기 (`EOC` 플래그 확인)
③	`HAL_ADC_GetValue()`	12-bit 결과 읽기 (0~4095)
④	`HAL_ADC_Stop()`	ADC 종료 (Busy 방지)
⑤	전압 계산식	`Vout = (ADC * 3.3) / 4095`
⑥	분압비 보정	`Vin = Vout × (R1 + R2) / R2`
⑦	반환	입력전압(Vin) 반환 (예: 12V 측정)

---

4. 수학적 변환식

ADC 변환 결과를 입력 전압으로 변환하기 위한 식은 다음과 같다.

$$V_{in} = \left(\frac{ADC_{value}}{4095} \right) \times 3.3 \times \frac{R_1 + R_2}{R_2}$$

10k / 2.2k 분압 시:

$$V_{in} = \left(\frac{ADC_{value}}{4095} \right) \times 3.3 \times 5.545$$

예:
ADC = 2620 → $(2620/4095) \times 3.3 \times 5.545 \approx 11.7V$

---

5. 실전 예시 (UART 출력 포함)

```
void Measure_Voltage(void)
{
    float voltage = Read_Voltage();
    printf("Input Voltage: %.2f V\r\n", voltage);
    HAL_Delay(500);
}
```

출력 예시:

```
Input Voltage: 11.96 V
Input Voltage: 12.02 V
Input Voltage: 11.98 V
```

---

6. FreeRTOS Task 내 응용 예시

```
void Task_VoltageMonitor(void *argument)
{
    for(;;)
    {
        float vin = Read_Voltage();
        printf("[Voltage Task] %.2f V\r\n", vin);
        vTaskDelay(pdMS_TO_TICKS(1000)); // 1초 주기
    }
}
```

→ `Read_Voltage()` 함수는 Polling 기반이지만,
FreeRTOS Task 내에서 비주기적 측정에 사용하기 적합하다.

---

7. 코드 개선 포인트

항목	개선 내용
오버샘플링	여러 번 측정 후 평균값 산출 (노이즈 저감)
보정계수 튜닝	실제 멀티미터 측정값과 비교하여 `(10k+2.2k)/2.2k` 보정
입력 보호	ADC 입력단에 100nF + 다이오드(Clamp) 추가
온도 보정	내부 온도센서 활용 시 ADC 오차 보정 가능

---

8. 오버샘플링 적용 예시

```
float Read_Voltage_Avg(void)
{
    uint32_t sum = 0;
    for (int i = 0; i < 10; i++)
    {
        HAL_ADC_Start(&hadc1);
        HAL_ADC_PollForConversion(&hadc1, HAL_MAX_DELAY);
        sum += HAL_ADC_GetValue(&hadc1);
        HAL_ADC_Stop(&hadc1);
    }

    float avg = (float)sum / 10.0f;
    float vout = (avg * 3.3f) / 4095.0f;
    float vin  = vout * 5.545f;

    return vin;
}
```

→ 10회 평균으로 노이즈 ±1 LSB 수준 제거 가능.

---

9. 디버깅 포인트

증상	원인	해결 방법
ADC 값이 0	핀 미설정	ADC 채널 초기화 확인
값이 불안정	Floating 입력	풀다운 저항 추가
과도하게 낮은 값	분압비 계산 오류	R1/R2 값 재확인
값이 고정됨	`HAL_ADC_Stop()` 누락	Stop 호출 추가

---

10. 요약

항목	설명
기능	ADC 측정값을 실제 전압으로 변환
핵심 공식	`Vin = (ADC * 3.3 / 4095) * ((R1+R2)/R2)`
분압비	10k / 2.2k → ×5.545 보정
입력 범위	최대 12 V 감시 가능
FreeRTOS 호환	Polling 기반으로 간단히 Task에 삽입 가능

✅ 핵심 요약

함수	역할
`HAL_ADC_Start()`	변환 시작
`HAL_ADC_PollForConversion()`	완료 대기
`HAL_ADC_GetValue()`	ADC 결과 읽기
`HAL_ADC_Stop()`	변환 종료
`Read_Voltage()`	ADC 측정 → 전압 환산 → 보정값 반환

> 결론적으로,
> `Read_Voltage()`는 STM32의 ADC 계측 루틴의 **표준화된 구현 템플릿**으로,
> 단일 함수 안에서 **ADC 초기화** → **변환** → **보정** → **반환**의 모든 절차를 처리한다.
> Smart-Tank 프로젝트 등에서 전원, 배터리, 센서 전압 모니터링에 그대로 활용할 수 있다.

- # 전압 보정식 `(ADC_Value * 11.67 / 2065) + 0.36`

1. 개요

STM32의 ADC는 이론적으로 12비트(0~4095) 분해능을 가지며,
아날로그 입력 전압과 다음과 같은 관계를 가진다.

$$V_{in} = \frac{ADC_{value}}{4095} \times V_{ref}$$

그러나 실제 시스템에서는

- 저항 허용오차 (±1~5%)
- ADC 기준전압 오차 (3.3V ≠ 정확히 3.300V)
- ADC 내부 비선형성, 노이즈
- 회로 납땜 및 배선 저항
- GND 기준 레벨 오차

등으로 인해 **측정값과 실제 전압 간의 오차가 발생**한다.

이를 보정하기 위해, 실험적으로 유도된 **보정식(Calibration Formula)** 을 사용한다.

---

2. 실험 기반 보정의 원리

1. 여러 기준 전압(예: 3V, 6V, 9V, 12V)을 입력한다.
2. ADC가 반환하는 `ADC_Value` 값을 기록한다.
3. 이 데이터를 선형 회귀(linear regression)로 분석한다.
4. 보정 계수(기울기 + 절편)를 도출한다.

결과적으로

$$V_{in} = a \times ADC_{value} + b$$

형태로 표현된다.

---

3. `(ADC_Value \* 11.67 / 2065) + 0.36` 해석

이 식은 다음과 같이 정리할 수 있다.

$$V_{in} = \frac{ADC_{value} \times 11.67}{2065} + 0.36$$

(1) 기울기(a)

$$a = \frac{11.67}{2065} = 0.00565$$

즉, ADC 카운트 1 증가당 약 5.65 mV 상승.

(2) 절편(b)

$$b = 0.36$$

ADC 값이 0일 때도 실제 회로 상 0.36V 정도의 오프셋이 존재함을 의미한다.

---

4. 수치적 의미

항목	값	설명
ADC 분해능	12-bit (0~4095)	STM32F103 기준
입력 분압비	약 5.545 (10k/2.2k)	외부 회로 감쇠
보정 계수 11.67 / 2065	실험 기반 스케일 팩터	실제-이론 오차 반영
오프셋 0.36V	회로·ADC 내부 오차 보정	케이블, 저항, GND 차이 반영

---

5. 예시 계산

ADC_Value	계산식	결과
0	(0 × 11.67 / 2065) + 0.36	0.36 V
1000	(1000 × 11.67 / 2065) + 0.36	5.99 V
1500	(1500 × 11.67 / 2065) + 0.36	8.81 V
2000	(2000 × 11.67 / 2065) + 0.36	11.64 V
2065	(2065 × 11.67 / 2065) + 0.36	12.03 V

→ 즉, **ADC=2065일 때 약 12V로 보정**되도록 설계된 식임을 알 수 있다.

---

6. 실제 코드 구현

```c
float Read_Voltage_Calibrated(void)
{
    HAL_ADC_Start(&hadc1);
    HAL_ADC_PollForConversion(&hadc1, HAL_MAX_DELAY);
    uint32_t adc_val = HAL_ADC_GetValue(&hadc1);
    HAL_ADC_Stop(&hadc1);

    // 실험 기반 보정식 적용
    float vin = (adc_val * 11.67f / 2065.0f) + 0.36f;
    return vin;
}
```

> 예를 들어, 12V 전원 측정 회로(10k/2.2k 분압)를 사용한 경우
> ±0.1V 이내의 정확도로 측정 가능하다.

---

7. 보정계수 도출 과정 예시

실제 전압 (V)	ADC 측정값	이론 계산(Vout)	오차(V)
3.00	510	2.90	+0.10
6.00	1023	5.85	+0.15
9.00	1537	8.80	+0.20
12.00	2065	11.64	+0.36

→ 이 데이터를 기반으로 선형 회귀 수행 시
기울기 ≈ 11.67/2065, 절편 ≈ 0.36 이 도출됨.

---

8. 오차 보정 효과

구분	보정 전	보정 후
3V 입력 시	2.85V	3.00V
6V 입력 시	5.75V	6.02V
9V 입력 시	8.70V	9.04V
12V 입력 시	11.60V	12.00V

> 보정 후 ±0.05~0.1V 수준으로 오차가 줄어듦.

---

9. 비교 — 일반식 vs 보정식

구분	공식	설명
이론식	$V_{in} = \frac{ADC}{4095} \times 3.3 \times 5.545$	이상적 회로 기준
보정식	$V_{in} = \frac{ADC \times 11.67}{2065} + 0.36$	실제 측정값 기반 보정
정확도	±3~5%	±1% 이하

---

10. 주의사항

- 보정식은 **특정 회로 구성과 MCU 보드에 종속적**이다.
  (저항, ADC 기준전압, PCB 레이아웃에 따라 달라짐)
- 다른 환경에서는 반드시 **새로운 보정 데이터**로 갱신해야 한다.
- 장기 사용 시 **온도 변화**에 따른 저항 및 Vref 오차도 고려할 것.

✅ 핵심 요약

항목	설명
보정식	`(ADC_Value * 11.67 / 2065) + 0.36`
기능	ADC 측정값 → 실제 전압 변환
의미	기울기(스케일) + 절편(오프셋) 보정
정확도 향상	±1% 이내로 개선
적용 범위	0~12 V 감시용 분압 회로 (10k / 2.2k)

> 요약하자면,
> 이 보정식은 10kΩ/2.2kΩ 분압 회로에서 **ADC=2065일 때 약 12V**를 출력하도록
> 실험적으로 보정된 선형 관계다.
> 실제 산업용 계측 시스템에서 이와 같은 **1차 보정식**은
> 회로 제작 오차, 부품 편차, 온도 변화 등에 따른
> 현실적 오차를 보완하기 위한 핵심 교정 단계로 사용된다.

• 아날로그 전압 + LM35 온도센서 측정

1. 개요

STM32F103의 ADC는 0~3.3V 범위의 아날로그 신호를 디지털 값(0~4095)으로 변환할 수 있다.
이 기능을 활용하면 하나의 ADC 모듈로

- **일반 전압 측정 회로 (예: 12V 분압 감시)**
- **온도센서 LM35 출력(섭씨 온도 전압 변환)**

을 동시에 읽을 수 있다.

> ✅ LM35는 **출력전압 10 mV/°C (0.01V/°C)** 의 선형 온도센서이다.
> 예를 들어 25°C → 0.25V, 100°C → 1.00V.

---

2. 하드웨어 연결

항목	핀 연결	설명
LM35 VCC	+5V	전원

항목	핀 연결	설명
LM35 GND	GND	기준
LM35 OUT	`PA1 (ADC1_IN1)`	온도 입력
분압 회로 출력(Vout)	`PA0 (ADC1_IN0)`	전압 측정 입력

STM32F103 ADC1 다중채널 구조 예시:

```
          +-------------+
          | STM32F103C8T6 |
          |               |
          |  PA0 → ADC1_IN0 (전압 센서) |
          |  PA1 → ADC1_IN1 (LM35)      |
          +-------------+
```

---

3. ADC 다중 채널 설정 (CubeMX 기준)

- **ADC Instance**: ADC1
- **Scan Conversion Mode**: Enabled
- **Continuous Mode**: Disabled
- **Discontinuous Mode**: Disabled
- **Channel Configuration**
  - Channel 0 → Rank 1 (Voltage)
  - Channel 1 → Rank 2 (LM35)
  - Sampling Time: `55.5 cycles` 이상

---

4. 원리 요약

(1) 전압 측정 회로

분압 회로로 감쇠된 입력을 ADC에 인가:

$$V_{in} = \frac{ADC \times 3.3}{4095} \times \frac{R_1 + R_2}{R_2}$$

10kΩ / 2.2kΩ 기준

$$V_{in} = \frac{ADC \times 3.3}{4095} \times 5.545$$

(2) LM35 온도 변환식

LM35의 출력 전압은

$$V_{out} = 10mV/^\circ C$$

따라서:

$$T(^\circ C) = \frac{ADC \times 3.3}{4095} \times 100$$

5. 예시 코드

```
#include "main.h"
#include <stdio.h>

extern ADC_HandleTypeDef hadc1;

// 전압과 온도 측정값 저장 변수
float Vin = 0.0f;
float Temperature = 0.0f;

void Read_Analog_Values(void)
{
    ADC_ChannelConfTypeDef sConfig = {0};

    // ------------------------------
    // 1. 전압 측정 (ADC1_IN0)
    // ------------------------------
    sConfig.Channel = ADC_CHANNEL_0;
    sConfig.Rank = ADC_REGULAR_RANK_1;
    sConfig.SamplingTime = ADC_SAMPLETIME_55CYCLES_5;
    HAL_ADC_ConfigChannel(&hadc1, &sConfig);

    HAL_ADC_Start(&hadc1);
    HAL_ADC_PollForConversion(&hadc1, HAL_MAX_DELAY);
    uint32_t adc_voltage = HAL_ADC_GetValue(&hadc1);
    HAL_ADC_Stop(&hadc1);

    float vout = (adc_voltage * 3.3f) / 4095.0f;
    Vin = vout * ((10.0f + 2.2f) / 2.2f); // 10k / 2.2k 분압보정

    // ------------------------------
    // 2. LM35 온도 측정 (ADC1_IN1)
    // ------------------------------
    sConfig.Channel = ADC_CHANNEL_1;
    sConfig.Rank = ADC_REGULAR_RANK_1;
    sConfig.SamplingTime = ADC_SAMPLETIME_55CYCLES_5;
    HAL_ADC_ConfigChannel(&hadc1, &sConfig);

    HAL_ADC_Start(&hadc1);
    HAL_ADC_PollForConversion(&hadc1, HAL_MAX_DELAY);
    uint32_t adc_temp = HAL_ADC_GetValue(&hadc1);
    HAL_ADC_Stop(&hadc1);

    float vtemp = (adc_temp * 3.3f) / 4095.0f;
    Temperature = vtemp * 100.0f; // 10mV/°C 변환

    // ------------------------------
    // 3. UART 출력
```

```
    // ------------------------------
    printf("Voltage: %.2f V | Temp: %.2f °C\r\n", Vin, Temperature);
}
```

6. 출력 예시

```
Voltage: 11.95 V | Temp: 27.3 °C
Voltage: 11.94 V | Temp: 27.2 °C
Voltage: 12.01 V | Temp: 27.4 °C
```

7. FreeRTOS Task 예시

```
void Task_SensorRead(void *argument)
{
    for(;;)
    {
        Read_Analog_Values();
        vTaskDelay(pdMS_TO_TICKS(1000)); // 1초 주기
    }
}
```

> `Read_Analog_Values()`는 Polling 기반이지만,
> 주기적 FreeRTOS Task 내부에서 안정적으로 동작한다.

8. 결과 해석

항목	계산식	예시 결과
ADC_Voltage	`(ADC * 3.3 / 4095)`	2.15 V
Vin (보정)	`Vout × 5.545`	11.93 V
LM35 출력전압	`(ADC * 3.3 / 4095)`	0.275 V
Temperature	`Vout × 100`	27.5 °C

9. 정밀도 향상 방법

방법	설명
오버샘플링	여러 번 측정 후 평균값으로 노이즈 저감
Vref 보정	내부 Vrefint(1.2V) 측정 후 스케일 보정
오프셋 캘리브레이션	LM35의 0°C 오프셋(±2°C) 보정

방법	설명
전원 안정화	LM35에 별도 디커플링(0.1μF + 1μF) 권장
ADC Input Filter	입력단 100nF 커패시터 추가 (LPF)

10. 하드웨어 참고 회로

```
    +12V (감시 전원)
      |
      |
     [R1] 10kΩ
      |
      ├──────→ PA0 (ADC_IN0)
      |
     [R2] 2.2kΩ
      |
     GND

    LM35 Sensor:
    +5V ──────┬────── VCC
              |
            [LM35]
              |
              ├──────── PA1 (ADC_IN1)
              |
              └──────── GND
```

✅ **핵심 요약**

구분	항목	계산식 / 비고
전압 측정	분압회로 (10k / 2.2k)	$V_{in} = (ADC \times 3.3/4095) \times 5.545$
온도 측정 (LM35)	10mV/°C 출력	$Temp = (ADC \times 3.3/4095) \times 100$
ADC 채널	PA0 (Voltage), PA1 (Temp)	ADC1_IN0 / ADC1_IN1
Vref	3.3V 기준	±1% 오차
정확도	±0.1V / ±1°C 수준	보정 후

> 결론적으로,
> STM32F103의 다중채널 ADC를 활용하면
> **한 번의 변환 루틴으로 전압 감시 + 온도 계측을 병행**할 수 있다.
> LM35는 선형 출력 특성 덕분에 코드 구현이 간단하고,
> 분압 회로와 함께 사용하면 **전원상태 + 온도상태를 동시에 모니터링하는**
> 스마트 수조(Smart Tank), IoT 센서 노드, 산업 제어 시스템 등에서 매우 유용하다.