12. 디버깅 및 최적화

12.1 디버깅

- ## ST-LINK 실시간 디버깅

⚙️ 개요

ST-LINK는 STM32 MCU와 PC 개발 환경(예: STM32CubeIDE, Keil, IAR)을 연결하는 **JTAG/SWD 인터페이스 디버거**다.
단순한 코드 업로드를 넘어, **Break, Step, Watch, Memory View, Live Expression** 등 다양한 실시간 디버깅 기능을 지원한다.

실시간 디버깅(Real-Time Debugging)이란 MCU가 **실행 중에도 변수·레지스터·메모리를 관찰하고 변경할 수 있는** 기능을 말한다.
이를 통해 **임베디드 시스템의 동작을 중단시키지 않고** 내부 상태를 추적할 수 있다.

🔌 1. 연결 구성

```
┌──────────────────┐
|  PC / CubeIDE |
| (GDB Client)  |
└──────┬───────────┘
       | USB
┌──────▼──────┐
|  ST-LINK V2 |──> SWCLK  → MCU
| (or V3)     |──> SWDIO  → MCU
|              ──> NRST   → MCU (optional)
└─────────────┘
```

- **SWD (Serial Wire Debug)** 인터페이스 사용 (TCK/TMS 대신 SWCLK/SWDIO)
- NRST 핀 연결 시 디버거가 리셋 제어 가능
- **Virtual COM Port (VCP)** 기능으로 UART 로그 병행 가능

🧩 2. 주요 디버깅 기능

기능	설명
BreakPoint / Step	코드 실행 중단, 한 단계씩 수행
Run / Resume	중단된 프로그램 재시작
Watch / Live Watch	변수 값을 실시간 관찰
Memory View	RAM/Flash 주소 직접 접근
Peripheral View	레지스터 상태 모니터링

기능	설명
SWO Trace (ITM)	printf 대체 실시간 로깅
Core Register View	PC, LR, SP, PSR 등 즉시 확인

🧭 3. CubeIDE 설정

(1) 디버거 선택

```
Project → Properties → Run/Debug Settings → Debugger
```

- Debug probe: **ST-LINK (OpenOCD)**
- Interface: **SWD**
- Reset Mode: **Connect under reset (권장)**

(2) Live Expressions 활성화

```
Window → Show View → Expressions
```

- `Add New Expression` 클릭 → 변수 이름 입력
- "**Continuous refresh**" 체크 → 실행 중 실시간 업데이트

(3) Semihosting 비활성화

`printf` 사용 시 실행 중 멈춤 방지:

- `Project → Properties → C/C++ Build → Settings → Tool Settings → MCU Settings`
- "Semihosting enable" 해제

⚡ 4. 실시간 변수 모니터링 예시

```
volatile float distance_mm;
volatile float weight_kg;
volatile uint8_t pump_state;
```

이 변수들을 CubeIDE **Live Expressions** 창에 등록하면, MCU가 동작 중이라도
다음과 같이 실시간으로 갱신된다.

변수명	값
`distance_mm`	132.6
`weight_kg`	1.241
`pump_state`	1

> `volatile` 키워드는 최적화에 의한 제거를 방지하여 실시간 갱신이 가능하게 한다.

🔄 5. Watchpoint (데이터 접근 감시)

특정 변수의 **읽기/쓰기 접근을 감시**할 수 있다.
프로그램이 해당 변수에 접근하면 즉시 Break가 발생한다.

예시:

```
uint8_t alarm_flag = 0;
```

CubeIDE → **Breakpoints** → **Add Watchpoint** → **alarm_flag**

- Access: Write
- Size: 1 byte

> 예: `alarm_flag`가 1로 변경될 때 코드 자동 중단

🧮 6. 실시간 Trace (ITM / SWO 출력)

(1) 기본 개념

SWO (Serial Wire Output)는 ARM Cortex-M 코어에 내장된 **실시간 Trace 채널**이다.
`printf()`를 중단 없이 실시간으로 IDE 콘솔에 출력할 수 있다.

(2) CubeIDE 설정

- `Debug Configurations → Startup → Enable SWV`
- Core Clock 입력 (예: 72 MHz)
- SWO Viewer 열기: `Window → Show View → SWV → SWV ITM Data Console`

(3) 코드 예시

```
#include "stdio.h"
int _write(int file, char *ptr, int len)
{
    for (int i = 0; i < len; i++)
        ITM_SendChar(*ptr++);
    return len;
}
```

이후 `printf("Distance=%.1f\n", distance_mm);` 호출 시
MCU 중단 없이 실시간 콘솔 출력 확인 가능.

🧠 7. 디버깅 시 주의사항

항목	설명
최적화(-O2, -O3)	변수 추적 불가 → `-Og` 또는 `-O0` 권장
Interrupt 중단 시 디버깅	실시간 응답 영향 있음
FreeRTOS 사용 시	`configUSE_TRACE_FACILITY`, `configUSE_STATS_FORMATTING_FUNCTIONS` 활성화 시 Task 모니터링 가능
SWO 사용 시 핀 확인	STM32F1: PB3 (SWO), STM32F4: PB3 또는 PB10 등

🔬 8. FreeRTOS Task 디버깅

CubeIDE 상단 → **FreeRTOS** → **Task List View**

- 각 Task의 상태 (Running, Blocked, Suspended)
- Stack 사용량, Priority, CPU 점유율
- Heap 사용량 (heap_4.c 기준) 실시간 표시

```
vTaskList()      // Task 목록 문자열로 출력
vTaskGetRunTimeStats() // CPU 점유율 출력
```

✅ 결론

ST-LINK 실시간 디버깅은 STM32 개발의 핵심 도구로,
중단 없는 실시간 모니터링과 오류 분석이 가능하다.

$$\boxed{ST-LINK+SWD+SWO=\text{완전한실시간분석환경}}$$

추천 조합:

- **CubeIDE + ST-LINK V3 + SWO Trace**
- **Live Watch + FreeRTOS Task View**
- **Watchpoint 기반 Fault 추적**

이 조합을 활용하면 **펌프 제어, 수위 감시, 저전력 모드 전환 등** 모든 동작을
멈추지 않고 안정적으로 검증할 수 있다.

- # UART 로그 출력 (`printf`)

⚙️ 개요

임베디드 시스템에서 **UART 로그**는 디버깅과 시스템 상태 추적의 핵심 도구다.
STM32 시리즈는 HAL 라이브러리 기반으로 `HAL_UART_Transmit()` 또는 `printf()` 리다이렉션을 통해 UART 콘솔로 실시간 로그를 출력할 수 있다.

`printf`를 UART로 연결하면 다음과 같은 이점이 있다:

- **디버거 연결 없이** 실시간 상태 확인
- **센서 값 / 이벤트 기록** 시각화
- **FreeRTOS Task 간 메시지 로깅** 통합

🔌 1. 기본 구성

하드웨어 연결 (예: STM32F103C8T6)

MCU 핀	기능	연결 대상
PA9	USART1_TX	USB-UART (TXD)
PA10	USART1_RX	USB-UART (RXD)
GND	공통 GND	USB-UART GND

UART 변환기로 **CH340 / CP2102 / FT232** 등을 사용하면 된다.
PC에서는 **TeraTerm**, **PuTTY**, **CoolTerm** 등으로 로그를 확인한다.

⚙️ 2. CubeMX 설정

1. **USART1 활성화**
   - Mode: Asynchronous
   - Baud rate: 115200
   - Word length: 8 bits
   - Stop bits: 1
   - Parity: None
   - Hardware Flow Control: None
2. **NVIC 설정 (선택)**
   - “USART1 global interrupt” 활성화 (RX 이벤트 필요 시)
3. **Code Generate → Generate Code**

🧩 3. HAL 기반 기본 송신 함수

```c
#include "usart.h"
#include <string.h>

void UART_Log(const char *msg)
{
    HAL_UART_Transmit(&huart1, (uint8_t *)msg, strlen(msg), HAL_MAX_DELAY);
}
```

사용 예시:

```c
UART_Log("System Init OK\r\n");
```

출력 결과:

```
System Init OK
```

🖨️ 4. `printf()` 리다이렉션

표준 입출력 함수(`printf`, `puts`, `fprintf`)를 UART로 연결하려면 `_write()` 함수를 재정의한다.
C 표준 라이브러리의 `syscalls.c` 파일 또는 main.c 하단에 추가한다.

예시:

```c
#include "usart.h"
#include <stdio.h>

int _write(int file, char *ptr, int len)
{
    HAL_UART_Transmit(&huart1, (uint8_t *)ptr, len, HAL_MAX_DELAY);
    return len;
}
```

이후 모든 `printf()`가 UART를 통해 전송된다.

```c
printf("Distance: %.1f mm, Weight: %.2f kg\r\n", distance, weight);
```

출력:

```
Distance: 128.4 mm, Weight: 1.23 kg
```

🧠 5. FreeRTOS 환경에서의 안전한 출력

멀티태스킹 환경에서는 여러 Task가 동시에 `printf()`를 호출하면
출력 내용이 뒤섞일 수 있다.
이를 방지하려면 **Mutex 보호**를 추가한다.

```
osMutexId_t uartMutexHandle;

void Safe_Printf(const char *fmt, ...)
{
    char buf[128];
    va_list args;
    va_start(args, fmt);
    vsprintf(buf, fmt, args);
    va_end(args);

    osMutexAcquire(uartMutexHandle, osWaitForever);
    HAL_UART_Transmit(&huart1, (uint8_t *)buf, strlen(buf), HAL_MAX_DELAY);
    osMutexRelease(uartMutexHandle);
}
```

사용 예:

```
Safe_Printf("[Sensor] Level=%.1f mm, Temp=%.2f C\r\n", level, temp);
```

🧩 6. 수신 콜백 기반 로그 (옵션)

UART RX 인터럽트를 활용하면 PC에서 명령을 받아 처리 가능하다.

```
uint8_t rx_data;

void HAL_UART_RxCpltCallback(UART_HandleTypeDef *huart)
{
    if (huart->Instance == USART1)
    {
        printf("[RX] %c\r\n", rx_data);
        HAL_UART_Receive_IT(&huart1, &rx_data, 1); // 재시작
    }
}
```

⚡ 7. 성능 팁

항목	설정	설명
Baud rate	115200 ~ 921600	속도 향상 가능
DMA 송신	가능	CPU 부하 감소

항목	설정	설명
SWO Trace 병행	디버거 연결 시 printf 대체 가능	
Non-blocking 모드	`HAL_UART_Transmit_IT()` 또는 `DMA` 로 백그라운드 전송	

🗒️ 8. 예제: 통합 센서 로그

```
void Log_SensorData(float distance, float weight, float level)
{
    printf("[DATA] D=%.1f mm, W=%.3f kg, L=%.1f mm\r\n",
           distance, weight, level);
}
```

출력 예시:

```
[DATA] D=127.8 mm, W=1.242 kg, L=53.5 mm
```

✅ 결론

`printf()` 를 UART로 리다이렉션하면
MCU 상태를 직관적으로 모니터링할 수 있으며,
디버거 없이도 실시간 로그를 확보할 수 있다.

$$\boxed{HAL\_UART\_Transmit() + printf() = \text{실시간텍스트기반진단인터페이스}}$$

추천 조합:

- USART1 @ 115200bps
- `_write()` 리다이렉션
- FreeRTOS Mutex 보호
- SWO Trace 병행 (디버깅 시)

• CubeIDE Watch / Live Variables

📘 개요

STM32CubeIDE는 STMicroelectronics가 제공하는 통합 개발 환경(IDE)으로,
STM32 MCU의 **디버깅, 실시간 변수 감시, 메모리 분석, FreeRTOS Task 모니터링**을 지원한다.
그중 **Watch Window**와 **Live Expressions(Variables)** 기능은
프로그램을 중단하지 않고도 MCU 내부 변수를 실시간으로 확인할 수 있게 해준다.

이 기능은 펌웨어 동작을 분석하거나, FreeRTOS 환경에서
Task 간 변수 상태를 모니터링할 때 매우 유용하다.

⚙️ 1. Watch Window (변수 감시)

Watch Window는 디버깅 중 특정 변수의 값을 실시간으로 추적하는 도구이다.
중단점(Breakpoint)에서 프로그램이 정지했을 때,
또는 Step Into / Step Over 중에도 변수 상태를 계속 확인할 수 있다.

사용 방법

1. 코드 내에 감시할 변수를 선언한다.

```
float distance_mm = 0.0f;
uint32_t weight_raw = 0;
```

2. **디버깅 시작 (Debug As → STM32 Cortex-M C/C++ Application)**
3. 상단 메뉴에서 **Window → Show View → Expressions** 선택
4. "+" 버튼을 클릭해 감시할 변수를 추가
   - 예: `distance_mm`, `weight_raw`, `hx711_data[offset]`
5. Breakpoint에서 멈추거나 Step 실행 시, 변수 값이 자동으로 갱신됨

특징

- 실시간 RAM 값 읽기
- 구조체, 포인터, 배열 형태도 지원
- 부동소수점(float) 변수 자동 변환 표시
- 값 변경 가능 (더블 클릭 → 수동 수정)

주의

- **정지 시점(Step / Break)** 에서만 값이 업데이트된다.
- 프로그램이 실행 중(Run)일 때는 값이 변하지 않는다 → 이를 해결한 것이 Live Variables 기능이다.

⚡ 2. Live Expressions (Live Variables)

Live Expressions는 MCU가 동작 중일 때도
변수의 값을 주기적으로 읽어오는 실시간 감시 기능이다.
이는 **SWD(Single Wire Debug)** 기반의 비침투적(non-intrusive) 메커니즘으로 동작한다.

활성화 방법

1. **디버깅 세션 시작**
2. 상단 탭에서 **Live Expressions** 창을 연다
   (없을 경우: Window → Show View → Live Expressions)
3. `+` 버튼 클릭 → 실시간 감시할 변수 등록
   예:

```
distance_mm
temperature
sensor_status
```

4. MCU가 실행 중이어도 주기적으로 값이 갱신됨

특징

- 코드 중단 없이 실시간 변수 갱신
- FreeRTOS Task 간 공유 변수 감시 가능
- 그래프 형태로 시각화 가능 (Plot 기능)
- CPU 점유율에 거의 영향 없음

제한사항

- SWD 인터페이스 사용 필수 (ST-LINK 필요)
- 너무 많은 변수를 등록하면 업데이트 속도 저하
- 초당 약 5~10회 정도의 샘플링 주기 (MCU 클럭/디버그 속도에 따라 다름)

🧠 3. 실무 활용 예시

(1) 센서 디버깅

```
float distance_mm;
float weight_kg;
float water_level;
```

→ Live Expressions에 3개 변수 추가
→ HC-SR04, HX711, 수위센서 측정 결과를 실시간 모니터링 가능

(2) FreeRTOS Task 상태 확인

```
uint32_t SensorTask_ExecCount;
uint32_t ControlTask_ExecCount;
```

→ 각 Task의 실행 횟수를 실시간으로 관찰
→ Task 스케줄링 문제, starvation 여부를 분석 가능

(3) 제어 루프 확인 (PID 루프)

```
float pid_error;
float pid_output;
```

→ Live Variables에서 두 변수를 Plot하면 PID 제어 안정성 시각 분석 가능

🔍 4. 데이터 그래프 시각화

Live Expressions 창에서 변수 우클릭 →
"Show in: Plot" 선택하면 실시간 파형으로 시각화된다.
이 기능은 외부 Serial Plotter 없이 MCU 내부 연산 결과를
직접 그래프로 확인할 수 있어, 센서 신호나 제어 루프 튜닝 시 매우 유용하다.

예시:

- `distance_mm` — 초음파 센서 거리 파형
- `pid_output` — 제어 출력 변화
- `weight_kg` — 필터링 전후 비교

⚙️ 5. 내부 동작 구조

- ST-LINK 디버거를 통해 SWD 채널로 변수의 RAM 주소를 직접 폴링(polling)
- CubeIDE의 **GDB 서버**가 주기적으로 값 요청 → IDE에 표시
- MCU는 디버그 레지스터(DEMCR, DWT) 기반으로 비침투적 접근 수행
- 프로그램 흐름에는 영향을 주지 않음 (단, SWO Trace 활성화 시 오버헤드 증가 가능)

🧩 6. 주의 사항

항목	설명
최대 변수 수	약 20~30개 권장
업데이트 주기	100ms ~ 500ms 사이
최적화 옵션 (-O2 이상)	컴파일러가 변수를 레지스터로 올릴 경우 값 표시 불가
`volatile` 선언	Live Expressions로 감시할 변수는 반드시 `volatile` 권장

예시:

```
volatile float distance_mm;
volatile uint32_t loop_counter;
```

✅ 요약

기능	Watch Window	Live Expressions
동작 방식	정지 시점에 값 표시	실행 중 실시간 표시
코드 중단	필요	불필요
데이터 변경	가능	불가

기능	Watch Window	Live Expressions
사용 목적	디버깅 중 변수 확인	런타임 모니터링
권장 변수	상태, 제어 변수	센서 출력, PID 변수

결론:
STM32CubeIDE의 **Watch**와 **Live Variables** 기능은
UART 로그보다 빠르고, 실시간 변수 추적이 가능하며,
FreeRTOS 기반 시스템의 디버깅 효율을 극대화한다.

$$\boxed{UART = \text{텍스트로그}, \quad LiveExpressions = \text{실시간변수시각화}}$$

12.2 성능 분석

- # Timer 정확도 검증

📘 개요

STM32의 **타이머(TIMx)** 는 매우 정밀한 하드웨어 타이밍을 제공하지만,
클럭 설정, 프리스케일러, 인터럽트 지연, FreeRTOS 스케줄러 영향 등에 따라
실제 주기 정확도(Accuracy)와 안정도(Stability) 가 달라질 수 있다.

이 섹션에서는 **STM32 Timer의 정확도를 검증하는 방법**을
실무 기준으로 정리한다.

🧩 1. 검증 목적

항목	설명
주기 정확도(Period Accuracy)	설정한 주기(예: 1ms, 1s)와 실제 측정된 주기의 오차 확인
지터(Jitter)	주기가 순간적으로 변동하는 정도 (Timer → ISR 진입 지연 포함)
FreeRTOS 영향	OS Tick, Context Switch가 타이밍에 미치는 영향 검증
온도/전압 안정성	HSI, LSI 사용 시 클럭 안정도 확인

⚙️ 2. 기본 설정 확인

(1) System Clock Configuration

CubeMX → **Clock Configuration** 탭에서 다음 확인:

- HSE (Crystal) 사용 시: ±20ppm 수준의 정확도
- HSI 사용 시: ±1% 정도 (지터 있음)
- APB1 / APB2 Prescaler 값 확인 (TIM Clock 계산 필요)

예시 (STM32F103C8T6):

```
SYSCLK = 72 MHz
APB1 = 36 MHz → TIM2-7 = 72 MHz (×2 보정)
```

---

(2) Timer 설정 예시

항목	값
Prescaler	71 (72MHz ÷ 72 = 1MHz, 1μs 단위)
Period (ARR)	999 → 1ms 주기
Mode	Upcounting
Interrupt	Enable (Update Event)

```
// CubeMX 생성 코드 (HAL_TIM_PeriodElapsedCallback)
void HAL_TIM_PeriodElapsedCallback(TIM_HandleTypeDef *htim)
{
    if (htim->Instance == TIM2)
    {
        HAL_GPIO_TogglePin(GPIOC, GPIO_PIN_13);  // 주기 확인용 토글
    }
}
```

🧪 3. 오실로스코프 / 로직분석기 측정

(1) GPIO 토글 측정

Timer ISR에서 GPIO 토글 → 오실로스코프에서 주기 확인

```
void HAL_TIM_PeriodElapsedCallback(TIM_HandleTypeDef *htim)
{
    if (htim->Instance == TIM2)
        HAL_GPIO_TogglePin(GPIOB, GPIO_PIN_0);
}
```

- 측정 결과 예:
  - 설정: 1.000 ms
  - 측정: 0.9997 ms
    → 오차 = -0.03% (정상)
- **지터 확인**:
  오실로스코프의 “Persistence” 모드에서 펄스 폭의 변동 확인
  - 정상: ±1~2μs
  - FreeRTOS 활성화 시: ±10~50μs 가능

---

(2) 주기 오차 계산식

$$\text{오차}(\%) = \frac{\text{측정주기} - \text{이론주기}}{\text{이론주기}} \times 100$$

예시:
1ms 설정 → 1.002ms 측정
→ 오차 = +0.2%

🔍 4. 소프트웨어 기반 검증 (RTC / SysTick 비교)

Timer가 정확한지 내부 기준으로 검증하려면
RTC 또는 HAL_GetTick() 기반 비교 가능.

```
uint32_t tick_start = HAL_GetTick();
uint32_t count = 0;

while (count < 1000)  // 1000회 타이머 인터럽트
{
    if (flag_timer_elapsed)
    {
        flag_timer_elapsed = 0;
        count++;
    }
}

uint32_t tick_end = HAL_GetTick();
printf("1000 cycles took %lu ms\n", tick_end - tick_start);
```

→ 1000회 × 1ms = 1000ms ± 오차
→ 타이머 ISR 누락 여부, 누적 오차 확인 가능

⏱️ 5. FreeRTOS 환경 검증

FreeRTOS 사용 시 Timer의 정확도는 **SysTick 주기, Context Switch 지연**에 영향받는다.

(1) 소프트웨어 타이머 (`xTimerCreate`)

- 정확도: ±1 tick
- Tick 주기(1ms) 설정 시 약 ±1ms 오차 가능

(2) 하드웨어 타이머 (TIMx)

- ISR 실행은 FreeRTOS 스케줄링에 영향받지 않음
- 단, **ISR 내부에서 Task Notify / Queue 전송 시 지연** 가능

개선 방법

- **IRQ 우선순위 조정 (NVIC)**
  `HAL_NVIC_SetPriority(TIM2_IRQn, 5, 0);`
- FreeRTOS configLIBRARY_MAX_SYSCALL_INTERRUPT_PRIORITY 이하로 설정

⚡ 6. 클럭 원인 오차 분석

클럭 소스	정확도	특징
HSE (Crystal)	±20 ppm (~0.002%)	고정밀, 권장
HSI (RC)	±1% 이상	내부 온도 변화에 민감
LSE (32.768kHz Crystal)	±50 ppm	RTC 정확도용
LSI (RC)	±1~2%	대기모드용, 정확도 낮음

→ Timer 정확도 향상을 위해 HSE 사용을 권장.
→ FreeRTOS Tick도 HSE 기반으로 설정 시 지터 최소화 가능.

🔧 7. 보정(Calibration) 및 개선

방법	설명
Clock Trim	HSI 사용 시 `HAL_RCCEx_CRSConfig()` 로 자동 보정 가능
DMA 타이밍 측정	DMA Transfer Complete 타이밍 비교로 주기 검증
DWT Cycle Counter	ARM Cortex-M3 이상에서 CPU 사이클 단위 측정 가능

DWT 예시:

```
// DWT 기반 1ms 측정
DWT->CYCCNT = 0;
uint32_t start = DWT->CYCCNT;
HAL_Delay(1);
uint32_t end = DWT->CYCCNT;
printf("Elapsed cycles: %lu\n", end - start);
```

→ 72MHz 기준 72,000 cycles ≈ 1ms → 정확도 검증 가능

✅ 요약

항목	검증 방법	기대 정확도
GPIO 토글	오실로스코프 측정	±0.1% 이하

항목	검증 방법	기대 정확도
HAL_GetTick 비교	누적 시간 비교	±1ms 수준
FreeRTOS 환경	Task Notify 지연 포함	±1~3 tick
DWT 사이클 카운터	CPU 레벨 정확도	±0.001%

📌 **결론:**

- 타이머 정확도 검증은 **GPIO 토글 + 오실로스코프 측정**이 가장 신뢰성 높음
- FreeRTOS 환경에서는 **ISR → Task Notify** 경로의 지연도 함께 분석
- **DWT Cycle Counter**를 병행 사용하면 **µs 단위 정밀 검증** 가능

• Task 주기성 측정

📘 개요

FreeRTOS에서 여러 Task가 병렬로 실행될 때,
각 Task의 **주기성(Periodicity)** — 즉 "정해진 주기로 정확히 실행되는가" — 는
시스템 안정성과 제어 성능을 결정하는 핵심 요소다.

이 절에서는 **Task 주기 오차를 측정하고 분석하는 표준적인 방법**을
HAL + FreeRTOS 기반으로 정리한다.

⚙️ 1. Task 주기성의 정의

Task 주기성은 "**이전 실행 완료 시점**과 **다음 실행 시점** 사이의 간격"으로 정의된다.
FreeRTOS는 이를 위한 두 가지 메커니즘을 제공한다:

방식	설명	특징
`vTaskDelay()`	단순 지연 (현재 시점 기준 Delay)	누적 오차 발생 가능
`vTaskDelayUntil()`	기준 Tick 기준 절대 지연	**정확한 주기 보장 (권장)**

🧩 2. 기본 측정 구조

```c
#include "cmsis_os.h"
#include "main.h"

void SensorTask(void *argument)
{
    TickType_t last_wake_time;
    const TickType_t period = pdMS_TO_TICKS(1000); // 1초 주기
    uint32_t tick_now;

    last_wake_time = xTaskGetTickCount();

```

```
    for (;;)
    {
        tick_now = xTaskGetTickCount();
        printf("[SensorTask] Tick: %lu\n", tick_now);

        // 주기성 유지
        vTaskDelayUntil(&last_wake_time, period);
    }
}
```

설명:

- `last_wake_time`은 이전 주기의 시작 Tick을 저장.
- `vTaskDelayUntil()`은 다음 주기까지 자동으로 계산하여 **드리프트 없는 주기 실행**을 보장.
- UART를 통해 Tick 값을 주기적으로 출력하면, Task 실행 간격을 직접 검증 가능.

📊 3. 주기 오차 측정 방법

(1) Tick 기반 오차 분석

```
static TickType_t prev_tick = 0;

void DisplayTask(void *argument)
{
    TickType_t last_wake_time = xTaskGetTickCount();
    const TickType_t period = pdMS_TO_TICKS(500);

    for (;;)
    {
        TickType_t now = xTaskGetTickCount();
        if (prev_tick != 0)
        {
            int32_t diff = (int32_t)(now - prev_tick);
            printf("[DisplayTask] Period: %ld ticks (%ld ms)\n", diff, diff);
        }
        prev_tick = now;

        vTaskDelayUntil(&last_wake_time, period);
    }
}
```

결과 예시 (UART 로그):

```
[DisplayTask] Period: 500 ticks (500 ms)
[DisplayTask] Period: 501 ticks (501 ms)
[DisplayTask] Period: 499 ticks (499 ms)
```

→ 평균 주기 500ms, 오차 ±1ms
→ Tick 정확도 및 FreeRTOS Scheduler 안정성 확인 가능

---

(2) DWT Cycle Counter 기반 μs 단위 측정

ARM Cortex-M3 이상 MCU는 **DWT(Cycle Counter)** 를 통해
CPU 클럭 단위로 정밀 시간 측정이 가능하다.

```
static inline void DWT_Init(void)
{
    CoreDebug->DEMCR |= CoreDebug_DEMCR_TRCENA_Msk;
    DWT->CYCCNT = 0;
    DWT->CTRL |= DWT_CTRL_CYCCNTENA_Msk;
}

void TimingTask(void *argument)
{
    DWT_Init();
    uint32_t prev = DWT->CYCCNT;

    for (;;)
    {
        uint32_t now = DWT->CYCCNT;
        uint32_t delta = now - prev;
        prev = now;

        float elapsed_ms = (float)delta / (SystemCoreClock / 1000.0f);
        printf("[TimingTask] Δt = %.3f ms\n", elapsed_ms);

        osDelay(1000);
    }
}
```

측정 예시:

```
Δt = 1000.031 ms
Δt = 1000.019 ms
Δt = 999.987 ms
```

→ ±0.03ms 수준의 고정밀 주기성 유지 확인 가능.

🔍 4. 여러 Task 간 주기성 비교

두 개 이상의 Task가 서로 다른 주기로 실행될 때,
실행 타이밍이 겹치거나 지연되는 현상을 **상호 간섭(Task Interference)** 이라 한다.

```
void FastTask(void *argument)
{
    TickType_t last = xTaskGetTickCount();
    for (;;) {
        printf("FastTask\n");
        vTaskDelayUntil(&last, pdMS_TO_TICKS(100));
    }
```

```
}

void SlowTask(void *argument)
{
    TickType_t last = xTaskGetTickCount();
    for (;;) {
        printf("SlowTask\n");
        vTaskDelayUntil(&last, pdMS_TO_TICKS(1000));
    }
}
```

UART 로그에서 실행 순서가 일정하게 유지되면 스케줄러가 정상 동작 중임을 의미.
우선순위 설정이 잘못되면 100ms Task가 지연될 수 있음 → `configUSE_PREEMPTION` 확인 필요.

🧮 5. 주기성 평가 지표

항목	설명	계산식
주기 평균값 (Mean Period)	전체 측정값의 평균	ΣTi / N
주기 표준편차 (Jitter)	주기의 변동 정도	√(Σ(Ti - Mean)² / N)
최대 지연 (Max Delay)	목표 주기 대비 최대 초과 시간	max(Ti - Ttarget)

예시 결과:

```
Target Period = 1000 ms
Mean Period = 1000.12 ms
Jitter (σ) = 0.05 ms
Max Delay = 0.15 ms
```

⚡ 6. 주기성 저하 원인 및 개선

원인	설명	개선 방안
UART 출력 지연	`printf()` 블로킹	DMA UART 전송 사용
ISR 과부하	높은 인터럽트 빈도	ISR 최소화, DMA 사용
우선순위 역전	낮은 우선순위 Task가 지연	Mutex with Priority Inheritance
Tickless Idle 영향	Sleep 진입 시 Tick 오차 발생	RTC 기반 주기 보정

✅ 요약

항목	방법	정밀도	장점
Tick Count 기반	`xTaskGetTickCount()`	ms 단위	간단, 실용적
DWT Cycle Counter	`DWT->CYCCNT`	µs 단위	고정밀
오실로스코프 + GPIO 토글	실시간 측정	하드웨어 수준	절대 시간 검증

📎 결론:

- `vTaskDelayUntil()` 은 FreeRTOS에서 가장 정밀한 주기 유지 방법이다.
- 주기성은 Tick 기반 혹은 DWT 기반으로 실시간 측정 가능하며,
  로그 또는 GPIO 토글로 검증 시 하드웨어 오차까지 평가할 수 있다.
- 실제 제어 시스템에서는 **PID 루프나 센서 샘플링 주기 유지**를 위해
  이 기법을 반드시 사용해야 한다.

• CPU 부하 측정

📘 개요

임베디드 시스템에서 **CPU 부하율(CPU Load, Utilization)** 은
현재 시스템이 얼마나 바쁘게 동작하고 있는지를 나타내는 지표다.

FreeRTOS 기반 STM32 프로젝트에서는
`Idle Task` 의 실행 시간을 기준으로 간접적으로 부하를 측정할 수 있으며,
또는 DWT(Cycle Counter)나 GPIO 토글 기반으로 직접 측정할 수도 있다.

🧩 1. CPU Load 정의

$$\text{CPU Load (\%)} = 100 - \frac{T_{\text{Idle}}}{T_{\text{Total}}} \times 100$$

- T_{total} : 전체 측정 구간의 시간
- T_{idle} : 그 구간 동안 Idle Task가 실제로 실행된 시간

즉, Idle Task가 차지하는 시간이 줄어들수록 CPU 부하는 높아진다.

⚙️ 2. Idle Hook 기반 측정

FreeRTOS 설정 파일(`FreeRTOSConfig.h`)에서 **Idle Hook** 을 활성화한다.

```
#define configUSE_IDLE_HOOK    1
```

그 후, `vApplicationIdleHook()` 함수를 구현한다.

```c
volatile uint32_t idle_counter = 0;

void vApplicationIdleHook(void)
{
    idle_counter++;
}
```

이 함수는 **CPU가 아무 일도 하지 않을 때마다 호출**되므로,
일정 기간 동안 증가한 `idle_counter` 값으로 부하를 계산할 수 있다.

🧮 3. 주기적 부하 계산 루틴

```c
uint32_t idle_count_prev = 0;
uint32_t cpu_load = 0;

void CPULoadTask(void *argument)
{
    const TickType_t period = pdMS_TO_TICKS(1000); // 1초 주기
    TickType_t last_wake = xTaskGetTickCount();

    for (;;)
    {
        uint32_t idle_count_now = idle_counter;
        uint32_t delta = idle_count_now - idle_count_prev;
        idle_count_prev = idle_count_now;

        // 1초 동안 Idle Task가 실행된 횟수를 이용한 상대적 부하 계산
        cpu_load = 100 - (delta * 100 / IDLE_CALIBRATION_VALUE);
        if (cpu_load > 100) cpu_load = 100;

        printf("[CPU] Load = %lu%%\n", cpu_load);

        vTaskDelayUntil(&last_wake, period);
    }
}
```

🧩 4. 기준(IDLE_CALIBRATION_VALUE) 보정

Idle 루프가 MCU 클럭 속도에 따라 다르기 때문에,
부하 측정 전에 **100% Idle 상태에서 보정값**을 측정해야 한다.

```c
void Calibrate_IdleLoop(void)
{
    idle_counter = 0;
    HAL_Delay(1000);   // 1초 동안 IdleTask만 동작
    IDLE_CALIBRATION_VALUE = idle_counter;
    printf("Calibration: %lu\n", IDLE_CALIBRATION_VALUE);
}
```

이 값을 이후 계산에서 기준으로 사용한다.

🧭 5. DWT 기반 정밀 측정

Idle Hook 대신 **DWT Cycle Counter**를 이용하면
μs 단위의 정밀 부하 계산이 가능하다.

```
static uint32_t idle_start, idle_total, measure_total;

void DWT_Init(void)
{
    CoreDebug->DEMCR |= CoreDebug_DEMCR_TRCENA_Msk;
    DWT->CYCCNT = 0;
    DWT->CTRL |= DWT_CTRL_CYCCNTENA_Msk;
}

void vApplicationIdleHook(void)
{
    idle_start = DWT->CYCCNT;
    while (__get_IPSR() == 0) ; // context switch 감지
    idle_total += DWT->CYCCNT - idle_start;
}

void CPULoadTask(void *argument)
{
    uint32_t prev_idle = 0;
    uint32_t prev_total = 0;

    for (;;)
    {
        HAL_Delay(1000);
        uint32_t now_total = DWT->CYCCNT;
        uint32_t delta_total = now_total - prev_total;
        uint32_t delta_idle = idle_total - prev_idle;

        float load = 100.0f * (1.0f - (float)delta_idle / delta_total);
        printf("CPU Load = %.2f%%\n", load);

        prev_idle = idle_total;
        prev_total = now_total;
    }
}
```

- 장점: μs 단위의 실시간 부하 측정 가능
- 단점: DWT 사용 불가한 MCU(Cortex-M0 등)에서는 지원 안 됨

🧠 6. GPIO 기반 외부 측정

Idle Hook에서 **GPIO 토글**을 사용하면
로직 분석기나 오실로스코프를 이용한 시각적 측정이 가능하다.

```
void vApplicationIdleHook(void)
{
    HAL_GPIO_TogglePin(GPIOC, GPIO_PIN_13);
}
```

- **Low 상태** → IdleTask 실행 중
- **High 상태** → 다른 Task 동작 중
  → PWM Duty 비율로 CPU Load를 관찰할 수 있음

📊 7. FreeRTOS Trace API 기반 부하 측정

FreeRTOS는 **Runtime Statistics** 기능을 통해
각 Task가 점유한 CPU 시간 비율을 직접 계산할 수 있다.

`FreeRTOSConfig.h` 설정:

```
#define configGENERATE_RUN_TIME_STATS        1
#define configUSE_STATS_FORMATTING_FUNCTIONS 1
#define configUSE_TRACE_FACILITY             1
```

Timer 또는 DWT를 runtime counter로 지정:

```
void configureTimerForRunTimeStats(void)
{
    DWT_Init();
}

uint32_t getRunTimeCounterValue(void)
{
    return DWT->CYCCNT;
}
```

사용 예시:

```
char stats[512];
vTaskGetRunTimeStats(stats);
printf("%s\n", stats);
```

출력 예시:

```
Task Name       Time     %
---------------------------------
SensorTask      2456     12%
DisplayTask     1389      7%
RTCTask          754      4%
Idle           14567     77%
```

✅ 요약

방법	정밀도	구현 난이도	특징
Idle Hook 기반	중간	★☆☆	간단, 실시간 추정
DWT 기반	높음	★★★	µs 단위, 고정밀 분석
GPIO 토글	낮음	★☆☆	외부 측정 장비 필요
Trace API	높음	★★☆	Task별 CPU 점유율 분석 가능

☝🏻 **결론:**

- 시스템 전반의 부하율을 보려면 **Idle Hook 방식**이 가장 단순하고 실용적이다.
- Task별 점유율을 보고 싶다면 **FreeRTOS Trace API + vTaskGetRunTimeStats()** 를 사용하라.
- 고정밀 실험 환경에서는 **DWT Cycle Counter** 를 활용하여 µs 단위로 CPU Load를 계산할 수 있다.

12.3 최적화

- # DMA 적용 (ADC, I²C)

1. 개요

DMA(Direct Memory Access)는 CPU를 거치지 않고 주변장치(Peripheral)와 메모리 간에 데이터를 직접 전송하는 하드웨어 모듈이다.
STM32에서 DMA를 사용하면 **ADC 변환 데이터**나 **I²C 송수신 데이터**를 자동으로 버퍼에 저장하여 CPU 부하를 최소화할 수 있다.
특히 FreeRTOS 환경에서는 DMA를 활용해 **비동기 데이터 수집 및 송신**을 효율적으로 구현할 수 있다.

---

2. DMA 구조

STM32 DMA 컨트롤러는 여러 개의 **Channel/Stream** 으로 구성되며, 각 채널은 다음과 같이 동작한다.

- **Source Address** : 데이터의 출발지 (예: `&ADC1->DR`, `&I2C1->DR`)
- **Destination Address** : 데이터의 목적지 (예: `adc_buffer[]`, `i2c_rx_buffer[]`)
- **Direction** : Peripheral → Memory 또는 Memory → Peripheral
- **Mode** :
  - Normal (한 번 전송 후 종료)

- Circular (버퍼 반복 사용)
- Double Buffer (이중 버퍼 구조)

---

3. ADC + DMA 연동

ADC는 변환 완료 시 DMA를 트리거하여 결과값을 자동으로 메모리에 저장한다.
이는 **고속 샘플링**, **연속 데이터 취득**에 적합하다.

CubeMX 설정

1. ADC → Continuous Conversion Mode : Enabled
2. DMA Settings → Mode : Circular / Data Width : Half Word
3. NVIC → DMA Interrupt : Enabled

코드 예시

```
#define ADC_BUFFER_LEN 16
uint16_t adc_buffer[ADC_BUFFER_LEN];

HAL_ADC_Start_DMA(&hadc1, (uint32_t*)adc_buffer, ADC_BUFFER_LEN);

void HAL_ADC_ConvCpltCallback(ADC_HandleTypeDef* hadc)
{
    if (hadc->Instance == ADC1)
    {
        uint32_t sum = 0;
        for (int i = 0; i < ADC_BUFFER_LEN; i++)
            sum += adc_buffer[i];

        float avg = (float)sum / ADC_BUFFER_LEN;
        printf("ADC Avg: %.2f\n", avg);
    }
}
```

- DMA가 자동으로 `adc_buffer[]`에 변환 결과를 채운다.
- Circular 모드 사용 시 자동 반복 샘플링이 가능하다.

---

4. I²C + DMA 연동

I²C를 DMA와 결합하면 대량의 데이터를 빠르게 송수신할 수 있다.
OLED, EEPROM, 센서 데이터 블록 읽기 등에 적합하다.

수신 예시

```
#define RX_SIZE 32
uint8_t i2c_rx_buffer[RX_SIZE];

HAL_I2C_Master_Receive_DMA(&hi2c1, (uint16_t)(0x48 << 1), i2c_rx_buffer, RX_SIZE);

void HAL_I2C_MasterRxCpltCallback(I2C_HandleTypeDef *hi2c)
{
    printf("I2C RX Done. First Byte: 0x%02X\n", i2c_rx_buffer[0]);
}
```

송신 예시 (OLED 등)

```
HAL_I2C_Master_Transmit_DMA(&hi2c1, (uint16_t)(0x3C << 1), oled_buffer,
sizeof(oled_buffer));
```

---

5. FreeRTOS 환경에서의 DMA 주의점

- DMA 콜백 내에서 **FreeRTOS API** 직접 호출 금지 → `vTaskNotifyGiveFromISR()` 사용
- DMA 버퍼는 **전역 또는 static 메모리**에 선언해야 함
- I²C DMA는 Multi-Slave 환경에서 **Mutex 보호** 필요

예시

```
void HAL_I2C_MasterTxCpltCallback(I2C_HandleTypeDef *hi2c)
{
    BaseType_t xHigherPriorityTaskWoken = pdFALSE;
    vTaskNotifyGiveFromISR(OLEDTaskHandle, &xHigherPriorityTaskWoken);
    portYIELD_FROM_ISR(xHigherPriorityTaskWoken);
}
```

---

6. 요약

구분	DMA 미적용	DMA 적용
CPU 개입	모든 전송 CPU 처리	하드웨어 자동 전송
응답 지연	높음	매우 낮음
FreeRTOS 부하	큼	최소화
전송 효율	낮음	높음
응용 예	단일 ADC, 단일 I²C 송신	다중 센서, 실시간 로깅, OLED 출력

- # Tickless Idle Mode

1. 개요

Tickless Idle Mode는 FreeRTOS의 저전력 기능으로,
CPU가 유휴 상태(Idle Task 수행 중)일 때 **SysTick 타이머를 일시 중단**하고,
다음 태스크 실행 시점까지 MCU를 **Sleep / Stop 모드**로 진입시키는 방식이다.

기본적으로 FreeRTOS는 1ms 주기로 **SysTick Interrupt**를 발생시켜 스케줄링을 수행한다.
하지만 유휴 상태에서도 지속적인 Tick Interrupt는 불필요한 전력 소모를 유발한다.
Tickless Idle 모드는 이 불필요한 인터럽트를 제거하여 소비전류를 크게 줄인다.

---

2. 동작 원리

1. Idle Task가 실행됨
2. 커널이 다음 태스크의 **지연 시간(Delay, Timeout 등)** 을 계산
3. 해당 기간 동안 Tick을 중단하고 MCU를 Sleep 또는 Stop 모드로 진입
4. RTC 또는 SysTick 재설정 후 **다음 태스크 준비 시점**에 깨어남
5. 시스템 시간(Tick count)을 보정하여 연속성 유지

시간 흐름 예시

```
|<-- Active -->|<------- Idle (Sleep) ------>|<-- Active -->|
SysTick ON           SysTick OFF (no interrupt)     SysTick ON
```

---

3. CubeMX 설정 절차

1. **FreeRTOS** → **Config Parameters** → **Enable Tickless Idle** 활성화
2. `configUSE_TICKLESS_IDLE` = 1
3. (Optional) `configPRE_SLEEP_PROCESSING()` / `configPOST_SLEEP_PROCESSING()` 콜백 함수 구현

CubeIDE에서는 자동으로 `FreeRTOSConfig.h`에 다음 매크로가 삽입된다.

```
#define configUSE_TICKLESS_IDLE         1
#define configEXPECTED_IDLE_TIME_BEFORE_SLEEP  2
```

---

4. 전력 절감 메커니즘

모드	클럭	유지 기능	소비전류 (STM32F1 기준)
Run	모든 클럭 활성	전체 기능	$_{10}$15 mA
Sleep	CPU 정지, 주변기기 유지	RTC, DMA, GPIO	~1 mA
Stop	대부분 클럭 차단	RTC, LSE만 유지	~50 μA

모드	클럭	유지 기능	소비전류 (STM32F1 기준)
Standby	RAM/RTC 제외 전원 차단	RTC 백업만 유지	~2 μA

Tickless Idle은 일반적으로 **Stop Mode 진입 전단계**로 활용된다.

---

5. 사용자 콜백 처리

Tickless Idle 중 전력 모드를 세밀하게 제어하려면
FreeRTOS의 **Pre/Post Sleep Hook**을 활용할 수 있다.

```
void PreSleepProcessing(uint32_t *ulExpectedIdleTime)
{
    /* Sleep 전 수행할 동작 */
    HAL_SuspendTick();          // SysTick 정지
    HAL_PWR_EnterSTOPMode(PWR_LOWPOWERREGULATOR_ON, PWR_STOPENTRY_WFI);
}

void PostSleepProcessing(uint32_t *ulExpectedIdleTime)
{
    /* Sleep 복귀 후 동작 */
    SystemClock_Config();       // 클럭 복구
    HAL_ResumeTick();           // SysTick 재시작
}
```

---

6. 동작 시 고려사항

- DMA, UART RX, RTC Alarm 등 **Wake-up Source**가 설정되어 있어야 복귀 가능
- Tickless Idle 진입 시점에는 모든 태스크가 **Block 상태**여야 함
- **SysTick 기반 Delay** (`HAL_Delay()`)는 Sleep 중 정지되므로 사용 자제
- **RTC 또는 LPTIM 기반 시간 기준**을 사용하는 것이 권장됨

---

7. 예시 흐름 (RTC 알람 기반 주기 측정과 결합)

1. 측정 루틴 완료 → 모든 Task Block 상태 진입
2. Idle Hook → Tickless Idle → STOP 모드 진입
3. RTC Alarm 발생 → Wake-up → FreeRTOS Tick 복구
4. 주기적 측정 루틴 재개

```
void vApplicationIdleHook(void)
{
    __WFI(); // Idle 상태에서 저전력 진입
}
```

---

8. 요약

항목	Tickless 미적용	Tickless 적용
SysTick 동작	항상 활성	Idle 중 정지
소비전류	높음	매우 낮음
RTC / Wake-up 지원	필요 없음	필수
시간 정밀도	항상 1ms 주기	Sleep 구간 RTC 보정
FreeRTOS 설정	기본	`configUSE_TICKLESS_IDLE = 1`

Tickless Idle Mode는 FreeRTOS 기반의 **저전력 시스템 설계의 핵심 요소**로,
RTC Alarm, DMA, Interrupt 기반 Task 구조와 함께 사용하면
측정 → 전송 → Sleep → Wake-up의 완전한 저전력 루프를 구현할 수 있다.

- # Static Task Allocation

1. 개요

FreeRTOS는 태스크, 큐, 세마포어, 타이머 등의 객체를 생성할 때
동적 메모리 할당(Dynamic Allocation) 또는 **정적 메모리 할당(Static Allocation)** 방식을 선택할 수 있다.

기본적으로 `pvPortMalloc()`을 사용하는 동적 방식(`xTaskCreate`)이 일반적이지만,
임베디드 환경에서는 **heap 오버플로**나 **메모리 단편화(fragmentation)** 문제를 피하기 위해
정적 메모리 할당(static allocation) 이 권장된다.

정적 할당 시 FreeRTOS는 **사용자가 직접 스택과 TCB(Task Control Block)**를 선언하고,
커널은 런타임 시점에 이를 그대로 사용한다.

---

2. 설정 조건 (`FreeRTOSConfig.h`)

정적 할당 기능을 활성화하려면 다음 매크로를 설정한다.

```
#define configSUPPORT_STATIC_ALLOCATION   1
```

또한, 동적/정적 병행 사용도 가능하며,
동적만 허용하려면 `configSUPPORT_DYNAMIC_ALLOCATION`을 `1`,
정적만 사용할 경우 `0`으로 지정한다.

```
#define configSUPPORT_DYNAMIC_ALLOCATION  1
#define configSUPPORT_STATIC_ALLOCATION   1
```

---

3. Static Task 생성 구조

FreeRTOS는 `xTaskCreateStatic()` API를 제공하며,
스택 및 TCB(Task Control Block)를 직접 전달받는다.

```
TaskHandle_t xHandle;
StaticTask_t xTaskBuffer;
StackType_t xStackBuffer[256]; // 256 words (1KB stack on Cortex-M)

xHandle = xTaskCreateStatic(
              vTaskFunction,       // 실행 함수
              "MyTask",            // 태스크 이름
              256,                 // 스택 크기 (word 단위)
              (void *)NULL,        // 매개변수
              tskIDLE_PRIORITY+1,  // 우선순위
              xStackBuffer,        // 스택 버퍼
              &xTaskBuffer         // TCB 버퍼
          );
```

특징

- 스택과 TCB가 전역 또는 static 영역에 존재 → 해제되지 않음
- `vTaskDelete()`로 삭제해도 메모리 반환 없음
- Heap이 전혀 필요하지 않음 → `heap_4.c`, `heap_5.c` 불필요

---

4. Idle Task, Timer Task의 정적 생성

FreeRTOS는 내부적으로 Idle Task와 Timer Service Task를 자동 생성한다.
정적 메모리 사용을 강제하려면 다음 **Hook 함수**를 반드시 구현해야 한다.

```
void vApplicationGetIdleTaskMemory(StaticTask_t **ppxIdleTaskTCBBuffer,
                                   StackType_t **ppxIdleTaskStackBuffer,
                                   uint32_t *pulIdleTaskStackSize)
{
    static StaticTask_t xIdleTaskTCB;
    static StackType_t  xIdleStack[configMINIMAL_STACK_SIZE];

    *ppxIdleTaskTCBBuffer   = &xIdleTaskTCB;
    *ppxIdleTaskStackBuffer = xIdleStack;
    *pulIdleTaskStackSize   = configMINIMAL_STACK_SIZE;
}

void vApplicationGetTimerTaskMemory(StaticTask_t **ppxTimerTaskTCBBuffer,
                                    StackType_t **ppxTimerTaskStackBuffer,
                                    uint32_t *pulTimerTaskStackSize)
{
    static StaticTask_t xTimerTaskTCB;
    static StackType_t  xTimerStack[configTIMER_TASK_STACK_DEPTH];

    *ppxTimerTaskTCBBuffer   = &xTimerTaskTCB;
```

```
    *ppxTimerTaskStackBuffer = xTimerStack;
    *pulTimerTaskStackSize   = configTIMER_TASK_STACK_DEPTH;
}
```

이 두 함수가 없으면 FreeRTOS는 내부적으로 **동적 할당**을 시도하므로
`configSUPPORT_DYNAMIC_ALLOCATION`이 0인 경우 컴파일 오류가 발생한다.

---

5. 장점

구분	동적 할당	정적 할당
메모리 관리	런타임 `malloc/free` 사용	컴파일 타임 고정
안정성	힙 오버플로 위험	안전, 예측 가능
메모리 단편화	발생 가능	없음
런타임 유연성	높음	낮음
실시간 성능	약간의 지연	즉시 생성

임베디드 및 RTOS 기반의 실시간 시스템에서는
예측 가능한 메모리 동작이 매우 중요하므로 정적 할당이 선호된다.

---

6. 실습 예제

DisplayTask / ControlTask / RTCTask를 정적으로 생성하는 예시:

```
StaticTask_t xDisplayTCB, xControlTCB, xRTCTCB;
StackType_t xDisplayStack[256], xControlStack[256], xRTCStack[256];

void Init_FreeRTOS(void)
{
    DisplayTaskHandle = xTaskCreateStatic(DisplayTask, "Display", 256, NULL, 2, xDisplayStack, &xDisplayTCB);
    ControlTaskHandle = xTaskCreateStatic(ControlTask, "Control", 256, NULL, 2, xControlStack, &xControlTCB);
    RTCTaskHandle     = xTaskCreateStatic(RTCTask, "RTC", 256, NULL, 3, xRTCStack, &xRTCTCB);

    vTaskStartScheduler();
}
```

---

7. 요약

항목	설명
설정 매크로	`configSUPPORT_STATIC_ALLOCATION = 1`
생성 함수	`xTaskCreateStatic()`
힙 의존성	없음
Idle/Timer Task	`vApplicationGetIdleTaskMemory()` / `vApplicationGetTimerTaskMemory()` 필요
장점	안정성, 예측성, 전력 효율성
단점	유연성 감소, 고정 메모리 사용량 증가

정적 태스크 할당은 **안정적이고 결정적인 동작**을 요구하는
저전력 FreeRTOS 시스템에서 필수적인 설계 기법이다.