8. FreeRTOS 실전

8.1 FreeRTOS 구조

- ## 커널 구조, Task, Scheduler

1. 개요

FreeRTOS 커널(Firmware Kernel) 은 임베디드 시스템에서 다중 작업(Task)을
효율적으로 관리하기 위한 **스케줄링 기반 실시간 운영체제(RTOS)** 이다.
STM32와 같은 Cortex-M 마이크로컨트롤러 환경에서는
CMSIS-RTOS2 레이어 또는 **FreeRTOS API**를 통해 커널 자원에 접근할 수 있다.

커널의 핵심 구성 요소는 다음과 같다.

구성 요소	역할
Task	개별 실행 단위 (스레드)
Scheduler	실행 우선순위 결정 및 전환 관리
Queue	Task 간 메시지 전달
Semaphore / Mutex	자원 접근 동기화
Timer (SW Timer)	비동기 이벤트 관리
Idle Task	유휴 상태 처리 및 저전력 진입

---

2. 커널 구조

FreeRTOS는 **우선순위 기반 선점형(Preemptive) 스케줄러**를 채택한다.
즉, 항상 **가장 높은 우선순위를 가진 준비 상태(Ready)** Task가 CPU를 점유한다.

(1) 주요 내부 구조

```
┌──────────────────────────────┐
|       Application Layer      |
|   ├── SensorTask             |
|   ├── DisplayTask            |
|   └── ControlTask            |
├──────────────────────────────┤
|     FreeRTOS Kernel Core     |
|   ├── Scheduler              |
|   ├── Ready List (Tasks)     |
|   ├── Blocked List           |
|   ├── Delayed List           |
|   ├── Timer Service Task     |
|   └── Idle Task              |
├──────────────────────────────┤
```

```
|      HAL / Driver Layer   |
|   ├── GPIO, I2C, UART     |
|   ├── TIM, ADC, RTC       |
|   └── FreeRTOS Hook APIs  |
└──────────────────────────┘
```

(2) Task 상태 전이

상태	설명
Running	현재 CPU를 점유 중
Ready	실행 대기 중 (우선순위별 리스트에 존재)
Blocked	이벤트(Queue, Delay, Semaphore) 대기 중
Suspended	임시 중지 상태 (`vTaskSuspend`)
Deleted	메모리 반환 대기 상태

Task는 **vTaskDelay()**, **xQueueReceive()**, **xSemaphoreTake()** 등에 의해
Blocked 상태로 전이되며, 이벤트 발생 시 Ready로 복귀한다.

---

3. Task 구조

각 Task는 독립적인 **스택(Stack)** 과 **제어 블록(TCB: Task Control Block)** 을 가진다.

(1) 생성

```
xTaskCreate(
    SensorTask,        // Task 함수 포인터
    "Sensor",          // Task 이름
    256,               // Stack 크기 (Word 단위)
    NULL,              // 파라미터
    2,                 // 우선순위
    &sensorHandle      // 핸들 포인터
);
```

(2) 주기적 Task 패턴

```
void SensorTask(void *argument)
{
    for (;;)
    {
        Read_Sensors();
        Process_Data();
        vTaskDelay(pdMS_TO_TICKS(100));  // 100ms 주기
    }
}
```

(3) Stack 및 Context 관리

- 각 Task는 독립 스택을 보유하며, Context는 **레지스터 세트 + 스택 포인터** 형태로 저장됨
- Task 전환 시 커널은 **PendSV Handler**를 통해
  현재 Task의 Context를 저장하고 다음 Task의 Context를 복원함

---

4. Scheduler 구조

(1) 스케줄러 종류

모드	설명
Preemptive Mode	높은 우선순위 Task가 즉시 실행
Cooperative Mode	현재 Task가 CPU를 자발적으로 양보
Time-Slicing Mode	동일 우선순위 Task 간 시분할 실행

STM32에서 기본적으로 사용하는 모드는 **Preemptive + Time Slicing** 이다.

(2) 스케줄링 과정

1. **SysTick 인터럽트 (1ms 주기)** 발생
2. `xTaskIncrementTick()` 호출로 Tick 카운트 증가
3. Delay 완료 Task를 Ready List로 이동
4. `vTaskSwitchContext()` 호출로 다음 실행 Task 결정
5. **PendSV 예외** 발생 → Context Switch 실행

---

5. 주요 커널 Hook 함수

함수	설명
`vApplicationIdleHook()`	Idle Task 루프
`vApplicationTickHook()`	매 Tick마다 호출
`vApplicationMallocFailedHook()`	메모리 할당 실패 시
`vApplicationStackOverflowHook()`	Stack Overflow 감지 시

예시:

```
void vApplicationIdleHook(void)
{
    Enter_Sleep_Mode(); // 저전력 진입
}
```

---

6. FreeRTOS 주요 구성 매크로

매크로	의미	예시
`configUSE_PREEMPTION`	선점형 스케줄링 활성화	1
`configUSE_TICKLESS_IDLE`	Tickless 저전력 모드	1
`configMAX_PRIORITIES`	최대 Task 우선순위 개수	5
`configMINIMAL_STACK_SIZE`	Idle Task 스택 크기	128
`configTOTAL_HEAP_SIZE`	전체 동적 메모리 크기	10 * 1024
`configUSE_TIME_SLICING`	시분할 실행 활성화	1

---

7. 커널 동작 시퀀스

1. `HAL_Init()` → `SystemClock_Config()`
2. `MX_FREERTOS_Init()` (Task 생성 및 Queue 설정)
3. `osKernelStart()` → 스케줄러 실행
4. 가장 높은 우선순위의 Ready Task부터 실행
5. SysTick/PendSV를 통해 Context 전환 반복

---

8. 결론

- FreeRTOS 커널은 **Task, Scheduler, IPC** 구조를 기반으로
  효율적인 병렬 처리를 지원한다.
- Task는 독립된 스택과 상태를 가지며,
  Scheduler가 우선순위 기반으로 CPU를 분배한다.
- SysTick → PendSV를 통한 Context Switch는
  정밀한 주기 제어 및 실시간 응답성 확보의 핵심 메커니즘이다.

• Tick, SysTick Handler

1. 개요

Tick 은 RTOS 내에서 시간의 최소 단위를 의미하며,
커널의 **스케줄링, Delay, Timer 동작의 기준 주기**로 사용된다.

SysTick 은 Cortex-M 코어에 내장된 24-bit 다운카운터 타이머로,
FreeRTOS 및 HAL 모두 시스템 Tick 발생원으로 활용한다.
기본적으로 **1ms 주기(1kHz)** 로 설정되어,
커널의 Task 전환, Delay 처리, Software Timer 관리 등에 사용된다.

---

2. SysTick 타이머 개요

(1) 레지스터 구성

레지스터	설명
`SYST_CSR` (Control and Status Register)	타이머 Enable, 인터럽트 제어
`SYST_RVR` (Reload Value Register)	카운트 주기 설정 (24bit)
`SYST_CVR` (Current Value Register)	현재 카운트 값
`SYST_CALIB` (Calibration Register)	보정용 상수 (10ms 기준)

(2) 클럭 구조

SysTick은 **프로세서 클럭 (HCLK)** 혹은 **HCLK/8** 을 소스로 사용한다.
주기 계산식은 다음과 같다.

$$\text{Reload} = \frac{HCLK}{\text{TickFrequency}} - 1$$

예를 들어,
HCLK = 72 MHz, Tick = 1000 Hz (1ms) 이면:

$$\text{Reload} = 72,000,000/1000 - 1 = 71,999$$

즉, SysTick이 1ms마다 인터럽트를 발생시킨다.

---

3. SysTick 초기화

HAL 또는 FreeRTOS는 `HAL_InitTick()` 또는 `xPortStartScheduler()` 내에서
SysTick을 자동으로 설정한다.

HAL 기반 초기화 예시

```
HAL_Init();
SystemClock_Config();
HAL_InitTick(TICK_INT_PRIORITY);
```

내부 동작:

```
void HAL_InitTick(uint32_t TickPriority)
{
    uint32_t uwTickPrio = TickPriority;
    HAL_SYSTICK_Config(SystemCoreClock / 1000);  // 1ms
    HAL_NVIC_SetPriority(SysTick_IRQn, uwTickPrio, 0);
}
```

---

4. FreeRTOS에서의 Tick 동작

FreeRTOS는 SysTick을 시스템 시계로 사용하며,
매 Tick마다 **커널 시간(tick count)** 을 증가시킨다.

주요 흐름

1. SysTick 인터럽트 발생
2. `xPortSysTickHandler()` 호출
3. `xTaskIncrementTick()` → Tick Count 증가
4. Blocked Task의 Delay 만료 검사
5. 필요 시 `vTaskSwitchContext()` → Context 전환
6. PendSV 인터럽트 트리거 → Task 전환 수행

Tick Handler 예시

```
void SysTick_Handler(void)
{
    HAL_IncTick();              // HAL Delay용 Tick 증가
    osSystickHandler();         // FreeRTOS Tick 처리
}
```

또는 RTOS 전용 코드에서는 다음과 같이 정의됨:

```
void SysTick_Handler(void)
{
    xPortSysTickHandler();
}
```

---

5. Tick과 Delay 함수의 관계

함수	역할	Tick 사용 여부
`HAL_Delay(ms)`	HAL 레벨 지연	O
`vTaskDelay(tick)`	FreeRTOS Task 대기	O
`vTaskDelayUntil()`	절대 Tick 기반 주기 제어	O
`HAL_GetTick()`	HAL Tick 값 반환	O

FreeRTOS에서는 `vTaskDelay()` 호출 시 현재 Tick 값을 기준으로
Task를 **Blocked 상태**로 전환하고,
Tick Handler에서 지정된 시간만큼 경과하면 Ready 상태로 복귀시킨다.

---

6. Tickless Idle 모드

Tickless Idle은 FreeRTOS의 저전력 모드로,
Idle 상태가 지속될 경우 **SysTick 인터럽트를 일시 정지**하고
MCU를 Sleep/Stop 모드로 전환한다.

핵심 매크로:

```
#define configUSE_TICKLESS_IDLE 1
#define configEXPECTED_IDLE_TIME_BEFORE_SLEEP 5
```

Tickless 동작 시 SysTick이 중단되며,
RTC 또는 저전력 타이머가 시스템을 깨운 후
Tick 누락분을 보정한다.

---

7. SysTick 인터럽트 우선순위

SysTick, PendSV, SVC는 FreeRTOS의 핵심 예외이며,
모두 **최저 우선순위**로 설정되어야 한다.

인터럽트	역할	권장 우선순위
`SysTick`	주기적 Tick 발생	Lowest
`PendSV`	Context Switch 수행	Lowest
`SVC`	RTOS 초기화 호출	Low

설정 예시 (`FreeRTOSConfig.h`):

```
#define configKERNEL_INTERRUPT_PRIORITY         255
#define configMAX_SYSCALL_INTERRUPT_PRIORITY    191
```

---

8. Tick 정확도와 드리프트

SysTick은 CPU 클럭에 종속되므로,
PLL 또는 HSE 불안정 시 Tick 간격이 변할 수 있다.
장기적인 RTC 기반 시간 유지에는 LSE(32.768kHz)를 병행 사용하는 것이 바람직하다.

Tick 주기 오차는 다음 식으로 평가할 수 있다.

$$\text{Drift} = \frac{\Delta f_{HCLK}}{f_{HCLK}} \times 100\%$$

---

9. 핵심 요약

- **SysTick**은 Cortex-M 코어에 내장된 정밀 24-bit 타이머로
  RTOS 커널의 시간 기준을 제공한다.
- **Tick Handler**는 Task Delay, Timer, Scheduler를 구동하는 핵심 루틴이다.
- **PendSV**와 연계되어 Context Switch를 수행하며,
  실시간 멀티태스킹의 기반이 된다.
- **Tickless Idle**을 활용하면 μA 수준의 저전력 Sleep 모드 운용이 가능하다.

Context Switch & Stack 관리

1. 개요

Context Switch(문맥 전환) 은 멀티태스킹 OS(예: FreeRTOS)에서
CPU가 실행 중인 **Task의 상태(Context)** 를 저장하고,
다른 Task의 Context를 복원하여 실행을 전환하는 과정이다.

이때 Context에는 **CPU 레지스터, 프로그램 카운터(PC), 스택 포인터(SP), 상태 레지스터(xPSR)** 등이 포함된다.
STM32의 Cortex-M 구조에서는 하드웨어와 소프트웨어가 협력하여 이 과정을 수행한다.

---

2. Context의 구성

Cortex-M 프로세서에서 Context는 다음과 같은 레지스터 세트를 포함한다.

구분	레지스터	설명
자동 저장 (HW Stack)	R0–R3, R12, LR, PC, xPSR	예외 진입 시 자동으로 스택에 저장
수동 저장 (SW Stack)	R4–R11	PendSV 핸들러에서 RTOS가 직접 저장

즉, 인터럽트 진입 시 하드웨어가 R0–R3, R12, LR, PC, xPSR을 자동 저장하고,
RTOS는 나머지 R4–R11을 소프트웨어적으로 스택에 보관한다.

---

3. Stack 구조

각 Task는 독립적인 **Stack 영역**을 가진다.
Task 생성 시 FreeRTOS는 지정된 크기의 스택을 Heap에서 할당하고,
초기 Context를 미리 구성해둔다.

Task 생성 시 스택 구조

```
Top of Stack → xPSR (0x01000000)
                PC (Task entry)
                LR (Task exit handler)
                R12
                R3
                R2
                R1
                R0 (Task argument)
                R11
                R10
                ...
                R4
Bottom of Stack
```

이 구조는 **스택 초기화 함수 (`pxPortInitialiseStack()`)** 에서 설정되며,
처음 Task 실행 시 PendSV가 이 스택을 복원한다.

---

4. Context Switch 절차

(1) 개요

FreeRTOS는 **SysTick** 또는 기타 인터럽트를 통해
주기적으로 **스케줄링(Scheduling)** 을 수행하고,
다음과 같은 순서로 Context Switch를 실행한다.

(2) 단계별 흐름

1. **SysTick 인터럽트 발생**
   - 현재 Task 실행 시간이 만료되면 `xPortSysTickHandler()` 호출
   - 커널이 `xTaskIncrementTick()` 로 Tick Count 증가
2. **스케줄러 판단**
   - `xTaskIncrementTick()` → 다음 실행할 Task 결정
   - Context Switch 필요 시 **PendSV 예외 발생 요청**
3. **PendSV Handler 진입**
   - 현재 Task의 레지스터(R4–R11) 스택에 저장
   - 현재 스택 포인터(SP)를 Task Control Block(TCB)에 기록
4. **다음 Task 선택 및 복원**
   - 커널이 다음 실행할 Task의 TCB에서 SP 읽음
   - 해당 Task의 스택에서 R4–R11 복원
5. **하드웨어 복원**
   - 예외 복귀 시 자동으로 R0–R3, R12, LR, PC, xPSR 복원
   - 프로그램 카운터(PC)가 새 Task의 진입점으로 이동

이후 CPU는 새로운 Task의 Context로 실행을 재개한다.

---

5. PendSV 핸들러 핵심 코드

```
void PendSV_Handler(void)
{
    __asm volatile
    (
        "MRS R0, PSP                \n" // 현재 Task의 PSP 읽기
        "STMDB R0!, {R4-R11}        \n" // R4-R11 저장
        "BL vTaskSwitchContext      \n" // 다음 Task 선택
        "LDMIA R0!, {R4-R11}        \n" // R4-R11 복원
        "MSR PSP, R0                \n" // PSP 복원
        "BX LR                      \n"
    );
}
```

이 루틴은 **Cortex-M의 PSP(Process Stack Pointer)** 를 사용하여
User Task의 스택을 관리하며,
커널(Stack)은 **MSP(Main Stack Pointer)** 가 담당한다.

---

6. PSP / MSP 이중 스택 구조

Cortex-M은 두 개의 스택 포인터를 지원한다.

스택 포인터	용도	사용 영역
MSP (Main Stack Pointer)	인터럽트, 예외, 커널용	SysTick, PendSV, SVC
PSP (Process Stack Pointer)	사용자 Task용	각 Task의 독립 스택

RTOS는 Task 전환 시 PSP를 조작함으로써,
커널과 사용자 Task의 스택 충돌을 방지한다.

---

7. 스택 오버플로 검출

FreeRTOS는 다음 매크로로 스택 사용량을 감시할 수 있다.

```
#define configCHECK_FOR_STACK_OVERFLOW  2
```

수준별 동작:

값	설명
0	미사용
1	Task 스택 포인터가 최소 한계 하향 시 감지

값	설명
2	Context Switch 시 스택 영역 전체 검사

콜백 함수:

```
void vApplicationStackOverflowHook(TaskHandle_t xTask, char *pcTaskName)
{
    // 오류 처리 (LED 점멸, 로그 출력 등)
}
```

---

8. Context 저장 최적화

Cortex-M의 Lazy Stacking 기능(FPU 포함 시)으로,
FPU 레지스터(S0–S31)를 실제 필요 시에만 저장함으로써
Context Switch 오버헤드를 감소시킬 수 있다.

해당 기능은 `FPCA` 플래그와 `CPACR` 설정을 통해 활성화된다.

---

9. 요약

- Context Switch는 Task의 레지스터 상태를 저장/복원하여
  CPU 실행 흐름을 전환하는 RTOS 핵심 메커니즘이다.
- Cortex-M은 하드웨어 스택 자동화로 효율적 문맥 저장을 지원한다.
- FreeRTOS는 **PendSV** 를 이용해 지연된 Context Switch를 수행하며,
  PSP/MSP의 이중 스택 구조로 Task 간 격리를 보장한다.
- 스택 오버플로 감시, Lazy Stacking 등으로
  안정성과 성능을 향상시킬 수 있다.

8.2 STM32 포팅

- # CubeMX로 FreeRTOS 활성화

1. 개요

STM32CubeMX는 FreeRTOS를 자동 통합할 수 있는 **RTOS 미들웨어 구성 도구**를 제공한다.
사용자는 GUI 환경에서 Task, Queue, Semaphore, Timer 등의 객체를 생성하고,
자동으로 초기화 코드(`freertos.c`)를 생성할 수 있다.

이 과정을 통해 수동 포팅 없이 FreeRTOS 기반의 멀티태스킹 환경을 빠르게 구축할 수 있다.

---

2. 설정 절차

(1) 프로젝트 생성

1. **STM32CubeMX 실행**
2. MCU 또는 보드 선택 (예: STM32F103C8Tx)
3. **Project Manager** → **Toolchain / IDE** 에서 *STM32CubeIDE* 선택

---

(2) FreeRTOS 활성화

1. **Middleware** → **FREERTOS** 항목 클릭
2. "**Enabled**" 체크박스 활성화
3. Kernel 옵션에서 "**CMSIS_V2**" 또는 "**CMSIS_V1**" 중 선택
    - CMSIS_V2는 HAL + FreeRTOS 연동 시 권장
4. **Interface** → **CMSIS-RTOS v2** 로 설정

---

(3) 커널 설정 (Configuration → FreeRTOS)

- **Config Parameters** 탭에서 주요 설정 조정 가능

항목	설명	기본값
`configUSE_PREEMPTION`	선점형 스케줄링 활성화	Enabled
`configUSE_TIME_SLICING`	동일 우선순위 Task 간 라운드로빈	Enabled
`configUSE_IDLE_HOOK`	Idle Hook 함수 사용	Optional
`configUSE_TICK_HOOK`	Tick Hook 함수 사용	Disabled
`configTICK_RATE_HZ`	SysTick 주기 (Hz)	1000
`configMINIMAL_STACK_SIZE`	Idle Task 스택 크기	128
`configTOTAL_HEAP_SIZE`	커널 Heap 메모리	10240 (10 KB 권장)

---

(4) Task 생성

1. **Tasks and Queues** 탭 이동
2. "**Add**" 버튼 클릭 → Task 생성
3. Task 이름 및 우선순위, 스택 크기 설정

예시:

Name	Function	Priority	Stack Size
`defaultTask`	StartDefaultTask	Normal	128

Name	Function	Priority	Stack Size
`sensorTask`	StartSensorTask	Above Normal	256
`displayTask`	StartDisplayTask	Normal	256

---

(5) 코드 생성

1. **Project → Generate Code** 클릭
2. CubeIDE에서 자동 생성된 파일 확인

- `freertos.c` : FreeRTOS 초기화 및 Task 생성
- `FreeRTOSConfig.h` : 커널 설정 매크로
- `main.c` : `MX_FREERTOS_Init()` 호출

---

3. 생성된 코드 구조

`main.c`

```
int main(void)
{
  HAL_Init();
  SystemClock_Config();
  MX_GPIO_Init();
  MX_FREERTOS_Init();  // FreeRTOS 객체 초기화
  osKernelStart();     // Scheduler 시작

  while (1) { }
}
```

`freertos.c`

```
void MX_FREERTOS_Init(void)
{
  osThreadDef(defaultTask, StartDefaultTask, osPriorityNormal, 0, 128);
  defaultTaskHandle = osThreadCreate(osThread(defaultTask), NULL);

  osThreadDef(sensorTask, StartSensorTask, osPriorityAboveNormal, 0, 256);
  sensorTaskHandle = osThreadCreate(osThread(sensorTask), NULL);
}
```

예시 Task 함수

```
void StartSensorTask(void const * argument)
{
  for(;;)
  {
    Read_Sensor();
    osDelay(100);
  }
}
```

---

4. FreeRTOSConfig.h 주요 매크로

매크로	설명	권장값
`configUSE_PREEMPTION`	선점형 스케줄러 사용	1
`configUSE_TICKLESS_IDLE`	저전력 모드 지원	1 (선택적)
`configCPU_CLOCK_HZ`	CPU 클럭 주파수	`SystemCoreClock`
`configTICK_RATE_HZ`	SysTick 주기	1000
`configMAX_PRIORITIES`	최대 Task 우선순위	7~10
`configUSE_MUTEXES`	Mutex 사용	1
`configUSE_COUNTING_SEMAPHORES`	Counting Semaphore	1
`configTOTAL_HEAP_SIZE`	커널 Heap 크기	10KB 이상
`configCHECK_FOR_STACK_OVERFLOW`	스택 오버플로 검출	2

---

5. Heap 관리

CubeMX는 **Heap_4.c** (Best Fit + Free) 방식 기본 선택.
 다른 알고리즘은 다음과 같다.

파일	알고리즘	특징
`heap_1.c`	단순 증가	해제 불가
`heap_2.c`	단순 할당/해제	단편화 가능
`heap_3.c`	`malloc()` 래핑	표준 C 메모리
`heap_4.c`	병합 지원	권장
`heap_5.c`	다중 메모리 영역	고급용

---

6. 스케줄러 시작 및 동작

FreeRTOS는 `osKernelStart()` 호출 시 다음 순서로 동작한다.

1. Idle Task 생성
2. SysTick 설정
3. Task List 초기화
4. Context Switch 활성화
5. 첫 번째 Task 실행

---

7. 디버깅 및 모니터링

(1) FreeRTOS+Trace 지원

CubeIDE의 SWV Trace 기능과 함께 태스크 상태를 시각화 가능.

(2) `uxTaskGetSystemState()`

Task 런타임 상태를 프로그램 상에서 모니터링.

```
UBaseType_t uxArraySize;
TaskStatus_t pxTaskStatusArray[10];
uxArraySize = uxTaskGetSystemState(pxTaskStatusArray, 10, NULL);
```

(3) Stack/Heap 사용량 확인

```
uxTaskGetStackHighWaterMark(NULL);
xPortGetFreeHeapSize();
```

---

8. 요약

- **CubeMX → Middleware → FreeRTOS 활성화**
- Task, Queue, Semaphore를 GUI 기반으로 정의
- 자동 생성 코드로 빠르게 RTOS 통합 가능
- **freertos.c / FreeRTOSConfig.h** 가 핵심 구성 파일
- **CMSIS-RTOS v2 인터페이스**를 통해 HAL과 자연스럽게 연동
- **heap_4.c + Tickless Idle** 조합으로 효율적 저전력 RTOS 시스템 구현 가능

heap_4.c 메모리 모델

1. 개요

FreeRTOS의 동적 메모리 관리 기능은 `pvPortMalloc()`과 `vPortFree()` 함수를 통해 제공된다.
이는 표준 C의 `malloc()` / `free()`와 유사하지만, **RTOS 내부에서 안정적이고 결정적인 메모리 관리**를 위해 별도의 구현 파일(`heap_1.c`~`heap_5.c`)로 분리되어 있다.

그중 `heap_4.c`는 "**Best Fit + Coalescing(인접 블록 병합)**" 알고리즘을 사용하는 가장 권장되는 메모리 모델이다.
임베디드 환경에서 **동적 할당 + 해제 + 메모리 단편화 최소화**가 모두 가능한 구조를 제공한다.

---

2. 구조 및 특징

항목	설명
파일명	`FreeRTOS/Source/portable/MemMang/heap_4.c`
할당 방식	Best Fit (가장 작은 충분한 블록 선택)
해제 처리	인접 Free 블록 자동 병합
메모리 단편화	낮음 (heap_2.c 대비)
필요 함수	`pvPortMalloc()`, `vPortFree()`, `xPortGetFreeHeapSize()`
사용 권장도	★★★★★ (일반 프로젝트 표준)

---

3. 메모리 할당 원리

`heap_4.c`는 `FreeRTOSConfig.h`의 매크로로 지정된
정적 배열 `ucHeap[configTOTAL_HEAP_SIZE]`를 기반으로 작동한다.

```
static uint8_t ucHeap[ configTOTAL_HEAP_SIZE ];
```

RTOS 내부는 이 배열을 "Heap 영역"으로 사용한다.
`pvPortMalloc()` 호출 시 이 영역 내에서 **가장 작은 적합한 빈 블록**을 찾아 할당하고,
`vPortFree()` 호출 시 해당 블록을 **Free 리스트에 반환**한다.
반환 시 인접한 Free 블록이 존재하면 자동 병합(coalescing)하여 단편화를 최소화한다.

---

4. 데이터 구조

`heap_4.c` 내부는 다음과 같은 구조체로 블록 정보를 관리한다.

```
typedef struct A_BLOCK_LINK
{
    struct A_BLOCK_LINK *pxNextFreeBlock;  // 다음 Free 블록 포인터
    size_t xBlockSize;                     // 블록 크기 (Header 포함)
} BlockLink_t;
```

각 블록은 Header(`BlockLink_t`) + Payload(사용 데이터)로 구성되며,
Header는 링크드 리스트 형태로 Free 블록을 연결한다.

---

5. 주요 함수

(1) `pvPortMalloc(size_t xWantedSize)`

- 지정한 크기의 메모리를 동적으로 할당
- 내부적으로 정렬 및 최소 블록 크기 보장
- 성공 시 메모리 포인터 반환, 실패 시 `NULL`

```
void *ptr = pvPortMalloc(128);
if (ptr == NULL) {
    // Heap 부족 처리
}
```

(2) `vPortFree(void *pv)`

- 이전에 `pvPortMalloc()`으로 할당된 메모리 해제
- Free 리스트에 병합 처리 수행

```
vPortFree(ptr);
```

(3) `xPortGetFreeHeapSize()`

- 현재 남은 Heap 크기 반환
- 메모리 사용률 진단용

```
size_t freeHeap = xPortGetFreeHeapSize();
```

(4) `xPortGetMinimumEverFreeHeapSize()`

- 시스템 동작 중 최소 남은 Heap 크기 기록 반환
- Task 스택 크기 및 Heap 최적화에 사용

```
size_t minHeap = xPortGetMinimumEverFreeHeapSize();
```

---

6. 설정 매크로

`FreeRTOSConfig.h`

```
#define configTOTAL_HEAP_SIZE   (10 * 1024)
#define configAPPLICATION_ALLOCATED_HEAP  0
```

- `configTOTAL_HEAP_SIZE` : 전체 Heap 영역 크기 (바이트 단위)
- `configAPPLICATION_ALLOCATED_HEAP` : Heap을 사용자가 직접 정의할 경우 1로 설정

---

7. 장점 및 특징

항목	설명
동적 할당 및 해제 지원	Task, Queue, Semaphore, Timer 등 자유로운 생성/삭제 가능
자동 병합(Coalescing)	메모리 단편화 방지
실시간 안정성 확보	`malloc()` 대비 예측 가능한 동작
메모리 상태 모니터링 가능	잔여/최소 Heap 조회 API 제공
다수의 객체 관리에 유리	RTOS 구조체(Task, Queue 등) 동적 생성에 적합

---

8. 주의사항

1. **Thread-safe**하지 않으므로, FreeRTOS 내부에서만 안전하게 사용해야 한다.
   (`taskENTER_CRITICAL()` 보호 영역 내에서 동작)
2. Heap 영역이 부족하면 커널 객체 생성 실패 또는 HardFault 발생 가능.
3. ISR(인터럽트 서비스 루틴) 내부에서는 `pvPortMalloc()` / `vPortFree()` 사용 금지.
4. 메모리 누수를 방지하기 위해 Task 종료 시 생성한 객체는 반드시 해제해야 한다.

---

9. 메모리 디버깅 팁

함수	용도
`vPortDefineHeapRegions()`	사용자 정의 다중 Heap 영역 등록
`configUSE_MALLOC_FAILED_HOOK`	메모리 부족 시 콜백 호출
`heap_4.c` + `configCHECK_FOR_STACK_OVERFLOW`	스택/Heap 모니터링 병행

예시:

```
void vApplicationMallocFailedHook(void)
{
  printf("Error: Heap Memory Exhausted\r\n");
}
```

---

10. 요약

구분	heap_1	heap_2	heap_3	heap_4	heap_5
해제 가능	X	O	O	**O**	O
병합 기능	X	X	X	**O**	O

구분	heap_1	heap_2	heap_3	heap_4	heap_5
단편화 방지	낮음	중간	중간	**높음**	높음
다중 영역 지원	X	X	X	X	**O**
권장 용도	단순, 고정 Task	일반	PC 이식	**임베디드 표준**	고급 시스템

결론:

`heap_4.c`는 STM32 + FreeRTOS 프로젝트에서 가장 안정적이고 효율적인 메모리 관리 모델이다.
자동 병합 기능으로 장시간 동작 시에도 메모리 누수나 단편화 문제를 최소화하며,
대부분의 임베디드 RTOS 시스템에서 **기본 선택 모델**로 채택된다.

- ## CMSIS-RTOS2 API (`osThreadNew()`, `osDelay()`)

8.3 실습 및 구조화

- ## Task 분리
  - ### SensorTask : 센서 데이터 취득

1. 개요

CMSIS-RTOS2는 Arm에서 정의한 **표준 RTOS 인터페이스 레이어**로,
FreeRTOS, RTX, Zephyr 등 다양한 RTOS 커널 위에서 공통된 API로 동작하도록 설계되었다.
즉, 하위 커널에 종속되지 않고 일관된 함수 형태로 Task(스레드), Queue, Semaphore, Timer를 제어할 수 있다.

STM32CubeMX 및 HAL은 CMSIS-RTOS2 표준을 기반으로 FreeRTOS를 포팅하며,
모든 Task 및 동기화 객체를 `osThreadNew()`, `osDelay()`, `osSemaphoreNew()` 등의 API로 관리한다.

---

2. 핵심 특징

항목	설명
표준화된 API	Arm 공식 CMSIS 사양 기반
커널 독립성	FreeRTOS, RTX 등 다양한 RTOS 호환
RTOS 객체 구조화	Thread, Timer, EventFlags, Semaphore 등
HAL과 자연스러운 연동	CubeMX 자동 생성 코드에서 기본 사용
C언어 표준 인터페이스	C99 호환 함수 형태로 제공

---

3. 스레드 (Thread) 생성 — `osThreadNew()`

(1) 함수 원형

```
osThreadId_t osThreadNew(osThreadFunc_t func, void *argument, const
osThreadAttr_t *attr);
```

(2) 매개변수

인자	설명
`func`	실행할 스레드 함수 포인터 (`void (*func)(void *argument)`)
`argument`	스레드에 전달할 인자 포인터
`attr`	스레드 속성 구조체 (`osThreadAttr_t`) — 이름, 스택 크기, 우선순위 지정 가능

(3) 반환값

- 성공 시 `osThreadId_t` (스레드 핸들)
- 실패 시 `NULL`

(4) 예제 코드

```
#include "cmsis_os2.h"

osThreadId_t sensorTaskHandle;

void SensorTask(void *argument)
{
    for (;;) {
        Read_Sensor();
        osDelay(100);
    }
}

void MX_FREERTOS_Init(void)
{
    const osThreadAttr_t sensorTask_attr = {
        .name = "sensorTask",
        .stack_size = 256 * 4,
        .priority = osPriorityAboveNormal,
    };

    sensorTaskHandle = osThreadNew(SensorTask, NULL, &sensorTask_attr);
}
```

이 코드는 `SensorTask`를 생성하고, 주기적으로 센서를 읽는 FreeRTOS Task를 CMSIS-RTOS2 API로 정의한다.

---

4. 스레드 속성 구조체 — `osThreadAttr_t`

필드명	자료형	설명
`name`	`const char *`	스레드 이름 (디버깅용)
`attr_bits`	`uint32_t`	속성 비트 (기본 0)
`cb_mem`	`void *`	제어 블록 메모리 직접 지정
`cb_size`	`uint32_t`	제어 블록 크기
`stack_mem`	`void *`	스택 메모리 직접 지정
`stack_size`	`uint32_t`	스택 크기 (byte 단위)
`priority`	`osPriority_t`	우선순위 설정
`tz_module`	`uint32_t`	TrustZone 모듈 ID (보안 관련)

---

5. 지연 (Delay) 함수 — `osDelay()`

(1) 함수 원형

```
osStatus_t osDelay(uint32_t ms);
```

(2) 설명

- 지정한 **밀리초(ms)** 동안 현재 스레드를 일시 중단하고,
  다른 준비 상태(Ready)의 Task로 CPU를 양도한다.
- 내부적으로는 FreeRTOS의 `vTaskDelay()` 또는 `osDelayUntil()`을 래핑한다.
- 커널의 Tick 주기(`configTICK_RATE_HZ`)를 기반으로 동작한다.
  예: Tick 주기 = 1ms → `osDelay(100)` = 100ms 지연.

(3) 반환값

값	의미
`osOK`	정상 종료
`osErrorTimeoutResource`	지연 중 인터럽트 발생 등으로 조기 종료
`osErrorParameter`	인자 오류

(4) 예제

```
void DisplayTask(void *argument)
{
    for (;;) {
        OLED_Update();
        osDelay(200); // 200ms마다 화면 갱신
    }
}
```

---

6. 우선순위 관리

CMSIS-RTOS2는 다음과 같은 Enum을 통해 스레드 우선순위를 지정한다.

```
typedef enum {
  osPriorityNone          =  0,
  osPriorityIdle          =  1,
  osPriorityLow           =  2,
  osPriorityBelowNormal   =  3,
  osPriorityNormal        =  4,
  osPriorityAboveNormal   =  5,
  osPriorityHigh          =  6,
  osPriorityRealtime      =  7,
  osPriorityISR           =  8
} osPriority_t;
```

FreeRTOS 내부에서는 해당 값이 커널의 우선순위 정수(`tskIDLE_PRIORITY + n`)로 매핑된다.

---

7. 스레드 제어 API 요약

함수	설명
`osThreadNew()`	스레드(Task) 생성
`osThreadTerminate()`	스레드 종료
`osThreadSuspend()` / `osThreadResume()`	일시 중단 및 재개
`osThreadYield()`	동일 우선순위 스레드 간 CPU 양보
`osDelay()`	일정 시간 대기 (ms 단위)
`osDelayUntil()`	절대 Tick 기반 대기
`osThreadGetId()`	현재 스레드 ID 반환
`osThreadGetState()`	스레드 상태 조회

---

8. CMSIS-RTOS2와 FreeRTOS의 대응 관계

CMSIS-RTOS2 함수	FreeRTOS 내부 매핑
`osThreadNew()`	`xTaskCreate()`
`osDelay()`	`vTaskDelay()`
`osSemaphoreNew()`	`xSemaphoreCreateBinary()`
`osMutexNew()`	`xSemaphoreCreateMutex()`
`osMessageQueueNew()`	`xQueueCreate()`
`osKernelStart()`	`vTaskStartScheduler()`

---

9. 예제 — FreeRTOS 기반 CMSIS Task 구성

```
void StartTasks(void)
{
    const osThreadAttr_t task1_attr = {
        .name = "SensorTask",
        .priority = osPriorityAboveNormal,
        .stack_size = 256 * 4
    };

    const osThreadAttr_t task2_attr = {
        .name = "ControlTask",
        .priority = osPriorityNormal,
        .stack_size = 256 * 4
    };

    osThreadNew(SensorTask, NULL, &task1_attr);
    osThreadNew(ControlTask, NULL, &task2_attr);
}
```

각 Task는 병렬적으로 실행되며, 커널이 Tick 단위로 스케줄링한다.

---

10. 요약

- CMSIS-RTOS2는 **Arm 공식 RTOS 표준 인터페이스**
- FreeRTOS를 포함한 다양한 커널에서 동일 API로 동작
- `osThreadNew()`로 Task 생성, `osDelay()`로 주기 제어
- HAL, CubeIDE와 완전 호환되어 **STM32에서 표준 RTOS 포팅 방식**으로 사용됨
- 코드 이식성, 유지보수성, 가독성 측면에서 **가장 권장되는 FreeRTOS 사용 방식**

DisplayTask : UART / OLED 출력

1. 개요

`DisplayTask` 는 주기적으로 **센서 데이터, 시스템 상태, 디버그 정보** 등을
UART 시리얼 터미널과 **OLED 디스플레이(SSD1306 등)** 양쪽에 출력하는 역할을 수행한다.
이 Task는 RTOS 환경에서 동작하며, 다른 센서/연산 Task와 병렬 실행된다.

UART 출력은 로깅 및 모니터링 용도,
OLED 출력은 실시간 사용자 표시(UI) 용도로 사용된다.

---

2. 시스템 구조

```
+------------------+
| SensorTask       | → 센서 데이터 측정
+------------------+
         ↓ Queue or Global
+------------------+
| DisplayTask      | → UART, OLED로 상태 표시
+------------------+
```

DisplayTask는 큐나 전역 구조체를 통해 다른 Task가 생산한 데이터를 읽는다.
주기적 업데이트는 `osDelay()` 로 조절하며, CPU 점유율을 최소화한다.

---

3. 초기화

(1) UART 초기화

```
extern UART_HandleTypeDef huart1;  // CubeMX 생성 핸들 사용

void UART_Init(void)
{
    // CubeMX 자동 생성 코드 사용
    // HAL_UART_Init(&huart1); → 이미 MX_USART1_UART_Init()에서 수행
}
```

(2) OLED 초기화

```
#include "ssd1306.h"
#include "fonts.h"

void OLED_Init_Display(void)
{
    ssd1306_Init();
    ssd1306_Fill(Black);
    ssd1306_UpdateScreen();
}
```

---

4. Task 구현

```c
#include "cmsis_os2.h"
#include "ssd1306.h"
#include "string.h"
#include "stdio.h"

extern UART_HandleTypeDef huart1;

typedef struct {
    float distance_mm;
    float weight_g;
    float level_mm;
} SensorData_t;

extern SensorData_t gSensorData; // 다른 Task에서 갱신

void DisplayTask(void *argument)
{
    char msg[64];

    for (;;) {
        // UART 출력
        snprintf(msg, sizeof(msg),
                 "Dist: %.1f mm | W: %.1f g | Lvl: %.1f mm\r\n",
                 gSensorData.distance_mm,
                 gSensorData.weight_g,
                 gSensorData.level_mm);

        HAL_UART_Transmit(&huart1, (uint8_t*)msg, strlen(msg), 100);

        // OLED 출력
        ssd1306_Fill(Black);
        ssd1306_SetCursor(0, 0);
        ssd1306_WriteString("SYSTEM STATUS", Font_7x10, White);

        ssd1306_SetCursor(0, 16);
        sprintf(msg, "Dist: %.1f mm", gSensorData.distance_mm);
        ssd1306_WriteString(msg, Font_6x8, White);

        ssd1306_SetCursor(0, 28);
        sprintf(msg, "Weight: %.1f g", gSensorData.weight_g);
        ssd1306_WriteString(msg, Font_6x8, White);

        ssd1306_SetCursor(0, 40);
        sprintf(msg, "Level: %.1f mm", gSensorData.level_mm);
        ssd1306_WriteString(msg, Font_6x8, White);

        ssd1306_UpdateScreen();

        osDelay(500); // 0.5초마다 갱신
    }
```

```
}
```

---

5. 출력 예시

UART 로그 (시리얼 터미널)

```
Dist: 125.3 mm | W: 84.5 g | Lvl: 52.0 mm
Dist: 125.2 mm | W: 84.6 g | Lvl: 52.0 mm
...
```

OLED 화면

```
SYSTEM STATUS
Dist: 125.3 mm
Weight: 84.5 g
Level: 52.0 mm
```

---

6. Task 등록

```
void MX_FREERTOS_Init(void)
{
    const osThreadAttr_t displayTask_attr = {
        .name = "DisplayTask",
        .stack_size = 512 * 4,
        .priority = osPriorityNormal,
    };

    osThreadNew(DisplayTask, NULL, &displayTask_attr);
}
```

---

7. 동작 요약

항목	설명
입력	`gSensorData` 구조체 (센서 Task에서 갱신)
출력	UART 텍스트, OLED 시각정보
주기	500ms
통신 인터페이스	I²C (OLED), UART (로깅)
주요 API	`HAL_UART_Transmit()`, `ssd1306_WriteString()`, `osDelay()`

---

8. 확장 포인트

- **큐(Queue) 기반 메시지 전달**로 구조 개선 (`osMessageQueuePut()` 사용)
- **오류 상태 LED 표시** 추가
- **UI 페이지 전환 (버튼 입력 연동)**
- **RTC 시각 표시 기능 통합**
- **BLE/UART 동시 출력** 모드

결론

`DisplayTask`는 STM32 시스템의 **UI 및 로깅 담당 핵심 Task**로,
센서·통신 Task와의 비동기 협업 구조를 통해
UART 디버깅과 **OLED 실시간 표시**를 안정적으로 병행한다.
이는 RTOS 기반 임베디드 시스템의 표준적인 시각화 루틴 구조이다.

ControlTask : 밸브/펌프 제어

1. 개요

`ControlTask`는 **수위 센서, 무게 센서, 초음파 센서** 등의 측정값을 기반으로
밸브(Valve) 와 **펌프(Pump)** 를 자동 제어하는 **폐루프(Closed-loop) 제어 Task**이다.

이 Task는 **FreeRTOS 스케줄러** 하에서 주기적으로 실행되며,
임계값(Threshold), 히스테리시스(Hysteresis), 안전 조건(Safety Flag)을 고려하여
펌프의 동작과 밸브 개폐를 안정적으로 수행한다.

---

2. 제어 대상

제어기기	타입	GPIO 핀	제어 논리	설명
Pump_OUT	릴레이 출력	PB0	High = 펌프 ON	물 주입 또는 배수
Valve_OUT	솔레노이드 밸브	PB1	High = 밸브 개방	유입 밸브 또는 배수 밸브
Level Sensor	정전식 20단	I²C(0x48, 0x49)	Read-only	수위 피드백
Weight Sensor	HX711	GPIO 입력	Read-only	중량 보정용 피드백

---

3. 제어 전략

1. **Threshold 기반 수위 제어**
   - 목표 수위(`LEVEL_TARGET`)와 허용 오차(`LEVEL_TOLERANCE`)를 설정
   - 수위가 너무 낮으면 → **펌프 ON, 밸브 OPEN**
   - 수위가 충분하면 → **펌프 OFF, 밸브 CLOSE**
2. **히스테리시스 적용**
   - 상한 / 하한 경계 구간을 두어
     불필요한 반복 ON/OFF 진동을 방지
3. **보조 제어 로직**
   - 무게 기준 보정 (탱크 무게 과부하 시 강제 정지)
   - 초음파 센서 데이터로 수위 보조 판단
   - 알람/안전 플래그 감시 (오류 발생 시 즉시 정지)

---

4. 제어 파라미터

```
#define LEVEL_TARGET_MM        80.0f   // 목표 수위 (mm)
#define LEVEL_TOLERANCE_MM     5.0f    // 허용 오차 범위 (mm)
#define WEIGHT_MAX_G           500.0f  // 과부하 중량 (g)
#define SENSOR_TIMEOUT_MS      2000    // 센서 응답 제한 시간
```

---

5. Task 구현

```
#include "cmsis_os2.h"
#include "main.h"
#include "stdio.h"

extern float gLevel_mm;
extern float gWeight_g;
extern float gDistance_mm;

void Pump_On(void)   { HAL_GPIO_WritePin(GPIOB, GPIO_PIN_0, GPIO_PIN_SET); }
void Pump_Off(void)  { HAL_GPIO_WritePin(GPIOB, GPIO_PIN_0, GPIO_PIN_RESET); }
void Valve_Open(void){ HAL_GPIO_WritePin(GPIOB, GPIO_PIN_1, GPIO_PIN_SET); }
void Valve_Close(void){HAL_GPIO_WritePin(GPIOB, GPIO_PIN_1, GPIO_PIN_RESET); }

void ControlTask(void *argument)
{
    uint8_t pump_state = 0;
    uint8_t valve_state = 0;
    uint32_t last_action_tick = 0;

    for (;;) {
        float level = gLevel_mm;
        float weight = gWeight_g;
```

```
        float distance = gDistance_mm;

        // 기본 상태 출력 (UART 로그용)
        printf("CTRL | Level: %.1f mm | Weight: %.1f g | Dist: %.1f mm\r\n",
               level, weight, distance);

        // 1) 과부하 보호
        if (weight > WEIGHT_MAX_G) {
            Pump_Off();
            Valve_Close();
            printf("CTRL: Overweight Protection!\r\n");
            osDelay(500);
            continue;
        }

        // 2) 수위 기반 제어
        if (level < (LEVEL_TARGET_MM - LEVEL_TOLERANCE_MM)) {
            if (!pump_state) {
                Pump_On();
                pump_state = 1;
                printf("CTRL: Pump ON\r\n");
            }
            if (!valve_state) {
                Valve_Open();
                valve_state = 1;
                printf("CTRL: Valve OPEN\r\n");
            }
        }
        else if (level > (LEVEL_TARGET_MM + LEVEL_TOLERANCE_MM)) {
            if (pump_state) {
                Pump_Off();
                pump_state = 0;
                printf("CTRL: Pump OFF\r\n");
            }
            if (valve_state) {
                Valve_Close();
                valve_state = 0;
                printf("CTRL: Valve CLOSE\r\n");
            }
        }

        // 3) 주기적 상태 확인
        osDelay(200);
    }
}
```

6. Task 등록

```
void MX_FREERTOS_Init(void)
{
    const osThreadAttr_t controlTask_attr = {
        .name = "ControlTask",
        .stack_size = 512 * 4,
        .priority = osPriorityAboveNormal,
    };

    osThreadNew(ControlTask, NULL, &controlTask_attr);
}
```

---

7. 동작 시나리오

상태	수위(mm)	펌프	밸브	비고
저수위 감지	< 75	ON	OPEN	충수 시작
정상 수위	≈ 80	유지	유지	안정 상태
고수위 초과	> 85	OFF	CLOSE	충수 종료
과부하 감지	무게 > 500g	OFF	CLOSE	보호 모드

---

8. 보조 기능

- **Manual Override**: 외부 명령(UART/BLE)으로 강제 제어
- **Error Recovery**: 일정 시간 이상 반응 없음 → 자동 정지
- **Watchdog Trigger**: 주기적 피드백으로 시스템 헹잉 방지
- **LED 표시기 연동**: 펌프/밸브 상태 시각화

---

9. 결론

`ControlTask` 는 RTOS 기반의 **자동 수위 제어 루프**로,
센서 데이터를 실시간 반영하여 **밸브/펌프를 지능적으로 제어**한다.
히스테리시스 및 보호 로직을 적용함으로써
안정적 제어와 하드웨어 수명 보존을 동시에 달성한다.

RTCTask : 알람 스케줄링

1. 개요

`RTCTask` 는 **STM32 내부 RTC(Real-Time Clock)** 를 이용하여
시간 기반 이벤트 스케줄링을 수행하는 FreeRTOS Task이다.
이 Task는 주기적으로 RTC 시간을 확인하고,
알람(Alarm) 혹은 예약된 측정 이벤트를 등록·관리한다.

주요 기능은 다음과 같다.

- RTC 시간 읽기 및 표시
- `HAL_RTC_SetAlarm_IT()`을 통한 알람 인터럽트 설정
- 알람 발생 시 다른 Task로 **Notification** 또는 **Queue Message** 전달
- **저전력 모드(SLEEP / STOP)** 에서의 깨움(Wake-up) 제어

---

2. 시스템 구조

구성요소	역할
RTC Peripheral	LSE(32.768 kHz) 기반 실시간 시간 유지
HAL_RTC_GetTime/Date()	현재 시간 읽기
HAL_RTC_SetAlarm_IT()	특정 시각 알람 인터럽트 등록
RTC_Alarm_IRQHandler()	알람 인터럽트 진입점
FreeRTOS RTCTask	스케줄 관리 및 Task 간 이벤트 전달

---

3. 주요 기능 흐름

1. 시스템 부팅 시, RTC가 초기화되어 현재 시각이 설정됨 (`Set_RTC()`)
2. `RTCTask`는 주기적으로 RTC 시간을 읽어 UART 또는 OLED에 출력
3. 특정 주기(예: 매 1분, 매 10초 등)마다 `Set_Alarm()` 호출
4. 알람 발생 시, `RTC_Alarm_IRQHandler()` → `HAL_RTC_AlarmAEventCallback()` 호출
5. Callback 내부에서 RTOS Task에 Notification 전달
6. RTCTask는 알람 이벤트를 수신하고 측정 루틴 또는 ControlTask를 실행시킴

---

4. 예시 코드 구조

```
#include "cmsis_os2.h"
#include "main.h"
#include "rtc.h"
#include "stdio.h"

extern RTC_HandleTypeDef hrtc;
osThreadId_t rtcTaskHandle;

void RTCTask(void *argument)
{
    RTC_TimeTypeDef sTime;
    RTC_DateTypeDef sDate;
    RTC_AlarmTypeDef sAlarm;
    uint8_t alarm_flag = 0;
```

```
15
16      for (;;) {
17          // 1) 현재 시간 읽기
18          HAL_RTC_GetTime(&hrtc, &sTime, RTC_FORMAT_BIN);
19          HAL_RTC_GetDate(&hrtc, &sDate, RTC_FORMAT_BIN);
20
21          printf("RTC | %02d:%02d:%02d  %02d-%02d-%02d\r\n",
22              sTime.Hours, sTime.Minutes, sTime.Seconds,
23              sDate.Year + 2000, sDate.Month, sDate.Date);
24
25          // 2) 주기적으로 알람 설정 (10초 뒤)
26          sAlarm.AlarmTime.Seconds = (sTime.Seconds + 10) % 60;
27          sAlarm.AlarmTime.Minutes = sTime.Minutes;
28          sAlarm.AlarmTime.Hours   = sTime.Hours;
29          sAlarm.AlarmMask = RTC_ALARMMASK_DATEWEEKDAY;
30          sAlarm.Alarm = RTC_ALARM_A;
31          HAL_RTC_SetAlarm_IT(&hrtc, &sAlarm, RTC_FORMAT_BIN);
32
33          // 3) 알람 발생 이벤트 대기
34          if (alarm_flag) {
35              printf("RTC Alarm Triggered → Sensor Measurement Start\r\n");
36              alarm_flag = 0;
37          }
38
39          osDelay(1000);
40      }
41  }
42
43  // 알람 인터럽트 콜백
44  void HAL_RTC_AlarmAEventCallback(RTC_HandleTypeDef *hrtc)
45  {
46      BaseType_t xHigherPriorityTaskWoken = pdFALSE;
47
48      // 알람 발생 시 Task에 알림
49      vTaskNotifyGiveFromISR(rtcTaskHandle, &xHigherPriorityTaskWoken);
50      portYIELD_FROM_ISR(xHigherPriorityTaskWoken);
51  }
```

5. Task 생성 및 등록

```
1  void MX_FREERTOS_Init(void)
2  {
3      const osThreadAttr_t rtcTask_attr = {
4          .name = "RTCTask",
5          .stack_size = 512 * 4,
6          .priority = osPriorityNormal,
7      };
8      rtcTaskHandle = osThreadNew(RTCTask, NULL, &rtcTask_attr);
9  }
```

6. 스케줄링 시나리오

이벤트	동작
부팅 시	RTC 초기화 및 현재 시각 설정
주기적 실행	매 10초 후 알람 재등록
알람 인터럽트	RTCTask에 Notification 전달
RTCTask 처리	측정 Task 또는 ControlTask 실행
저전력 모드	STOP 모드 진입 → 알람으로 자동 Wake-up

---

7. 저전력 동작 연계

`RTCTask`는 **Tickless Idle Mode**와 함께 사용되어
다음 알람까지 MCU를 **STOP 모드**로 진입시킨 후,
RTC 알람 인터럽트가 발생하면 **자동으로 깨워서 측정 루틴 수행**을 트리거한다.

이 방식은 배터리 기반 **저전력 IoT 센서 노드** 설계에 필수적이다.

---

8. 결론

`RTCTask`는 시스템의 **시간 기반 이벤트 허브(Time Scheduler)** 로서,
다른 Task에 측정, 로깅, 제어 등의 실행 타이밍을 제공한다.
RTC 알람 인터럽트와 FreeRTOS Task Notification을 연계하여
정확하고 효율적인 주기 제어 및 저전력 동작을 동시에 실현한다.

- # Queue / Mutex / Semaphore 사용

1. 개요

FreeRTOS는 멀티태스킹 환경에서 **Task 간 안전한 데이터 공유**와 **동기화(Synchronization)** 를 위해
세 가지 핵심 객체를 제공한다: **Queue**, **Mutex**, **Semaphore**.

이들은 각각의 목적이 명확하며, STM32 환경에서도 HAL 드라이버와 함께 안정적으로 사용된다.

---

2. Queue (큐) — Task 간 데이터 전달

Queue는 여러 Task 간에 데이터를 **FIFO(First-In First-Out)** 방식으로 교환하는 구조체이다.
센서 데이터, 명령, 이벤트 플래그 등을 안전하게 전달하기 위해 사용된다.

2.1 주요 특징

- 데이터 복사 기반 (Pointer 전달 아님)
- 송신(`xQueueSend`)과 수신(`xQueueReceive`) 가능
- Queue가 가득 찬 경우 블록 대기 또는 Timeout 설정 가능
- RTOS 내에서 Task 간 우선순위 역전 없이 안전하게 동작

2.2 예시 코드

```c
#include "cmsis_os2.h"

typedef struct {
    float level_mm;
    float weight_g;
    float distance_mm;
} SensorData_t;

osMessageQueueId_t sensorQueue;

void SensorTask(void *argument)
{
    SensorData_t data;

    for (;;) {
        data.level_mm = Read_Water_Level();
        data.weight_g = HX711_Get_Value();
        data.distance_mm = Read_Ultrasonic();

        osMessageQueuePut(sensorQueue, &data, 0, 0);
        osDelay(500);
    }
}

void DisplayTask(void *argument)
{
    SensorData_t recv;

    for (;;) {
        if (osMessageQueueGet(sensorQueue, &recv, NULL, osWaitForever) == osOK) {
            printf("Level: %.1f mm | Weight: %.1f g | Dist: %.1f mm\r\n",
                recv.level_mm, recv.weight_g, recv.distance_mm);
        }
    }
}

void MX_FREERTOS_Init(void)
{
    sensorQueue = osMessageQueueNew(8, sizeof(SensorData_t), NULL);
    osThreadNew(SensorTask, NULL, NULL);
    osThreadNew(DisplayTask, NULL, NULL);
}
```

---

3. Mutex (뮤텍스) — 자원 보호

Mutex (Mutual Exclusion) 는 공유 자원(예: OLED, UART, Flash)에 대한
단일 접근 보장을 위해 사용된다.

여러 Task가 동시에 동일 자원에 접근할 경우,
뮤텍스가 Lock된 상태에서는 다른 Task의 접근을 차단하여
데이터 충돌이나 비정상 동작을 방지한다.

3.1 주요 특징

- Binary Semaphore와 유사하나 **소유권(Task Ownership)** 이 존재
- 동일 Task가 여러 번 Lock 가능 (Recursive Mutex 지원)
- 우선순위 역전(Priority Inversion) 방지 기법 포함

3.2 예시 코드

```
osMutexId_t uartMutex;

void UART_Print(const char *msg)
{
    osMutexAcquire(uartMutex, osWaitForever);
    printf("%s", msg);
    osMutexRelease(uartMutex);
}

void TaskA(void *arg)
{
    for (;;) {
        UART_Print("Task A reporting\r\n");
        osDelay(1000);
    }
}

void TaskB(void *arg)
{
    for (;;) {
        UART_Print("Task B reporting\r\n");
        osDelay(1500);
    }
}

void MX_FREERTOS_Init(void)
{
    uartMutex = osMutexNew(NULL);
    osThreadNew(TaskA, NULL, NULL);
    osThreadNew(TaskB, NULL, NULL);
}
```

---

4. Semaphore (세마포어) — 동기화 신호

Semaphore는 Task 간의 이벤트 동기화나 하드웨어 인터럽트 신호 전달에 사용된다.
특히 **Binary Semaphore**는 "이벤트 플래그" 역할을 하며,
Counting Semaphore는 리소스 개수 제한이나 다중 이벤트 처리에 적합하다.

4.1 주요 용도

- 인터럽트에서 Task로 신호 전달
- 특정 Task의 실행 조건 제어
- 리소스 사용 카운트 관리

4.2 예시: 인터럽트 → Task 동기화

```
osSemaphoreId_t echoSem;

void HAL_GPIO_EXTI_Callback(uint16_t GPIO_Pin)
{
    if (GPIO_Pin == ECHO_PIN) {
        osSemaphoreRelease(echoSem);  // 초음파 Echo 완료 시 Task 깨움
    }
}

void UltrasonicTask(void *argument)
{
    for (;;) {
        Trigger_Ultrasonic();
        if (osSemaphoreAcquire(echoSem, 100) == osOK) {
            float dist = Get_Distance();
            printf("Distance: %.1f mm\r\n", dist);
        } else {
            printf("Echo Timeout\r\n");
        }
        osDelay(500);
    }
}

void MX_FREERTOS_Init(void)
{
    echoSem = osSemaphoreNew(1, 0, NULL);
    osThreadNew(UltrasonicTask, NULL, NULL);
}
```

---

5. 비교 요약

기능	Queue	Mutex	Semaphore
목적	데이터 전달	자원 보호	동기화 신호
데이터 포함 여부	있음	없음	없음
소유권(Task Ownership)	없음	있음	없음
사용 예시	센서 데이터 교환	UART / OLED 접근 보호	IRQ → Task 이벤트 전달
블록/대기 가능	O	O	O

---

6. 결론

`Queue`, `Mutex`, `Semaphore`는 FreeRTOS에서 **Task 간 통신 및 자원 보호의 핵심 도구**이다.
적절히 조합하면, STM32 기반 계측 시스템에서
데이터 일관성, 동기화, 안정성을 모두 확보할 수 있다.

특히 센서 데이터 전달(Queue) → 출력 보호(Mutex) → 인터럽트 동기화(Semaphore) 구조로
RTOS 기반 **모듈화된 안정 시스템**을 구현할 수 있다.

- # Tickless Idle (Sleep) 적용

1. 개요

Tickless Idle은 FreeRTOS의 저전력 기능으로,
시스템이 유휴 상태(Idle Task만 실행 중)일 때 **SysTick 인터럽트를 정지**시키고,
MCU를 **Sleep 모드**로 전환하여 전력 소모를 최소화한다.

기본적으로 FreeRTOS는 주기적으로 SysTick(1ms 단위)을 발생시켜 Task 스케줄링을 수행하지만,
Tickless 모드를 사용하면 일정 기간 동안 이벤트(Task Ready, Interrupt 등)가 없을 경우
그 기간만큼 Tick을 "건너뛰고" 깨어나므로 불필요한 CPU 활동이 사라진다.

---

2. 기본 원리

1. **RTOS Idle 상태 진입 감지**
   - 모든 Task가 Blocked 상태가 되어 Scheduler가 Idle Task만 수행.
2. **다음 이벤트까지 남은 시간 계산**
   - `xNextTaskUnblockTime`을 참조하여 다음 Task가 깨어날 시간(`expectedIdleTime`) 계산.
3. **SysTick 타이머 중지**
   - SysTick Interrupt를 정지하고 남은 시간을 백업.
4. **MCU Sleep 모드 진입**
   - `__WFI()` (Wait For Interrupt) 명령어로 Sleep 상태 진입.
5. **인터럽트 또는 RTC 알람 등으로 Wake-up**
   - 외부/내부 이벤트 발생 시, 시스템이 즉시 깨어남.
6. **누락된 Tick 보정 후 재시작**
   - 실제 Sleep 시간만큼 OS Tick Counter를 조정.

---

3. CubeMX 설정 절차

1. **FreeRTOS 활성화**
   - Middleware → FreeRTOS → Configuration → "Enable" 선택.
2. **Tickless Idle 기능 활성화**
   - FreeRTOS 설정 탭에서
     → *"Enable FreeRTOS tickless idle"* 체크.
   - `configUSE_TICKLESS_IDLE` 매크로가 자동으로 `1`로 설정됨.

3. **Idle Hook 설정(선택)**
    - `vApplicationIdleHook()` 콜백을 사용해 진입 전/후 동작 추가 가능.

---

4. 주요 설정 매크로

`FreeRTOSConfig.h` 파일에서 다음과 같이 지정된다.

```
#define configUSE_TICKLESS_IDLE        1
#define configEXPECTED_IDLE_TIME_BEFORE_SLEEP   2
#define configUSE_IDLE_HOOK            1
#define configCPU_CLOCK_HZ             (SystemCoreClock)
#define configTICK_RATE_HZ             ((TickType_t)1000)
```

옵션 설명:

매크로	설명
`configUSE_TICKLESS_IDLE`	Tickless Idle 기능 활성화
`configEXPECTED_IDLE_TIME_BEFORE_SLEEP`	Sleep 진입 전 최소 대기 Tick 수
`configUSE_IDLE_HOOK`	Idle Hook 함수 사용 여부
`configTICK_RATE_HZ`	SysTick 주기 (일반적으로 1kHz)

---

5. 사용자 Idle Hook 예시

```
void vApplicationIdleHook(void)
{
    /* 유휴 시 전력 절감용 Sleep 진입 */
    __WFI();   // Wait For Interrupt
}
```

위 코드는 FreeRTOS가 Idle 상태일 때 CPU를 자동으로 Sleep 모드에 진입시킨다.
SysTick이 멈춘 상태에서도 인터럽트(예: RTC, EXTI, UART Rx 등)가 발생하면 즉시 깨어난다.

---

6. Tickless Idle의 FreeRTOS 동작 예시

기존 (Normal Tick)

```
Tick 1 → Scheduler → Tick 2 → Scheduler → Tick 3 → ...
```

Tickless (Idle 구간 포함)

```
Tick 1 → [Idle] → (Sleep) → Wake-up → Tick 120 → Scheduler 재개
```

즉, Sleep 중에는 **Tick Interrupt가 완전히 비활성화**되고,
Wake-up 시 OS는 누락된 Tick을 계산하여 **Task 지연 시간 보정**을 수행한다.

---

7. RTC 기반 Tickless 보정

Tickless 모드는 일반적으로 **SysTick Timer**의 대체 수단으로
RTC(LSE 32.768kHz) 또는 **LPTIM**을 사용한다.
이 방식은 장기 Sleep에서 오차를 최소화하며 초저전력 설계에 적합하다.

예시 (RTC Alarm으로 깨움):

```
void vPortSuppressTicksAndSleep(TickType_t xExpectedIdleTime)
{
    StopSysTick();
    ConfigureRTCWakeUp(xExpectedIdleTime);
    __WFI();  // Sleep
    RestartSysTick();
}
```

---

8. 응용 시나리오

- **데이터 로거**: RTC Alarm 주기(예: 5초)로 깨어나 측정 → Sleep
- **BLE Sensor Node**: BLE 이벤트 없을 시 Sleep → 인터럽트 수신 시 재개
- **배터리 기반 IoT 시스템**: 90% 이상 Idle 유지 시 전류 소모 수 μA 단위까지 절감

---

9. 주의사항

항목	설명
디버깅 시 Sleep 진입 금지	Sleep 중 디버거 연결이 끊길 수 있음
SysTick 사용 중단 주의	HAL_Delay() 등 SysTick 기반 함수 사용 제한
인터럽트 우선순위 관리	Wake-up 원인이 되는 IRQ 우선순위 확인 필요
전원 관리 함수 동기화	Sleep 진입 전 주변장치(ADC, UART 등) 클럭 정지 필요

---

10. 결론

Tickless Idle은 FreeRTOS 기반 STM32 시스템에서
전력 효율을 극대화하기 위한 핵심 기능이다.
SysTick 중단, Sleep 진입, RTC 기반 보정을 통해
CPU 활동을 최소화하며,
센서 기반 장치나 IoT 노드에서 수 배의 배터리 수명을 확보할 수 있다.